Deep Reinforcement Learning applied to the game Bubble Shooter

Laurens Samson
10448004

Bachelor thesis
Credits: 18 EC

Bachelor Opleiding Kunstmatige Intelligentie

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
Efstratios Gavves

Institute for Language and Logic
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 24th, 2016

Abstract

In 2013 Google Deepmind invented a new algorithm, deep reinforcement learning, which led to revolutionary results. The algorithm was able to play on 46 different Atari games and beat a human expert in some of the games. In this research deep reinforcement learning will be applied to the game Bubble Shooter in order to create the self-learning agent AlphaBubble. A different game state representation is used than in the original algorithm, also experiments have been conducted with bigger action spaces. The performance of AlphaBubble has clearly become better than a random Bubble Shooter agent, although it has not been able to exceed human play yet. However, AlphaBubble should be able to invent superhuman behaviour in the future.

Acknowledgements

First of all, I would like to thank my supevisor Efstratios Gavves and co-supervisors Matthias Reisser, Changyong Oh and Berkay Kicanaoglu for their help during this project. Their investments have been essential to achieve the results that are accomplished.

Contents

1 Introduction

Artificial intelligence aims to reach or even outperform human-intelligence. In the past chess computers were created, which were able to outperform humans. The chess computer DeepBlue beat the world champion in chess over six matches(Campbell, Hoane, & Hsu, 2002). This algorithm was based on several techniques, such as tree-search and databases of grandmasters.

Nowadays machine learning is being used to create artificial intelligence. Different types of algorithm have been developed over the last past decades, which established the possibility to let computers learn. The basic idea of machine learning is gathering as much information and thereafter feed this information to a machine learning algorithm(Schapire, 2003). For example, in order to train a spam detector, the algorithm should be fed with as many spam and non spam mails.

In the last past decades great achievements have already been accomplished in several games with machine learning. Reinforcement learning is a machine learning technique that is able to invent behaviour for particular game environments. Lately the game of Go was mastered with the algorithm AlphaGo, which was based on several machine learning techniques, such as reinforcement learning, deep learning and supervised learning(Silver et al., 2016). The algorithm was able to beat the world champion and achieved a win rate of 99.8 percent against other Go programs. Also, the game Backgammon was mastered with the machine learning technique reinforcement learning, the level of play was estimated very close to world's best human players(Tesauro, 1994). One of the big advantages of reinforcement learning is that there is no need to collect data sets, since the data is achieved by discovering different situations in games.

Reinforcement learning is similar to the way humans learn. Humans learn how to behave by exploring different types of actions. For example, when learning how to ride a bicycle, people experience quickly that falling is not preferable. Computers learn by maximising the reward they receive for taking actions(Sutton & Barto, 1998). If a computer learns to ride a bicycle, then it would receive a positive reward for continuing riding and a negative reward for falling.

Other improvement in the domain of machine learning has been made in deep learning, which has led to better image recognition, voice recognition and speech recognition(LeCun, Bengio, & Hinton, 2015). Due to the improvement of software, hardware and algorithms this improvement has been possible. One of the great benefits of deep learning is that it can extract features from the data, which always have been done by humans(LeCun et al., 2015). Creating features can possibly cost a lot time and moreover expertise is necessary. However, Deep Learning requires still a lot of experimenting with different architectures.

In 2013 Google Deepmind invented a machine learning technique, called deep reinforcement learning, which is based on deep learning and reinforcement learning. This algorithm was a breakthrough in reinforcement learning and the achievements were very impressive. A self-learning agent reached master level

of play in most of the Atari games and beat a human expert in some of the games(Mnih et al., 2013, 2015). In addition, the same network was able to learn all the different Atari games.

Therefore, in this thesis the deep reinforcement learning algorithm will be recreated to learn the agent AlphaBubble play the game Bubble Shooter. The achievements that were accomplished in the previous research have shown that this algorithm can lead to superhuman behaviour. The combination of reinforcement learning and deep learning seems very promising, since deep learning can extract features from the data and reinforcement learning can invent the optimal policy. The possibilities of deep reinforcement learning for societal matters seem endless. For example, the algorithm could contribute to invent a superhuman policy for self-driving cars.

The goal of Bubble Shooter is to clear the field of balls by shooting a random coloured ball next to matching balls. If the ball is shot next to two or more matching balls, then those balls will disappear, but when a single ball reaches a particular line at the bottom of the screen the episode is terminated. Bubble Shooter is an interesting game for deep reinforcement learning, since the action space is theoretically infinite. The agent has to determine a shooting angle between 0 and 180 degrees.

In this thesis the necessarily theoretical background will be explained in order to have a good understanding of deep reinforcement learning. Therefore the basic concepts of deep learning and reinforcement learning will be described first and subsequently the algorithm deep reinforcement learning will be explained. Also, relevant literature will be provided in order to justify the choice for deep reinforcement learning. The method section will indicate how deep reinforcement learning is applied to the game of Bubble Shooter. Thereafter the results from different experiments of AlphaBubble will be shown. In the conclusion the results of this research will be summarised and discussed whether deep reinforcement learning can be applied to Bubble Shooter.

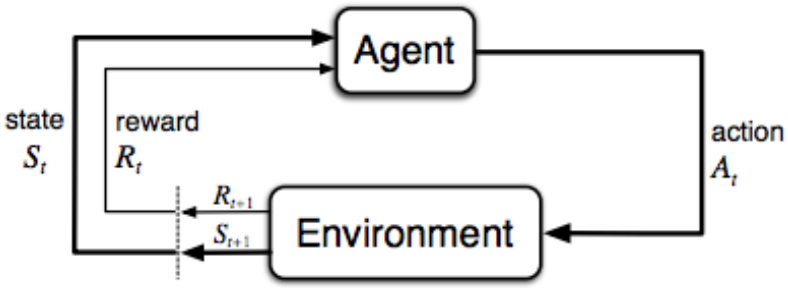
2 Theoretical background

2.1 Reinforcement Learning

In figure 1 the model of reinforcement learning is shown. The concept of reinforcement learning is to achieve a goal by learning from the interaction with the environment. The agent and the environment interact continually. After every action of the agent the state in the environment changes and subsequently the agent has to perform another action(Sutton & Barto, 1998).

At every time step the environment provides the current state to the agent. On the basis of this state the agents has to select an action. As a consequence of the action the agent receives a numerical reward and the agent finds itself in a new state(Sutton & Barto, 1998). At every state the agent decides which actions to perform on the basis of his policy, which is trained by his experience in the past. The goal of the agent is to maximise the reward it receives over a long period by finding the optimal policy.

Figure 1: Model of reinforcement learning (Sutton & Barto, 1998)



In other words, the agent has to maximise the return, G_t, which is the sum of rewards, R_t, over time t.

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{1}$$

where T is a terminal state(Sutton & Barto, 1998). Terminal states occur when an episode is finished, for example when a game of chess is won, draw or lost. Conversely, if the agent would maximise the return, it would in some environments prefer playing forever than finish the game. For example in the game of Bubble Shooter, it would rather shoot balls forever than winning the game, because eventually this will lead to a higher reward. Because of this problem, rewards in the future should be less attractive than close rewards. Hence, the following equation should be maximised by the self-learning agent:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

where γ is the discount factor, which lays between 0 and 1. Rewards received k time steps in the future is worth only γ^{k-1}. Because of the discounted rewards, the agents prefers solving the problem more quickly. So, the goal of the agent is the maximise the discounted return(Sutton & Barto, 1998).

In reinforcement learning the value function estimate how much future reward the self-learning agent will receive for a given state following its policy π. This function is called state-value function.

$$v_\pi(s) = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \tag{3}$$

This functions returns the expected value starting at state s following policy π.

Similarly, the action value function or Q-value calculates the return for the current state and a particular action for policy π.

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] \tag{4}$$

This function can be estimated by the experience of the agent(Sutton & Barto, 1998). To acquire the optimal behaviour, the policy with the highest expected return should be chosen.

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{5}$$

In order to invent the optimal policy a trade-off between exploration and exploitation is used in reinforcement learning. To discover the actions which obtain the highest reward, the agent must see as many different states of the learning environment. During exploitation the agent performs the best actions according to what it has learnt so far in order to obtain reward. During exploration the agent moves randomly in order to invent a better policy in the future. The trade-off between exploitation and exploration results in discovering many different states of the environment, this will help the agent to invent the optimal policy(Sutton & Barto, 1998).

One of the most acquainted algorithms in reinforcement learning is Q-learning, which is an algorithm that directly approximates the optimal action value(Sutton & Barto, 1998). This algorithm was a breakthrough in reinforcement learning, because it immediately estimates the action-value function and moreover the algorithm converges quick. The definition of Q-learning is the following:

$$q(s, a) = q(s, a) + \alpha[R_{t+1} + \max_a q(s_{t+1}, a) - q(s, a)] \tag{6}$$

Q-Learning converges to the optimal policy with the probability 1, if in all states all actions are repeatedly sampled and the action-values are represented discrete(Watkins & Dayan, 1992). In fact this is almost impossible when the action or state space becomes really big.

In the past some promising results of reinforcement learning have been accomplished. Researchers created an self-learning agent that performed a good

level of play in the game Ms. Pacman(Bom, Henken, & Wiering, 2013). Handcrafted features were created as input for the neural network and these features resulted in a well generalised agent that was able to play at different mazes. Although only seven features were created, it still needs a lot of prepossessing. One of the main problems of machine learning is to decide what features to use and how to combine them(Blum & Langley, 1997). Blum & Langley(1997) also state that determining what features are relevant is a difficult task. Moreover, by creating handcrafted features, the agent will learn playing what the creator of the algorithm indicates as important. In other words, to create good features expertise is necessary.

However, good performance have been achieved using reinforcement learning with handcrafted features. For example the game Backgammon was mastered with the algorithm temporal difference learning(Tesauro, 1994). When only providing the raw data of the game state to the algorithm, the level of play was estimated as average. However, when adding handcrafted features the game level increased to a really high level. But, in some case knowledge is not explainable. The phenomena that knowledge is so ingrained that it can not be explained properly is called tacit knowledge(Nemati, Steiger, Iyer, & Herschel, 2002). For example, when the best footballer in the world has to explain what makes him the best player of the world, he will not be able to provide a very precise answer about how he does his movements. Thus, at some domains of behaviour it might be difficult to acquire good features, since some behaviour is tacit knowledge. Therefore, it would be valuable if features can be extracted automatically from data.

2.2 Deep Learning

The last past years deep learning has improved speech recognition, image recognition and many other domains(LeCun et al., 2015). Deep learning discovers structures in data sets by adapting the parameters that compute the outcome of the network with the algorithm backpropagation.

For decades machine learning has been limited, because the techniques were not able to process raw data. Building a machine learning system required good preprocessing in order to create features. Representation learning are methods that are able to process raw data and automatically discover the representations that are necessary for classification or regression. Deep learning is a representation learning method, obtained by non-linear modules that each transform a representation into a higher and more abstract representation(LeCun et al., 2015). For classification, these higher representations consolidate the important aspects of the input and inhibit the unimportant aspects of the input. For example, during classification of images the input are pixel values. In the first layer of representation typically represents the presence or absence of edges in the images, the next layer of representation illustrates shapes. Hence, the benefit of deep learning is that features are created automatically by consolidating important aspects of the data(LeCun et al., 2015).

One of the most successful types of deep learning are convolutional neural

networks, these are made to process data in the form of multiple arrays(LeCun et al., 2015). The architecture of these network is structured in different stages. The firs stages are built with convolutional and pooling layers. The role of the convolutional layer is to detect local conjunctions of features from the previous layer, the role of the pooling layer is to merge similar features into one feature. The stages after the convolutional and pooling layers are mostly fully connected layers. Nowadays, convolutional neural network architectures have between 10 and 20 layers(LeCun et al., 2015). Training these networks would have cost about two weeks a few years ago, because of the progress in software, hardware and algorithms the training time has been reduced to a few hours.

Deep learning has shown some revolutionary results in the last few years. As is said before in several domains the performance has improved drastically. The performance of image recognition has been improving the last few years. Researchers achieved an 15,3 percent error rate on image recognition with a convolutional neural network consisting of five convolutional layers, this was the breakthrough for deep learning(Krizhevsky, Sutskever, & Hinton, 2012). The results of this research were very impressive, since the second best algorithm achieved an error rate of 26,2.

Also, in Speech recognition a lot of progress have been made because of deep learning. Google voice input transcribes short messages from mobile devices, the vocabulary of this task is very large(Hinton et al., 2012). Eventually, Google achieved an error rate of 11.8 percent, which was again a huge improvement relative to the previous method. Deep learning is a very promising machine learning technique, which will have many more successes, since deep learning requires very little engineering(LeCun et al., 2015). Moreover, the development of new architectures and new algorithms will accelerate this process.

2.3 Deep Reinforcement Learning

In 2013 Google DeepMind published a paper wherein the algorithm deep reinforcement learning was implemented, a combination of reinforcement learning and deep learning(Mnih et al., 2013). The model was a convolutional neural network trained with Q-Learning. This self-learning algorithm was able to learn different policies for different Atari 2600 games using the same network.

2.3.1 The algorithm

At every time step the agent receives the current game state as an image s and based on this state the agent performs an action a in the environment. The agent acts according to the trade-off between exploration and exploitation. In other words the agent sometimes performs a random move and sometimes the best move to what he has learnt so far. The environment changes because of the action taken by the agent. Subsequently the agent receives the image of the new game state s' and in addition the reward r for the taken action. After every action a tuple with s, a, r, s' is stored in the replay memory and a random mini batch is taken from the replay memory. Then, the network is trained using

these mini batches. The use of replay memory has several advantage of standard Q-learning(Mnih et al., 2015). First, every action can potentially be used for training several times, which is efficient. Secondly, training for a consecutive sequence is not efficient, because of the high correlations between all the states in the sequence. Randomised samples avoid these correlations and will reduce the variance. The goal of the agent is to interact with environment in a way that maximises the rewards it receives in the long run. The pixel values of the the current game state are the input for the convolutional neural network. The network estimates the action values for all the possible action from that game state. The pseudo-code of the algortihm is shown in figure 2(Mnih et al., 2013).

Figure 2: Pseudocode of the Deep Reinforcement Learning algorithm(Mnih et al., 2013)

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

2.3.2 Cost function

By making use of the Bellman equation the action value function is estimated in most reinforcement learning algorithms. The concept of the Bellman equation is the following: "if the optimal value $Q(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma Q^*(s', a')$".(Mnih et al., 2015).

$$Q^*(s, a) = E[r + \max_a q(s', a')] \tag{7}$$

In most reinforcement learning algorithm, such as Q-learning, the Bellman equation is used as an iterative update. Those algorithms converges to the optimal policy after an infinite amount of iterations. Mostly, linear approximator are used for this task, but sometimes non-linear approximator are used to estimate the action-value function(Mnih et al., 2015). In deep reinforcement there are two networks, the Q-network and the target network. The Q-network is

trained by optimising the parameters with backpropagation, the target network is necessary for the iterative loss update:

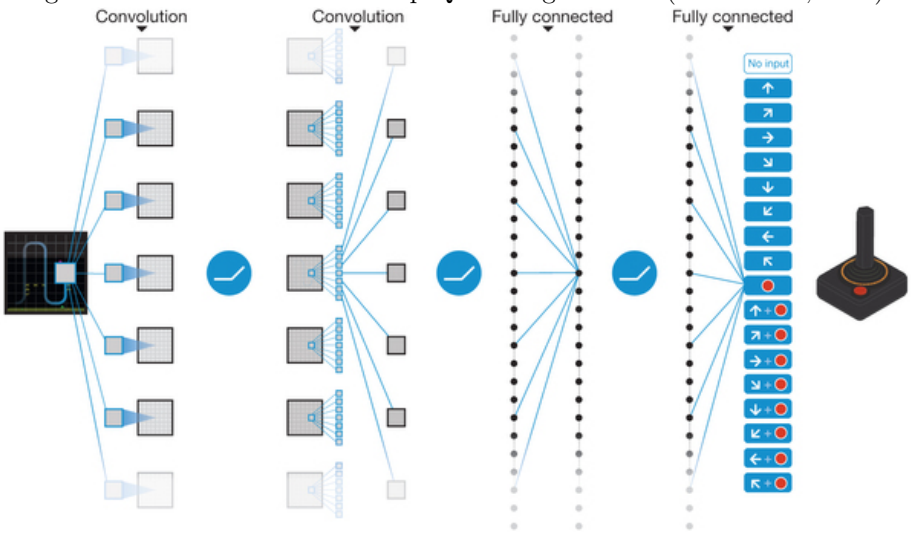
$$L_i(\theta_i) = E[r + \gamma \max_a q(s', a'; \theta^-) - q(s, a; \theta_i)] \tag{8}$$

where θ_i refers to the parameters of the Q-network and θ^- refers to the parameters of target network. The parameters of the target network θ^- the are hold from the previous iteration of the Q-network. These parameters are updated every certain number of iterations, because this makes the algorithm more stable.

2.3.3 Architecture

In figure 3 the architecture of the network is shown that was built for the Atari games, the same network was trained for all the games(Mnih et al., 2015). The pixel values of the current game state are the input of convolutional neural network. The network will train itself to estimate the action-value function for every possible actions. The network exists of three convolutional layers followed by two rectified linear units. The output of the network are the Q-values for every possible acton from the input state.

Figure 3: Architecture of the Deep Q-learning network (Mnih et al., 2015)



3 Research question

In this thesis deep reinforcement learning will be implemented in order to learn the agent AlphaBubble play the game Bubble Shooter. This leads to the following research question:

To what extent is a self-learning agent able to learn playing the game Bubble Shooter using Deep Reinforcement Learning?

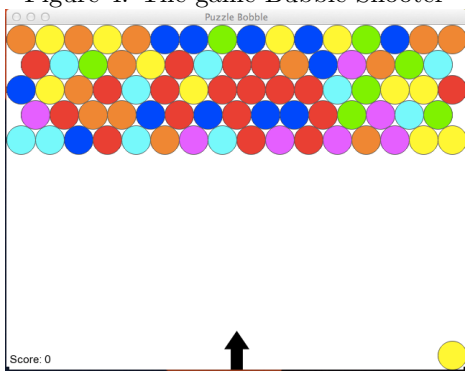
4 Method & Approach

In this section an explanation will be provided how deep reinforcement learning will be applied to the game Bubble Shooter. The adaptions compared to the Google Deepmind implementation will be described, such as architecture and game state representation. Further, the reward function and the action space will be pointed out in this section.

4.1 Bubble Shooter

The open source implementation of the game bubbleshooter was downloaded from https://github.com/justinmeister/bubbleshooter. The game contains one level, which is an random generated level. The number of colours and lines of balls can be adapted in the code, which is valuable for experimenting. Furthermore, adaptions have been made in order to run the programs faster, this will reduce the training time. The code will be attached.

Figure 4: The game Bubble Shooter



In figure 4 the game Bubble Shooter is shown. The goal is to clear the field of balls by shooting coloured balls next to matching coloured balls. If a group of 2 or more matching balls is hit with a matching coloured balls, then this group of balls will pop. When the whole field of balls is cleared, the player wins. The

players loses when a single balls reaches a certain line on the bottom of the screen. Different versions of Bubble Shooter exist. Initially, AlphaBubble will be trained on the version of Bubble Shooter, where no new ball come into the game. Eventually, an attempt will be made to let AlphaBubble play on a game, where every certain number of shots a new line of balls come in to the game.

4.2 The algorithm

The pseudo-code of the algorithm is shown in figure 2. The same algorithm will be implemented, but some adjustments are made. Because of a matter of time, a simple representation of the current game state will be used as input for the network. Also, the network architecture will differ from the Google Deepmind implementation.

4.3 Architecture

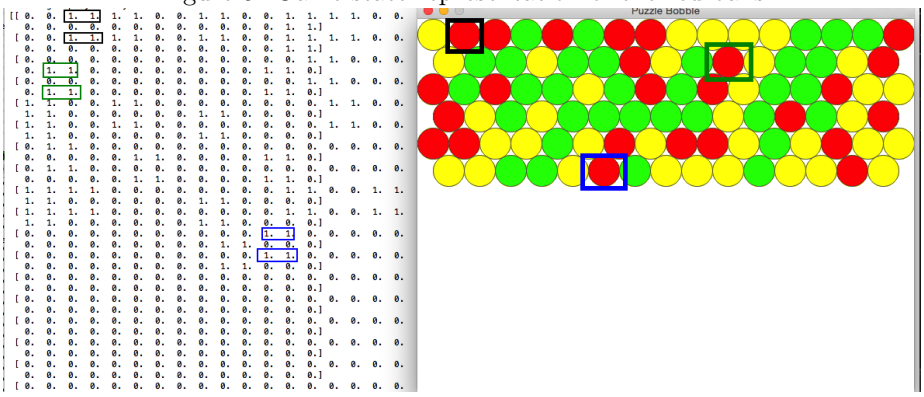
In figure 2 the architecture of the Deep Reinforcement Learning network of the Atari Games is shown(Mnih et al., 2015). This architecture will be the baseline for the network of AlphaBubble. Google Deepmind chose the pixel values of the image from the current game state as input for the network. In this thesis another game state representation will be opted, which will be explained in the next section. However, the surface of the game state representation will be smaller than an image, but the representation does have more dimensions. The output of the network are Q-values for all possible action from the input state, so the number of output nodes depends on the number of actions. In order to extract features out of the games states, convolutional layer will be part of the network. These layers will be able to find patters and structures in the data, which will be functioning as features. The convolutional layers are followed up with fully connected layer with rectifier non-linearity. The last layer is a fully connected layer without non-linearity.

4.4 Game State Representation

Google Deepmind utilised the pixel values of the image from the current game state as game state representation. A three-dimensional array is opted as representation of the current game state, one dimension of the representation is shown in figure 4. For every ball colour a grid containing ones and zeros is created. Every ball is represented as a square consisting of 4 ones. Ones will indicates where that particular ball colour occurs in the game, zeros indicates that there is no such ball at that place. In figure 4 the grid for the red ball colour is shown. Also, a dimension is created, which indicate where no balls occur. Because of the game state representation the complexity of the task will be reduced, since it basically detects objects perfectly. Therefore, less convolutional layers are probably necessary to accomplish the task of playing Bubble Shooter. Nowadays, object recognition can be done with deep learning and the

decision was made to omit this task in order to reduce the complexity(Szegedy, Toshev, & Erhan, 2013).

Figure 5: Game state representation of the red balls



To accelerate the training time, the average of the game state representation is decreased to zero. Averaging the array to zero will reduce the training time, since some functions, such as sigmoid function, have the steepest gradient close to zero. To accomplish an average of zero, all zero values are adapted to a value between zero and minus one and all the ones are reduced to a value between zero and one.

4.5 Language & Packages

AlphaBubble will be implemented in Python using the deep learning packages Theano and Lasagne. Theano offers the possibility calculate gradient, which is very valuable for backpropagation. With Lasagne networks can be built easily with many different sort of layers.

4.6 DAS4

For training the deep reinforcement network, the DAS4 computer of the university of Amsterdam will be utilised. The super computer contains more GPU and this will reduce the time for training the network. Moreover, the convolutional layer, which will be built into the network, requires CUDA, which is installed on the DAS4(Bal et al., 2016).

4.7 Rewards

In table 1 the rewards are shown for all possible events during the game.

Table 1: Reward function for possible events in Bubble Shooter

Events	**Rewards**
Win the game	20
Lose the game	- 15
Pop balls	1 * amount of ball poppd
Hit one matching ball	1
Miss	-1

In order to encourage AlphaBubble to pop as many balls as possibly, one point is given for every ball the agent pops. Every time the agent fails to hit a matching coloured ball, it receives a reward of minus one. The aim of giving a negative reward for missing is to achieve a win as quickly as possible. The discount factor will be 0.99, this means that rewards in the future are still attractive and the agent should keep rewards in the future in to account.

4.8 Action space

Since deep reinforcement learning is not able to do regression tasks, consequently it is not possible to predict the best angle. Instead deep reinforcement learning is only capable to classify the best action. Therefore, the number of actions has to be set. Probably, the training time of the network will increase, when the action space is larger, because the exploration time will enlarge. However, enlarging the action space will probably reduce the complexity of the game, since more different targets can be hit. Therefore, the hypothesis is that increasing the action space will not differ to much in training time. During this thesis, experiments will be conducted with different action spaces in order to see the effect of the different sizes and to find out what action space leads to the optimal policy.

During the experiments the range of the angle will lay between 10 and 170 degrees, because the angles below 10 and above 170 will not be very effective shots. Moreover, it could reduce the training time, since less exploration is required . The directions are calculated with the following formula:

Table 2: Formula for shooting directions of AlphaBubble

Number of actions	Formula
20	8 * action + 10
40	4 * action + 10
80	2 * action + 10

where action is the argmax over all the output nodes of the network, which represent the action-value function. During exploration the variable action is a

random integer between zero and the amount of actions.

4.9 Evaluation of AlphaBubble

In the experiment section AlphaBubble will be evaluated. The following evaluations will be executed:

1. AlphaBubble vs random agent
2. AlphaBubble with different action spaces
3. AlphaBubble vs human players
4. AlphaBubble with moving balls

First, AlphaBubble will be compared to a random agent, both the agents will have an action space of 20. The goal of this evaluation is to proof that AlphaBubble has learnt. Thereafter, AlphaBubble will be compared with different action spaces in order to see the effect of the different action spaces and moreover to see which action space is the best. In order to see if AlphaBubble is able to beat human players, an comparison is made between AlphaBubble and the average of ten human players. Last, AlphaBubble will play on a game, where new lines of balls come in to play every certain amount of shots.

5 Experiment

In this section the results of several experiments will be presented. Also, the hyper parameters and the architecture of the convolutional neural network will be shown. Further, some interesting results of initial experiments will be described.

Implementation of AlphaBubble:
https://github.com/laurenssam/AlphaBubble

Demonstration of AlphaBubble:
https://www.youtube.com/watch?v=DPAKFenNgbs

5.1 Initial experiments

Initially, AlphaBubble was trained on a game state, where two matching balls were placed next to each other. The goal for AlphaBubble was to hit one of those balls, this would result in popping the group of balls. The aim of the experiment was to confirm that the algorithm was able to learn and find out whether the action space was big enough to solve the game. The action space appeared to be sufficient to solve the game.

Subsequently, the algorithm was trained on a game with one ball. The goal for AlphaBubble was to invent a winning policy with two shots, therefore the agent needed to hit the first ball twice in a row. The aim of this experiment was to find out if AlphaBubble was capable of seeing future rewards, because the first shot would receive a reward of -1, since it would not pop any balls. For the following shot, it would receive a reward of 15 for winning the game. This experiment was insightful because this experiment showed that the agent was not able to see the future rewards in the first place. There appeared to be a problem with updating the target network

The following experiment was to test if AlphaBubble was able to win the game with two balls colours and two lines of balls. At this point the squared ball representation was not yet implemented, but the representation of a ball consists of two ones horizontally placed in the array. The algorithm seemed to struggle with balls that were placed vertically to each other, therefore the representation was changed to the squared representation, which immediately got rid of this problem. Another interesting problem, which was indicated in this experiment, was that AlphaBubble sometimes preferred to play forever by keeping a few balls into play. Probably, the future reward seemed more promising to AlphaBubble than winning the game, therefore the reward for winning was enlarged.

Thereafter, an experiment with three different ball colours and two layers of balls was executed. AlphaBubble did not appear able to invent a winning policy. First, adding more convolutional layers was attempted, but this initially was not the solution. Subsequently, a different convolutional layer of the Lasagne package was implemented, which drastically improved the performance of AlphaBubble. The reason for this improvement is still unclear, since the

hyper parameters of both convolutional layers are the same. The new convolutional layer makes use of CUDA wrappers, this might be the cause of the improvement, however this is unsure.

5.2 Architecture

During the thesis many experiment has been conducted with many different architectures. Eventually, the following network, shown in figure 6, was the outcome:

Figure 6: Code of network architecture

```
def build_network():
    l_in = lasagne.layers.InputLayer(
        shape=(None, 8, GRIDSIZE * 2, ARRAYWIDTH * 2)
    )
    l_conv1 = cuda_convnet.Conv2DCCLayer(
        l_in,
        num_filters=32,
        filter_size=(6, 6),
        stride=(1, 1),
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.HeUniform(), # Defaults to Glorot
        b=lasagne.init.Constant(.1),
        #dimshuffle=True
    )
    l_conv2 = cuda_convnet.Conv2DCCLayer(
        l_conv1,
        num_filters=64,
        filter_size=(2, 2),
        stride=(1, 1),
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.HeUniform(),
        b=lasagne.init.Constant(.1),
    )
    l_hidden1 = lasagne.layers.DenseLayer(
        l_conv2,
        num_units=512,
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.HeUniform(),
        b=lasagne.init.Constant(.1)
    )
    l_hidden2 = lasagne.layers.DenseLayer(
        l_hidden1,
        num_units=512,
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.HeUniform(),
        b=lasagne.init.Constant(.1)
    )
    l_out = lasagne.layers.DenseLayer(
        l_hidden2,
        num_units=NUMBEROFACTIONS,
        nonlinearity=None,
        W=lasagne.init.HeUniform(),
        b=lasagne.init.Constant(.1)
    )
    return l_out
```

The input of the network is the game state representation, the output of the network are the Q-values for all possible actions. The network consists of two convolutional layers, followed by two dense layers with a ReLU as non-linearity. The last layer of the network is a dense layer without non-linearity.

To initialise the parameters of the network, the default settings of the He-Uniform were used from the Lasagne package. The initialisation of the bias was set to a constant of 0,1.

During the research several experiments have been run to improve the performance of the algorithm. In order to make AlphaBubble more intelligent, the network was extended with extra layers. However, adding a third convolutional layer did not improve the performance of AlphaBubble immediately, neither did adding more dense layers. Eventually, the filter size of the first layer were 6 by 6 and the second layer 2 by 2, these filter sizes showed the best performance.

5.3 Hyper parameters

In table 3 the hyper parameters are presented. At the start the replay memory is filled up to 3000 random moves, thereafter the training starts with an exploration rate of 0.9. During training every 1000 iterations the exploration rate is decreased with 0.01 until it reaches 0.1. For optimisation of the loss function RMSprop is used, because AlphaBubble appeared to learn faster with RMSprop than Stochastic Gradient Descent. The discount factor is 0.99 in order to make the agent look in the future as far as possible. The rewards in the future stay valuable with a big discount factor and the agent will keep these future rewards in to account. The batch size was set the 128, because the documentation of the convolutional layer states that it acts optimal with batch sizes of multiple of 128.

Table 3: Hyper parameters of AlphaBubble

Hyper parameters	**Value**	**Description**
Batch size	128	*Number of training cases, which is computed each iteration*
Replay memory size	7500	*Maximum number of state transitions that are stored*
Target network update	2500	*Number of iterations that the target network is updated*
Discount factor	0.99	*Discount factor that is used for reinforcement learning*
Learning rate	0.00005	*Learning rate for RMSprop*
Momentum	0.9	*Momentum for RMSprop*
Rho	0.9	*Rho for RMSprop*
Epsilon	0.000001	*Epsilon for RMSprop*
Exploration decay	0.01/1000	*Every 1000 iterations the exploration rate is decreased with 0.01*
Initial exploration	0.9	*Initial value of epsilon for exploration*
Final exploration	0.1	*Final value of epsilon for exploration*
Replay memory start size	3000	*Before training the replace memory is filled with random moves.*

5.4 Comparison to random agent

In this section AlphaBubble will be compared to a random agent to confirm that AlphaBubble has learnt. The agents will be compared on a game of six lines of balls with four and five different ball colours. Both players have 500 shots to finish the game. If the whole field of balls is cleared, the player wins. If a single ball reaches the tenth layer, the player dies. Both agents have an action space of twenty, which means that they can shoot in twenty different directions. The result are shown in figure 5 and table 4:

Figure 7: Plot of win percentage AlphaBubble vs random agent with different amount of ball colours

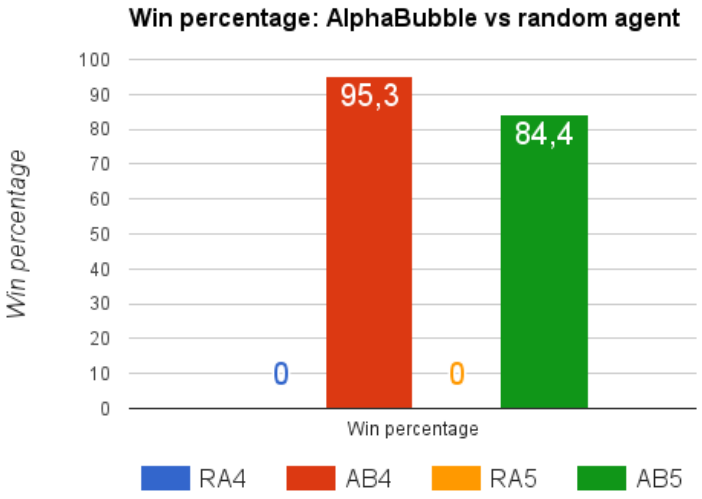


Table 4: Results AlphaBubble(AB) vs random agent(RA) with different amount of ball colours, the number behind the agents name indicates the amount of ball colours in the game

	RA4	**AB4**	**RA5**	**AB5**
Win percentage	0	95,3	0	84,4
Lose Percentage	100	3,7	100	12,5
Unfinished	0	0,9	0	3,1
Average amount of shots to win	-	75,9	-	136,6
Average score	13,63	150	77,85	207,7
Average amount of shots per game	16,5	78,79	14,75	144,6
Balls popped/shot	0,82	1,91	0,52	1,43

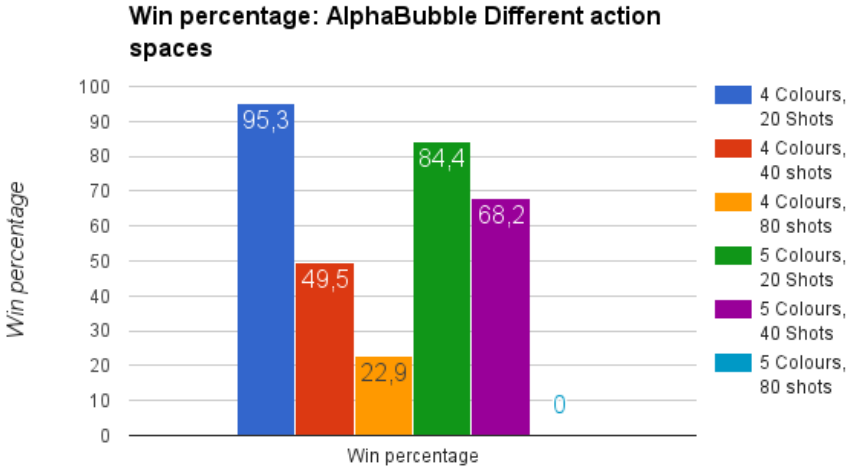
There is a clear difference between the two agents. AlphaBubble wins 95 percent of the games on a game with four different colours, while the random

agent loses all games. Since the random agent is only capable of popping 0.82 balls per shot, which is shown in table 4, it does make sense that the agent can impossibly win, because the number of balls in the game will increase with 0.18 balls per shot. AlphaBubble is able to pop almost 2 balls per shot, this is clearly a winning policy. Also, notice the difference in score between the agents. Although the score is not very important, since you don't score points for winning, only for popping balls. So, if the players want to maximise their scores, they should not try to win. In table 4 is shown that AlphaBubble needs approximately 76 shots to win a game with four colours, this number can probably be reduced when the action space is bigger. Because of the limited amount of moves, it will in some cases not be able to shoot in the optimal direction. Comparing the two agents on a game with five colours, the same phenomena are observed in the game with four colours. However, both agents performed worse because of the increase of complexity of the game.

5.5 Compare agent with different action spaces

In this section AlphaBubble with different action spaces will be analysed. The agent was trained for different action spaces of size 20, 40 and 80 with the same amount of training time, 400.000 iterations on games with 6 lines of balls and 4 or 5 different colours. The training time is kept the same for comparison, this way the effect of the action spaces will be visible.

Figure 8: Results AlphaBubble with different action spaces



In figure 8 is shown that enlarging the action space results in longer training time. The AlphaBubble that was trained with 20 actions and four colours has

found a winning policy, which is able to win almost all games. When adding more actions, the performance of the agent decreases. The AlphaBubble with 80 shots is only capable of winning 23 percent of the games. However, the game seems to become easier when there are more possible actions, since more targets can be hit. However, AlphaBubble probably needs more exploration time to visit enough states and discover actions with bigger action spaces before it converges to the optimal policy. When training with five colours, the effect of the action space is also decrease of performance. Controversy, the AlphaBubble, which was trained with 40 actions and five colours, performed better than the AlphaBubble that was trained with 40 actions on 4 colours, moreover the agent that was trained with 80 actions performed much worse. These results are surprising.

Table 5: AlphaBubble playing with different action spaces, the number behind C indicates the number of colours, S indicates the number of actions.

	4C20S	4C40S	4C80S	5C20S	5C40S	5C80S
Win percentage	95,3	49,5	22,9	84,4	68,2	0
Lose Percentage	3,7	32	61	12,5	27,7	79,9
Unfinished	0,9	18,5	16,1	3,1	4,1	20,1
Average amount of shots to win	75,9	187	244	136,6	171	0
Average score	150	252	232,2	207,7	215	2253
Average amount of shots per game	78,79	215	207	145	162	229
Balls popped/shot	1,91	1,16	1,12	1,43	1,32	0,98

In table 5 some statistics of AlphaBubble with different action spaces shown. The expectation was that a bigger action space would make the game easier to solve. However, when a comparison is made in amount of shots needed to win between the different action spaces, it is striking that the smaller action spaces solve the game faster. However, this probably is because the big action space have not converged yet and therefore are still having trouble with solving the problem. Also, notice the difference between the scores. The bigger the action space, the higher the score. As is said before, the score is not really relevant, since winning doesn't score a player points. The difference in scores is because these agents pop more balls, however they don't find a way to win the game.

5.6 Comparing to a human players

At this section AlphaBubble will be compared with human players. The agent will be compared on a game with 5 different ball colours and 6 lines of balls. Results of 5 human players were collected in order to compare these results to AlphaBubble.

Table 6: AlphaBubble vs human players on a game with 5 different colours and 6 lines of balls

	Humans	AlphaBubble
Win percentage	100	84,4
Lose percentage	0	12,5
Balls popped/shot	1,75	1,43
Amount of shots per game	110	136,6

Although AlphaBubble is only compared to five human players, it is striking that human still clearly perform better than AlphaBubble, which is shown in table 6. Mostly, human players are better in recognising dangerous situations, where AlphaBubble in some case does not understand that the game is almost lost. Moreover, human players easily see that balls are being blocked most of the time and that these balls are not reachable, while AlphaBubble is having trouble with identifying that balls are blocked.

Although humans are still performing better than AlphaBubble, the level of play does not differ too much. A little improvement of play would make the level similar to human players.

5.7 Balls moving down

In this section AlphaBubble will be evaluated on a Bubble Shooter game, where new lines of balls come into play during the game. Every 32 shots two new lines of balls are placed on the top of the already existing balls. Although AlphaBubble has not trained on this variant of Bubble Shooter, it is still interesting to see how it performance in this environment. The best performing agents, AlphaBubble with 4 and 5 colours with 20 actions, were picked for this task. The results are shown in table 7.

Table 7: AlphaBubble playing on a game where new lines of ball come in play

	4 Colours	5 Colours
Win percentage	50.3	3,1
Lose percentage	49.7	96,9
Balls popped/shot	2.14	1.73
Average amount of shots	94	70
Average amount of shots to win	124	159

Interestingly, AlphaBubble is still able to win half of the games with four different ball colours, although it has never trained on this type of play. For AlphaBubble all bubbles in the game have the same value, shooting balls on the top or at the bottom does not make any difference. In a game where new ball come into play, it is important that balls at the bottom are removed first, since a new line of ball could make the player lose. AlphaBubble is not aware of this problem, however its policy still wins 50 percent of the games. It would

be interesting to see if AlphaBubble is capable of finding this strategy, when training on a game, where new balls come in to play.

On a game of 5 colours the win percentage is very low, the problem is still too complex. The combination of extra balls coming in to the game and 5 different ball colours in the game is too difficult for AlphaBubble and therefore it is barely able to win a game.

6 Conclusion

In this thesis deep reinforcement learning is successfully applied to the game Bubbleshooter in order to create the agent AlphaBubble. The implementation of AlphaBubble has led to a decent Bubble Shooter player, which has not performed superhuman behaviour yet. However, AlphaBubble wins 95 percent of the games, when playing with four different ball colours and wins 65 percent of the games when playing with five different ball colours. When comparing AlphaBubble to humans, it is striking to see the difference in recognising dangerous situation and identify that balls are being blocked. Further, different experiments have be executed in order to see what the effect is of different action spaces. As expected, enlarging the action space results in longer training time, probably because the agent needs more exploration time.

To conclude, with deep reinforcement learning a machine can learn to play the game Bubble Shooter. Although a really high level of play hasn't been accomplished yet, there is still enough space for improvement of the algorithm. The algorithm network could be extended with extra layers in order to solve more complex problems. Also, some of the hyper parameters can be optimised, such as replay memory, action space and the reward function. To answer the research question "To what extend is a self-learning agent able to learn playing the game Bubble Shooter using deep reinforcement learning?, at this point the level of play is below average of human players, but with some extension of the network and conducting more experiments, it should be possible to exceed human performance in Bubble Shooter.

7 Discussion

AlphaBubble showed some decent performance, however improvement can still be made. In the usual Bubble Shooter game the number of ball colours in the game is five or six. AlphaBubble is able to play on these games, however is not able to win all games. Also, it has not been trained on a game where new lines of balls come into play. For future research it would be interesting to see if AlphaBubble is able to come up with a policy that is able to play in these sort of games. In order to accomplish that behaviour, there probably needs to be added an extra input node, which indicates how many shots there have been done so far.

An interesting aspect were the experiments with different action spaces, which pointed out that bigger action spaces probably need more exploration and therefore converges slower than small action spaces. Possibly, an adaption could be made to the algorithm, such that not the whole action space has to be calculated, but the algorithm will check only for good potential actions. Perhaps, an combination of three search and deep reinforcement learning could make this idea possible. Moreover, in future research experiments should be run with longer exploration time, this really makes sense to attempt, when training on a larger action space.

Also, improvement of AlphaBubble can be made in optimising all the hyper parameters. During the research experiments have been conducted with many different architectures and hyper parameters, still improvement could be obtained by optimising. Hyper parameters that have not been optimised yet: size of replay memory, the periods of exploration and exploitation, the discount factor, the reward function and the parameters for RMSprop.

Furthermore, deeper networks have showed to solve more complex problems. Possibly, this could help AlphaBubble to learn a policy that is able to invent a winning policy on even more colours or invent a winning policy for a game where new balls come into play. Therefore, experiments with deeper network should be conducted in future research.

The open source implementation of Bubble Shooter has led to several problems. During the thesis bugs have been detected in the game. In some cases, after shooting a ball, the ball suddenly disappears and does not participate in the game. Most of the bugs are fixed, but still in some cases strange events occur, but these event are rare. Fortunately, the self-learning agent has not noticed these bugs and does not abuse them as an advantage.

In the implementation of the game new lines of ball coming into play was not supported. Because of the way the game was implemented, it was hard to add one layer of balls during the game. Therefore, the decision was made adding two layers of balls in stead of one, which was implemented more easily. However, in the original game Bubble Shooter single lines of balls drop down in stead of two. For future research it is recommended to implement the game from scratch in order to prevent these problems.

The Lasagne package offers several convolutional layers. During experimenting with the layers a big difference in performance occurred, when changing from the *lasagne.layers.Conv2DLayer* to *lasagne.layers.Cudaconvnet*. It is still unclear why the CUDA convolutional layer is resulting in a better performance than the standard convolutional layer. Possibly the CUDA wrappers are the cause of the better performance, but this can't be confirmed.

Also, the *lasagne.layers.Cudaconvnet* had several restriction, which has led to some problems. The layer only supported input of the dimension 1, 2, 3 ,4 and multiple of 4. When AlphaBubble plays with 4 colours, there are 5 dimensions, one for every colour and one dimension for which indicates where no balls occur. Therefore, an array of 8 dimensions was created, where the non-necessary dimension were filled with zeros.

For future research it would be interesting if deep reinforcement learning could be utilised for societal goals. At this point, deep reinforcement learning is mostly applied to games and has not been to make a difference in society. Attempts should be made to create agents, which are valuable for the world. For instance, deep reinforcement learning might contribute to self-driving cars.

References

Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., ... Wijshoff, H. (2016). A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, *49*(5), 54–63.

Blum, A. L., & Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial intelligence*, *97*(1), 245–271.

Bom, L., Henken, R., & Wiering, M. (2013). Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *Adaptive dynamic programming and reinforcement learning (adprl), 2013 ieee symposium on* (pp. 156–163).

Campbell, M., Hoane, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, *134*(1), 57–83.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., ... others (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, *29*(6), 82–97.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... others (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Nemati, H. R., Steiger, D. M., Iyer, L. S., & Herschel, R. T. (2002). Knowledge warehouse: an architectural integration of knowledge management, decision support, artificial intelligence and data warehousing. *Decision Support Systems*, *33*(2), 143–161.

Schapire, R. E. (2003). The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification* (pp. 149–171). Springer.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press.

Szegedy, C., Toshev, A., & Erhan, D. (2013). Deep neural networks for object detection. In *Advances in neural information processing systems* (pp. 2553–2561).

Tesauro, G. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, *6*(2), 215–219.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.