university of groningen / faculty of science and engineering

Q-learning adaptations in the game Othello

D. Krol, d.j.krol.1@student.rug.nl,
J. van Brandenburg, j.c.van.brandenburg@student.rug.nl,
Supervisor: Dr M.A. Wiering

Abstract: Reinforcement learning algorithms are widely used algorithms concerning action selection to maximize the reward for a specific situation. Q-learning is such an algorithm. It estimates the quality of performing an action in a certain state. These estimations are continuously updated with each experience. In this paper we compare different adaptations to the Q-learning algorithm to learn an agent play the board game Othello. We discuss the use of a second estimator in Double Q-learning, the addition of a V-value function in QV- and QV2-learning, and we consider the on-policy variant of Q-learning called SARSA. A multilayer perceptron is used as a function approximator and is compared to the use of a convolutional neural network. Results indicate that SARSA, QV- and QV2-learning perform better than Q-learning. The addition of a second estimator in Double Q-learning does not seem to improve the performance of Q-learning. SARSA and QV2-learning converge slower and they struggle to escape local minima, while QV-learning converges faster. Results show that the multilayer perceptron outperforms the convolutional neural network.

1 Introduction

In January of 2019, one of the world's strongest professional Starcraft players was beaten by AlphaStar, an AI trained using Reinforcement Learning (Wender and Watson, 2012). It learned how to play using an algorithm called Q-learning (Watkins and Dayan, 1992), and adaptations of Q-learning. These algorithms are widely used in machine learning applications. They are used to generate music playlists (King and Imbrasaitė, 2015), to set prices in competitive marketplaces (Tesauro and Kephart, 1999) and even to maintain security and privacy in health-care systems (Shakeel et al., 2018).

Q-learning trains an agent to obtain the highest rewards. While the agent explores the environment, actions and states are evaluated. This evaluation corresponds to the expected sum of rewards obtained in the future. This way the agent knows how good it is to be in a state. Q-learning continuously updates these evaluations while the agent traverses through the environment. Board games are a perfect platform for testing reinforcement learning algorithms, such as Q-learning (Watkins and Dayan, 1992). They are often deterministic, multi agent, and have a fixed set of rules. There are many reinforcement algorithms based on Q-learning. These Q-learning adaptations can enhance the algorithm in different ways, such as faster learning speed and a higher performance. Some of these algorithms have already been implemented in the board game Othello before (van der Ree and Wiering, 2013). In this paper we discuss these Q-learning adaptations and others, which have not yet been applied to Othello.

One Q-learning variant is the on-policy variant called SARSA (Rummery and Niranjan, 1994). On-policy means that the agent updates its policy based on the performed actions. The following research question is asked: *How does Q-learning compare to the on-policy variant, SARSA, in terms of performance?*

In Q-learning the agent chooses the action that has the maximum expected Q-value. This can result in overestimation and positive bias. We explore a relatively new approach to Q-learning called Double Q-learning (Hado, 2010). In this algorithm two different networks are used to bypass the positive bias. The following research question is posed: *How does the performance of Double Q-learning compare to Q-learning?*

The QV-family is a set of learning algorithms, introduced by Wiering (2005). It combines Q-learning with original basic TD-learning methods. Two members of the family, QV- and QV2-learning,

are implemented to answer the question: *How does the performance of the QV-family compare to Q-learning?*

Q-learning is often used in combination with a neural network. A multilayer perceptron (MLP) can be used to estimate the Q-values removing the need to store individual state evaluations in a lookup table. This allows the agent to act in never before seen environments.

Convolutional neural networks (CNN) are frequently used in image analysis to extract complex spatial patterns (Schmidhuber, 2015). They show surprisingly good results when applied to board games, where the board can be seen as a two dimensional image (Smith, 2015). In previous research a CNN was trained on a huge set of data to learn to play the game Othello (Liskowski et al., 2018). This data set contained board compositions and moves of games played by expert human players. The use of a CNN showed to be an accurate approach to predict expert human moves. We are interested in the use of a CNN in combination with the previous mentioned algorithms. The following research questions is posed: *How does the usage of a convolutional neural network instead of a multilayer perceptron affect the performance of Q-learning algorithms?*

This research is not about making the best Othello playing agent. The interest goes out to the performance comparison of the learning algorithms. We compare the performance by training the agents against fixed opponents, while learning from their own moves.

Outline. In the next section we explain the board game Othello. In Section 3 the theory of reinforcement learning and different algorithms are discussed. Furthermore, it describes the application of the algorithms to Othello. Section 4 describes the experiments and Section 5 shows the results of these experiments. Finally we discuss the results in Section 6 and conclude our research in Section 7.

Figure 2.1: User interface of Othello. White starts. Yellow discs represent possible legal moves.

2 Othello

Othello is a two player strategy board game played on a 8×8 board. It is a variant of the original board game Reversi, invented in 1883, and differs in the initial setup of the board. In the game each player is represented by a color, either black or white. Discs, black on one side and white on the other, can be placed to make a play. The goal of the game is to have the most discs faced up with your color at the end of the game. The initial game setup starts with four discs in the center square of the board. The top right and bottom left discs are white and the two remaining discs are black. The player using white discs always starts the game. A disc has to be placed in a position, such that there is at least one straight occupied line, between the placed disc and another disc of the player. This horizontal, vertical or diagonal line may only consist of discs of the opponent. All enclosed discs of the opponent in this line are then flipped such that they now have the color of the player. The turn now alternates to the other player. If a player cannot make a move, the turn also alternates to the other player. The yellow discs in Figure 2.1 represent the legal moves for the player using white discs. The game finishes when both players consecutively are unable to make a move. The player with the most discs on the board wins the game. A game is tied when both players have the same amount of discs.

3 Reinforcement Learning

Finding a winning strategy in a game can often be done by using reinforcement learning (Ghory, 2004; Baxter et al., 2001; Mnih et al., 2015). It frames the problem of learning, by updating the probabilities of chosen actions, depending on a reward. Which action should be performed, is determined by a certain policy π. The goal of the reinforcement learning (RL) algorithm is to obtain a policy, that selects actions that result in the highest rewards as often as possible. In playing board games this can be seen as a policy that leads to winning the most games.

The Markov property states that the future is independent of the past given the present (Sutton and Barto, 1998). Problems that can be defined as a Markov Decision Process (MDP), can be solved by using reinforcement learning. An MDP is defined by a finite set of states $s \in S$ and finite set of actions $a \in A$. The transition function to go from state s to state s' by performing action a is defined by $T(s, a, s')$. The reward for the agent, when performing action a in state s is defined by $R(s, a)$. The discount factor $0 \leq \gamma \leq 1$ is used in weighing immediate rewards more heavily than future rewards. We will now explain several reinforcement learning algorithms that are applicable to the Markov Decision Process.

Temporal Difference learning Temporal difference learning is a reinforcement learning algorithm. It can learn directly from raw experiences without having a model of the environment (Sutton and Barto, 1998). When there is no clear reward for a state, it predicts what the reward will be for that state using previous estimations, essentially bootstrapping. Therefore, it can learn without having to wait for a final reward. A policy determines what action to take in a given state $s : a = \pi(s)$. The best policy is the one with the highest state-value for every state. The state-value estimation, calculated by the V-value function, is defined by the expected value, E[.], of the cumulative reward:

$$V^{\pi}(s) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, \pi\right] \tag{3.1}$$

After visiting a state, the TD-method updates its value estimation of the previous state. This reduces the difference between the current estimation and the previous estimation. The value is updated using the rule:

$$\hat{V}^{new}(s) \leftarrow \hat{V}(s) + \alpha(r + \gamma\hat{V}(s') - \hat{V}(s)) \tag{3.2}$$

Here $0 < \alpha \leq 1$ represents the learning rate, which controls effect of an experience (s, r, s') on the current estimate.

Q-learning Q-learning is a very popular and widely used reinforcement learning algorithm (Watkins and Dayan, 1992). Opposed to TD-learning not only the state is evaluated but also the accompanying action. They can be evaluated together as a state-action pair (s_t, a_t). The evaluation of this state-action pair is called a Q-value and is defined by the Q-value function:

$$Q^{\pi}(s, a) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, a_0 = a, \pi\right] \tag{3.3}$$

Board games like Othello are played with two players and are therefore not fully deterministic. The next state depends not only on the move of the agent, but also on the move from the opponent after that. Hence, the Q-value is calculated by using state s_{t+1}, which is the state after the opponent has made a move and it is the agent's turn again. The value function is defined in Equation 3.4. The $\max_a Q(s_{t+1}, a)$ part returns the maximal Q-value for all possible actions. This essentially evaluates all possible state-action pairs available from s_{t+1}.

$$Q(s_t, a_t) = E[r_t] + \gamma \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \max_a Q(s_{t+1}, a) \tag{3.4}$$

The estimated Q-value is updated using the temporal difference between the previous estimate and the current estimate. The Q-learning algorithm can be used to update the value function (Watkins and Dayan, 1992):

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{3.5}$$

The application of the Q-learning algorithm can be observed here:

Algorithm 3.1: Q-Learning

```
Initialize Q(s, a) arbitrarily
for each epoch do
    for each environment step do
        Observe state s_t
        Select a_t ~ π(s_t)
        Execute a_t, observe s_{t+1} and reward
          r_t = R(s_t, a_t)
        Q(s_t, a_t) ← Q(s_t, a_t) + α(r_t +
          γ max_a Q(s_{t+1}, a) − Q(s_t, a_t))
        s_t ← s_{t+1}
    end
end
```

SARSA The on-policy variant of Q-learning is called SARSA (State, Action, Reward, State, Action). Q-learning updates its estimates with the maximum possible Q-value of the successive state-action pair. SARSA updates with the estimate of the action performed in the next state while following the policy. SARSA's update rule is the following:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3.6}$$

Notice that Q-learning and SARSA are the same if the policy is defined as $\pi(s_t) = \max_a Q(s_t, a_t)$. Hence, SARSA is used in combination with other policies, which do not use a greedy action selection. For example, in combination with random exploration.

Double Q-learning Problems arise when using Q-learning in stochastic environments. Q-learning uses the maximum state-action value as an approximation for the maximum expected state-action value. Essentially learning estimates from estimates. Always taking the maximum of estimates introduces positive bias and results in overestimation of state-action pair values (Hado, 2010). Van Hasselt proposed the use of two estimators to bypass the overestimation of action values. This results in two independent Q-value functions: Q^A and Q^B. Both Q-value functions can estimate Q-values by using the opposite estimator as an action selector. Hereby decoupling the action selection from the state-action pair evaluation. Rewriting Equation 3.5 results in the following:

$$Q^A(s_t, a_t) \leftarrow Q^A(s_t, a_t) + \alpha(r_t + \gamma Q^B(s_{t+1}, \max_a Q^A(s_{t+1}, a)) - Q^A(s_t, a_t)) \tag{3.7}$$

$$Q^B(s_t, a_t) \leftarrow Q^B(s_t, a_t) + \alpha(r_t + \gamma Q^A(s_{t+1}, \max_a Q^B(s_{t+1}, a)) - Q^B(s_t, a_t)) \tag{3.8}$$

When updating, there is a 0.5 probability to use either Equation 3.7 or Equation 3.8. Either one can then be used as an estimator.

QV-Learning QV-learning is an on-policy reinforcement learning algorithm that keeps track of two value functions (Wiering, 2005). These are: the V-value function, which is used in the temporal difference learning algorithm (Eq. 3.2), and the Q-value function, which is used in the Q-learning algorithm (Eq. 3.5). QV-learning learns by first estimating the expected value of the state, using the V-value function. Then, this state evaluation is used to update the estimation of the corresponding state-action pair of the Q-value function. I.e. since s_{t+1} is the state after performing action a_t in state s_t, the estimation $Q(s_t, a_t)$ can be updated using the estimation $V(s_{t+1})$:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \beta(r_t + \gamma V(s_{t+1}) - Q(s_t, a_t)) \tag{3.9}$$

Here $0 < \beta \leq 1$ is the learning rate similar to α. After the Q-value function is updated, the V-value function is updated following the TD-learning rule (Eq. 3.2). QV-learning might be effective, because the V-value does not take the action into account. As a result, it may find an optimal value faster than the Q-value function. The Q-values can therefore be compared to how an action in a state leads to different successor states.

QV2-Learning Another adaptation of QV-learning is QV2-learning (Wiering and Van Hasselt, 2009). The update rule of the Q-value function is the same as Equation 3.9. The only difference is how the V-value function is updated. Instead of updating with its own estimates by using the rule in Equation 3.2, it updates using the estimates of the Q-value function:

$$V^{new}(s_t) \leftarrow V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - Q(s_t, a_t)) \tag{3.10}$$

Essentially, both algorithms are updating each other.

3.1 Application to Othello

Since the game Othello has a limited set of actions for every state and the states are fully observable, reinforcement learning is well applicable. It has shown good performances in previous research (van der Ree and Wiering, 2013). The evaluation of a finished game is simple, since there are only three outcomes: win, tie and loss. However, the reward for intermediate states is not present. Only in the final state does the agent know what the result of its previous set of actions was. The lacking of immediate rewards, can be solved by using the previously discussed reinforcement learning algorithms. In this section we will show the application of the previous discussed algorithms to the game Othello.

3.1.1 Value function approximators

Since the state space of Othello is very large (approximately 10^{28}), storing all estimations in a lookup table is not feasible (van Eck and Wezel, 2008). The V- and Q-value functions can however be approximated by using neural networks. The input layer consists of nodes that represent the board state. The output of the network is an estimated value of the state or state-action pair, depending on the used reinforcement learning algorithm. The neural networks have a learning rate for controlling the effect of training samples on the estimations over time. The learning rate in Equations 3.2 and 3.5 can thus be set to 1. The equations then simplify to Equation 3.11 and Equation 3.12 respectively.

$$V^{new} \leftarrow r_t + \gamma V(s_{t+1}) \tag{3.11}$$

$$Q^{new}(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a) \tag{3.12}$$

There are no intermediate rewards and the final reward can only be observed in the final state of the game. In all states except for the last, r_t is equal to zero. In the final state r_t is equal to the final reward of the game for the agent. Here $Q(s_{t+1}, a)$ and $V(s_{t+1})$ are equal to zero since s_{t+1} does not exist.

We compare two different neural networks as function approximators. The Multilayer Perceptron (MLP) is a feed-forward network, that maps a set of output variables to a given input, using a hierarchical structure of multiple nodes. The Convolutional Neural Network (CNN), often used in image recognition, can detect simple visual patterns and combine them into more complex patterns.

3.1.2 Algorithms

Q-Learning The neural network used for estimating Q-values is called the Q-network. The output layer of the Q-network consists of 60 nodes, each representing an action. Note here that four nodes have been removed since these are always occupied in the initial start of the game. Not every action is a legal action, since board locations can be occupied. A move is also illegal if the placed disc does not result in flipping of opponents discs. These illegal actions are filtered out. The legal action with the highest expected reward is performed. Hence, the policy function is defined as:

$$\pi(s) = arg\max_a \hat{Q}(s, a) \tag{3.13}$$

Always performing the action expected to have the highest reward, can limit the agent from finding high rewarding actions and new tactics. Actions can remain to have the highest reward even though they are not the best. *ϵ-decreasing* exploration takes care of this problem. With a probability, a random action can be performed instead of picking the one deemed optimal. This probability decreases over time, as best actions become more certain and less optimal actions should be avoided. This exemplifies the Exploration-Exploitation trade-off dilemma.

In every state except for the first the agent takes the following steps:

1. Observe state s_t
2. Compute $Q(s_t, a'_t)$ for all legal actions a'_t
3. Choose an action a_t using ϵ-decreasing policy π
4. Compute target value of $Q(s_{t-1}, a_{t-1})$ using Equation 3.12
5. Use Q-network to compute $Q(s_{t-1}, a_{t-1})$

6. Back propagate the difference between estimate and target as error
7. Execute a_t

When training the network, state s_{t-1} is forward propagated once. Therefore, all errors of the other output nodes are set to zero. Only the output node for the performed action is changed. The error is then back propagated through the network.

SARSA The application of SARSA is very similar to the application of Q-learning on Othello. It also uses a Q-network to approximate the Q-values. The steps used in the Q-learning algorithm are the same, except for step 4. Instead of using the update rule from Equation 3.12 it uses:

$$\hat{Q}(s_t, a_t) \leftarrow r_t + \gamma \hat{Q}(s_{t+1}, a_{t+1}) \tag{3.14}$$

Instead of using the highest estimate of all possible actions to update the network, the estimate of the performed action is used.

Double Q-Learning Van Hasselt (2016) shows an updated version of the Double Q-learning algorithm implemented with a deep neural network. The principle is the same. There are two networks each with an independent set of weights. One network, the online network Q^{online}, is used for action selection and finding the greedy policy. The other, the target network Q^{target}, is used for evaluating that policy. The online network is trained with the update rule while the target network is not. Action selection of the agent is now as follows:

$$a = \max_a Q^{online}(s_{t+1,a}) \tag{3.15}$$

The update rule (Eq. 3.12) is changed into the following:

$$Q^{online}(s_t, a_t) \leftarrow r_t + \gamma Q^{target}(s_{t+1}, a) \tag{3.16}$$

In every state except for the first the agent takes the following steps:

1. Observe state s_t
2. Compute $Q^{online}(s_t, a'_t)$ for all legal actions a'_t
3. Choose an action a_t with Eq. 3.15 while using ϵ-decreasing policy π
4. Compute target value of $Q(s_{t-1}, a_{t-1})$ with Equation 3.16
5. Use Q^{online} to compute $Q(s_{t-1}, a_{t-1})$
6. Back propagate the difference between estimate and target as error
7. Execute a_t

After a predetermined amount of training samples, the weights of the online network are copied to the target network.

QV-Learning QV-learning uses both the value function approximators of Q-learning and TD-learning. Therefore, it uses a similar Q-network as discussed in the Q-learning section. The neural network that approximates the V-value, hereafter V-network, has only one node in the output layer. This node returns the estimated value of the input state. The Q-network is used to play Othello and determines the next state, by choosing an action with policy from Equation 3.13. The V-network is the target network, which is used to update the weights of the Q-network. Updating is done in two steps. First the state is evaluated with the V-network using forward-propagation. Next, this evaluation is used to back-propagate both the V-network and the Q-network. This is done using Equation 3.11 and Equation 3.17 respectively.

$$\hat{Q}(s_t, a_t) \leftarrow r_t + \gamma \hat{V}(s_{t+1}) \tag{3.17}$$

QV2-Learning QV2-learning uses the same Q-network and V-network as QV-learning. Similarly, the update rule from Equation 3.17 is used to update the Q-network. We use a simple version of QV2-learning, in which the V-network is updated using the estimations of the Q-network. The update rule of the V-network becomes:

$$V^{new} \leftarrow r_t + \gamma Q(s_{t+1}, a) \tag{3.18}$$

3.1.3 Experience replay

Learning from sequential states can negatively impact the learning of neural networks due to the correlation between successive states. A biologically inspired replay memory mechanism is introduced which stores experiences in a buffer. The mechanism, called *experience replay*, allows to smooth the

distribution of training samples by taking a random uniformly distributed set of experiences from the replay memory. Effectively allowing retraining and thus refreshing on past experiences (Lin, 1992) but also removing correlation between states (Mnih et al., 2015).

An experience is a quadruple (s_t, a_t, r_t, s_{t+1}) and stored in a buffer with size D. An agent continuously interacts with its environment, storing each experience in the buffer. Whenever the buffer reaches its capacity D it will remove the oldest memory and adds the new memory. After a certain amount of environment steps, a random uniformly distributed sample of experiences is retrieved from the replay memory buffer. This sample is then presented to the learning algorithm.

The reinforcement learning algorithms slightly change in their application to Othello. In Algorithm 3.2 the pseudocode of Q-learning with experience replay can be observed. Other algorithms follow this changed procedure and only differ in the update steps previously mentioned. They are omitted for clarity.

Algorithm 3.2: Q-Learning with experience replay

```
Initialize Q(s, a) with random weights, replay buffer B
for each iteration do
    for each environment step do
        Observe state s_t
        Select a_t ~ π(s_t)
        Execute a_t, observe s_{t+1} and reward r_t = R(s_t, a_t)
        Store (s_t, a_t, r_t, s_{t+1}) in replay buffer B
    end
    for each update step do
        sample e_t = (s_t, a_t, r_t, s_{t+1}) ~ B
        Compute target value of Q(s_t, a_t) with Equation 3.12
        Use Q-network to compute Q(s_t, a_t)
        Back propagate difference between estimate and target as error
    end
end
```

4 Experiments

Multilayer perceptron The setup of the multilayer perceptron is based on previous research of van der Ree and Wiering (2013), which showed good performances with Q-learning. The input layer consists of 64 nodes, which represent the state of the 8×8 grid. Every node represents a single space in the grid. The state of a grid space is represented by a number, which is determined by the occupation of a disc. The value of the grid space is 1 if occupied by a white disc, -1 for a black disc and 0 when empty. The MLP has one hidden layer of 60 nodes. As previously mentioned, it has an output layer consisting of 60 nodes when used as a Q-network and one output node when used as a V-network. Sigmoid activation functions are used on the hidden and the output layer:

$$f(a) = \frac{1}{1 + e^{-a}} \tag{4.1}$$

Before training, all network weights are initialized randomly between the values -0.5 and 0.5. The learning rate of the neural network is set to 0.001. An Adam optimizer is used to update the network weights.

Convolutional Neural Network The use of a convolutional neural network (CNN) as a function approximator is tested against the use of a multilayer perceptron. The setup of the CNN was optimized in preliminary experiments. We chose smaller networks over more complex networks, since they showed similar performance and training time was considerably shorter. The CNN consists of 2 convolutional layers each with 64 feature maps with 3×3 receptive fields. The stride is set to 1 in all directions. There are no pooling layers in the CNN setup since the input, the Othello board state, is already very small. The input for each layer is padded with zeros on the border. Two fully-connected layers are added on top of the network. The hidden layer contains 64 nodes. Similar to the MLP, the output layer contains 60 nodes when used as Q-network and 1 node when used as a V-network. All layers, except for the output layer, use rectified linear units ($ReLU$) to process outcomes. The output layer uses the *sigmoid* activation function (Eq. 4.1). Network weights are randomly initialized at the start with values between -0.01 and 0.01. As

input, two 8×8 channels are used, one for each player. All values are zero except for the location of discs of the player, which are set to one. The two channels are combined into a $8 \times 8 \times 2$ matrix. This translation of the board state to a three dimensional representation is based on the research by Liskowski et al. (2018). The learning rate of the network is set to 0.001.

Starting states Games played by agents with a deterministic strategy will often end up with the same results. To prevent this, we do not use the initial starting state of Othello, but the 236 possible states after 4 turns. These are all used twice, since an agent can play as both white and black. These 472 positions are used in a random order for training and to measure performance during testing. When all 472 games are played, the order is shuffled again.

Opponents In all experiments, the agent trained with reinforcement learning is played against two opponents: the random agent and the heuristic agent. The random agent will always pick a random legal move on its turn. The heuristic agent is a positional agent. It performs the action that results in the state with the highest evaluation. Evaluating a state is done by:

$$V = \sum_{i=1}^{64} x_i w_i \tag{4.2}$$

Here i represents a square on the board. x_i is 0 if square i is empty, 1 if it is occupied by a disc of the agent and -1 if it is occupied by a disc of the opponent. w_i is the weight on square i, which is determined by a weighted piece counter (Szubert et al., 2011). This positional agent is used extensively in previous research in the game Othello (van der Ree and Wiering, 2013; Yoshioka et al., 1999; Lucas and Runarsson, 2006). The predetermined weights of the heuristic player are shown in Table 4.1. Playing against a positional agent is a good way to measure the performance, since it plays well and is deterministic. The random and heuristic agent were tested against each other. A test run consists of playing 472 games, using all starting states once. The average score of 1000 test runs is 0.18 for the random agent and 0.82 for the heuristic agent.

100	-25	10	5	5	10	-25	100
-25	-25	2	2	2	2	-25	-25
10	2	5	1	1	5	2	10
5	2	1	2	2	1	2	5
5	2	1	2	2	1	2	5
10	2	5	1	1	5	2	10
-25	-25	2	2	2	2	-25	-25
100	-25	10	5	5	10	-25	100

Table 4.1: Weighted piece counter of the heuristic agent

Rewards and parameters The reward for the end of a game, is valued at 1, 0.5 and 0 for win, tie and loss respectively. These rewards are used for training, in e.g. Equation 3.11 and Equation 3.12, and to measure performance when testing. The capacity size of the experience replay memory is set to 50,000 with a sample size of 200. We train on 200,000 games in total. After every game the networks are updated. The random exploration value ϵ is set to 0.1 and linearly decreases to 0 over the total amount of games. The discount factor is set to 1.

5 Results

All algorithms, Q-, DQ-, QV-, QV2-learning and SARSA, are trained against both the random and the heuristic agent. Each training session consists of playing 200,000 games. Every 2000 games the algorithm is tested. A test run consists of playing all 472 starting positions once. For each algorithm 10 training sessions are performed and all test scores are recorded. The average scores of these trials are shown in Figure 5.1 and 5.2. The best results, which are used to compare the performance of the algorithms, are presented in Tables 5.1 and 5.2.

To compare the performance of using a CNN instead of an MLP, all experiments are repeated but now with a CNN as function approximator. The results are presented in Figures 5.3 and 5.4 and Tables 5.3 and 5.4

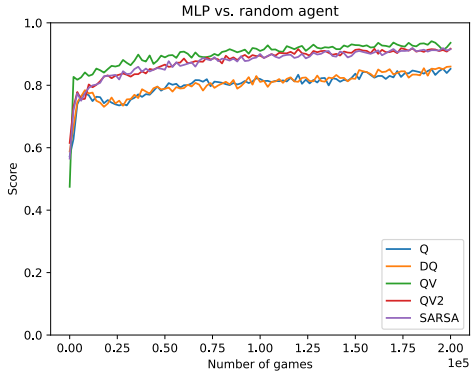

Figure 5.1: The average scores of 10 trials using an MLP. Trained for 200,000 games versus the random agent

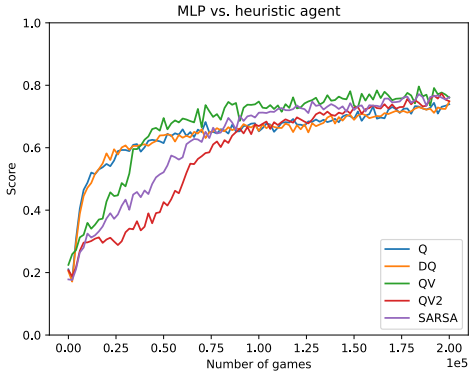

Figure 5.2: The average scores of 10 trials using an MLP. Trained for 200,000 games versus the heuristic agent

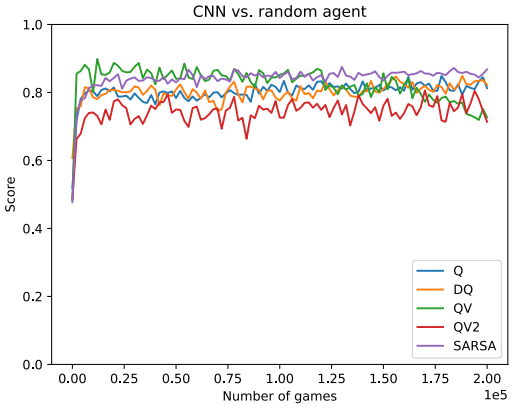

Figure 5.3: The average scores of 10 trials using a CNN. Trained for 200,000 games versus the random agent

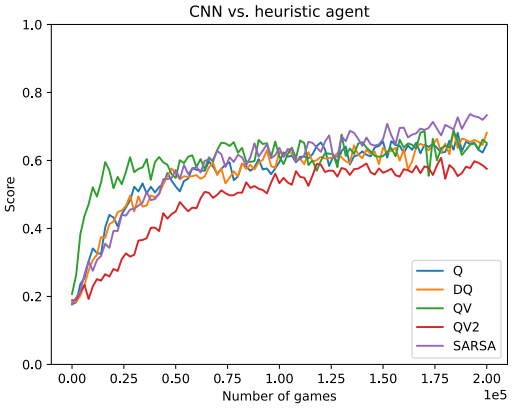

Figure 5.4: The average scores of 10 trials using a CNN. Trained for 200,000 games versus the heuristic agent

Algorithm	Score	St. Err.
Q	0.853	0.026
SARSA	0.919	0.014
DQ	0.860	0.021
QV	0.941	0.011
QV2	0.920	0.015

Table 5.1: Performances of MLPs trained while playing against a random agent. The score of the best test during the learning session is presented, with the standard error.

Algorithm	Score	St. Err.
Q	0.744	0.030
SARSA	0.772	0.028
DQ	0.748	0.043
QV	0.800	0.015
QV2	0.774	0.026

Table 5.2: Performances of MLPs trained while playing against a heuristic agent. The score of the best test during the learning session is presented, with the standard error.

Algorithm	Score	St. Err.
Q	0.848	0.034
SARSA	0.875	0.019
DQ	0.849	0.422
QV	0.900	0.020
QV2	0.806	0.095

Table 5.3: Performances of CNNs trained while playing against a random agent. The score of the best test during the learning session is presented, with the standard error.

Algorithm	Score	St. Err.
Q	0.681	0.052
SARSA	0.736	0.058
DQ	0.688	0.048
QV	0.690	0.167
QV2	0.608	0.165

Table 5.4: Performances of CNNs trained while playing against a heuristic agent. The score of the best test during the learning session is presented, with the standard error.

6 Discussion

Following our results we will look back to the research questions posed earlier:

How does the performance of SARSA compare to Q-learning?

The results in Tables 5.1 and 5.2 show that SARSA outperforms Q-learning while playing and learning against both the random (r) and the heuristic (h) agent. These differences are significant ($t_r(18) = 7.0678$, $p < .0001$ and $t_h(18) = 2.1577$, $p = .0447$). This indicates that learning on the action performed by the agent, may lead to a higher performance, than an off-policy approach. While using a policy with exploration steps, on-policy learning might converge slower, since it may cause larger variance. This can be seen in Figure 5.2.

How does the performance of Double Q-learning compare to Q-learning?

Following the results in Tables 5.1 and 5.2 we observe that Double Q-learning is on par with Q-learning when playing against the random and heuristic agent ($t_r(18) = .6623$, $p = .5162$ and $t_h(18) = .2413$, $p = .8121$). Observing Figures 5.1 and 5.2 we see that the development of the performance is the same for Q-learning and Double Q-learning. This seems to indicate that the addition of a second estimator does not affect the performance. These are interesting results as Double Q-learning often outperforms Q-learning in previous research (Hado, 2010). A probable cause might be the limited amount of games played before copying the online network weights to the target network. This could negate the effect of having a second estimator, since the networks would have been trained on a too similar set of experiences thus not reducing positive bias.

How does the performance of the QV-family compare to Q-learning?

Both QV- and QV2-learning perform better while learning against the random and the heuristic agent than Q-learning. This can be observed in Tables 5.1 and 5.2. The differences, compared to Q-learning, are significant for QV-learning ($t_r(18) = 9.8572$, $p < .0001$ and $t_h(18) = 7.0585$, $p < .0001$) and for QV2-learning ($t_r(18) = 5.2797$, $p < .0001$ and $t_h(18) = 2.3897$ $p = .0280$). The results of the QV-family are similar to SARSA, which is also an

on-policy learning algorithm. This again indicates that an on-policy variant has a better performance than Q-learning. This is in line with the results of previous research (Wiering, 2005).

When an agent, trained with QV- or QV2-learning, plays against a heuristic agent, convergence is slower. This can be observed in Figure 5.2. Similar to SARSA, this can be explained by the fact that it is on-policy and the policy function uses exploration steps. However, the performance of QV-learning surpasses the performance of Q-learning after roughly 40,000 games. This might be due to the fact that both networks are updated using state evaluations instead of state-action evaluations. Similarly, this might explain why the QV-learning converges faster than Q-learning, when playing against a random agent, as shown in Figure 5.1.

QV2-learning converges slower than Q-learning, when playing against both the random and the heuristic agent. This might be caused by the fact, that two random initialized networks update each other. In combination with the exploratory policy, converging might be slow.

How does the usage of a convolutional neural network instead of a multilayer perceptron affect the performance of the Q-learning adaptations?

The performance development when using a CNN, shows that the CNN is able to extract spatial information from the board and the agent is able to learn. Comparing Tables 5.2 and 5.4, we can observe that all algorithms using a CNN instead of an MLP as a function approximator, perform worse when playing against the heuristic agent. The algorithms do show similar development in performance when using an MLP or a CNN, except for QV-learning. QV-learning shows a faster increase of performance when using a CNN but using an MLP will result in a higher maximum performance. The result of playing against a random opponent show similar performances when using an MLP or CNN. Do notice the higher fluctuations in the development of performance when using a CNN.

7 Conclusion

In this research we have compared the performance of different adaptations of Q-learning applied to the game Othello. Also, we compared the performance when using a CNN instead of an MLP as a function approximator.

The results show that SARSA outperforms Q-learning when playing against both agents. The QV-family also performs better than Q-learning. QV2-learning converges slower, where QV-learning converges faster than Q-learning. Double Q-learning seems to perform similar to Q-learning. The usage of a CNN instead of an MLP as function approximator seems to decrease the performance, when playing against the heuristic agent. The performance of the trained agents versus the random agent is similar when using a CNN or MLP. QV-learning even shows a faster increase in performance when using a CNN but will not reach the highest score of QV-learning when using an MLP.

Since our implementation of Double Q-learning shows different results than previous research, more focus could be directed on the use of double estimators. Since it produced similar results as Q-learning, we are interested in optimizing the number of games until weights are copied over from the target network to the online network. The usage of a CNN shows a promising performance development when training in QV-learning. Although the usage of an MLP still outperforms our implementation of the CNN, it would be interesting to inspect the performance of more complex CNNs.

References

Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Reinforcement learning and chess. In *Machines that learn to play games*, pages 91–116. 2001.

Imran Ghory. Reinforcement learning in board games. *Department of Computer Science, University of Bristol, Tech. Rep*, 105, 2004.

Van Hasselt Hado. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI*

Conference on Artificial Intelligence, AAAI'16, page 2094–2100. AAAI Press, 2016.

James King and Vaiva Imbrasaitė. Generating music playlists with hierarchical clustering and q-learning. In *European Conference on Information Retrieval*, pages 315–326. Springer, 2015.

Long-ji Lin. Reinforcement learning for robots using neural networks. page 29, 1992.

Paweł Liskowski, Wojciech Jaśkowski, and Krzysztof Krawiec. Learning to play othello with deep neural networks. *IEEE Transactions on Games*, 10(4):354–364, 2018.

Simon M. Lucas and Thomas P. Runarsson. Temporal difference learning versus co-evolution for acquiring othello position evaluation. In *2006 IEEE Symposium on Computational Intelligence and Games*, pages 52–59. IEEE, 2006.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

G. Rummery and M. Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

Mohamed Shakeel, Baskar S., V.R. Sarma Dhulipala, Sukumar Mishra, and Mustafa Jaber. Maintaining security and privacy in health care system using learning based deep-q-networks. *Journal of Medical Systems*, 42, 10 2018. doi: 10.1007/s10916-018-1045-z.

Schuyler Smith. Learning to play stratego with convolutional neural networks. 2015.

R. Sutton and A. Barto. *Reinforcement learning: An introduction*. The MIT Press, 1998.

Marcin Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. Learning board evaluation function for othello by hybridizing coevolution with temporal difference learning. *Control and Cybernetics*, 40:805–831, 01 2011.

Gerald Tesauro and Jeffrey Kephart. Pricing in agent economies using multi-agent q-learning. *Autonomous Agent and Multi-Agent Systems*, 5, 10 1999. doi: 10.1023/A:1015504423309.

M. van der Ree and M. Wiering. Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 108–115, 2013.

M. van Eck and M. Wezel. Application of reinforcement learning to the game of othello. *Computers & Operations Research*, 35:1999–2017, 06 2008. doi: 10.1016/j.cor.2006.10.004.

Christopher J.C.H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

S. Wender and I. Watson. Applying reinforcement learning to small scale combat in the real-time strategy game starcraft:broodwar. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 402–408, 2012.

Marco Wiering. Qv (lambda)-learning: A new on-policy reinforcement learning algorithm. In *Proceedings of the 7th European workshop on reinforcement learning*, pages 17–18, 2005.

Marco Wiering and Hado Van Hasselt. The qv family compared to other reinforcement learning algorithms. In *Adaptive Dynamic Programming and Reinforcement Learning*, pages 101 – 108, 05 2009. doi: 10.1109/ADPRL.2009.4927532.

Taku Yoshioka, Shin Ishii, and Minoru Ito. Strategy acquisition for the game "othello" based on reinforcement learning. *IEICE Transactions on Information and Systems*, 82(12):1618–1626, 1999.

8 Contributions

D.K. and J.v.B. conceived of the presented idea, developed the overall program, implemented Q-learning and SARSA and carried out the experiments of Q-learning and SARSA. D.K. implemented Double Q-learning, carried out the Double Q-learning experiments and implemented experience replay. J.v.B. implemented QV- and QV2-learning and carried out the QV- and QV2-learning experiments. D.K and J.v.B. set up the design of the MLP and CNN, implemented it and carried out the CNN experiments. J.v.B. wrote the sections: QV- and QV2-learning QV- and QV2-learning application, SARSA and SARSA application. D.K wrote the sections: Othello, Double Q-learning, Q-learning application and Double Q-learning application. Both authors contributed to the remaining sections of thesis.