Survey Reinforcement Learning

Victor Dolk

September 6, 2010

Eindhoven University of Technology
Department of Mechanical Engineering
Den Dolech 2, 5600MB Eindhoven, Netherlands
v.s.dolk@student.tue.nl

Supervisor: ir. R.J.M. Janssen

Abstract

A Robocup team faces a lot of challenges and problems during the development of its autonomous soccer players. In some cases, the goal of a certain task is clear but the way of getting there is difficult to determine or too complex. Reinforcement learning is a powerful tool to deal with these problems. The purpose of this survey is to explain what reinforcement learning is and to give insight about what it might mean for Robocup. The paper will first explain the reinforcement learning theory and discuss several methods. The theory will be explained on the basis of a maze case. At least the survey will provide insight over the applicability of reinforcement learning in Robocup with some examples and ideas. It shows that reinforcement learning can be applied succesfully but there are still a lot of problems that have to be solved. So the applicability of reinforcement learning in Robocup has been limited until now.

Keywords: Robocup, reinforcement learning, machine learning

Contents

1 Introduction

The goal of a Robocup team is to develop autonomous robots which can win the Robocup Worldcup, a soccer tournament for robots. A wide range of techniques are being used to reach this goal. Reinforcement Learning is an attractive framework because it enables a robot to learn autonomous from experience. For this reason, the current line of research is establish efficient Reinforcement Learning algorithms that are applicable to the real-world. The technique is based on the idea of learning from rewards, comparable to the way how dogs can learn tricks.

The research question of this paper is: What might Reinforcement Learning mean for Robocup? The purpose of this paper is to give insight about what reinforcement learning is and what it is capable of.

The theory of reinforcement learning will be explained on the basis of the book *"Reinforcement Learning: An introduction"*, Sutton R.S. & Barto A.G., MIT press (1998)[2]. It will start with a short introduction to Reinforcement Learning, important terms and notations will be explained. Next, different methods will be discussed and compared with each other on the basis of a maze case. After that, areas of the reinforcement learning theory yet to be examined will be considered. At least, this survey will give some examples and ideas of reinforcement learning in Robocup.

2 Introduction to Reinforcement Learning

2.1 Reinforcement Learning

Machine learning is the science of designing and developing algorithms that allow computers to learn. A computer learns when it is able to evolve certain behavior based on empirical data. In general, machine learning distinguishes three forms of learning:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning algorithms learn from examples. The desired behavior is being demonstrated and the learning object tries to copy that behavior. So the input and the desired output are known. With supervised learning algorithms, a mapping from the input to the desired output can be constructed out of different examples. So it is a matter of correction. This type of learning can only be used if the desired output is known. Attending a lecture is an example of supervised learning.

Unsupervised learning uses only input data to evolve behavior. Categorizing, clustering and organizing input data or learning to estimate certain dimension are examples of this type of learning. The goal of unsupervised learning algorithms is finding a relation between different inputs. This form of learning is very useful to interpret and process huge amount of data. Processing and interpreting our senses could be seen as a form of intelligence. Face recognition is an example of unsupervised learning.

Reinforcement learning learns from rewards and punishments (negative reward). In some cases, the desired behavior/output is unknown even if the purpose/goal of the learning object is clear. If numerical rewards and punishments are defined well, the goal can be reached by a trial-and-error process. Reinforcement learning algorithms try to find a behavior in such matter so the sum of all rewards and punishments is maximal. Reinforcement learning makes use of so-called delayed rewards. It means that not only the immediate but also the future rewards are important for evolving the behavior. An example, in which delayed rewards are necessary, is playing chess. Sometimes a piece has to be sacrificed to obtain a better position. So reinforcement learning evaluates instead of supervised learning which corrects. Learning chess is a good example of reinforcement learning. The desired behavior is unknown because it is unknown how the opponent will react after each turn. Though the main purpose of the game is clear: checkmate the opponent. A better behavior will become clear after evaluating different games of chess by looking which movements were good and which not. The formal definition of Reinforcement learning is: how to map states to actions so as to maximize the numerical reward. The terms states, actions and reward will be discussed later on.

2.2 Advantages of Reinforcement Learning

As told, reinforcement learning is necessary when a desired behavior is unknown. This behavior is often unknown if it is impossible to predict the response of the surroundings. Examples of this case are: the opponent with playing chess, the wind during the steering of an aeroplane and the movements of opponents during a soccer game. The response can also be unpredictable simply because the surrounding is to complex to model. In some cases, the desired behavior is known but to complex to program. An example of this case is walking.

Reinforcement learning can be applied at low-level but also at high-level. Low-level means the direct influence on actuators, for example controlling voltages towards electric motors of a walking robot. High-level means making decisions between prescribed movements/strategies etc. For example the choice between a high and a low pass.

If the response of the surrounding changes in time, than the problem is called a non-stationary problem. Reinforcement learning can take this into account so the behavior adapts itself to the surrounding constantly.

The optimal behavior is not always the best behavior. If the surrounding is insecure, taking the optimal behavior can be risky. A safe behavior is better in this case. By choosing the right algorithm, reinforcement learning can take this into account too. A navigation system could choose an optimal route to drive, but if there is a big chance on a traffic jam, it is better to take a different way. This problem will be discussed later on in the report. Reinforcement learning is able to find the optimal solution and the "safe" solution.

2.3 Central idea of Reinforcement Learning: Evaluative feedback

The central idea of Reinforcement learning is learning by trial-and-error and evaluation. This central idea will be explained using the n-armed bandit problem as illustration. Suppose a casino has n-slot machines with arbitrary winning chances, then the expected profit (or loss) is different for each machine. Because it is unknown for a casino visitor which machine is the best, it has to be found by a trial-and-error process. There are two strategies for dealing with this problem:

- Trying each machine enough times so it will become clear which machine is the best. This process is called exploration
- Pulling the machine which performed the best at that moment. This process is called exploitation

Each strategy has a disadvantage. At the first strategy, a lot of trials are necessary to be able to tell with enough confidence which machine is the best. The second strategy is that there is a chance that not all the machines are tried. So the best machine might not be found.

A solution to this two disadvantages is a combination of the two strategies. So a balance has to be found between exploration and exploitation. This is possible with the next three methods:

- The ε-greedy method: With the chance of ε a random machine will be chosen (exploration). At other times, the machine which is the best at that time will be chosen (greedy). So if $\varepsilon = 1$, pure exploration will be executed. If $\varepsilon = 0$, only exploitation will be performed.
- The *softmax*-method: This method is almost the same as the ε-greedy method. But instead of choosing a total arbitrary machine with chance ε, machine which are better at that moment will be chosen more likely.
- Optimistic initial values: If the begin estimation of each machine's profit is very optimistic, all machines will be tried before starting with exploitation. So in the beginning there is pure exploration and later on only exploitation.

During the trials of the machines, the average profit or loss is being kept. This can be done by making in vector with all trial result in it. The problem then is that the memory required, is increasing in time. The solution is using an incremental implementation of the average. A new average can be calculated with the following formula:

$$\mu_{new} = \left(1 - \frac{1}{1+k}\right)\mu_{old} + \frac{1}{1+k}X \tag{1}$$

k is the number of trials and X the sample profit of the last trial.

In practice, the surrounding will often be non-stationary. For example, the casino owner can decide to change the winning changes each hour. The consequence is that beside exploitation, we constantly have to explore. Furthermore, old measured values have to weigh less for calculating the estimated average profit of each machine. This can be done by adjusting equation 1:

$$\mu_{new} = (1 - \alpha)\,\mu_{old} + \alpha X \tag{2}$$

$\alpha$is called the step-size parameter or learning rate and has to be between or equal to 0 and 1. The closer α gets to 1, the bigger the weight for recent measured sample profits.

2.4 Explanation of the maze case

During this report, a maze will be used to explain the various algorithms. For this, an existing program[1] is used and extended for this survey. The principle behind the maze case is simple: a smiley walks around a maze and has to find the exit as fast as possible after a certain number of trials using reinforcement learning. The positions in the maze are discrete.

A collision with the wall will result in a reward of -50. After the collision, the smiley stays in the same position. Because the smiley has to find the exit as fast as possible, each transition is a reward of -1. The smiley can't look ahead and only recognizes a wall when he collides at one. At each transition, the smiley will only receive the reward and its new position.

[1]Originally created by: Vivek Mehta and Rohit Kelkar under the supervision of Prof. Andrew W. Moore at the Robotics Institue of the Carnegie Mellon University of Pittsburgh.

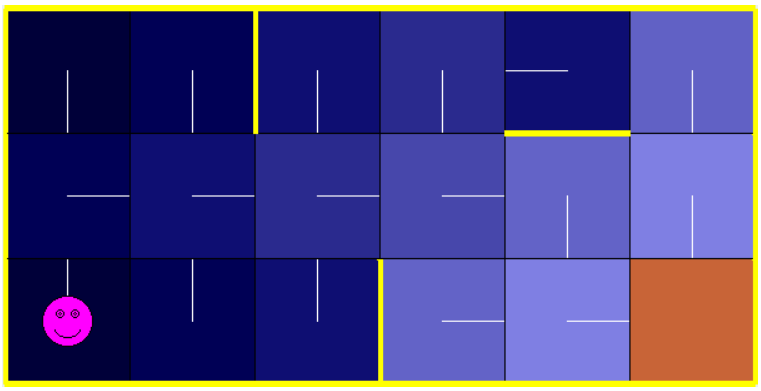

Figure 1: Maze example

Figure 1 shows an example of a maze of the program. The exit is colored orange. The white lines show the preferred directions of the purple figure. When the smiley reaches the exit, he will start all over again. That is why the maze is called a terminal problem.

In the program it is possible to add noise ($PJOG$). This means that there is a possibility that the smiley will go another direction without noticing it being the wrong direction. Suppose the purple figure in figure 1 is about to go upwards, but instead he goes left because of the noise, than the received reward for going upwards is equal to -50. The reason for this is not noticing going another direction because of the noise.

2.5 Terms

In this section, important terms of the Reinforcement learning theory will be explained using figure1.

Agent The agent is the object which learns. The smiley in the figure is the agent of the maze case. In a robot, only the part responsible for making decisions is called the agent. That is the computer most of the time. The arms of a robot for example belong to the environment.

Environment The environment is everything outside the agent. The environment of the maze is set of walls and the exit. The agent and the environment are interacting with each other.

State The state is the situation of the agent in respect to the environment. This situation exists only out of parameters related to the reinforcement learning problem. For a game of chess, it is not important whether it is in the kitchen or the living room. In the maze, the decisions will only depend on the agent's position. So each cell in the figure is a state.

Action The decisions an agent can make in each state are called actions. These actions often have influence on the state but that is not always the case. An action can also lead to the same state, for example when colliding a wall in the maze. In the maze, four different actions are possible: left, right, upwards and downwards. It is clear these actions have influence on the state of the agent (the position).

Policy The policy determines the behavior of the agent. It represents the probability distribution of choosing a certain action. This can be totally random (each action is selected with the same probability), deterministic (in each state, only one action will be chosen) and in between (for example ε-greedy). The greedy policy of the maze is illustrated by the white lines.

Goal The purpose of the agent. In the maze, it's the exit.

Reward A numerical value which will lead the agent towards its goal. The agent receives a reward after each action. If the agent maximizes the sum of all the rewards by choosing the right actions, the best behavior towards the goal is found.

Terminal-state The state in which the agent has reached its goal in a terminal problem. In the maze, it's the exit.

Episode The period from the beginning state towards the terminal state.

Return The sum of all future rewards from a certain state towards the goal. Suppose 10 steps are needed to reach the exit of the maze, in that case the return is equal to -10 since each step gets a reward of -1.

Discount-factor If the problem is continuous, there exists no terminal state. This would mean the return will go to infinity. To prevent this, the discount factor is introduced. With this factor, the sum won't go to infinity. Before a reward is added to the return, the return is multiplied by the discount factor. So if the discount factor is equal to zero, the return will be equal to the immediate reward. This factor is not important in the maze case.

Markov-property A process meets the Markov-property if the decisions can be based only on the current situation. So the history of the process is not important. In chess, someone is able to take over the place of another player without knowing which movements were made. So playing chess can be a Markov-process (unless the strategy is based on the behavior of the opponent during the game).

State-value-function This function gives the expected return as a function of each state under a certain policy. So this function tells whether it is good or bad to be in a particular state. The state-values of a maze are illustrated in figure 3.

Action-value-function This function is almost the same as a State-Value-function except it depends on state-action pairs instead of states. This is necessary for an unknown environment. In such an environment, it is not possible to say which state follows after an action. If the agent in the maze isn't able to see where the walls are, he can't predict when he will walk against the wall.

Optimal state/action-value-function This function is the State/Action-Value-function belonging to the optimal policy, the policy in which the agent gets most rewarded. So the values of this function are maximal.

2.6 Mathematical definitions and formulas

In the last section, various terms were explained. Some of these terms can be formulated mathematically. With these mathematical formulations, an important equation can be derived namely the Bellman-equation. The Bellman-equation forms the basis of most of the reinforcement learning algorithms. The definitions are as follows:

Return with discount factor:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{3}$$

If $T = \infty$, $0 \leq \gamma < 1$ and if $T \neq \infty$, $\gamma = 1$

Markov-property:

$$\Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} \tag{4}$$

The definition means that the response of the environment only depends on the current state and action.

State-value function:

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} \tag{5}$$

It says that the value-function is equal tot the expected return in state s under policy $\pi(s, a)$.

Action-value function:

$$Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\} \tag{6}$$

Probability to get from state s to s' after performing action a:

$$\mathcal{P}^a_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{7}$$

The expected value of the next reward by a transition form state s to s' after performing action a:

$$\mathcal{R}^a_{ss'} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{8}$$

The Bellman-equation relates the expected return value in the current state to the expected return value in the next state. This equation for the state-value-function can be derived as follows:

$$\begin{aligned} V^\pi(s) &= E_\pi\left\{R_t | s_t = s\right\} \\ &= E_\pi\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} | s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+2} | s_t = s\right\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma E_\pi\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+2}\right\}\right] \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned} \tag{9}$$

Similarly, the Bellman-equation for the action-value function can be derived:

$$Q^\pi(s,a) = \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \pi(s',a') Q(s',a')\right] \tag{10}$$

The Bellman optimality equations are the Bellman equations under the optimal policy. These equations are as follows:

$$V^*(s) = \max_\pi V^\pi(s) = \max_{a \in A(s)} \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \tag{11}$$

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q(s',a')\right] \tag{12}$$

The objective of Reinforcement learning algorithms is finding the optimal state-/action-value-function. The optimal behavior/policy can be derived with this function.

3 Elementary Methods

In this chapter, three elementary methods for solving the Reinforcement problem (finding the optimal state-/action-value-function) will be discussed. Each method has its advantages and disadvantages in respect to applicability, efficiency and rate of convergence to the optimal solution. In the next chapter, there will be explained how to combine these three elementary methods in order to reduce their disadvantages and increase their performance.

3.1 Dynamical programming

Dynamical programming is a set of algorithms for finding the optimal policy under the assumption that a perfect model of the environment is given. The problem also has to meet the Markov-property. The algorithms work on the basis of the Bellman-equation (equation 9). So dynamical programming is a form of planning and not learning because the behavior of the agent will not be based on real-time experience but on a fully known model/data. Though the theory behind this method is necessary to understand the algorithms which will be discussed later on. In this section, two dynamical programming algorithms will be explained: Policy iteration and Value iteration. At the end, these two methods will be compared with each other. Provisionally, we will assume the set of actions and states are discrete. The backup diagram of dynamical programming is shown in figure 2. It shows which values are used to backup the state-value function. A white dot represents a state and a black dot an action. A T is a terminal state.

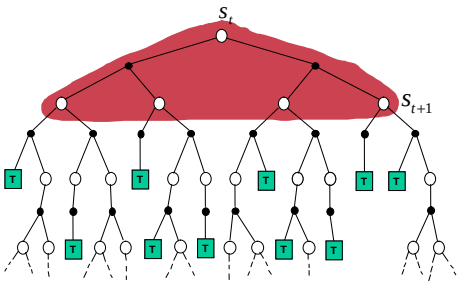

Figure 2: Backup diagram of Dynamical Programming

3.1.1 Policy Iteration

At Policy Iteration, three steps are being executed separately. These three steps are:

- Initialization
- Policy evaluation

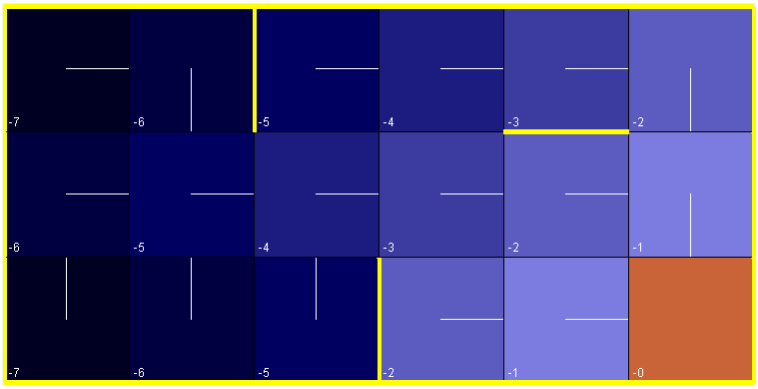

Figure 3: Maze with the optimal state-value in each cell

- Policy improvement

In the first step, the policy and the estimated state-value-function are being initialized arbitrary. Because of the converging iteration process, the choice of the initial values can be arbitrary and different to each other. The initial values of the state-value-function don't have to meet the Bellman-equation. The second step will adjust the state-value-function so it meets the Bellman-equation. Because the set of states is discrete, the state-values can be represented in a matrix. The number in each cell of the maze, represents the state-value (figure 3).

Policy evaluation, the second step of Policy Iteration, finds the state-value-function with an iterative process. In each iteration, the right hand of the Bellman-equation is filled in for each state to obtain a better estimation for the state-value-function (line 8 in algorithm 1). This update is also called a backup. The correct value-function is found if the state-value-function meets the Bellman-equation in each state. By repeating step 2, the estimated state-value function under policy π will converge to the correct state-value function. If the agent doesn't reach the terminal state by following policy π, the repetition has to be terminated. The statement for this termination can be a maximum number of iterations or by a maximum of the state-value-function. In figure 4, four iterations of policy evaluation are shown. The initial policy in this figure is upwards in each state. Notice that the values will go to infinity by running more iterations.

The third step of Policy Iteration is policy improvement. This step determines the action in each state which will give the greatest expected return. The return is calculated with the Bellman-equation (line 16 of algorithm 1. If there is no change to the policy after this step, then the current policy is equal to the optimal policy. In this case, the policy is stable. If policy improvement would be executed on the basis of the fourth iteration of figure 4, the policy would look like figure 5. The pseudo script of Policy Iteration is given in algorithm 1.

Algorithm 1 Pseudo-algorithm of Policy Iteration

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$arbitrarily for all $s \in S$
2. Policy Evaluation
   Repeat
   $\triangle \leftarrow 0$
   For each $s \in S$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]$
   $\triangle \leftarrow \max(|v - V(s)|, \triangle)$
   until $\triangle < \theta$(precision)
3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in S$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]$
   If $b \neq \pi(s)$ then policy-stable $\leftarrow$ false
   If policy-stable, then stop else go to 2

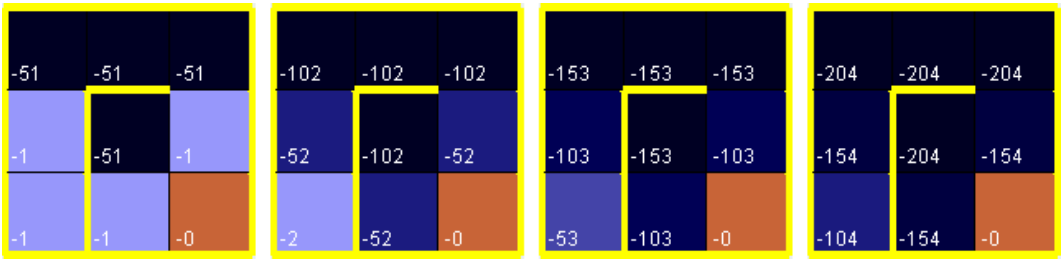

Figure 4: Four iterations of the policy evaluation process

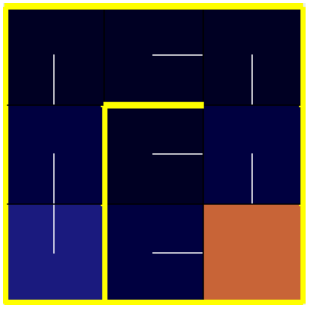

Figure 5: Policy improvement on the basis of the state-values of the fourth iteration of figure 4

Algorithm 2 Pseudo-algorithm of Value Iteration

Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$arbitrarily for all $s \in S$
Repeat
  $\triangle \leftarrow 0$
  For each $s \in S$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]$
    $\triangle \leftarrow \max(|v - V(s)|, \triangle)$
until $\triangle < \theta$(precision)

Output a deterministic policy π, such that
$\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]$

3.1.2 Value Iteration

As told, policy evaluation and policy improvement are two separate steps in the Policy Iteration algorithm. Value Iteration makes these two step at once. Improving the estimated state-value function still happens with the Bellman-equation but under a greedy policy. A greedy policy chooses the action which is estimated as best at that moment, so the action which leads to the greatest estimated return (line 6 of algorithm 2). This iteration is performed over the whole state-set. The repeat will stop if the optimal state-value-function is found.

The optimal policy can be derived from the optimal state-value-function by looking which action gives the greatest outcome on the right hand of the Bellman-equation. The pseudo script of Value Iteration is given in algorithm 2. $\triangle$ denotes the maximum change of the state-value-function after each policy evaluation step. θ is the precision decide whether the policy evaluation step is converged enough. If $\triangle < \theta$, the second step is converged enough. Variable b is used to check whether the policy is stable or not.

3.1.3 Comparison between Policy and Value Iteration

In common, Policy Iteration has to run more often through all the states than Value Iteration because of the policy evaluation step. If a policy doesn't lead to the terminal state from all states, Policy Iteration runs through all the states often unnecessarily. Value Iteration must run trough all the state-action pairs to determine the greedy policy. At a high action/state ratio, Policy Iteration will perform better than Value Iteration. So the bigger the maze, the better Value Iteration performs compared to Policy Iteration.

During the policy evaluation step of Policy Iteration, it is not necessary to converge completely. It is also possible to perform policy improvement after a few steps of policy evaluation. In this way, an algorithm can be made between Policy Iteration and Value Iteration. This can improve the performance.

More states cause longer calculations. The iteration must be made more often and has to go through more states. So a very large state-set is a problem

for Value or Policy iteration. But often it is not necessary to run through the whole state set during an iteration. An improvement of the policy can already be made after iterating only a part of all the states. Only evaluating the relevant states is a solution for very large state sets. In the maze, the states on the way to the end are the most important to evaluate.

3.1.4 Bootstrapping

Dynamical programming does bootstrapping. It implies that estimates are calculated using other estimations. This only works for processes which meet the Markov-property because then the estimated values are independent from each other. Also the Bellman-equation is derived with the assumption that a value-function meets the Markov-property.

3.2 Monte Carlo Method

The Monte Carlo Method in contrast to Dynamical Programming is a learning method. So with this method, the agent is able to make decisions based on real-time experience. This implies that a perfect model of the environment is not required. If the environment is unknown, the action-value function has to be found instead of the state-value function.

3.2.1 Central idea behind the Monte Carlo Method

The Monte Carlo Method can only be applied on terminal tasks. The sequence from begin state up to the terminal state is called an episode. The Monte Carlo Method retains the return value of each state which has been passed during the episode. This can be done by counting up all the received rewards after leaving a certain state for the first time (first-visit Monte Carlo) or every time and averaging the calculated return (every-visit Monte Carlo).

The action-value of a state-action pair is estimated by taking the average of all the returns obtained from that state-action pair. After updating the action-values after each episode, the policy will be improved. After an infinity number of episode, the action-value function and the policy are equal to the optimal action-value function and optimal policy respectively.

As told the problem has to be finite. Another requirement is that every chosen policy must lead the agent to the terminal state. For that reason, the best policy in the beginning is a random policy because it is unknown how to get to the terminal state. With a random policy, there is always a chance to get in the terminal state. During the learning process, exploration is necessary to be sure every state is visited.

Learning with exploration can be realized in two ways: on-policy and off-policy. On-policy means that policy being followed (behavior policy) is equal to the policy being evaluated (evaluated policy). Off-policy means that the behavior policy is independent from the evaluated policy. An off-policy algorithm

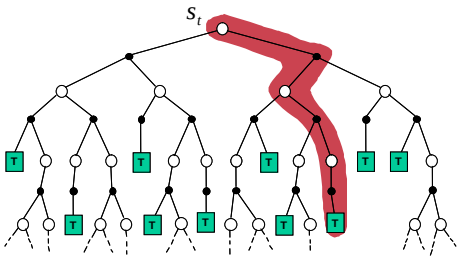

Figure 6: Backup diagram of a Monte Carlo Method

often uses the greedy policy as evaluated policy and a random or ε-greedy policy as behavior policy for exploration.

The backup diagram of Monte Carlo is shown in figure 6. This figure shows which values a Monte Carlo method uses as backup for the action-value function.

3.2.2 Monte Carlo Algorithm

In this survey, only the on-policy algorithm will be attended. The off-policy algorithm requires more explanation and is not necessary for understanding the rest of the survey. Furthermore, the off-policy Monte Carlo algorithm performs worse than the on-policy algorithm because with the off-policy algorithm not all the passed state-action pairs that have been passed will be updated. So it takes more episodes to find the optimal action-value function. However, in the next chapter the difference between on- and off-policy will be discussed. The algorithm that will be discussed in this section is the first-visit algorithm.

The pseudo script of the on-policy Monte Carlo Method is presented in algorithm 3. At first, the action-state value is initialized. In the maze program there's been chosen for random values between 0 and -1. The reason for this is that the greedy policy is also random in the beginning which increases the chance of getting to the terminal state. The policy used is the ε-greedy policy. This means that $\pi(s, a) > 0 \, \forall \, s, a$. So there is always a chance to get in the terminal state.

After initialization, an infinite loop follows. If the problem is stationary, the policy will converge to the optimal policy. In case of a non-stationary problem, the policy will constantly adapt itself to the environment. The loop exists of three steps. In the first step, an episode is performed by the agent using policy π. The episode starts in the initial state which is the lower-left corner in the maze case. During this episode, the return is memorized for each state-action pair that has been visited for the first time. Notice that in algorithm 3, an incremental average is used. In the third step, the current policy is improved.

Algorithm 3 Pseudo script of the on-policy Monte Carlo Method

Initialize for all $s \in S$, $a \in A(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s,a) \leftarrow$ an arbitrary ε-greedy policy
  $R \leftarrow 0$
Repeat forever:
  (a) Generate an episode using exploring starts and π
  (b) For each pair s, a appearing in the episode:
    $R \leftarrow$ return following the first occurrence of s, a
    $Q(s,a) \leftarrow Q(s,a)\frac{k}{1+k} + R\frac{1}{1+k}$
  (c)For each s in the episode:
    $a^* = \arg\max_a Q(s,a)$
    For all $a \in A(s)$
    $$\pi(s,a) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|A(s)| \text{ if } a = a^* \\ \varepsilon/|A(s)| \text{ if } a \neq a^* \end{cases}$$

3.2.3 Performance of the Monte Carlo Method

An important difference between the Monte Carlo Method and other methods discussed in this report is that Monte Carlo does not bootstrap. In some cases it is possible to use this method for non-Markov processes.

Monte Carlo has more advantages compared to Dynamical Programming. Monte Carlo doesn't require a perfect model of the environment because Monte Carlo can learn from experience. Monte Carlo is also able to focus on a subset of states. In the maze the states on the route to the exit are the most important. States far away from that can be more or less neglected by Monte Carlo. Finally, it is sometimes possible to simulate single runs but not possible to determine the probability of certain transitions. The Monte Carlo Method is able to learn from these simulated episodes.

In practice, the Monte Carlo Method is not efficient. In the maze case, it takes a lot of random steps before the agent has reached the exit. Because of this, the action-values become very high after the first run. A lot of episodes are necessary to average the action-values to a reasonable value. So the convergence behavior of Monte Carlo is very slow. A solution to this problem is giving recent measured points more weight. However, a wrong exploration step can take the agent in a risky situation which also leads to high return values. If these return-values are weighed to heavy, it will have a bad influence on the new policy. For this reason, an extra function for Monte Carlo is added to the maze program. In the beginning of the learning process, the function weighs recent values very heavy. Later on in the learning process, all values are weighed equal. So the definition of the learning rate is as follows: $\alpha(t = 0) = 1$ and $\alpha(t \to \infty) = \frac{1}{1+k}$. This only works for stationary environments. If the environment is non-stationary, an adaptive learning rate is necessary for improvement. As soon as a change in the environment is observed, the learning rate should go to 1 again.

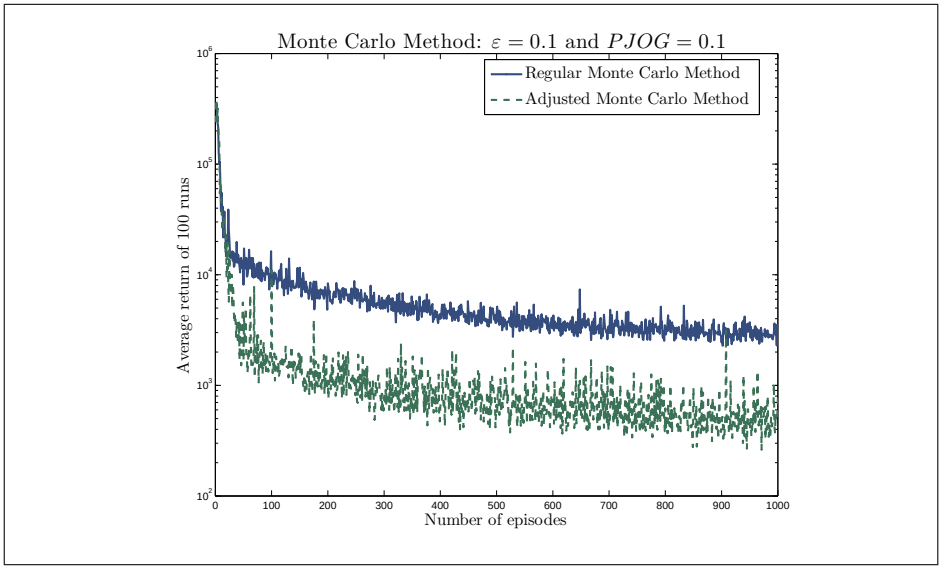

Figure 7: Regular Monte Carlo Method versus Adjusted Monte Carlo Method

The performance of the maze with and without the extra function is shown if figure.

In spite of the bad performance of Monte Carlo, the idea behind Monte Carlo is important to develop a more efficient algorithm in the next chapter.

3.3 Temporal Differences

Dynamical programming uses a full backup to calculate estimated state-values. A full backup stands for using the state-values of all possible following states as backup. Monte Carlo uses a deep backup which is a backup based on all the received rewards during an episode. Temporal Differences (TD) is the third possibility, it makes use of shallow and sample backups (the opposite of deep and full backup respectively). This means that Temporal Differences is able to learn from experience like Monte Carlo. But TD backups an action-value-pair after each step instead of after a whole episode. Dynamical Programming uses the state-value of all possible following states, TD uses the action-value of only one following state. So TD also does bootstrapping. The backup diagram of TD is shown in figure 8.

In this section, two algorithms will be explained: on-policy TD (also known as Sarsa-learning) and off-policy TD (also known as Q-learning). Finally the performance of these two methods will be compared to each other.

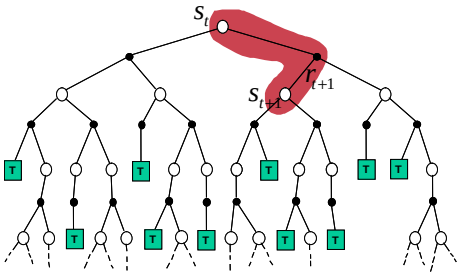

Figure 8: Backup diagram of Temporal Differences

Algorithm 4 Pseudo-script of Sarsa-learning (on-policy TD)

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode)
  Initialize s
  Choose a from s using policy derived from Q (e.g. ε-greedy)
  Repeat (for step of episode)
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s', a') - Q(s,a)\right]$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

3.3.1 Sarsa-learning

The name Sarsa comes from $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ which is in essence the description of the backup diagram of Sarsa (see figure 8). Sarsa uses the current action-value, the reward and the action-value belonging to the next state and action to backup the current action-value. An exploring policy has to be followed e.g. ε-greedy. A combination of the Bellman-equation and the incremental average (equation 2) is used to derive the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{13}$$

Because of the incremental average, the probabilities on a certain transition performing action a is not required. The pseudo-script of Sarsa-learning is shown in algorithm 4.

First the action-value function is initialized. The initial values can be chosen arbitrarily. High (optimistic) initial values will encourage exploring in the beginning. In the maze program, the initial values are zero. The next line is a loop which is, like Monte Carlo, infinite. At the beginning of an episode, an initial state is chosen which is the lower-left corner in the maze case. The policy used in the program is the ε-greedy policy. Notice that the action-value is being updated after each step in contrast to Monte Carlo which updates after each episode.

Algorithm 5 Pseudo-script of Q-learning (off-policy TD)

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode)
  Initialize s
  Repeat (for step of episode)
    Choose a from s using policy derived from Q (e.g. ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

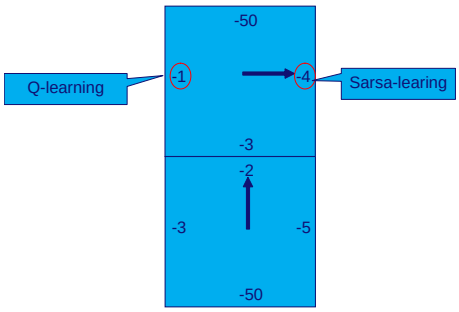

Figure 9: Difference between Sarsa- and Q-learning. The second move is taken arbitrary. Sarsa would use -4 for the backup and Q-learning -1.

3.3.2 Q-learning

The difference between Sarsa- and Q-learning is very small. However, the next subsection will show, that the difference between the two methods will certainly have consequences. With an off-policy method, the evaluating policy is independent from the behavior policy. In Q-learning, theε-greedy policy is being followed but the greedy policy is being evaluated. This leads to a small alteration to equation 13:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{14}$$

So instead of using the action-value of the next state action-pair, the maximal action-value of the next state is used for the backup. So the greedy policy is being evaluated. The algorithms is shown if algorithm 5. Figure 9 illustrates the difference in backup between Q- and Sarsa-learning.

3.3.3 Sarsa-learning versus Q-learning

The difference between on- and off-policy seems to be very small. In most of the cases, the performances will be equal. Though there is a difference in the convergence. Q-learning converges to the optimal solution. Sarsa-learning

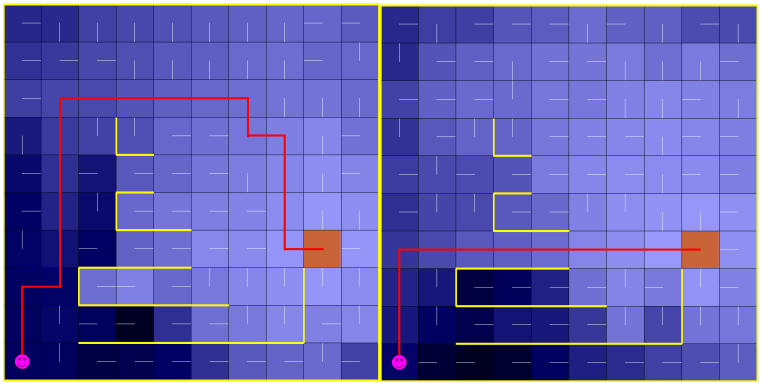

Figure 10: Difference of convergence between Sarsa- (left) and Q-learning (right)

converges to the "safe" solution. The optimal solution, is not always the best solution. This is determined by the uncertainties of the environment. As told, a navigation system doesn't always tell the best way to go because of the probability on traffic jams. In figure 10 the difference in convergence to a certain policy is shown. It is difficult to judge which method is the best. The relative performance of the two methods depends from the problem. Most of the cases, Sarsa is better because of the uncertainties in the real world. But sometimes it is necessary to take risk for example playing roulette. Safe players won't make huge profits. So in this case Q-learning will give a better solution because the chance of winning a lot of money is bigger. Though, the chance of losing a lot of money is also bigger. So coincidence is an important factor.

3.3.4 Performance of TD in relation to Monte Carlo

Temporal Differences methods perform a lot better than the Monte Carlo Method. This can be explained by the fact that TD already learns during an episode. So the disadvantages of Monte Carlo don't count for TD. The difference in performance has not yet been proved mathematically, though it can be proved empirical (see figure 11 and 12). TD does bootstrapping, so the method only works for Markov processes.

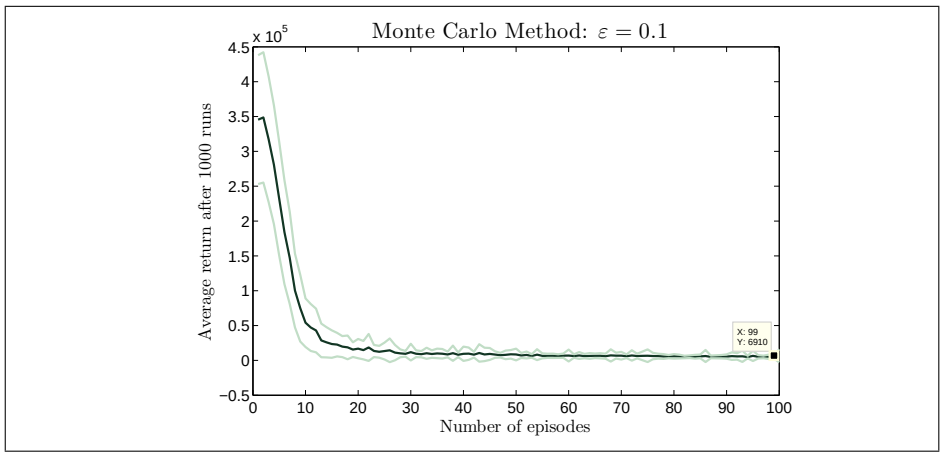

Figure 11: Performance of the Monte Carlo Method

4 Combination of methods

In the previous chapter, three important methods of the reinforcement learning theory were explained. Every method has its advantages and disadvantages. In spite of the big differences, the methods can be combined. So the methods aren't alternatives to each other. This chapter will explain how to combine the elementary methods. A balance will be found between deep backup (using a whole episode) and shallow backup (using one step) and between planning (full backup) and learning (sample backup).

4.1 Eligibility Trace

The similarity between a Monte Carlo Method and TD is that they both perform a sample backup. The difference is the depth of the backup. They are each others extremes in that prospect. It is possible to define a method which is in between. An example is the n-step TD prediction. Instead of using one step, n-steps can be used for making a backup. The notation of a n-step TD prediction is TD(n-step). So the previously discussed TD method is the TD(1-step) method. The backup diagram of different TD(n-steps) methods is shown in figure 13.

4.1.1 Central idea of Eligibility Trace

TD uses a single step return for the backup an Monte Carlo a T step return, where T is the number of steps in an episode. The definition of the n-step return is as follows:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \qquad (15)$$

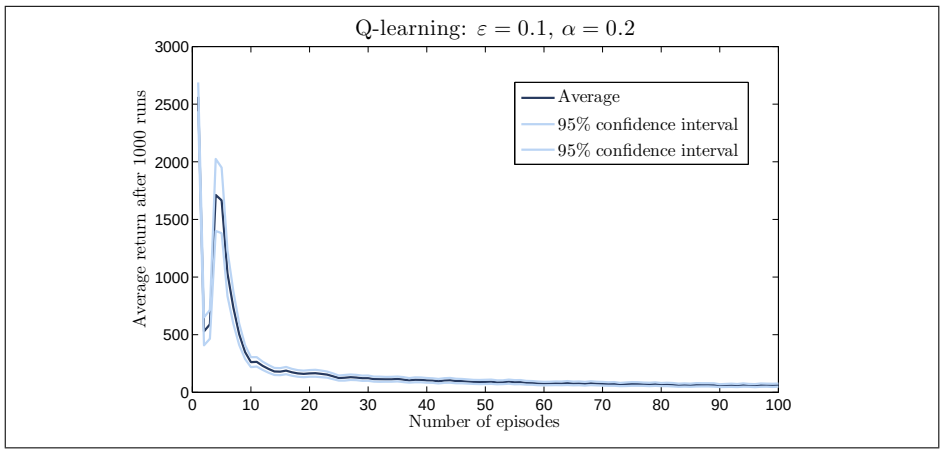

Figure 12: Performance of Q-learning

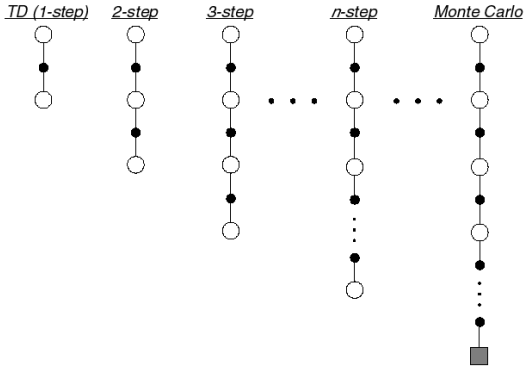

Figure 13: backup diagram of TD(n-step)

Different n-step returns can be combined by averaging, for example the 2-step and the 4-step return:

$$R_t^{ave} = \frac{1}{2}R_t^{(2)} + \frac{1}{2}R_t^{(4)} \tag{16}$$

The backup diagram of this combined return is shown in figure 14. By making use of the average of different n-step returns, the backups of the estimated action-values are based on more than one estimated value. So a better performance can be expected of an algorithm using this technique.

One way to use all the n-step returns for calculating a return is making use of the so-called λ-return:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)} \tag{17}$$

The notation of a TD method making use of the λ-return is TD(λ). The backup

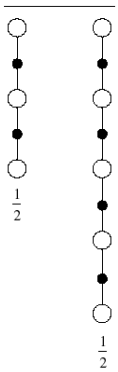

Figure 14: backup diagram of combined n-step return

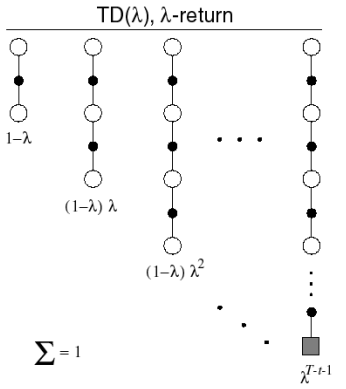

Figure 15: backup diagram of TD(λ)

diagram of this method is shown in figure 15. Notice that the sum of all the returns' weight factors is equal to one. If lambda is equal to one, the backup diagram is equal to the backup diagram of the every-visit Monte Carlo Method. Though the TD(1) and the every-visit Monte Carlo Method are not completely the same. TD(1) learns during an episode and can be applied to discounted continuing tasks. So TD(1) will perform better than the regular Monte Carlo Method. The TD(0) method is equal to the regular TD method.

Implementing equation 17 to the Sarsa- and Q-learning algorithm is not possible since the equation is not causal. This implies, unknown data of the future is used in that equation. Instead of trying to predict the future (forward view of TD(λ)), we can recall current data to the past (backward view TD(λ)). Chapter 7.4 of the book Introduction to Reinforcement Learning shows that these two views are equivalent. δ_t is the TD error and e_t the eligibility trace. The eligibility trace determines the influence of a TD error on a backup. The

Algorithm 6 Pseudo-script of Sarsa(λ)

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
Repeat (for each episode)
  Initialize s
  Choose a from s using policy derived from Q (e.g. ε-greedy)
  Repeat (for step of episode)
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ε-greedy)
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s, a) \leftarrow e(s, a) + 1$
    For all s, a :
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
      $e(s, a) \leftarrow \gamma\lambda e(s, a)$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

TD error and the (accumulating) eligibility trace are calculated as follows:

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{18}$$

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{19}$$

A backup can be made on the basis of the following formula:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a) \qquad \text{for all } s, a \tag{20}$$

This backup can be implemented in the Sarsa- and Q-learning algorithms. The algorithms are then notated as Sarsa(λ) and Q(λ).

4.1.2 Sarsa(λ)

The Sarsa(λ) algorithm is shown in algorithm 6. Notice that if $\lambda = 0$, the algorithm is identical to the regular Sarsa algorithm. In the Sarsa(λ) algorithm, more action-values are updated in one step. The degree on which they are updated depends on λ. The eligibility trace is illustrated in figure 16. Notice that the trace can be bigger than one and that some actions are caused by noise.

4.1.3 Q(λ)

The Watkins' Q(λ) algorithm is shown in algorithm 7. The first difference with Sarsa(λ) is instead of using $Q(s', a')$ for the backup, $Q(s', a^*)$ is used (a^* is the greedy action). The second difference is that Q(λ) cuts off the trace after making an exploring action. So if the behavior policy is particularly exploring, the trace will constantly be cut off and short. In this case Q(λ) converges only a fraction faster than Q-learning. It is possible to make some variations to the algorithm such as Peng's Q(λ)[3] or naive Q(λ). These methods will not be explained in this survey.

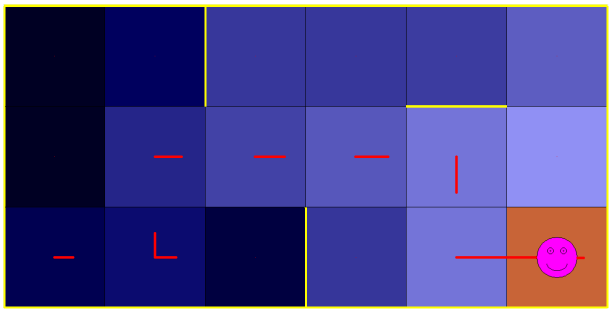

Figure 16: Eligibility trace

Algorithm 7 pseudo-script of Q(λ)

Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$, for all s,a
Repeat (for each episode)
  Initialize s
  Choose a from s using policy derived from Q (e.g. ε-greedy)
  Repeat (for step of episode)
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ε-greedy)
    $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s, a :
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      If $a' = a^*$, then $e(s,a) \leftarrow \gamma\lambda e(s,a)$
                  else $e(s,a) \leftarrow 0$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

4.1.4 Replacing Trace

The eligibility trace discussed in section 4.1.1, can become bigger than one if the same state-action pair is visited more than once. In practice, a replacing trace performs better. Instead of the eligibility trace raises with one when a state-action pair is visited (which happens with the accumulating trace discussed in section 4.1.1), a replacing trace becomes one. So the definition of the replacing trace is as follows:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{if } s = s_t \text{ and } a \neq a_t; \\ 1 & \text{if } s = s_t \text{ and } a = a_t; \end{cases} \tag{21}$$

Another variation of the eligibility trace is a small adjustment recommended by Singh and Sutton (1996)[5]. In this approach, the trace of all the other actions than the chosen action in a certain state is set to zero. This eligibility trace is defined as follows:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t \\ 0 & \text{if } s = s_t \text{ and } a \neq a_t; \\ 1 & \text{if } s = s_t \text{ and } a \neq= a; \end{cases}$$

This method performs better because in this way a bad action doesn't leave a trace. This variant of the replacing eligibility trace is applied in the maze program.

4.1.5 Performance of an eligibility trace

An eligibility trace forms a bridge between TD and Monte Carlo. As told, a Monte Carlo method has advantages during non-Markov processes by comparison with TD. Because the eligibility trace brings TD closer to Monte Carlo, the TD(λ) method also benefits during non-Markov processes by comparison with other bootstrapping methods.

A disadvantage of the eligibility trace is not knowing the optimal value of the variable λ.It can only be determined empirically. The right balance between Monte Carlo and TD can't be found theoretically. The optimal value also depends on the choice of $\alpha$and γ. So an empirical search to the optimum is very difficult. The optimum is always in between Monte Carlo and TD so:

$$0 < \lambda_{optimum} < 1$$

Using eligibility trace will lead to a faster learning algorithm because it uses the estimated action-values in a more efficient way than TD. This means in case of the maze that less episodes are needed to find the optimal policy (see figure 17). This is an advantage if the agent is learning on-line (in practice). For off-line learning, the number of episodes needed to learn a certain behavior is not important but the total calculation time. Using eligibility traces will be at the expense of the calculation time. So during off-line learning, it is not useful to use eligibility traces. During off-line learning, the preference goes to TD(0).

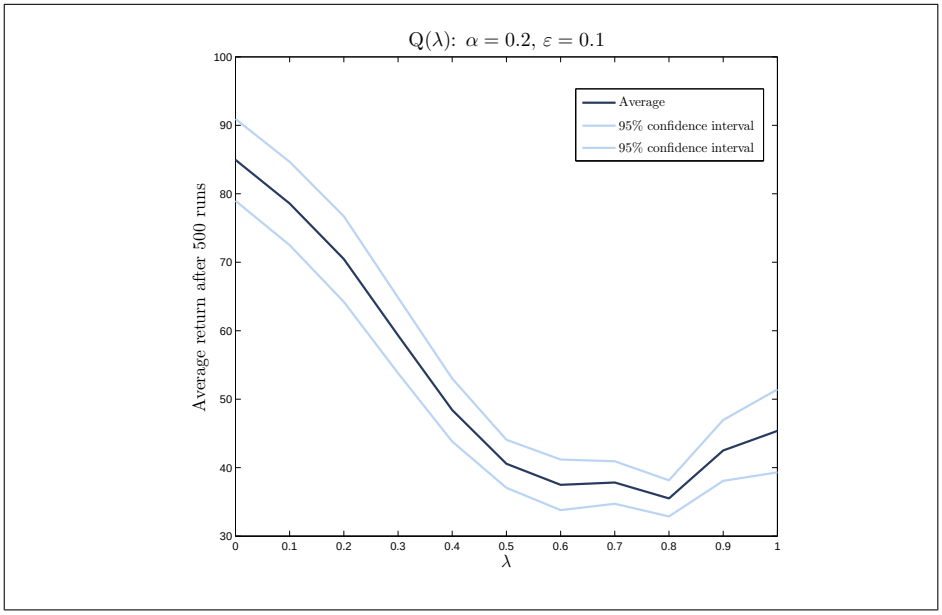

Figure 17: Average return as function of λ

4.2 Planning and learning

The essential difference between Dynamical Programming and the other two elementary methods is the requirement of a perfectly described model of the environment. If such a model is available, it is possible to perform full-backup instead of a sample backup. During a Monte Carlo or a TD learning process, the gathered information has been managed inefficiently. For example, a received reward will only be used for a single backup and then be thrown away. This section explains how to use this gathered information more efficiently so it will be possible to increase the performance of a learning algorithm considerably.

4.2.1 Main idea behind planning and learning

While applying the Monte Carlo and TD algorithm in the real-world, the action-values have only been updated on-line. The gathered information (rewards and next states) has been used only a single time for improving the state-value estimations and has not been stored into the memory. If the agent would have stored this information, he could have used it as a model for performing back-ups off-line. The input of this model is the current state-action pair, the outputs of this model are the associated reward and successive state. This idea is schematically drawn in figure 18. This principle can be depictured as an alternation between learning (on-line) and planning (off-line).

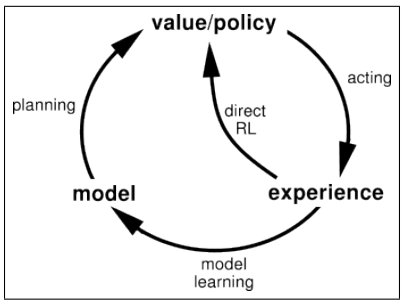

Figure 18: Relationship among learning, planning and acting

Algorithm 8 Pseudo script of Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in A$ and $a \in A$
Do forever:
- (a) $s \leftarrow$current (non terminal) state
- (b) $a \leftarrow \varepsilon - greedy(s,Q)$
- (c) Execute action a; observe resultant state s', and reward r
- (d) $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
- (e) $Model(s,a) \leftarrow s', r$ (assuming an deterministic environment)
- (f) Repeat N times:
  - $s \leftarrow$random previously observed state
  - $a \leftarrow$random action previously taken in s
  - $s', r \leftarrow Model(s,a)$
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$

4.2.2 Dyna-Q

An example of interchanging learning and planning is the Dyna-Q algorithm shown in algorithm 8. The steps from a to d are identical to the Q-learning algorithm. In step e, the model of the environment is being updated. Notice that the model assumes a deterministic environment. It doesn't store the frequencies/ probabilities of certain rewards and successive states from a certain state-action pair which is vital information for non-deterministic environments. Step f performs back-ups off-line. Instead of using a perfect description of the environment, the on-line obtained model is used to apply off-line Q-learning. The state-action pair to be evaluated is chosen randomly from all previous visited states. Step f is being repeated N times. Increasing N will to some extent cause a decrease in the number of episodes needed for finding the optimal/best policy.

In a non-stationary environment, the model has to adapt itself constantly to the changing environment. The agent has to be stimulated to explorer states which haven't been visited in a long time to assure that the model is "up to date".

Algorithm 9 Pseudo script of Prioritized Sweeping

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a and $PQueue$ to empty
Do forever:
- (a) $s \leftarrow$current (non terminal) state
- (b) $a \leftarrow policy(s,Q)$ (e.g. ε-greedy)
- (c) Execute action a; observe resultant state, s' and reward, r
- (d) $Model(s,a) \leftarrow s', r$
- (e) $p \leftarrow |r + \gamma \max_{a'} Q(s',a') - Q(s,a)|$
- (f) if $p > \theta$, then insert s,a into $PQueue$ with priority p
- (g) Repeat N times, while $PQueue$ is not empty:
  - $s,a \leftarrow first(PQueue)$
  - $s', r \leftarrow Model(s,a)$
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  - Repeat, for all $\bar{s},\bar{a}$ predicted to lead to s:
    - $\bar{r} \leftarrow$predicted reward
    - $p \leftarrow |\bar{r} + \gamma \max_a Q(s,a) - Q(\bar{s},\bar{a})|$
    - if $p > \theta$ then insert $\bar{s},\bar{a}$ into $PQueue$ with priority p

4.2.3 Prioritized Sweeping

In step f of the Dyna-Q algorithm, s and a are chosen randomly with useless back-ups as consequence. This is improved in the prioritized sweeping algorithm. This method makes use of a priority queue of state-action pairs. The state-action pair with the highest priority will be chosen first. The priority of a state-action pair depends on the absolute value of the (expected) TD error p. The higher the priority of a state-action pair, the higher the position in the priority queue.

The algorithm is shown in algorithm 9. Step e calculates the TD error which is equal to the priority of a state-action pair. If the TD error of a state-action pair is larger than a certain threshold, the state-action pair will be added to the priority queue (step f). The action value of state-action pairs with a low priority will not be evaluated off-line. Step g evaluates the state-value belonging to the state-action pair with the highest priority. If the action-value belonging to the state-action (s,a) is changed, priorities of state action pairs that have s as possible successive state will also change. These state-actions pairs can be predicted with the model. The second repeat of g calculates the priorities of these predicted pairs and adds them to the priority queue if the priority is high enough. Step g will be repeated N times. The repeat also stops when the priority queue is empty.

4.2.4 Performance of planning and learning

In figure 19 the influence of the number of off-line backups on the average return as function of the number of episodes is shown. The graph is based on data of the maze application. The performance improves a lot when the number of off-line back-ups is increased. With the regular Q-learning, it takes 20 episodes

to find the optimal solution. Dyna-Q with 50 off-line backups per step takes only 6 episodes to find the optimal solution. Notice that there is nearly no difference in performance between 50, 100 and 1000 off-line back-ups per step. So in this case, it is not necessary to perform more than 50 back-ups per step. The performance of prioritized sweeping is even better than Dyna-Q because the off-line backups are chosen more efficient.

The disadvantage of Dyna-Q and Prioritized Sweeping is that it can't be combined with function approximation. Function approximation will be discussed in the next section.

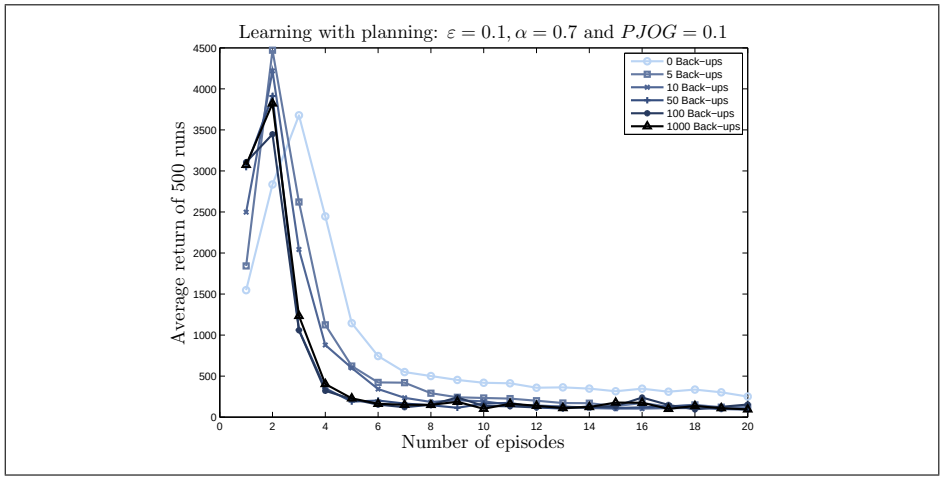

Figure 19: Performance of the Dyna-Q algorithm

5 Function Approximation

Until now, an action-value or a state-value function could be pictured as a look-up table. Each element of this table represented a state or a state-action pair. If the state or action set is becoming very large or even continues, this way to represent an action- or a value-function isn't possible anymore. The bigger the state or action set, the larger the amount of memory needed and the longer the calculation/ learning time. The only way to solve this problem is making use of generalization. This generalization can be made using supervised learning: artificial neural networks (ANN), gradient descent method, pattern recognition and statical curve fitting.

The action- or state-value function will be represented with a set of features and depends from a set of parameters. The parameters can be updated with the so-called gradient descent method or in case of an ANN with back-propagation. So the principle of reinforcement learning maintains but the representation and ways of making back-ups of an action- or a value-function are different. The different ways of making back-ups are based on decreasing the mean squared error of the desired output and the approximated output by tuning the parameters.

Function approximation is a separate subject of machine learning. For this reason, the exact working of function approximation will not be discussed. Though, a short explanation of the linear gradient descent method is in appendix II and of neural networks is in appendix I.

Function approximation is a good method to handle big or continuous state and/or actions sets. Though it has a disadvantage. Function approximation methods in combination with Reinforcement Learning still has poorly understood in the sense of its convergence behavior. It is hard to predict whether an RL algorithm with function approximation will converge or diverge to the right solution.

An example of a case which can only be solved with function approximation is the mountain car problem (see figure 20). Consider a car which is in between two hills and has not enough power to climb to the top. The only solution is moving up and down to build up enough inertia to reach the top of the right hill. This problem has a continuous set of states. The car first has to move away from to goal before it is able to reach it. The three actions are: do nothing (no throttle), full throttle forward and full throttle reverse. The states are dependent from the cars horizontal position and speed. The linear gradient descent Sarsa(λ) method (see appendix II) is used to solve this problem. The result after optimizing the parameters is shown in figure 20.

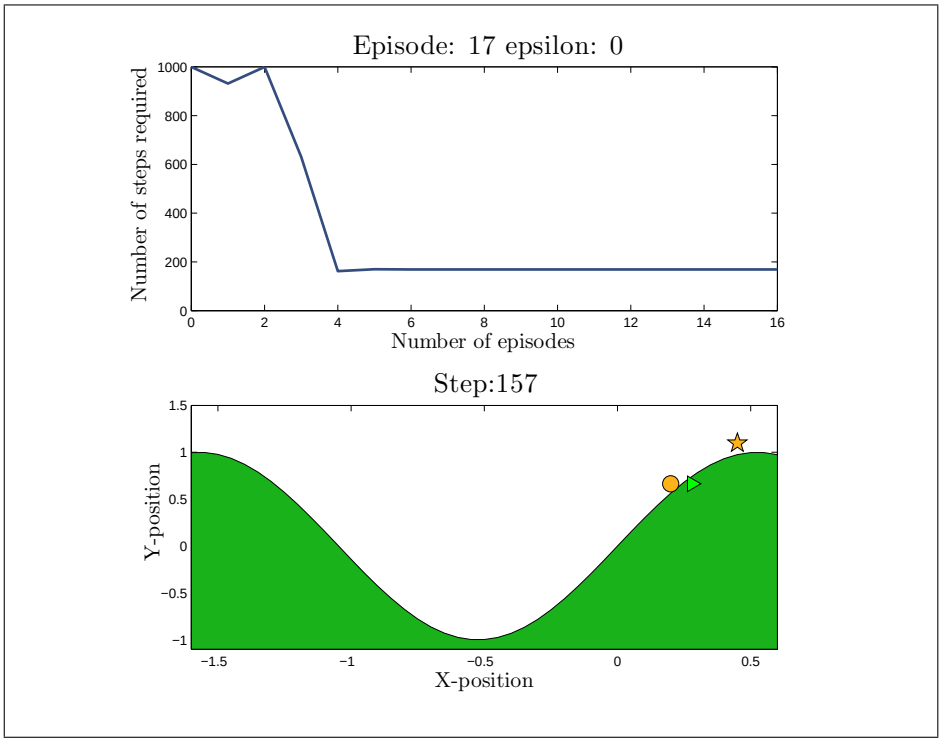

Figure 20: Mountain Car problem

6 Enumeration of the discussed dimensions of Reinforcement Learning

The previous chapters have shown among others that such thing as an optimal reinforcement algorithm doesn't exist. Choices between the different approaches will depend on the reinforcement problem. In this chapter, the dimensions of reinforcement learning discussed in this report will be enumerated.

Exploration $\lhd\rhd$ Exploitation Discussed in section 2.3. The ε-greedy policy is an example of a policy which alternates exploitation and exploration.

Stationary environment $\lhd\rhd$ Non-stationary environment This determines the choice of the value of α.

Environment is known $\lhd\rhd$ Environment is not known If there exists a perfectly described model of the environment, Dynamical programming, among the reinforcement learning, is the most efficient approach to solve the problem. Still a choice has to be made between Value and Policy Iteration which depends on the actions/states ratio.

Action-Values $\lhd\rhd$ State-Values If the environment is known, state-values will be used for solving the reinforcement learning problem because it is clear what the successive state is after an action from a certain state. If the consequence of an action is not known, action-values have to be found to solve the problem.

Full-backup $\lhd\rhd$ Sample backup Dynamical Programming uses a full-backup (see figure 2 and 21). This is only possible when a perfectly described model of the environment is available. A full-backup uses all the state-values of all the possible successive states. Sample-backup only uses one state-value of the next state. If the environment is not known, it is impossible to try all the potential successive states.

Deep backup $\lhd\rhd$ Shallow backup Monte Carlo uses a deep backup (see figure 6 and 21). TD and Dynamical Programming use a shallow backup (see figure 8 and 2). A balance between the deep backup of Monte Carlo and the shallow backup of TD can be made with eligibility traces. The optimum is never pure TD or pure Monte Carlo but always in between.

Accumulating traces $\lhd\rhd$Replacing traces Two different definitions of the eligibility trace. A replacing trace will never exceed one.

Episodic $\lhd\rhd$ Continuing Reinforcement learning problems can be episodic or continuing. An episodic problem has a terminal state. Monte Carlo can only be applied to episodic tasks.

Discounted $\lhd\rhd$ Undiscounted If a problem/task is continuing, returns will go to infinity if the sum of all the rewards is undiscounted. That's why a discount factor is used to make the returns finite. For episodic tasks (like the maze), the discount factor is equal to one.

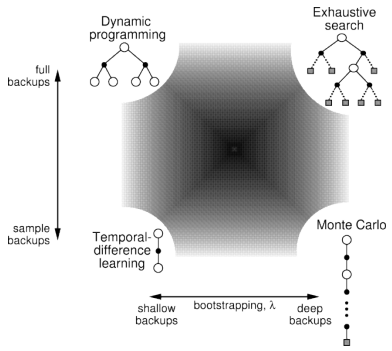

Figure 21: Space of reinforcement learning methods

Bootstrapping ◁▷ Non-bootstrapping Bootstrapping methods perform better than non-bootstrapping methods in most of the cases but can't be used for non-Markov processes.

Real ◁▷ Simulated Backups can use real experience but also simulated experience or both. The best balance between real and simulated experience depends on the quality of the simulated experience.

On-line ◁▷ Off-line On-line learning is learning from current received data from the real world. Off-line learning is learning from a theoretical (Dynamical Programming) or empirical (Dyna-Q and Prioritized Sweeping) model.

Location of backups During off-line learning, states and actions has to be chosen. This choice influences the performance of the learning algorithm like we've seen at Dyna-Q and Prioritized Sweeping.

Timing of backups Monte Carlo backups after each episode. TD backups after each step. The timing of the backups has an important influence on the performance of an algorithm.

On-policy ◁▷ Off-policy On-policy means evaluating the policy which is being followed and Off-policy means evaluating a different policy than the policy which is being followed. On-policy methods are used for finding the "save" solution and Off-policy methods are used for finding the optimal solution.

Discrete states and actions ◁▷Continues states and actions If the state- and action-set are discrete and not too large, value-functions can be represented in a look-up table. In other cases, function approximation is needed.

Low-level ◁▷ High-level Reinforcement learning can be applied at low-level: direct control of actuators, but also at high-level: decision making on strategic level (e.g. choose whether to pass or to dribble)

7 Areas yet to be examined

The reinforcement theory still has a lot of open questions to be researched. In this chapter, few of these questions will be introduced.

There is still no mathematical proof for TD performing better than the Monte Carlo Method. Also the convergence to the optimal solution isn't proved for all the mentioned methods, especially methods which are using function approximation.

A lot of research is being done towards applying reinforcement learning on non-Markov processes. In most of the cases, it is about making smart signal construction in such a matter that non-Markov processes can be treated as Markov-processes. The theory of Partially Observable Markov Decision Processes (POMDP) is an example of such an approach. This theory is about states which are not fully observable for example because of noise on the sensor or a limited vision. In this case, other signals will be used to predict a chance of being in a certain state.

Previously discussed methods are missing an interesting potential extent to reinforcement learning namely the idea of hierarchy and modularity. With the current reinforcement learning methods, an agent learns a behavior for a certain task or problem. If the agent must learn a new task but familiar to the one he has already learned (modularity), he still has to start the learning process all over again. Humans are able to choose learned behaviors as primitive actions. So we are able to learn different levels and to relate them to each other. In this way, complex problems can be solved by first solving small problems.

Another research area in the reinforcement learning theory is dealing with a multi-agent environment (like a Robocup team). In such an environment, multiple agents must interact and incorporate with each other, for example they must be able to divide tasks.

At last, a learning algorithm would be perfect if it combines all the three forms of machine learning: supervised, unsupervised and reinforcement learning. The methods discussed in this report, require pre-defined states and actions. With unsupervised learning, the agent is able to categorize input data. Together with reinforcement learning, there has to be a way that an agent can define the states and actions by itself. With supervised learning, the agent must, besides generalization, also be able to deal with hints and advice. Unfortunately, this integrated form of machine learning has not yet been developed.

8 Reinforcement Learning and Robocup

Reinforcement learning can mean a lot for Robocup and can be used in different ways. In this chapter, several existing examples and some potential ideas will be given about Reinforcement Learning and Robocup.

The German Brainstormers Tribots Robocup[4] team uses reinforcement learning to let the robot learn how to dribble with the ball. Dribbling is an example of a continuing tasks with a continuous state and action set. For this reason, neural networks are used to approximate the action-value-function. Ball control is a process which is difficult to program since it is difficult to determine the best way of moving and taking the ball with you. Figure 22 shows the difference between the hand-coded and the learned dribble behavior.

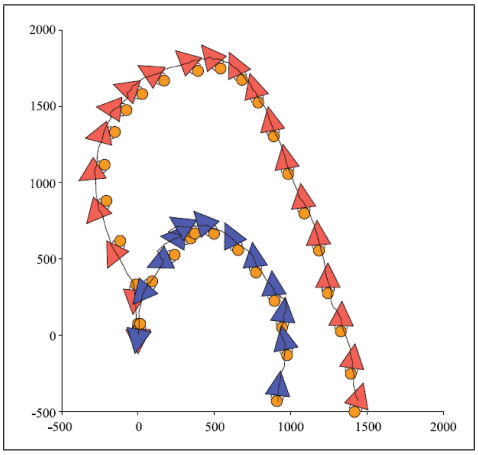

Figure 22: Comparison of hand-coded (red) and learned dribbling behavior (blue)

Another approach of Reinforcement Learning is applying it to a good simulation of the environment and implement the found desired behavior to the real-world. The German team Tribots used this approach for ball interception. The advantage of this approach is that the number of episodes or number of trials isn't important. This makes it possible to find solutions to more difficult problem which requires a lot of trials. The calculation time is the only limiting factor.

The ability to learn from experience instead of a perfect described model of the environment is one of the big advantages of Reinforcement Learning. During a soccer game, it is unknown how the opponent will react/play. The playing style of the opponent can be traced with Reinforcement Learning. For example, the keeper could adapt it's play to the shooting technique of the opponent. An offensive player can also learn which corner of the opponent's goal is the best to aim for.

Tribots also applied Reinforcement Learning at the lowest level, namely

Figure 23: Keepaway soccer simulator

controlling the DC motors. Fast, accurate and reliable motor speed control is a central requirent in many real world applications.

Reinforcement Learning can also be applied at high level. It is difficult to program a complex decision such as to decide whether to pass or dribble. The university of Texas[6] has made a keepaway simulator for the simulation league (see figure). This simulator contains a number of keepers, players with the task to keep the ball in possession as long as possible, and a number of takers, it is their task to intercept the ball. Examples of actions in this case are: HoldBall (remain stationary while keeping possession of the ball), PassBall(k) (pass the ball to keeper k) and GoToBall().

Recently, the Techunited Robocup team is working on a walking platform with six legs. The way of propelling of such a platform is a difficult process. The movements will among others be dependent on frictions of the ground and the type of ground. It is hard to take these effects into account while programming. The precise desired behavior is unknown. That is why the reinforcement learning technique could mean a lot to the walking platform.

Applying reinforcement learning to Mid-size league Robocup isn't that easy. A lot of difficult challenges have to be solved. In the Mid-size league, the sensors have a lot of noise, so it is difficult to determine a state. Another challenge is the enormous state-space set. In the field are 10 robots, each with its own position, rotation and speed. Big state-sets require difficult approximations and a lot of trials before getting to a desired behavior. All the independent players are learning simultaneously which is still an obscure area of reinforcement learning. At last, the effects of actions have often a long and variable delay. So the basic reinforcement learning theory is in the cases, which have the above mentioned problems, not enough and still requires a lot of research. But in some cases it might be a useful tool.

Appendix I: Neural Networks

The main idea behind a neural network is to create a complex system by gathering simple elements[1]. The simple elements in a neural network are nodes and connections between nodes. The behavior of the whole complex system can't be explained by the behavior of a single element which makes the system emergent. First a biological network will be explained briefly. Next the relation between biological and artificial neural networks (ANN) will be clarified. At least, back-propagation will be explained.

In a biological neural network, the network is formed by neurons. The connections between these neurons are chemical synapses and electrical gap junctions. If the sum of the potentials at the post synaptic membrane is high enough, an action potential will occur which transmits a signal to other neurons. Synapses can transmit inhibitory post synaptic potentials and excitatory post synaptic potentials which will respective decrease and increase the chance for a future action. This dynamic behavior is called synaptic plasticity. This phenomenon makes it able to learn from activity or experience.

An artificial neural network is an abstraction of the complexity of a biological neural network. An artificial neuron consists of inputs (like synapses), which are multiplied by weights and then computed by a mathematical function which determines the activation (a summation for example). Then another function is used to compute the output (a threshold for example). The weights can be positive (excitatory) and negative (inhibitory). The weights can be adjusted by different algorithms to obtain the desired output from the network. The number of different ANN's is enormous because of all the different choices for the functions, topology, learning algorithms etc.

One of the most famous algorithms is back-propagation. Back-propagation first calculates how the error (square of the desired output minus the actual output) depends on the weights. After that the weight can be adjusted by using the method of gradient descent:

$$\triangle w_{ji} = -\eta \frac{\delta error}{\delta w_{ji}} \tag{22}$$

η is a constant which determines the size of the adjustment. Using this algorithm, the error will converge to zero.

The current line of research of this area is among other developing chips that behave like synapses. This can for example be realized with semiconducting polymers which are also used in LEC's. These polymers show a hysteresis effect while applying a potential. But this type of chips are still in its infancy. This research takes place at the Chemistry department on the TU/e

Appendix II: Linear Gradient Descent Method

In linear methods, the value approximate function V_t can be represented as a linear function of the parameter vector $\overrightarrow{\theta_t}$. Corresponding to every state s, there is a column vector of features $\overrightarrow{\phi_s} = (\phi_s(1), \phi_s(2), \ldots, \phi(n))^T$, with the same number of components as vector $\overrightarrow{\theta_t}$. The features can be constructed in several ways e.g. with tile coding, coarse coding, radial basis functions and Kanerva coding. Independent from the construction, the approximate state-value function is given by:

$$V_t(s) = \overrightarrow{\theta_t^T}\overrightarrow{\phi_s^T} = \sum_{i=1}^{n} \theta_t(i)\phi_s(i) \tag{23}$$

Gradient-descent updates are used to update the linear function approximation. The gradient of the approximate value with respect to $\overrightarrow{\theta_t}$ in this case is:

$$\nabla_{\overrightarrow{\theta_t}} V_t(s) = \overrightarrow{\phi_s}$$

The mean squared error (MSE) is equal to:

$$MSE(\overrightarrow{\theta_t}) = \sum_{s \in S} P(s) \left[V^\pi(s) - V_t(s)\right]^2$$

The objective is to decrease this error. This can be done as follows:

$$\begin{aligned}
\overrightarrow{\theta}_{t+1} &= \overrightarrow{\theta_t} - \frac{1}{2}\alpha \left[\nabla_{\overrightarrow{\theta_t}} \left[V^\pi(s) - V_t(s)\right]^2\right] && (24)\\
&= \overrightarrow{\theta_t} + \alpha \left[V^\pi(s) - V_t(s)\right] \nabla_{\overrightarrow{\theta_t}} V_t(s_t) && (25)\\
&= \overrightarrow{\theta_t} + \alpha \left[v_t - V_t(s)\right] \nabla_{\overrightarrow{\theta_t}} V_t(s_t) && (26)
\end{aligned}$$

v_t is a backed-up value as mentioned in the previous chapters.

Algorithms 10 and 11 show how the linear gradient descent method is applied to Sarsa(λ) and Q(λ) respectively.

Algorithm 10 Pseudo-script of linear gradient-descent Sarsa(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
    $\vec{e} = \vec{0}$
    $s, a \leftarrow$ initial state and action of episode
    $\mathcal{F}_a \leftarrow$set of features present in s, a
    Repeat (for each step of episode):
        For all $i \in \mathcal{F}_a$:
            $e(i) \leftarrow e(i) + 1$ (accumulation traces)
            or $e(i) \leftarrow 1$ (replacing traces)
        Take action a, observe reward r and next state s
        $\delta \leftarrow r - \Sigma_{i \in \mathcal{F}_a} \theta(i)$
        With probability $1 - \varepsilon$:
            For all $a \in \mathcal{A}(s)$:
                $\mathcal{F}_a \leftarrow$set of features present in s, a
                $Q_a \leftarrow \Sigma_{i \in \mathcal{F}_a} \theta(i)$
            $a \leftarrow$arg $\max_a Q_a$
        else
            $a \leftarrow$a random action $\in \mathcal{A}(s)$
            $\mathcal{F}_a \leftarrow$set is features present in s, a
            $Q_a \leftarrow \Sigma_{i \in \mathcal{F}_a} \theta(i)$
        $\delta \leftarrow \delta + \gamma Q_a$
        $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
        $\vec{e} \leftarrow \gamma \lambda \vec{e}$
    until s is terminal

Algorithm 11 Pseudo-script of linear gradient descent Watkins' Q(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
    $\vec{e} = \vec{0}$
    $s, a \leftarrow$ initial state and action of episode
    $\mathcal{F}_a \leftarrow$set of features present in s, a
    Repeat (for each step of episode):
        For all $i \in \mathcal{F}_a$:
            $e(i) \leftarrow e(i) + 1$ (accumulation traces)
            or $e(i) \leftarrow 1$ (replacing traces)
        Take action a, observe reward r and next state s
        $\delta \leftarrow r - \Sigma_{i \in \mathcal{F}_a} \theta(i)$
        For all $a \in \mathcal{A}(s)$:
            $\mathcal{F}_a \leftarrow$set of features present in s, a
            $Q_a \leftarrow \Sigma_{i \in \mathcal{F}_a} \theta(i)$
        $\delta \leftarrow \delta + \gamma Q_a$
        $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
        With probability $1 - \varepsilon$:
            For all $a \in \mathcal{A}(s)$:
                $Q_a \leftarrow \Sigma_{i \in \mathcal{F}_a} \theta(i)$
            $a \leftarrow$arg max$_a Q_a$
            $\vec{e} \leftarrow \gamma \lambda \vec{e}$
        else
            $a \leftarrow$a random action $\in \mathcal{A}(s)$
            $\vec{e} = 0$
    until s is terminal

Appendix III: Manual of the maze program

The maze program is programmed in Java. The program was originally built by Vivek Mehta and Rohit Kelkar under supervision of Prof. Andrew W. Moore[2] at the Robotics Institute of the Carnegie Mellon University in Pittsburgh. For this report, the program is exented among others with eligibility trace, the Dyna-Q algorithm and the Monte Carlo method. To run the program, Java Runtime Environment (JRE) 1.4.2 or above must be installed which is available on the website of Java Sun. The program can be started by running the file *"maze.jar"*. To recompile, adjust and execute the source code, the Java Development Kit (JDK) and an editor such as JCreator LE must be installed. The main program is stored in the file *"MainUI.java"*.

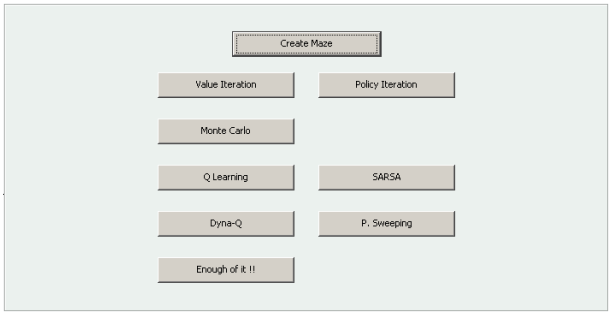

Figure 24: Main screen of the maze program.

Appendix IV:

The mountain car problem is written in MatLab. It is an extended version of a program originally written by Jose Antonio Martin H. PhD[3] of the Universidad Complutense de Madrid. The program is extended with an eligibility trace which increased the performance considerably. To run the program, open *"Demo.m"* and hit F5. The maximum number of steps per episode is 1000. In the file *"MountainCarDemo.m"*, parameters like $\alpha, \gamma, \varepsilon$ and λ can be set.

[2]Website: http://www.cs.cmu.edu/∼awm/rlsim/
[3]Website: http://www.dia.fi.upm.es/∼jamartin/

List of Algorithms

List of Figures

Index

References

[1] *Neural Networks - A Systematic Introduction*. Springer-Verlag.

[2] A.G. Barto and R.S. Sutton. *Reinforcement Learning: An introduction*. MIT press, 1998.

[3] J. Peng and R.J. Williams. Incremental multi-step q-learning. *Proceedings of the Eleventh International Conference on Machine Learning*, pages 226–232, 1994.

[4] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Springer Science*, 27:55–73, 2009.

[5] S.P. Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1), 1996.

[6] P. Stone, R.S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[7] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1989.