FREDERIK HØJLUND & KENNETH RAVNSHØJ MADSEN

Simulation Framework for Supervised Reinforcement Learning

SOFTWARE ENGINEERING 2012 AALBORG UNIVERSITY

Aalborg University
Department of Computer Science

Title:
Simulation Framework for Supervised Reinforcement Learning

Topic:
Machine Intelligence - SW10

Project Period:
February 1st, 2012 - June 8th, 2012

Project Group: sw109f12
Frederik Højlund
Kenneth Ravnshøj Madsen

Supervisor:
Paolo Viappiani

Number of appendices: 1
Total number of pages: 65
Number of pages in report: 62
Number of reports printed: 4

Abstract:

The goal for this project was to gain knowledge of MDP based reinforcement learning techniques in relation to implementation in real world learning environments. A robot framework developed in the previous semester served as a motivation for this. Dynamic programming, Monte Carlo and temporal difference methods were explored as well as generalisation techniques and supervised learning theory.

We have developed a learning framework that incorporates this theory and conducted learning experiments with a toy-like cat and mouse game. The experiments showed that it is possible to speed up learning by using a supervisor and use radial basis functions to approximate value functions and to allow generalisation. The supervisor was shown to be useful even if the supervisor and the trained agent were trained in different environments.

PREFACE

This report is the result of Software Engineering group sw109f12's 10th semester project at the Department of Computer Science, Aalborg University. The report is documentation for the project, which proceeded in the period from 1st of February 2012 until 8th of June 2012.
The topic for this semester is reinforcement learning and its applicability for the robot framework, developed in the previous semester. We explore reinforcement learning theory, while developing a learning framework based on a simulator. The simulator is used to conduct several experiments that shows generalisation techniques and supervised learning in practice.

The reader is expected to have understanding of programming corresponding to a student that has completed the 9th semester of Software Engineering, as well as basic knowledge about probability theory, Markov decision processes and reinforcement learning.

Unless otherwise noted, this report uses the following conventions:

- Cites and references to sources will be denoted by square brackets containing the authors' surname, and the year of publishing. The references each corresponds to an entry in the bibliography on page 63.
- Abbreviations will be represented in their extended form the first time they appear.
- When a person is mentioned as *he* in the report, it refers to *he/she*.

Throughout the report, the following typographical conventions will be used:

- References to classes, variables and functions in code listings are made in `monospace` font.
- Omitted unrelated code is shown as "..." in the code examples.
- Lines broken down in two are denoted by a ↵.

The code examples in the report are not expected to compile out of context.

Appendices are located at the end of the report. The source code for the software project, referenced papers as well as the report in PDF are included on the attached CD-ROM on page 67.

CONTENTS

CHAPTER 1

INTRODUCTION

Learning has always been an important aspect in the lives of humans. The ability to acquire new knowledge and skills have shaped what we are today. Until recently, being able to learn from interaction and experience as well as build upon what is already known have been a privilege of living beings.

Today, machines are able to use computational approaches in order to learn based on interaction. Reinforcement learning is a goal driven method for learning from interaction. Rather than telling the machine or agent specifically what to do, it is told what to achieve. Through rewards, goals can be defined for the agent, and through maximising the amount of rewards received the agent can learn how to achieve the goal.

What the agent learns is a policy that can be used for sequential decision making. The agent is presented with information about the current configuration of its environment and is required to select an action to carry out. The policy provides this action and provides a mechanism for the agent to store learned knowledge in.

In this project we will explore different reinforcement learning methods for training agents that interacts with an environment. In the autumn semester of 2011, we developed a robot framework that uses a Microsoft Kinect® as a positioning device. We wish to explore how reinforcement learning techniques could be applied in a real world environment like this.

1.1 Initial Idea

This section describes the initial idea of the project as well as the result of the preliminary work performed in the project of the previous semester.

As a preparation for this master thesis, we spent the previous semester on developing a robot framework[Højlund and Madsen, 2012]. The idea was to construct a real world robot environment and examine how to apply machine learning techniques to this. Decision making under uncertainty makes sense in an environment like this where the robots are subject to uncertainty in relation to how they move. An overview of the physical platform is shown in figure 1.1.

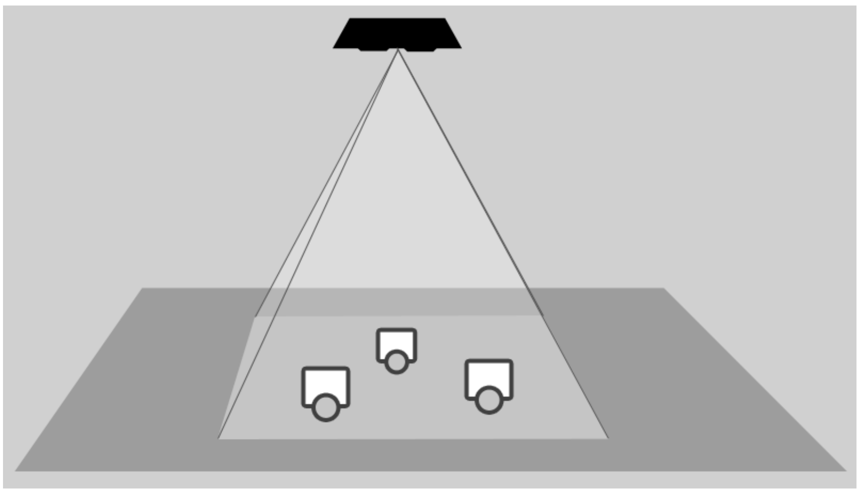


Figure 1.1: Overview of the physical Kinect platform.

The physical platform consists of a Microsoft Kinect® mounted on the ceiling and a number of LEGO® robots that are located on the floor and detectable by the Kinect. The positions of the robots are monitored by the Kinect by using its depth-map capabilities. To control the robots, we developed a framework that is able to make the robots move in any direction and any distance within the viewing field of the Kinect. The viewing field of the Kinect is approximately 2 × 1.5 meters and measures this area with a resolution of 320 × 240 pixels. In theory, this framework is able to detect a robot in 76800 different positions. Since the robots are moving on a slippery surface and the LEGO motors have backlash in their internal gears, the movements of the robots are subject to uncertainty.

During the previous semester we studied Markov decision processes(MDPs) and partially observable MDPs as well as how to use them for decision making under uncertainty. It was examined how MDPs and POMDPs can be solved using dynamic programming techniques such as value iteration including different approximation techniques. The idea was to try different scenarios with the framework, in order to show its capabilities and using the studied theory. However, value iteration is not the only way to solve decision making tasks and for this project the focus is on reinforcement learning. One of the disadvantages of using value iteration is that the complete model of the environment must be known in order to obtain a policy. In our example, the exact probabilities of how the LEGO robots move must be obtained. Reinforcement learning does not have this requirement which is why it is thought to have potential for application in real world environments where it often is intractable to obtain a complete model.

Application of reinforcement learning in a real world environment like the robot framework serves as motivation for exploring reinforcement learning in this project. The fundamental techniques and algorithms are examined as well as the importance of correct tuning of learning parameters. To experiment with the algorithms we develop a learning framework that is able to train agents of a simple cat and mouse game.

Because a real world environment have certain limitations in relation to doing online reinforcement learning a simulator can be very useful. In the real world environment actions take some time to execute since the robots move slowly around on the floor. Because the agent might need to train for thousands of episodes, reinforcement learning can be very time consuming. Other practical issues like battery power and general wear are also factors that can have a negative impact. However it is required to simulate the environment in a realistic manner if a policy obtained on a simulator is to be used in the real world. In many cases it is possible to create a model that to some degree approximates the real probabilistic dynamics of the environment. An advantage of simulation is that it is possible to train for a large number of episodes in a small amount of time. Furthermore, using simulations can reduce the cost of and the risks involved with training in a real world environment.

Even though a simulated model could be used to do value iteration it makes more sense to use reinforcement learning because of the direct interaction with the environment. Examples of interaction serves as the foundation of reinforcement learning where agents bases its behaviour on its own experience. Another notion is that an agent can train using a simulator initially and then use this knowledge to interact with the real world and refine its policy to include the real probabilistic dynamics of the environment. We show that this is possible in experiments using a supervised learning algorithm where the agent can draw on the knowledge of the supervisor. Even though the supervisor have trained in a different environment it can still guide the agent and improve online learning performance.

Large state spaces and the requirement to train many episodes can be a problem in relation to doing reinforcement learning in real world environments. To address this, generalisation can be used. Standard reinforcement learning utilise lookup tables that increase in size with the state space. Using function approximation instead makes it possible to greatly compress the representation of the value function. This makes it possible to generalise between states that are similar. Since states share the same function approximation it is possible to use experience learned in one state to determine the value of other states.

1.2 Approach

We review some fundamental concepts of reinforcement learning useful for our learning framework. This includes dynamic programming(DP), Monte Carlo and temporal difference(TD) methods. These methods lay the groundwork for the Q-learning [Watkins, 1989] and Sarsa [Rummery and Niranjan, 1994] algorithms as well as a supervised Q-learning algorithm [Rosenstein and Barto, 2004]. In our project, a supervised learning algorithm uses a supervisor that the learning agent can use as tutor in order to learn both from its own but also from the supervisors experience. Furthermore, generalisation theory is examined and covers function approximation using tile coding and radial basis functions. Methods for using a supervisor in a reinforcement learning algorithm is examined as well.

This theory will lay the groundwork for developing a learning framework. This framework will consist of a simulator and learning algorithms to provide the basis for conducting experiments with reinforcement learning. The simulator is intended to be flexible in order to simulate different environments. This includes, for example, easy swapping of maps and the possibility to change the probability dynamics.

The learning framework will be used to conduct experiments using the reinforcement learning theory. Value function approximation is used to compute the state values in few episodes for a simple game using radial basis functions. This underpins the idea that function approximation makes it possible to obtain good enough value functions using a lot less episodes than standard lookup table algorithms. Multiple experiments with supervised learning is carried out in order to show that learning agents can speed up their learning rates and improve online learning performance using a supervisor. Even a supervisor with a suboptimal policy or a policy obtained in an different environment can be useful for the learning agent.

1.3 The Cat and Mouse Game

Throughout the project we have used the cat and mouse game as the testing ground for reinforcement learning. The inspiration for the cat and mouse game is taken from [Eden et al., 2012]. Even though the project only uses this game, the algorithms and theory is not specific or limited to this game.

The cat and mouse game consists of a mouse, a cat and a piece of cheese in a gridbased area. Given randomly starting positions, the objective of the mouse is to collect pieces of cheese while trying to avoid being eaten by the cat. The mouse collects the cheese when it moves into the position of the cheese. If a mouse collects a piece of cheese, it gets a positive reward and the cheese is randomly positioned elsewhere in the confined area. The cat eats the mouse when they are at the same position, which also ends the game. If the mouse is eaten it gets a negative reward.

We measure the performance of the mouse by using equation 1.1.

$$\text{Mouse Performance Ratio} = \frac{\text{Deaths}}{\text{Collected Cheese}} \tag{1.1}$$

From the equation it can be seen that the lower the ratio, the better the mouse is doing.

The state of the game is defined as the combination of the positions of the mouse, cat and cheese, as shown in Equation 1.2.

$$\text{State} = \left\{Mouse_x, Mouse_y, Cat_x, Cat_y, Cheese_x, Cheese_y\right\} \tag{1.2}$$

In order to move around in the area, both the mouse and the cat can choose an action that moves them in one of the following directions: North, Northeast, East, Southeast, South, Southwest, West or Northwest. The actions is shown in figure 1.2.

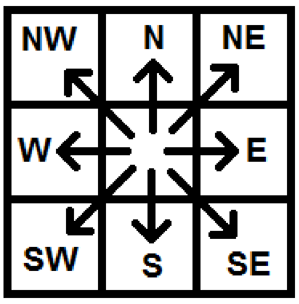


Figure 1.2: Overview of the 8 actions the agents can take.

The area in which the agents move is a $n \times m$ grid and may contain special cells, such as obstacles and teleports. It is not possible for the agents to move through an obstacle, so if they try, they will end up in the same position as where they began the move. Also the agents cannot move out of the grid. The mouse is the only agent that can use the teleports, where the agent is transported from an entrance to an exit.

The game is sequential, so the mouse moves first, and then the cat. This means that the cat can base its action on what the mouse did. The cat is scripted to go directly towards the mouse if possible. If the direct action towards the mouse is into an obstacle, the cat chooses a random action instead. This is done in order to make it harder for the mouse to predict where the cat is going and thereby make the game more challenging.

In a realistic setting, an agent has to face a degree of uncertainty in the resulting outcome of an action. This is modelled by a transition function, that maps triples of $(state, action, newstate)$ to probalities. This transition function is unknown for the agent.

We use two versions of the game, one with and one without uncertainty. In order to show how uncertainty can be modelled in a simulator we define a probabilistic transition function with the following probabilites: The agent will with probability 0.8 move in the selected direction, with 0.1 move to the right and with 0.1 move to the left in relation to the selected direction. Two examples of these uncertainties are shown in figure 1.3, which concerns the North and Northeast actions. Figure 1.3(a) shows that when taking the North action, there is a probability of 0.8 that the agent will actually go North, whereas with probability 0.1 go Northeast and with probability 0.1 go Northwest. Figure 1.3(b) shows what can happen when the Northeast action is taken.

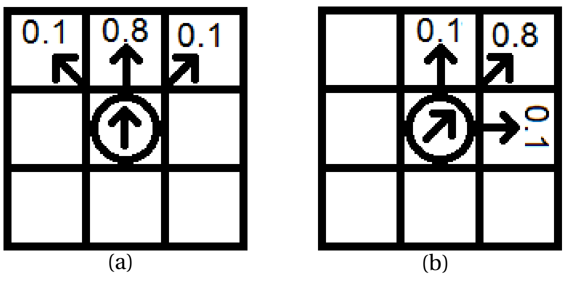


Figure 1.3: Example of the uncertainties in the movement. 1.3(a) shows the North action and 1.3(b) shows the Northeast action.

1.3.1 Maps

The first map, *Map1*, is a 5 × 5 grid with three obstacles in the middle as seen in figure 1.4.

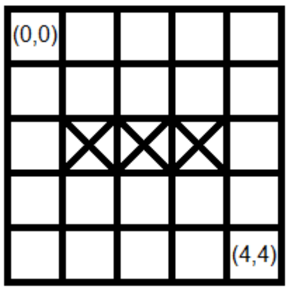


Figure 1.4: Overview of *Map1*.

The grid position (0,0) is at the top left corner, and position (4,4) is at the bottom right corner, as shown in the figure. The obstacles is located in (1,2), (2,2) and (3,2) and are marked with a cross in the figure.

The second map, *Map2*, is based on *Map1*, but with the addition of teleports as seen in figure 1.5.
As seen in the figure, the teleport entrance 1 is marked with “EN1” and has a corresponding exit marked as “EX1”. If the mouse is positioned in (0,0), which is the field marked with “EX4”, and chooses action North, it will use the teleport entrance “EN1”, to get to position (0,4) instead of (0,0) as it would have been, if the same action was performed at (0,0) in *Map1*.

Throughout the experiments these maps are used in combination with the different probabilites for the transition function.

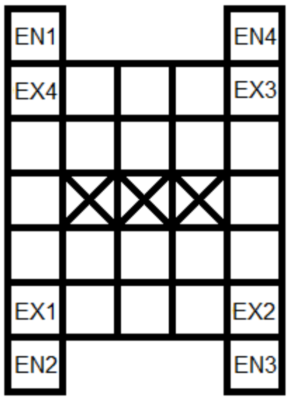


Figure 1.5: Overview of *Map2*.

CHAPTER 2

BACKGROUND

This chapter elaborates on basic concepts of reinforcement learning. This form of machine learning is based on a computationally framework for learning through interaction with an environment. The foundation of reinforcement learning are MDPs. If the problem can be be defined using a MDP, reinforcement learning can be used to solve it. One of the advantages of solving MDPs with reinforcement learning is that the exact dynamics of the MDP does not need to be known.

This chapter first defines MDPs and then elaborates on three methods to solve the reinforcement learning problem: Dynamic programming, Monte Carlo and temporal difference methods. The first method is model-based whereas the two latter are model-free.

Furthermore, since the basic methods does not scale that well when the complexity of the problem rises, generalisation is introduced in order to cope with this.

The idea that a supervisor can be able to speed up the learning of an agent have been supported in related work. We elaborate on some of this work and describe a supervised Q-learning algorithm that uses a supervisor in order to speed up learning.

2.1 Markov Decision Processes

This section describes a framework for sequential decision making: Markov decision processes (MDPs). In this project the MDPs are considered finite which means they consist of the following:

S: a finite set of states.

A: a finite set of actions.

Transition Function: Defined by a probability distribution:

$$P_a(s,s') = P(s_{t+1} = s' | s_t = s, a_t = a), \tag{2.1}$$

which is the probability that taking action a at time t in state s will lead to state s' in the next time step $t+1$.

Reward Function: The expected immediate reward. Given a state s, an action a and a next state s' the reward is defined by a probability distribution given by:

$$R_a(s,s') = E(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'), \tag{2.2}$$

This is a general definition of the reward function. In many cases the rewards are deterministic and simply based on the new state s'.

In some decision making tasks both these probability distributions are fully known. This means that it is possible to model the environment and know exactly how actions affects it. But this is not always possible and many realistic robot tasks are hard to model completely. For example, in a game of chess it is known beforehand exactly how the board will look after a certain move is made. Control of an automated helicopter is a task where it seems impossible to completely model the environment and be able to fully determine the probabilistic outcome of actions.

2.2 Reinforcement Learning

This section covers the basics of reinforcement learning and try to outline the different approaches and discuss them in relation to learning in the cat and mouse game. The source for this section is [Sutton and Barto, 1998] and [Russell and Norvig, 2006].

In short, reinforcement learning concerns agents that learn by interaction with an environment. The agent learns how to make decisions in order to achieve some goal. The agent decides which actions to take based on the states of the environment, and the retrieved rewards support the evaluation of these decisions. The goal for the agent is described using the rewards. How and when the reward function awards or punishes the agent, can be seen as a formulation of the goal. The behavior of the agent is defined by a policy that maps states to actions. It is assumed that the environment can be modelled as a MDP with a finite set of states and actions. The dynamics of the MDP are not always fully known and this has a significant influence in how learning can be done.

When the agent interacts with the environment it is through a sequence of steps $t = 0, 1, 2, ...$ At each step t, the agent gets information about the current state $s \in S$, where S is all possible states of the environment. The agents selects an action $a \in A(s_t)$ at each of these steps, where $A(s_t)$ is the set of all possible actions available in state s_t. When this action is executed a new step is reached: $t + 1$. This means that the agent receives information about the new state s_{t+1} as well as a reward $r_{t+1} \in R$. R is a reward function that returns a numerical value. This framework of agent-environment interaction is shown in figure 2.1.

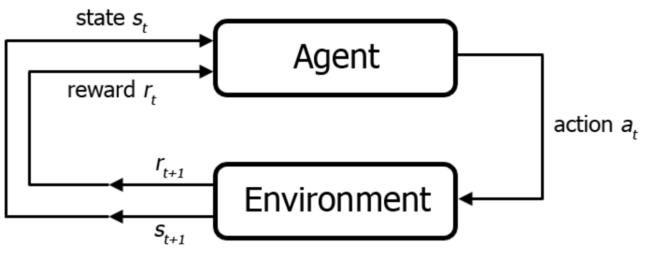


Figure 2.1: Agent-environment interaction (Credit: [Sutton and Barto, 1998]).

When the agent selects actions to execute it uses a policy that maps states to actions. The policy π_t can be seen as a probability $\pi_t(s, a)$ that action a will be selected in state s. A deterministic policy maps each state to exactly one action. This means that one action has probability 1 of being selected whereas the others have probability 0. All MDPs have an optimal deterministic policy.

The goal of reinforcement learning is to develop a policy such that the agent maximises the total received reward. Reinforcement learning methods describes how this policy is changed according to the interactions the agent experiences with the environment.

The total received reward can be seen as a sequence of rewards the agent obtains in a sequentiel interaction with the environment as shown in equation 2.3.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T. \tag{2.3}$$

This sequence accumulates the received rewards and ends with T being the final time step. In many cases this final time step, where the environment is in a terminal state, makes sense in many tasks where reinforcement learning can be applied. These tasks can be called episodic because each sequence ends in a terminal state and the agent-environment interaction can be split naturally into episodes. After each episode the environment is reset to a start state and the interaction restarts. Some tasks can not be naturally split into episodes, and to avoid the return R_t being infinite a discount factor is introduced. In continuing tasks the discount factor ensures that R_t is finite by diminishing the rewards depending on when they are received.
This discounted return is shown in equation 2.4.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.4}$$

The discount factor γ also makes sense for episodic tasks because it can make the agent favor rewards received in the near future over rewards received in the far future. This can be used to encourage the agent to, for example, complete a task sooner instead of later because the rewards are larger when received early. γ is usually a numerical factor from 0 to 1. A low factor will make the agent only focus on immediate rewards, and when γ approaches 1, the agent will take rewards received further in the future more into account.

2.2.1 Reinforcement Learning Algorithms

This section elaborates on how to obtain a policy for reinforcement learning tasks. A dynamic programming based method, a Monte Carlo method and a combination of these called Temporal-Difference learning are presented in this section.

2.2.1.1 Dynamic Programming

First of all, it is assumed that the underlying dynamics of the environment can be described as a MDP with finite state and action sets. Most of the algorithms are based on estimating a value function for either states or state-action pairs. In order to develop and improve policies it is required to define the value of a state or state action pairs given a policy π. For state values it is defined as:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \tag{2.5}$$

This can be understood as the expected total reward received when starting in state s and then follow policy π. The value for state-action pairs is very similar to state-values. A value for each action is defined in each state:

$$Q^{\pi}(s,a) = E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{2.6}$$

These Q values are defined as the expected return when starting in state s, taking action a and then following the policy π afterwards. It is seen that there is a recursive relationship between values of states, where the value of a state is defined by its successor states and so on. According to the transition function, states can lead to multiple states and the expected rewards of these states are weighted by the probability to transition to them. This can be formalised with the Bellman equation for V^{π} where the value of a state is:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P_a(s,s') \left[R_a(s,s') + \gamma V^{\pi}(s') \right] \tag{2.7}$$

The equation first sums all the possible actions that can be taken from the current state. The environment can now transition into multiple states according to the transition function, and yield a reward according to the reward function. The equation sums all possible outcomes with the probabilities that they will occur. The Bellman equation forms the basis for value iteration which is an algorithm that can be used to obtain optimal policies for fully observable finite MDPs. The dynamic programming algorithm works by sweeping over all states of the MDP and updating their values according to the following update rule:

$$V(s) = \max_{a} \sum_{s'} P_a(s,s') \left[R_a(s,s') + \gamma V(s') \right] \tag{2.8}$$

In each sweep, all state values are updated once with this rule. The value of the state is based on the possible next reachable states. When the update is applied the value of the state is updated with the action that yields the best immediate reward. As in the Bellman equation in 2.7 these rewards are weighted by the probability to get them. It can be visualised in a backup diagram as seen in figure 2.2.

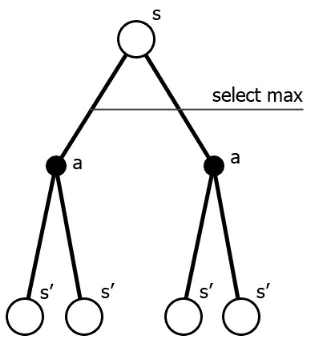


Figure 2.2: Backup diagram for V^* (Credit: [Sutton and Barto, 1998]).

The figure shows how state values are updated in order to converge to the optimal values of the function V^*. In theory, when these values are updated infinitely often they will reach an equilibrium where the values must be the ones of the optimal policy. In practice, the algorithm is terminated when the change of the values is very small as defined by some constant. Listing 2.1 shows the algorithm.

```
function ValueIterationAlgorithm(){
  foreach state s in S:
     $V(s) = r_{min}$
  repeat until convergence:
        foreach state s:
            $V(s) = \max_a \sum_{s'} P_a(s,s')\left[R_a(s,s') + \gamma V(s')\right]$
    return $V$
}
function Policy(state s, valuefunction V){
    return $\arg\max \sum_{s'} P_a(s,s')\left[R_a(s,s') + \gamma V(s')\right]$
}
```

Listing 2.1: Pseudo Code for the Value Iteration algorithm.

When the optimal value function have been computed it is simple to derive the optimal policy. It is a matter of selecting the action that yields the highest expected reward in the current state. Value iteration is quite efficient at computing optimal value functions that can be used to construct optimal policies, but it has limited applicability. First of all, the transition and reward function must be known in order to execute the Bellman updates. In some task these functions are hard to obtain and sometimes it might not be possible at all. Especially the transition function is problematic because it requires exact information about how the environment behaves to be known. Large state spaces are a general problem in machine intelligence, but since this algorithm sweeps over the entire state space each iteration, it performs really poor on tasks with large state spaces.

2.2.1.2 Monte Carlo methods

This section elaborates on how an agent can learn a policy when the exact workings of the environment is unknown. When the transition function is not available the agent can use experience instead. This experience is gained in direct interaction with the environment.

Monte Carlo methods require only a sample set consisting of a sequence of states, actions and rewards. This works when the task is episodic and it is possible to record an agents interaction from the start state to a terminal state. This record of a full episode of interaction is used to update a policy for the agent.

Let us consider how a Monte Carlo method can be used to estimate a value function for a policy when the model of the environment is unknown. If we want to estimate $V^{\pi}(s)$ and we have a number of episode records, the idea is to simply average the returns observed in the episodes after visits to the state s. When the state s is first visited in a episode, the return is saved and averaged over other first-visit returns of the same state in other episodes. The pseudo code for this Monte Carlo first-visit method is shown in listing 2.2.

```
function MCFirstVisitEvaluation(Policy $\pi$){
    $V$ = an arbitrary state-value function
    Returns(s) = an empty list for all states s
    repeat until convergence:
        Generate an episode using $\pi$
        For each state in the episode
```

```
            $R$ = return after first visit to s
            Returns(s).Append($R$)
            $V(s)$ = Returns(s).Average()
    return $V(s)$
}
```

Listing 2.2: Pseudo Code for the Monte Carlo first-visit method.

Each of the returns can be seen as an estimate of $V^{\pi}(s)$ and when the number of recorded first visits goes towards infinity the value function will converge to the true $V^{\pi}(s)$.

However, in tasks where the model is unknown, a value function for states cannot be used to derive a policy. In order to be able to select actions when given a state, the agent will need a value for each action. These pairs of states and actions are contained in the $Q^{\pi}(s,a)$ value function. So instead of averaging returns following a state, returns following a state-action pair is averaged. It is however required that all state-action pairs are visited in order for the function to converge. If the policy used to generate episodes is deterministic and always selects the same action given a state, many state action pairs will never be visited. In order to be able to learn the optimal value function it is required to explore, which means trying state-action pairs that are not currently favored by the policy. The balance between exploring untried pairs and exploiting favored pairs is an important topic in reinforcement learning. The solution in the following section is an assumption called exploring starts. Here, the idea is to select a random state-action pair as the starting point for each episode, where all pairs must have a nonzero chance of being selected. Because this is not useful for all tasks, an alternative is to consider stochastic policies that with some probability will select a random action in a state.

Now we consider using a Monte Carlo method for developing a policy based on interaction with the environment. The algorithm aims to estimate the $Q^{\pi}(s,a)$ and update the policy accordingly. The pseudo code in listing 2.3 shows the algorithm. It is similar to the policy evaluation algorithm in listing 2.2, but it is based on state-action pairs and it modifies the policy if a pair turns out to give a better return.

```
function MCControlExploringStarts(){
    $Q(s,a)$ = an arbitrary state-action pair value function
    $\pi(s)$ = an arbitrary policy
    Returns(s,a) = an empty list for all pairs
    repeat until convergence:
        Generate an episode using $\pi$ with exploring starts
        For each state-action pair in the episode
            $R$ = return after first visit to s,a
            Returns(s).Append($R$)
            $Q(s,a)$ = Returns(s,a).Average()
        For each s in the episode
            $\pi(s) \leftarrow \arg\max_a Q(s,a)$
    return $\pi(s)$
}
```

Listing 2.3: Pseudo Code for the Monte Carlo Control Algorithm.

For the algorithm to work it is assumed that infinite many episodes are used as well as exploring starts. After each episode the returns for each state-action pair is appended and averaged making the values asymptotically approach the real $Q(s,a)$ values. Then the policy is updated accordingly, where the state-action pair with the highest return is selected as the best action in a state.

The assumption about exploring starts is however unrealistic in many tasks. Starting in every state might be problematic or impossible depending on the nature of the task. The assumption can be avoided however if the algorithm evaluates and improves policies that are soft, meaning that the probability of selecting an action is higher than zero for all possible actions available in a state. ϵ-greedy policies assign probabilities to actions in the following way: The actions that not currently are the best action are given probability $\frac{\epsilon}{|A(s)|}$ of being selected. The best action is assigned $1-\epsilon+\frac{\epsilon}{|A(s)|}$. ϵ is commonly a small nonzero number. These policies will select the best action most of the time, but with probability ϵ they will select a completely random action from the set of currently available actions.

An advantage of Monte Carlo methods over dynamic programming approaches is that it is possible to learn even when the exact workings of the environment are unknown. The episode records used for training can be generated or collected without using the exact model as required by the dynamic programming methods.

2.2.1.3 Temporal Difference

The next method called temporal difference (TD) can be seen as a combination of dynamic programming and Monte Carlo methods. Like the Monte Carlo methods it does not need the model of the environment and can learn based on direct experience. But unlike Monte Carlo the method does not need a full episode of interaction in order to be able to update values for state-action pairs. Temporal difference learning methods can update value estimations based on other estimations instead of waiting for the final return as the Monte Carlo methods does. This can be seen as a way of bootstrapping and can be an advantage because the agent will learn at every step of the interaction instead of just by the end of the episode. If the episodes are long, this is especially convenient.

The basic idea is that at every step t a TD method takes an action in a state and updates that state-action pair based on the reward and the next state. This can be formalised as:

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[r_{t+1} + \gamma V(s_t+1) - V(s_t)\Big] \tag{2.9}$$

Where α is the learning rate and γ is the discount factor. Like the Monte Carlo methods, TD methods uses sample backups and like the DP methods, TD methods bootstrap by basing estimates on estimates. The successor state and the reward is used to calculate the value for a state-action pair the agent has visited. But unlike the Monte Carlo methods, TD methods uses a single transition and does not do full backup like the DP methods. Full backups are when the value of a state-action pair is updated based on all possible successors instead of a sample successor. A simple TD algorithm is shown in listing 2.4.

```
function TDEvaluation(Policy $\pi$){
   $V$ = an arbitrary state-value function
   $\pi(s)$ = policy to be evaluated
   $s$ = state
   repeat for each episode:
      repeat for each step
         select action $a$ using $\pi(s)$
         take action $a$
         receive reward $r$ and next state $s'$
         $V(s) \leftarrow V(s) + \alpha\Big[r + \gamma V(s') - V(s)\Big]$
         $s \leftarrow s'$
}
```

Listing 2.4: Pseudo Code for a simple TD algorithm.

The algorithm is used to evaluate a policy based on the TD method. It estimates state values based on the reward and the value of the successor state. Next, we consider how TD methods can be used for control in order to obtain policies for tasks. The need to balance exploration and exploitation is still essential here. The algorithms utilise ϵ-greedy algorithms and are split into two main classes: on-policy and off-policy. On-policy means that the algorithm is updating the same policy it is using to select actions at each step. An off-policy algorithm updates a different policy than the one it is using to select actions. This is a difference in relation to how exploration affects the learning. On-policy algorithms will update the policy and include the effects of the random exploration based on the ϵ variable. Off-policy algorithms will, even though actions are sometimes selected at random, update a policy based on deterministic action selection.

In the following the on-policy Sarsa algorithm and the off-policy Q-learning algorithm is elaborated. Sarsa learns state-action values $Q^{\pi}(s,a)$ for the policy π for all state-action pairs. During learning the agent will during the episode experience a sequence of states, actions, and rewards. At each time step the agent will observe a state, take an action, receive a reward and observe a new state. When the successor state is observed, the state-action pair the agent chose in the predecessor state will be updated according to the following rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\Big] \tag{2.10}$$

Equation 2.10 is similar to the update rule for TD estimation in equation 2.9 except that it is based on state-action pairs. Now the important part is the $Q(s_{t+1},a_{t+1})$ part of the equation. This is the value of the state-action pair in the successor state. In order for Sarsa to be on-policy the action could be selected using an ϵ-greedy policy. This means that Sarsa will not always update with the value of the most favored state-action pair but simply update with the pair the policy chooses. The outlines of the Sarsa algorithm is shown in listing 2.5.

```
function Sarsa(){
   $Q(s,a)$ = an arbitrary state-action pair value function
   $s$ = state
   repeat for each episode:
   Use the $\epsilon$-greedy policy $\pi$ to select action $a$ in $s$
      repeat for each step
         take action $a$
         receive reward $r$ and next state $s'$
         Use the $\epsilon$-greedy policy $\pi$ to select action $a'$ in $s'$
         $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
         $s \leftarrow s'$
         $a \leftarrow a'$
}
```

Listing 2.5: Pseudo Code for the Sarsa.

a' is selected using the ϵ-greedy policy making Sarsa evaluate and update the same policy it is currently using to interact with the environment. This means that the policy it learns will take the random action selection into consideration and compensate for it. The backup diagram of Sarsa in figure 2.3 shows how the action a is updated using the successor state and the action a' selected by the ϵ-greedy policy.

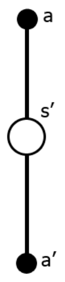


Figure 2.3: Backup diagram for Sarsa (Credit: [Sutton and Barto, 1998]).

This action selection is the major difference with the off-policy Q-learning elaborated in the following.

Q-learning is an off-policy algorithm because it follows one policy but updates another. The update at each step in Q-learning is defined in equation 2.11.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{2.11}$$

Like Sarsa, it is based on state-action pairs, but unlike Sarsa the Q value in the successor state is calculated using the action currently favored by the deterministic policy. This value, $Q(s_{t+1}, a)$, is the Q value of the best action in state s_{t+1} and can not be selected stochastically. So even though Q-learning uses a ϵ-greedy policy to select which actions to explore/exploit, it is developing a deterministic policy. The algorithm is shown in listing 2.6.

```
function Q-learning(){
   $Q(s,a)$ = an arbitrary state-action pair value function
   $s$ = state
   repeat for each episode:
      repeat for each step
         Use the $\epsilon$-greedy policy $\pi$ to select action $a$ in $s$
         take action $a$
         receive reward $r$ and next state $s'$
         $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
         $s \leftarrow s'$
}
```

Listing 2.6: Pseudo Code for Q-learning.

Given enough episodes and transitions, Q-learning will converge to the optimal value function Q^* and have learned a deterministic policy to solve the task. Using the ϵ-greedy policy will guarantee that all state-action pairs are visited and updated. The backup diagram can be seen in figure 2.4.

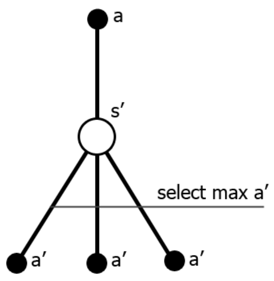


Figure 2.4: Backup diagram for Q-learning (Credit: [Sutton and Barto, 1998]).

The figure shows how the action a is updated using the successor state s' and the maximum of the possible actions a' that can be taken in s'.

The next section elaborates on how these algorithms can be used without the need for large lookup tables.

2.2.2 Generalisation

This section elaborates on how to cope with larger state spaces using generalisation and function approximation. The sources for this section are [Russell and Norvig, 2006] and [Sutton and Barto, 1998].

The Sarsa and Q-learning methods and algorithms described in the previous section all utilise a lookup table for the value function $V(s)$ or $Q(s,a)$. The size of these tables are dependent on the environment. If the environment have many states and many possible actions these tables becomes very large. This is a problem in relation to both memory usage but also in relation to the reinforcement learning itself. A larger table means more values to be estimated and this means training will take longer time.

To address this problem, some kind of generalisation is needed. This means that the decision making in unvisited states will be based on a generalisation of experiences in previously visited similar states. The idea is to use function approximation for the value function instead of a lookup table. Based on example data points, an approximate function can be constructed. This is not necessarily the real value function because it might not be possible to represent the true value function using the chosen form for the approximation function.

As an example, the value of the state s can be represented using a weighted function:

$$V_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s). \tag{2.12}$$

In equation 2.12 the f functions are a set of basis functions. The features can be described by n parameters and this number is supposed to be many times smaller than the total state representation of the value function. The weights denoted by θ are supposed to be learned using reinforcement learning. The strenght is that updating one state will in turn update other states as well. Since the update is done on the function that is "shared" among the states, it is possible to approximate the value of a state that have never been visited before.

To use generalisation, let us first look at estimating a value function $V^\pi(s)$ for some policy π using a gradient descend method. Of course, it all depends on what form of function we use to represent the value function with. An example from [Russell and Norvig, 2006] uses a simple 4 × 3 gridworld example where the features of each gridtile is simply the x and y coordinate.

$$V_\theta(x,y) = \theta_0 + \theta_1 x + \theta_2 y. \tag{2.13}$$

Equation 2.13 shows a simple representation of the state value where the value is obtained by plotting the x and y values into the formula. The weights will be learned using a learning algorithm that updates the weights. For a Monte Carlo algorithm, example records for interaction with the environment can be used to do this. These records consists of a state and the total return or reward obtained starting in that state. For example, an episode starting in (0,0) returns a value that is different than the one currently predicted by the value function. The weights of the value function will need to be updated to incorporate the return of this episode. The value function update of, for example, Q-learning does not translate directly when weights are update instead of state values.

Instead, an update function is defined in order to reduce the error between the existing value and the new return. The error is defined as the squared difference between the two values divided by two. This is done because it is desired to minimise the squared error in relation to doing linear regression. Linear regression is fitting a function to a set of noisy sample values. This error in relation to each of the feature parameters (eg. x and y) is given using partial derivates of the approximation function. The update for a given θ when evaluating the value of $V^{\pi}(s)$ is:

$$\begin{aligned}\theta_i &= \theta_i - \frac{1}{2}\alpha\nabla_{\theta_i}\Big[V^{\pi}(s) - V(s)\Big]^2 \\ &= \theta_i + \alpha\Big[V^{\pi}(s) - V(s)\Big]\nabla_{\theta_i}V(s) \qquad (2.14)\end{aligned}$$

α can be seen as a learning rate that controls how much the weights are shifted to reduce the error. $\nabla_{\theta_i}V(s)$ denotes the partial derivatives:

$$\frac{\partial V_{\theta}(s)}{\partial \theta_i} \qquad (2.15)$$

To apply this to the value function in equation 2.13 will give an update rule for each weight θ. These update rules are listed in equation 2.16.

$$\begin{aligned}\theta_0 &= \theta_0 + \alpha\Big[V^{\pi}(s) - V(s)\Big] \\ \theta_1 &= \theta_1 + \alpha\Big[V^{\pi}(s) - V(s)\Big]x \\ \theta_2 &= \theta_2 + \alpha\Big[V^{\pi}(s) - V(s)\Big]y \qquad (2.16)\end{aligned}$$

2.2.2.1 Coarse Coding

The linear function approximation based on the simple features, x and y in the example, is not very usefull for all learning tasks. Interactions between the features cannot be represented. For example, the value of feature i might be highly dependent on feature j. The value for i might only be high if feature j is present. The idea is to model these interactions by using conjunctions of features.

If we consider a world where the state is given as a two dimensional position it could make sense to use circle features as shown in figure 2.5.
A feature is present if the position is inside the circle. The features are called binary because they are either present or not. In order to generalise between states the idea is that the circles will have a certain overlap. This means that one state can cause multiple features to be present and that different states can share a feature. When updating the weight for a feature, other states that share this feature are affected as well. In figure 2.5, state 1 makes three fea-

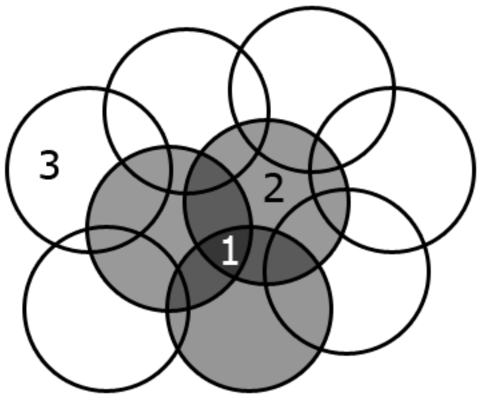


Figure 2.5: Coarse coding features. (Credit: [Sutton and Barto, 1998]).

tures present. State 1 and 2 share one feature which means that there will be some generalisation between these states. State 1 and 3 does not share any features so there will not be any generalisation between these two.

The degree of approximation is highly dependent on the design of these features. More small features will yield a finer approximation than using less but broader features. Of course, the more features, the more computation is needed.

2.2.2.2 Tile Coding

Tile coding is a a kind of coarse coding where the features are defined using tilings. These tilings partion the state space into binary feature tiles. For example in the example with the state given by a 2D position the tiling could simply look like figure 2.6.

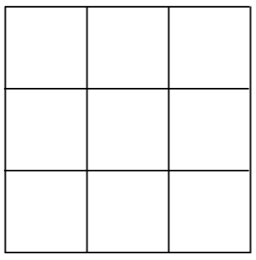


Figure 2.6: A tiling consisting of 9 tiles (Credit: [Sutton and Barto, 1998]).

Like the circles, the feature tiles are present if the position is inside the tile. There is, however, no overlap which is why multiple tilings is used. Different tilings will be offset by different amounts meaning that one position can cause multiple tile features to be present. Figure 2.7 shows how the state 1 causes two tile features to be present.

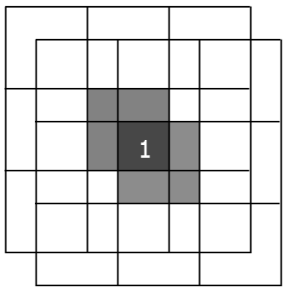


Figure 2.7: State 1 causes two tiles to be present. (Credit: [Sutton and Barto, 1998]).

Again the density of the tile in the tilings determines how accurate the approximation will be.

2.2.2.3 Radial Basis Function

Binary features makes computation simple in the way that only present tiles needs to be updated. Depending on the density, the number of present tiles is much smaller than the total number of tiles. Another approach is to have features that can be present with a certain degree. This makes more advanced generalisation possible. How much a feature is present can be represented by a number between 0 and 1 as a contrast to binary features where this number is either 0 or 1. Radial basis functions(RBF) are functions whose values depend only on the distance from some origin. For example, instead of using the circles where the diameter defines whether the state is present or not, a radial basis function determines the degree to which the feature is present based on the distance to the center of the circle. The closer to the center of the cirle, the more the feature is present. The are different radial basis functions that can be used and a common one is a Gaussian distribution. Equation 2.17 shows a Gaussian function based on the distance between the state s and the center of the feature c_i.

$$\phi_s(i) = exp\Big(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\Big) \tag{2.17}$$

An example of two radial basis functions are show in the graph in figure 2.8.
In this 1 dimensional example, the input variable x will cause these two RBF features to be present with a degree between 0 and 1. Using RBF features makes it possible to approximate more smoothly and might in some cases fit the target value function better than tile coding of features. The downside is that more computation time is needed. At every update most of the features are present, even though they might be to very small degrees. They still need their weights updated based on the state transition and reward received.

Designing good features for a given learning task is an interesting problem. One thing might work for one task and not for another. Since there are no rigid guidlines for how to design features it is a topic where art and science overlaps slightly.

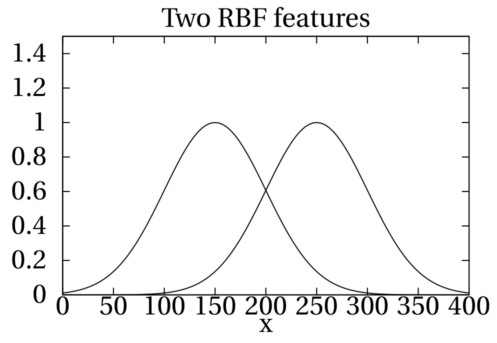


Figure 2.8: Two radial basis function features with centers at 150 and 250. σ is 50.

2.2.3 Supervised Learning

In real world learning tasks, there might be more feedback available to the learner other than simple rewards. As an example, consider a soccer trainer who is teaching a player to perform a kick of the ball into the goal. The player kicks the ball in different ways in order to develop the motor skills required to succesfully hit the goal. Feedback of his actions is provided by visual confirmation of whether the goal was hit or not. He can then adjust his kick based on his observations of the trajectory of the ball. This "reward" will help the player to make corrections to his kick based on his own experience.

The trainer can give more evaluative feedback to the player in a verbal form. As an expert the trainer can also provide the player with pointers to how the kick should be performed. This can be seen as a way for the player to ask the trainer how he would have made the kick. The idea is that the player will learn based on a combination of his own experimentation but also on the feedback and advice provided by the trainer.

Normally, reinforcement learning agents only base their learning on their own observations, so how can this idea of a supervised learning be incorporated into the existing algorithms? In [Rosenstein and Barto, 2004] a framework is suggested for using supervised learning in reinforcement learning algorithms. The supervisor can be seen as another agent that is able to provide actions as well as reward feedback on the actions the learning agent takes. This framework is visualised in figure 2.9 which is similar to [Rosenstein and Barto, 2004].

The figure is similar to figure 2.1 except that the update function of the agent is shown. In the case of Q-learning this update function will be similar to equation 2.11 where the outcome of the taken action is used to update the Q function of the agent. The agent will be able to have the supervisor select which action to take based on the supervisors policy. Additionally the supervisor will aid the updating of the policy by providing a reward. This should help the agent to learn faster because of the expert "advice" of the supervisor.

Like the soccer player, the agent will need to learn from both its own actions and the feedback provided by the supervisor. To achieve this, the agent will need a mechanism to choose whether to select an action on its own or have the supervisor select one. In

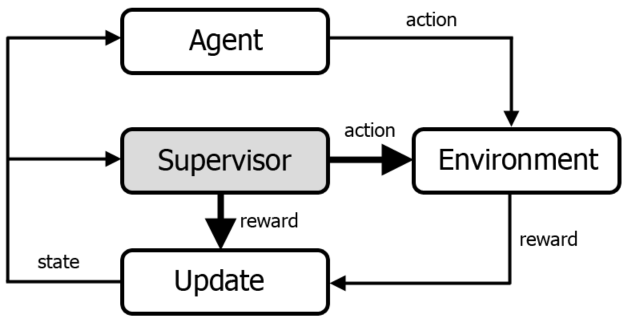


Figure 2.9: Supervised reinforcement learning architecture.

[Rosenstein and Barto, 2004] the action is selected based on a weighted sum of the agents and the supervisors action:

$$a \leftarrow k a^{agent} + (1-k) a^{supervisor} \tag{2.18}$$

Equation 2.18 computes an action based on the k parameter. The only problem is that actions are assumed to be continuous scalar values because the framework in [Rosenstein and Barto, 2004] is designed for continous domains. This action selection does not directly transfer to tasks where there is a finite number of discrete actions. In the cat and mouse game, described in section 1.3, the actions are to go in one of eight directions (north, northeast, east and so on) and k-weighted sum can not be calculated based on these actions. In order to cope with this, it is proposed in [Rosenstein and Barto, 2004] that the k parameter will be seen as a probability to select the agents own action instead of the action adviced by the supervisor. This is not exactly the same as the smooth mixing between the scalar actions of equation 2.18, but since the action selection will be done repeatedly, the probabilistic selection is a good approximation.

The update rule is also weigthed by the k parameter in the following way:

$$w \leftarrow w + k w^{RL} + (1-k) w^{SL} \tag{2.19}$$

In [Rosenstein and Barto, 2004] w is a parameter vector for the parameterised policy function π. This can be translated into Q values for a discrete domain. Equation 2.19 is used to calculate the update as a k-weighted value of the normal reinforcement learning update of the agent kw^{RL} and the supervised learning update w^{SL} from the supervisor. This works for discrete domains as well, because these updates are done on numerical rewards and state values. However, in order to update with the w^{SL} value, the supervisor will need to supply state values that are in the same format as the agents own.

In [Jensen et al., 2010] the update equation is further developed based on equation 2.19. This so called combined target is defined as:

$$ct_t = k(s_t)(r_{t+1} + V(s_{t+1})) + (1 - k(s_t))(o_{sup}(s_t)) \tag{2.20}$$

This update reflects the update equations from [Rosenstein and Barto, 2004] and works for discrete domains by using a value function $V(s)$. This replaces $kw^{RL} + (1-k)w^{SL}$ in equation 2.19. $o_{sup}(s_t)$ is the output of the supervisor that should be in the same format as the agents own. Each time the agent updates, the supervisor provides its state value which can be seen as a reward received from the supervisor.

The k parameter can be used to control how the agent learns from the supervisor. If the agent can control k it can lower the value indicating that it needs advice from the supervisor. The supervisor could be able to completely take control of the action selection if the supervisor sets $k = 0$. This could be powerful in online non-simulated learning tasks where it would be possible for the supervisor to make the agent avoid taking very risky actions. The k parameter could even be controlled by a human that could assess whether the agent is doing fine or needs help from the supervisor.

Another approach is to have a k value for each state which is actually the case in equation 2.20. When a state is visited the k value could be raised to indicate that the agent needs less advice from the supervisor in that state. When a state have been visited many times, the k value will approach zero meaning that the agent will only take its own action and update on the rewards received from the environment. The idea is that the supervisor will improve the learning speed in the beginning and then gradually stop interfering and make the agent learn on its own.

Modification of the Q-learning algorithm (listing 2.6), with the combined target update of equation 2.20 and the supervised action selection is elaborated in the following section. An ϵ-greedy action selection is still performed but with probability k the agent will take the action provided by the supervisor. The algorithm is based on [Rosenstein and Barto, 2004] and uses the update from [Jensen et al., 2010]. This algorithm updates a value function based on state-action pairs $Q(s, a)$ instead of state values $V(s)$. Listing 2.7 shows the algorithm.

```
function Supervised Q-learning(){
  $Q(s,a)$ = an arbitrary state-action pair value function
  $s$ = state
  $k(s)$ = k parameter for states
  repeat for each episode:
    repeat for each step
    With probability k:
      Use the $\epsilon$-greedy policy $\pi$ to select action $a$ in $s$
    Else
      Get action $a$ from supervisor
    take action $a$
    receive reward $r$ and next state $s'$
    receive $Q_{sup}(s,a)$ value from supervisor
    $ct_t = k(s)(r + \gamma \max_{a'} Q(s',a')) + (1-k(s))(Q_{sup}(s,a))$
    $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[ct_t\right]$
    $s \leftarrow s'$
```

```
        update $k(s)$ to indicate a visit to $s$
}
```

Listing 2.7: Pseudo Code for Supervised Q-learning.

2.3 Summary

This chapter presented the reinforcement learning theory that lays the groundwork for this project. MDPs are the foundation of reinforcement learning and serves as a framework for incorporating all the necessary state and actions as well as the transition and reward function. It was defined how rewards is defining how an agent will behave while trying to maximise the received reward.

Dynamic programming is a simple method used to derive a policy π when the dynamics of the MDP is completely known. The value iteration algorithm is used to continuously sweep over the states and update the state values in order to compute the optimal value function.

When the exact dynamics of the environment is unknown, the agent could use sampled episodic experience with the environment instead. In Monte Carlo methods, the agent is given a number of episodes that is used to calculate the average return for states. Given enough episodes the value function for states can be obtained.

In order to use a policy when the environment dynamics are unknown, a value function based on state-action pairs is used instead. The returns received when following an action in a state are averaged to calculate the value of the state-action pair.

To avoid the assumption of exploring starts, soft policies was introduced. ϵ-greedy policies have a probability of selecting a random action in any given state.

Temporal difference methods are a combination of dynamic programming and Monte Carlo methods. Instead of updating the value function after each episode, it is updated after each action. TD methods use sample backups like Monte Carlo methods but bootstraps like the dynamic programming methods.

Generalisation was introduced in order to cope with large state space and long computation time. Coarse and tile coding as well as radial basis functions could be used to generalise between similar states.

The idea of supervised reinforcement learning was introduced and a learning algorithm that utilised the policy of a supervisor in order to speed up learning was presented.

Chapter 3
Analysis

This chapter presents the analysis of the learning framework, to examine how the framework should be developed in order to suit our needs.

We aim to develop a framework that supports implementation of the different reinforcement learning algorithms and allows for easy tuning of the learning parameters. In order to be able to do learning experiments we desire a flexible system that is able to log progress during training. To be able to verify that the mouse is able to learn we wish to have visual representation of the cat and mouse game in progress.

3.1 Use Case

The main functionality of the framework must be identified. We are planning to use the framework for training and playing with the agent. The use case diagram is shown in figure 3.1.

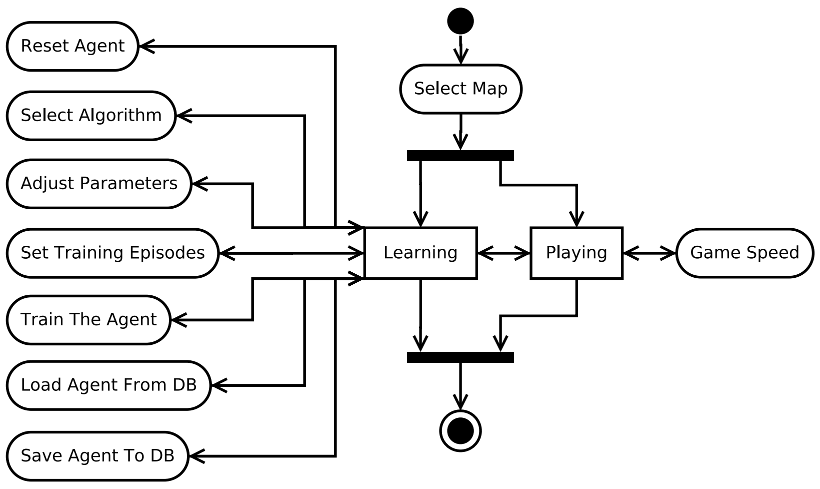


Figure 3.1: The use case diagram of the learning framework.

As seen in the diagram, the usage of the application is limited to one specific map or environment at a time. To change the environment, the application must be restarted.

The "Learning" and "Playing" boxes are states, and the rest are activities. For example, if the application are in the "Learning" state, the "Select Algorithm" activity can be performed, which will set the algorithm used for the training of the agent.

3.2 Architecture

This section describes the analysis of the architecture of the learning framework.

As seen in the Use Case diagram, see figure 3.1, there are two main parts in the application: the Learning part and the Playing part. These two parts are sharing some central functionality in the framework. For example, the training of an agent creates a policy, that is used when playing the game. This means that the policy must be accessible for both the learning part and the playing part.

In order to find the shared functionality, the different algorithms was examined to find out what they need from the framework. This lead to a number of interactions between the parts of the application. These interactions are illustrated in figure 3.2.
The figure shows that the policy and the environment is shared between the two parts.

The architecture should reflect that the main usage of the program is learning and playing with the agent. In the learning part it should be possible to adjust the parameters for the algorithms, so this means that the learning part provides a user interface where the parameters

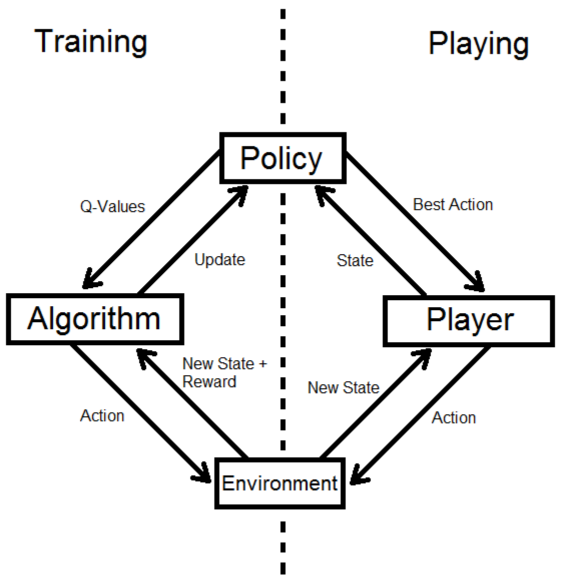


Figure 3.2: The interactions in the learning framework.

can be adjusted. When playing the game, it should be possible to see how the mouse and cat interacts in an graphical user interface.

In order to design the learning framework to support these requirements, two possible solutions are elaborated below.

3.2.1 Multilayered architecture

The multilayered architecture[Microsoft, 2012] separates similar parts of the application into layers, as shown in figure 3.3.

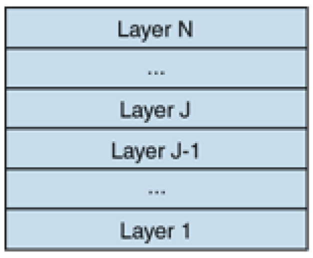


Figure 3.3: The multilayered architecture with N layers (Credit: [Microsoft, 2012]).

As seen in the figure an application can consists of many layers, with each layer having a specific grouping of functionality. A layer has a upwards and downwards interface, meaning that layer J only interact with layer $J+1$ and $J-1$. Typical the number of layers is three, and are separated in a presentation layer, a business logic layer and a data access layer.

Figure 3.4 shows that layers can also contain subsystems or components.

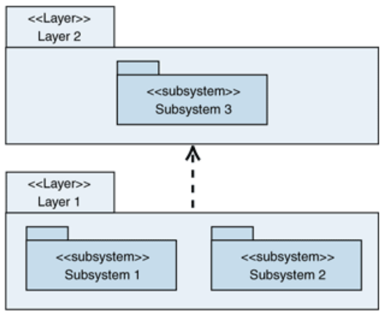


Figure 3.4: The multilayered architecture with subsystems or components (Credit: [Microsoft, 2012]).

The advantage of using a multilayered architecture is that it offers a low coupling between the different components and creates high cohesion, meaning that related functionality is grouped together.

3.2.2 Model-View-Controller Architectural Pattern

The Model-View-Controller (MVC)[Fowler, 2012] pattern is an architectural pattern and divides a software application into three parts:

Model contains business logic and the code for accessing the data needed by the program.

View is the user interface of the application. A view will be rendered on request from the controller and can be changed based on data from the model.

Controller handles the user interaction and modifies the model state.

An example of MVC pattern can be seen in figure 3.5.
When an event occurs, the event is passed to the controller. The controller will trigger the rendering of views or changes of the model state, and if necessary the views receive data from the model that is used to render the proper view. The MVC pattern separates the three aspects of the application which helps reducing the complexity of the system. Another advantage is that the three components can also be tested separately because of the loose coupling between the components.

3.2.3 Choice of Architecture

For the learning framework we have chosen to use the multilayered architecture, as it is suitable for our needs. The MVC pattern is more suitable for larger systems, with multiple users, as it separates the actual data from the users. As this is a learning framework prototype, using the multilayered architecture provides the sufficient basis to construct the application.

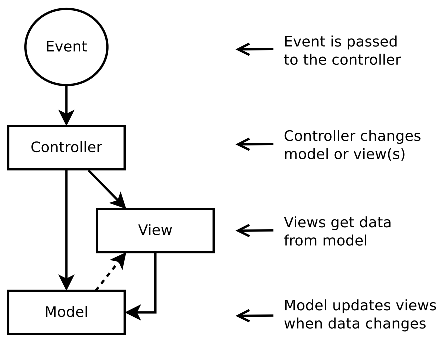

Event
Controller
View
Model
Event is passed to the controller
Controller changes model or view(s)
Views get data from model
Model updates views when data changes

Figure 3.5: The MVC architectural pattern.

3.3 Summary

In this chapter we have analysed the important parts of the learning framework in order to make choices on how the framework should be constructed. The expected use of the framework were elaborated by constructing a use case diagram. Lastly the architecture of the framework was determined to be based on the multilayered architecture.

Chapter 4

Design

This chapter elaborates on the design of the learning framework based on the analysis. It presents the design of the chosen architecture, as well as elaborates on the design of the partial user interfaces for the Learning and Playing layers. The chapter also describes the design of the Algorithm component, that aims for easy implementation of additional algorithms in the learning framework.

4.1 Architecture

This section elaborates the design of the architecture of the learning framework. The design of the architecture for the learning framework is shown in figure 4.1.

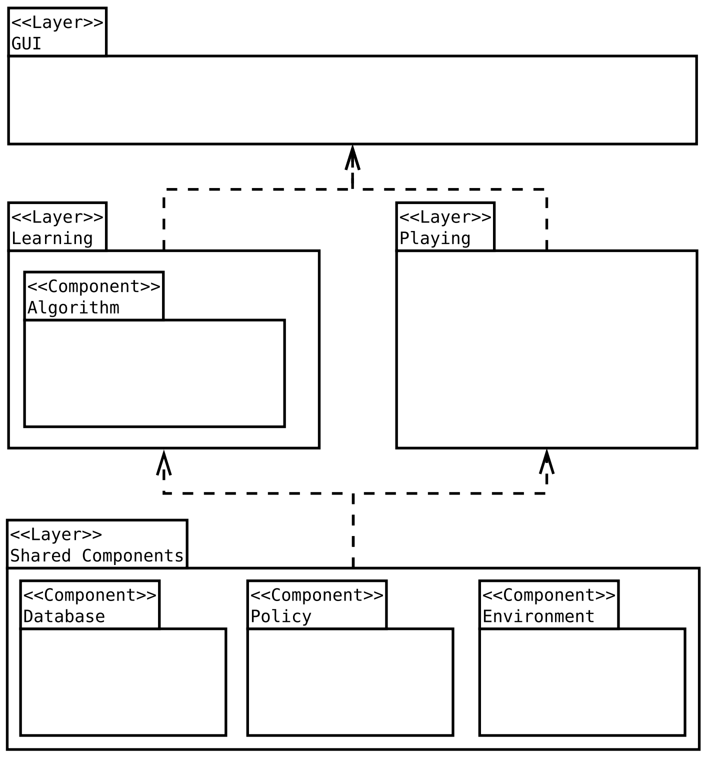


Figure 4.1: The architecture of the learning framework.

The architecture is based on a multilayered architecture with a presentation layer at the top, a logic layer in the middle and instead of a pure data layer in the bottom, we have a layer that contains the shared components including the database component. The remainder of this section briefly describes the purpose of each layer and component.

4.1.1 GUI Layer

The GUI layer will provide the overall user interface, as well as make it possible to incorporate user interfaces from other layers, to for example enable the Learning layer to provide a settings interface, that can be used for adjusting the current algorithm.

4.1.2 Learning Layer

The Learning layer offers a user interface, where the user can choose between which algorithm to use in the training, as well as adjusting the different rewards the agent receives during the training.

4.1.3 Algorithm Component

The Algorithm component consists of the different algorithms, used for training the agent. It provides an algorithm interface, such that implementing and incorporating a new algorithm into the framework is easy.

4.1.4 Playing Layer

This layer contains the logic for playing a game with the current trained agent in the chosen environment. It also offers a user interface where the statistics of the game is shown as well as an graphical overview of the game.

4.1.5 Database Component

The Database component has the responsibility for the connection to the database. This component is used for saving the policy of a trained agent as well as the settings of the algorithm that was used to train the agent.

4.1.6 Policy Component

The Policy component is responsible for making it possible for learning algorithms to store and update a policy.

4.1.7 Environment Component

This component handles the environment which is the map and the uncertainty setting. The Algorithm component or Playing layer interacts with the environment as shown previously in figure 3.2 on page 31.

4.2 Learning Layer

This section describes the design of the Learning layer.

In order to make it possible to control the training of the agent, the Learning layer provides a user interface that contains the relevant training functionality. A mockup of the user interface for the Learning layer is shown in figure 4.2.
The user interface has an algorithm selector in the form of a drop-down box in order to make possible to select which algorithm to use during the training. In order make it easy to adjust the learning parameters for the reinforcement learning algorithms the user interface will contain fields for these parameters. It should be possible to adjust the discount factor, learning rate and exploration factor easily between each training session. When the learning algorithm is changed in the drop-down box, the different fields are updated with the default values for the chosen algorithm.

Right next to the adjustable parameters for the algorithm are similar fields for adjusting the different rewards the agent gets during the training. The rewards is made adjustable in order to change the behaviour of the mouse.

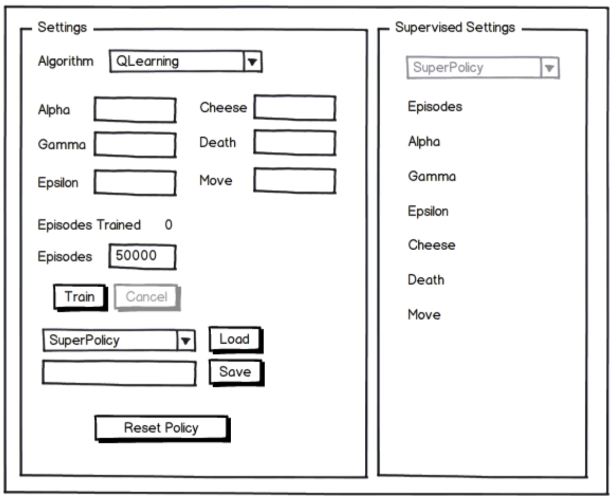

Algorithm: QLearning
Episodes Trained: 0
Episodes: 50000
SuperPolicy

Figure 4.2: The mockup of the user interface in the Learning layer.

Just below the parameters are the “Train” button and a field where the number of episodes to train for is entered. The agent can be trained multiple times with different amount of episodes and each training session will continue to update the same policy. The policy can be reset in order to start the agent from scratch and maybe try other parameters for the algorithm. A training session can be cancelled by pressing the “Cancel” button that is only enabled during training. If a training session is cancelled the policy is not reset, but keeps its value at the time of the cancellation.

The layer also provides interaction with the database in order to save a policy and use it later. By specifying a name for the trained policy it can be saved to the database and retrieved later. A policy can be retieved in order to play, continue training or to use it as a supervisor policy.

If the selected algorithm is a supervised algorithm, the panel to the right is enabled. This panel provides a policy-selector that is used for choosing the policy of the supervisor. The selectable policies is loaded from the database along with the settings that were used to train them. When switching between the policies the settings are loaded into the fields below in order to show how the policy was trained.

4.3 Playing Layer

This section describes the design of the Playing layer.

The Playing layer provides a user interface, where the game can be played and the statistics of the game is shown. A mockup of the user interface for the Playing layer is shown in figure 4.3. As seen in the figure, an visual overview of the game is located in the top left. The statistics are shown to the right and includes the current time step, current episode, the number of times the cat has eaten the mouse, the number of times the mouse has collected the cheese and the mouse performance ratio, calculated using Equation 1.1 on page 4.

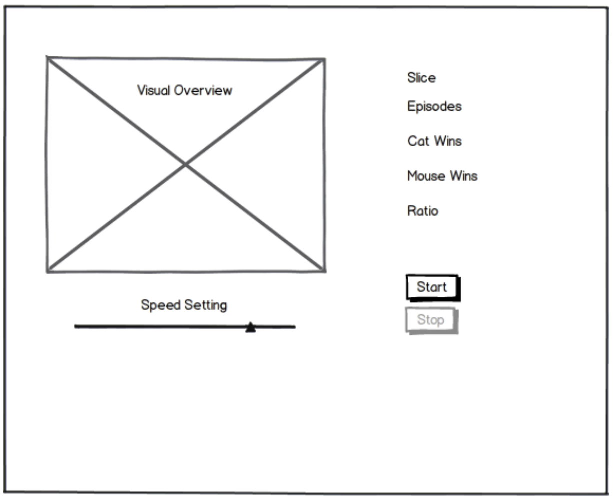


Figure 4.3: The mockup of the user interface in the Playing layer.

Below the statistics there are two buttons. One for starting the game and one for stopping the game. Only one of the buttons will be enabled at a time.

The slider below the visual overview controls the frame rate of the game.

4.4 Algorithm Component

This section describes the design of the Algorithm component.

The class diagram for the Algorithm component is shown in figure 4.4.

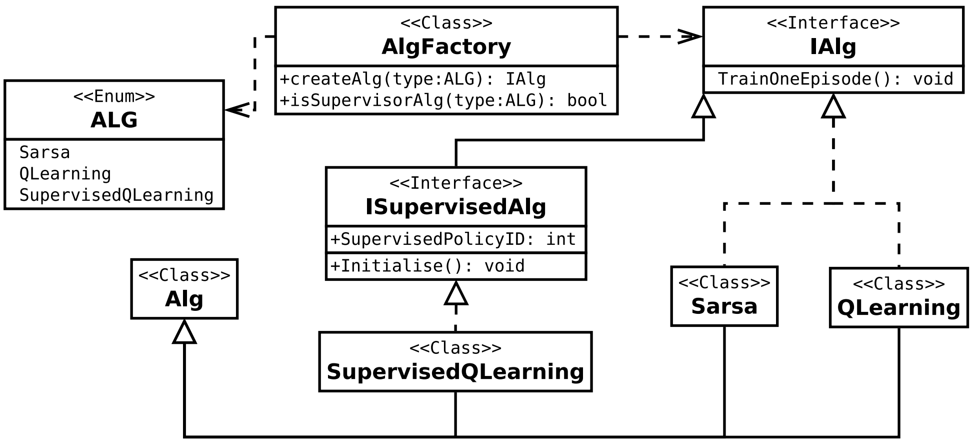


Figure 4.4: The class diagram of the Algorithm component.

As seen in the class diagram, the main class for the component is the `AlgFactory` class. This class is based on the Factory design pattern[Freeman et al., 2004] that makes a decoupling between the client of the algorithms and the actual algorithms implementation. This means that adding a new algorithm to the program only requires change in the `AlgFactory` class and in the `ALG` enum. The enum is sent as a parameter to the factory method to specify which algorithm to retrieve without referring to the exact implementation.

The `IAlg` interface specifies the required methods the individual algorithms must implement in order to be used in the Learning layer. This includes, for example, the `TrainOneEpisode` method which is the core of an algorithm.

The interface `ISupervisedAlg` contains the extra methods that concerns supervised algorithms. These methods are required as supervised algorithms need a connection to its supervisor and also need to fetch the policy of its supervisor. This is done in the `Initialise` method.

To differentiate between supervised algorithms and normal algorithms, the method `isSupervisedAlg` in the `AlgFactory` class is used. This makes it easy to execute supervised specific code.

From the class diagram it can also be seen that all algorithms classes inherits from the `Alg` class. This is because the share common properties like the discount factor, learning rate and exploration probability.

4.5 Summary

This chapter elaborated the design decisions for the learning framework. It presented the chosen design of the architecture which is based on the multilayered architecture with a shared component layer at the bottom layer. Each layer and component of the architecture and their responsibilities was described.

Two layers, the Learning and Playing layers, were elaborated and the designed user interface for the layers was depicted.

The Algorithm component was also elaborated and the purpose of important classes and interfaces were explained.

Chapter 5

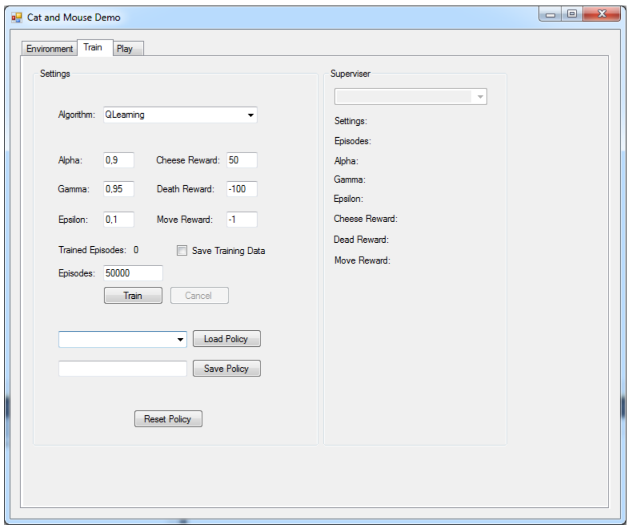
Implementation

This chapter elaborates the important implementation details of the developed learning framework. It focuses on the Learning layer, the Playing layer and the Algorithm component, as these are the most interesting parts of the framework.

5.1 Learning Layer

This section describes the implementation of the Learning layer.

A screenshot of the implemented user interface for the Learning layer is shown in figure 5.1.


Cat and Mouse Demo

Environment | Train | Play

Settings

Algorithm: QLearning

Alpha: 0,9
Cheese Reward: 50
Gamma: 0,95
Death Reward: -100
Epsilon: 0,1
Move Reward: -1

Trained Episodes: 0
Save Training Data
Episodes: 50000

Superviser

Settings:
Episodes:
Alpha:
Gamma:
Epsilon:
Cheese Reward:
Dead Reward:
Move Reward:

Figure 5.1: A screenshot of the implemented user interface for the Learning layer.

As seen in the figure the design is similar to the mockup in figure 4.2 on page 38. The only difference is the "Save Training Data" checkbox, that is used to save the data used for the different plot in the experiments. This functionality was added in order to save only the relevant training sessions and thereby minimising the space used for training results.

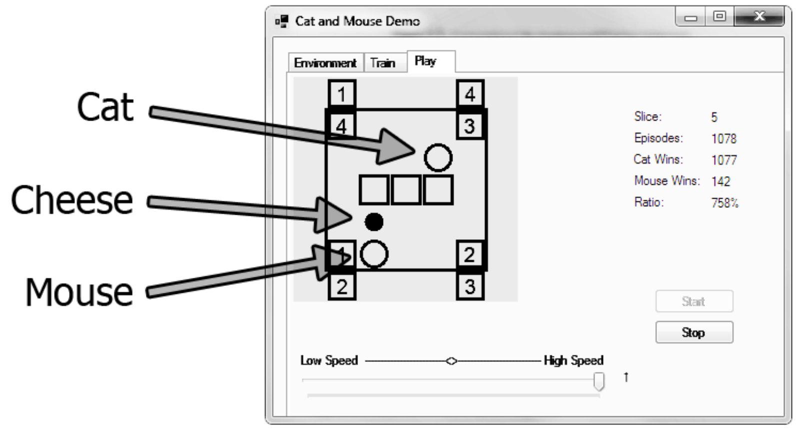
5.2 Playing Layer

This section describes the implementation of the Playing layer.

A screenshot of the implemented user interface for the Playing layer is shown in figure 5.2.
As seen in the figure the design is similar to the mockup in figure 4.3 on page 39. The screenshot is taken from a game where the agent has not been trained, which means that the mouse moves randomly. It can be seen in the statistics area, that the mouse performance is 758%, which means it dies approximately 8 times before collecting one cheese.

The visual overview shows the map used for the game which is *Map2* with teleports. The map is represented with the large black square and in the middle of the map there are three small black squares, that represents the obstacles.

It is difficult to spot the difference between the cat and the mouse in this screenshot because it is black and white. The cat is at position (3,1), the mouse is at position (1,4) and the cheese is just above the mouse, as indicated by the arrows.



Slice: 5
Episodes: 1078
Cat Wins: 1077
Mouse Wins: 142
Ratio: 758%

Figure 5.2: A screenshot of the implemented user interface in the Playing layer.

Below the visual overview the slider is positioned close to the right and just besides the slider there is a number that shows the chosen time delay between the actions. This time delay is controlled by the slider and is measured in ms.

5.3 The Algorithm Component

This section describes how to incorporate an algorithm into the learning framework, as well as elaborating the implementation of the supervised Q-learning algorithm used in the experiments.

5.3.1 Incorporate an Algorithm

To incorporate an algorithm into the learning framework, three simple steps are performed. When the three steps are completed and the class is fully implemented it is selectable from the user interface when training the agent.

This section describes the implementation of the Q-learning algorithm in order to show the three steps in details. The theory behind the algorithm is described in section 2.2.1.3. The algorithm is implemented in the `QLearning` class.

5.3.1.1 Add member to the `ALG` Enum

To let the framework know about the new algorithm it is required to specify a new member of the `ALG` Enum. Source code 5.1 shows the source code of the `ALG` Enum with the newly added member `QLearning`.

```
public enum ALG
{
    QLearning,
    Sarsa,
    SupervisedQLearning
}
```

Source Code 5.1: The ALG Enum.

5.3.1.2 Updating the `AlgFactory` class

The `createAlg` method in the `AlgFactory` class needs to be updated, such that the client of the factory can request an instance of the new algorithm. Additionally, if the new algorithm is a supervised one, the `isSupervisedAlg` method must be updated as well. Since the Q-learning algorithm we are implementing here is not a supervised algorithm, we only need to update the `createAlg` method, as shown in source code 5.2.

```
public static IAlg createAlg(ALG alg, ref IEnvironment environment, ↵
    ref IPolicy policy)
{
    switch (alg)
    {
        case ALG.QLearning:
        {
            return new QLearning(ref environment, ref policy);
        }
      ... // Continues with a case for each of the existing members in ↵
        the ALG enum.
    }
}
```

Source Code 5.2: The `createAlg` method in the `AlgFactory` class.

As seen in the source code the `createAlg` method is, besides the `ALG` type parameter, called with a reference to both the environment and the policy. This is done to ensure the algorithm shares the environment and policy with the rest of the program, especially the Playing layer. The references are passed along to the new instance of the requested algorithm.

5.3.1.3 Implementing the `IAlg` interface

The `QLearning` class needs to implement the `IAlg` interface in order to implement the specific methods used by the Learning layer. In order to make the `QLearning` class simple it extends the `Alg` class and inherits the learning parameters.

```
public QLearning(ref IEnvironment environment, ref IPolicy policy)
    : base(ref environment, ref policy)
{
    Alpha = 0.9;
    Gamma = 0.95;
    Epsilon = 0.1;
}
```

Source Code 5.3: The constructor of `QLearning`.

Since it inherits the different properties, the `QLearning` class can specify the default values for each of the properties as shown in the source code.

The core of the algorithm is implemented in the `TrainOneEpisode` method, which is executed once per episode. The `TrainOneEpisode` method for the Q-learning algorithm is shown in source code 5.4.

```
 1 public void TrainOneEpisode()
 2 {
 3     environment.ResetScores();
 4     state = environment.Reset();
 5    ...
 6     while (!environment.IsTerminal())
 7     {
 8         action = selectAction(state);
 9         newstate = environment.NextState(action);
10         reward = environment.Reward;
11         this_Q = policy.getQValue(state, action);
12         max_Q = policy.getMaxQValue(newstate);
13         new_Q = this_Q + alpha * (reward + gamma * max_Q - this_Q);
14         policy.setQValue(state, action, new_Q);
15         state = newstate;
16     }
17     if (saveData)
18     {
19         LogController.Instance.TrainingLog(ref environment);
20     }
21 }
```

Source Code 5.4: The `TrainOneEpisode` method.

The implementation of the Q-learning algorithm is straightforward. The first line resets the scores in the environment and invokes the `ResetWorld` method in line 4 resets the environment and brings the environment to a random initial state, which is returned. The code in line 7 through 16 is executed as long as the mouse is alive. Inside the loop an action is selected based on the current state in line 8. The `selectAction` method is a local method, that selects an action using an ϵ-greedy policy.

Performing the chosen action in the environment in line 9 leads to a new state and the reward is retrieved in line 10. Afterwards the Q value for current state and chosen action is retrieved from the policy, as well as the maximum Q value from the new state. These Q values are then used to calculate the new Q value for the current state and chosen action, by using the inherited `alpha` and `gamma` variables. This calculation is based on the update equation in line 9 in listing 2.6 on page 19. The policy is then updated with the new Q value in line 14, before the state is set to the new state and the loop continues.

The last few lines of the `TrainOneEpisode` method contains the functionality to save the training data by using the `LogController` class. This class handles the logging of relevant data to a file in order to measure how the agent performs during training. The `LogController` class is based on the Singleton design pattern[Freeman et al., 2004]. This means that there is only one instance of the logger and ensures that the log file is only written to by one process at a time.

5.3.2 Supervised Q-Learning

This section elaborates the implementation of the supervised Q-learning algorithm which is described in section 2.2.3.

As the `SupervisedQLearning` class implements the `ISupervisedAlg` interface, it is initialised before the training begins in order to fetch the selected supervisor policy from the database. This is done in the `Initialise` method that also initialises the array for storing the state visits.

The `TrainOneEpisode` method is shown in source code 5.5.

```
public void TrainOneEpisode()
{
   environment.ResetScores();
    state = environment.Reset();
    ... // Initialise local variables
    while (!environment.IsTerminal())
    {
        updateVisitTable(state);
        action = selectAction(state);
        newstate = environment.NextState(action);
        reward = environment.Reward;
        this_Q = policy.getQValue(state, action);
        max_Q = policy.getMaxQValue(newstate);
      ct = ((k * (reward + gamma * max_Q)) + ((1 - k) * ↲
          SPolicy.getQValue(state, action)));
        new_Q = (1 - alpha) * this_Q + alpha * ct;
        policy.setQValue(state, action, new_Q);
        state = newstate;
    }
    ... // Save training data
}
```

Source Code 5.5: The `TrainOneEpisode` method.

The algorithm itself is very similar to the normal Q-learning algorithm. The first difference is in line 8, with the `updateVisitTable` method and the content of this method is shown in source code 5.6.

```
1  private void updateVisitTable(int[] state)
2  {
3      StateVisitTable.SetValue(((int)StateVisitTable.GetValue(state)) + ↲
           1, state);
4      k = 1 - Math.Pow(0.9, (int)StateVisitTable.GetValue(state));
5  }
```

Source Code 5.6: The `updateVisitTable` method.

The purpose of the `updateVisitTable` method is to keep track of the number of visits in the different states. This statistics is used for calculating the `k` variable in line 4, that is used for deciding whether the agent should ask the supervisor or not.

The second difference in the `TrainOneEpisode` method is inside the `selectAction` method in line 9. In this method the `k` variable is used, as shown in source code 5.7.

```
private int selectAction(int[] state)
{
   ... // Initialise local variables
    // Ask Supervisor:
    if (RandomGenerator.Instance.getDouble() > k)
    {
        ... // Get the best action from the supervisor
    }
    // Explore:
    else if (RandomGenerator.Instance.getDouble() < epsilon)
    {
      ... // Choose a random action
    }
    // Exploit:
    else
    {
        ... // Get the agents own best action
    }
    return selectedAction;
}
```

Source Code 5.7: The `selectAction` method.

The `RandomGenerator` class is a singleton and is used to get a random number between 0 and 1. Depending on that number and the value of `k`, the agent will ask the supervisor for its best action. Otherwise, it will resume as normal Q-learning with an ϵ-greedy action selection.

The last difference in the `TrainOneEpisode` method is in line 14 and line 15, where the Q value is updated. This update lines corresponds to line 14 and line 15 in listing 2.7 on page 26.

5.4 Summary

This chapter presented the important implementation details of the learning framework.

It covered the implementation of the two main parts in the framework which are the Learning and the Playing layers. The implemented user interface was shown for each layer.

The implementation of the Algorithm component described how to add an algorithm to the framework in three simple steps and elaborated the implementation details of the supervised Q-learning algorithm.

CHAPTER 6

EXPERIMENTS

This chapter elaborates on the experiments done to examine how reinforcement learning can be used to train agents in tasks where the model is unknown. The cat and mouse game, as described in section 1.3, is used throughout these experiments to show different aspects of how the theory works and what influence the different parameters have on learning. The two maps, *Map1* and *Map2*, are used in this section and shown in figure 6.1.

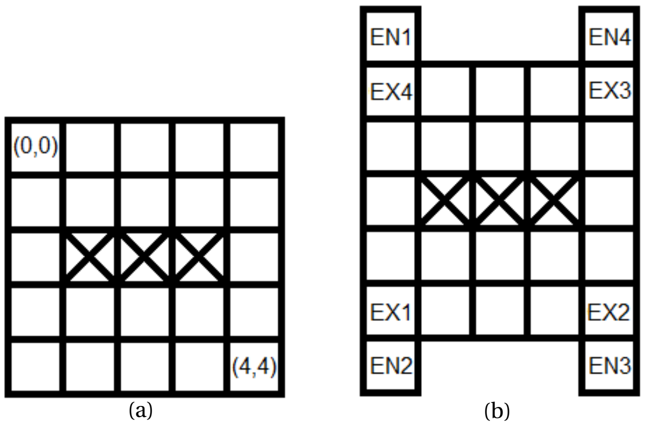


Figure 6.1: The maps used in this chapter: 6.1(a) is *Map1* and 6.1(b) is *Map2*.

An experiment with function approximation based on radial basis functions is carried out to show how it is possible to generalise between different states.

Experiments with supervised learning shows how the learning speed of an agent can be increased when a supervisor provides guidance in the form of actions and rewards. Furthermore, it is tested how useful advice from a supervisor is, if it is trained in a different environment than the learning agent.

The evaluation of online learning is done by measuring the amount of cheese the mouse collects in a single episode and by measuring the total length of an episode in action steps. Offline performance is evaluated using the ratio between collected cheese and number of deaths as described in equation 1.1.

6.1 Generalisation

This section elaborates on an experiment done where radial basis functions are used to approximate a value function for a simple version of the cat and mouse game. There is no control here meaning that a deterministic policy is evaluated in order to approximate the value function. Since this is a demonstration of policy evaluation, the policy is scripted and is just simply to move the mouse in the direction of the corner where the cheese is located. The game consist of a 5 × 5 grid with the cheese in the upper right corner and no cat. The game starts by giving the mouse a random position and ends when the mouse collects the cheese. The mouse receives a positive reward for collecting a cheese. Figure 6.2 shows the game map.

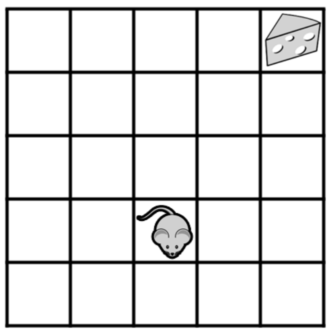


Figure 6.2: A simple version of the cat and mouse game without a cat.

The state is simply given by the x and y coordinate of the mouse. The cheese is always in the upper right corner.

The x and y parameters were split into 5 features and assumed to be independent. The value function for the x parameter is shown in equation 6.1. It is the sum of the 5 RBF features with their respective weights.

$$V_\theta(x) = \theta_1 f_1(x) + \theta_2 f_2(x) + \cdots + \theta_5 f_5(x). \tag{6.1}$$

At each step the weights of the radial basis functions were updated to a degree based on the distance from the feature centers. The derivative $\nabla_{\theta_i} V(s)$ from equation 2.14 on page 21 is the degree to which feature i is present. The higher this degree the more the weight is updated. For example there is a feature for the x value with the center in 3 and with a σ value of 0.5. When this feature is updated the partial derivative is the one in equation 6.2. In the example state the x parameter is 2.

$$\nabla_{\theta_i} V(s) = exp\Big(-\frac{\|2-3\|^2}{20.5^2}\Big) \approx 0.135335 \tag{6.2}$$

Listing 6.1 shows the algorithm used for the experiment. The algorithm is based on [Sutton and Barto, 1998] as well as the principles of [Papierok et al., 2008].

```
function RBF-policyEvaluation(){
   $V(s)$ = an arbitrary state value function
   $s$ = state
   repeat for each episode:
      repeat for each step
         Use scripted policy $\pi$ to select action $a$ in $s$
         take action $a$
         receive reward $r$ and next state $s'$
         $\delta = r + \gamma V(s') - V(s)$
         for all features $i$
            $\theta_i = \alpha\theta_i + (1-\alpha)\delta\nabla_{\theta_i} V(s)$
         $s \leftarrow s'$
}
```

Listing 6.1: Pseudo Code for RBF-policy evaluation.

The scripted policy is used to take actions and at each step all RBF features are updated based on the received reward. This was done for 50 episodes which resulted in the value function in figure 6.3.
The graph clearly shows that the closer the mouse is to the cheese the higher the value of the state is. After only 50 episodes it is possible to get a value for all states even though they might not have been visited many times or even at all. Testing showed that even after a few episodes, the value function already begins to have a top in the corner.

On a task like the one in this experiment, it would be easy to use value iteration in order to compute the state values if it was assumed that the model was known. The example was simplified in order to have a simple implementation and be able to visualise the value function on a graph. This experiment still shows how radial basis functions can be succesfully used to approximate a value function for a given reinforcement learning problem where the model is unknown. The main advantages is the generalisation between states and that it is possible to estimate the value function using fewer episodes.

6.2 Supervised Q-learning

This section elaborates on the experiments with the supervised Q-learning algorithm described in section 2.2.3. The implementation of the algorithm is described in section 5.3.2.

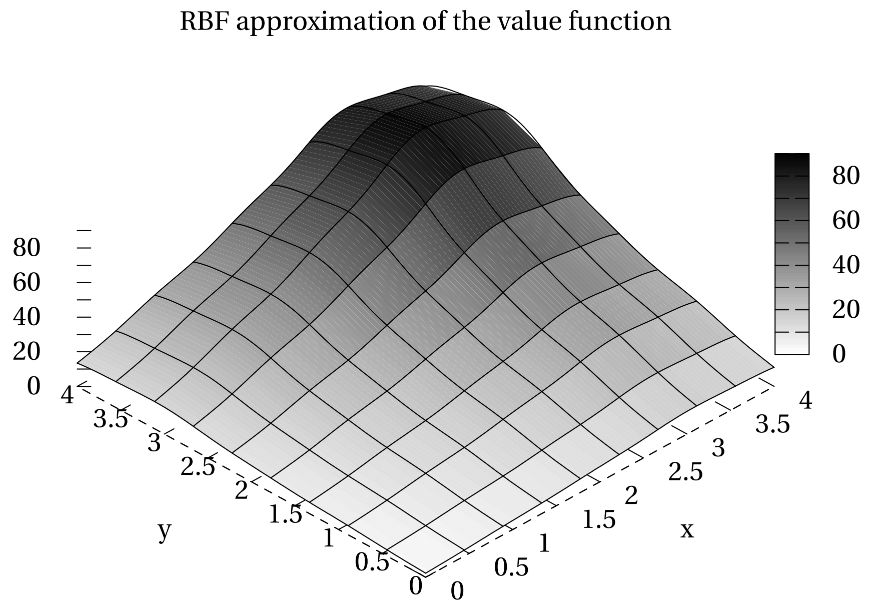


Figure 6.3: Approximated value function using RBF features.

The experiments were run on the cat and mouse game and aimed to see if the supervised Q-learning could learn faster than the normal Q-learning. *Map1* was used in this experiment without any uncertainty in relation to actions. A supervisor was trained using the normal Q-learning algorithm but learning was stopped early and before the $Q(s,a)$ function had converged. The idea behind this is that the supervisor does not need to know the optimal policy, but can accelerate the learning in the beginning in spite of this. At some point during training, the learning agent is supposed to surpass its supervisor and develop a better policy than the one of the supervisor. This is where the k parameter becomes important. The faster k approaches 1, the faster the learning agent will stop using the supervisor. If k is approaching 1 at a slower rate, the learning agent might be hindered by the supervisor and will have a harder time surpassing the supervisor.

In our implementation each state has a counter, denoted $visits(s)$, that is increased by one each time the agent visits the state. The value $k(s)$ is then calculated using the following formula:

$$k(s) = 1 - n^{visits(s)} \tag{6.3}$$

This equation is implemented in source code 5.6. Depending on n, $k(s)$ is a function that starts at zero and then approaches 1 slower and slower. This experiment will try three different values for n in order to see how it affects the k parameter and thereby the supervised learning. The three values for n: 0.99, 0.97 and 0.9 describes the functions seen in figure 6.4.

The value of k approaches 1 with different speeds depending on the n values we selected. The selection was done after observing how many times each state was visited using a normal Q-

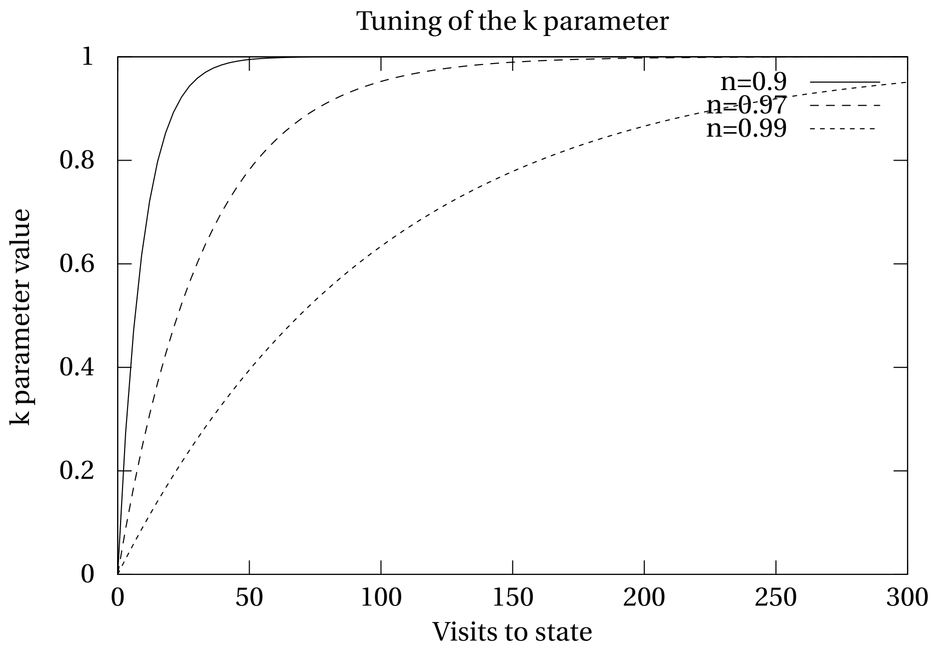


Figure 6.4: The $k(s)$ parameter as a function of visits to the state s.

learning algorithm. The number of visits is around a few hundreds per state when training for 100000 episodes. Experiments were run using the cat and mouse game on the map without teleports. The mouse was trained against a scripted cat using the following parameters:

Learning rate $\alpha = 0.9$

Discount Factor $\gamma = 0.95$

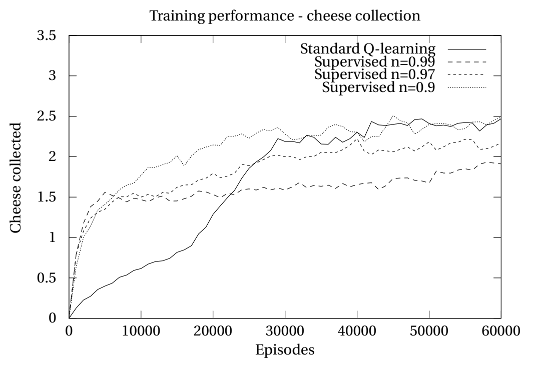
Epsilon (Exploration probability) $\epsilon = 0.1$

Cheese reward $r = 50$

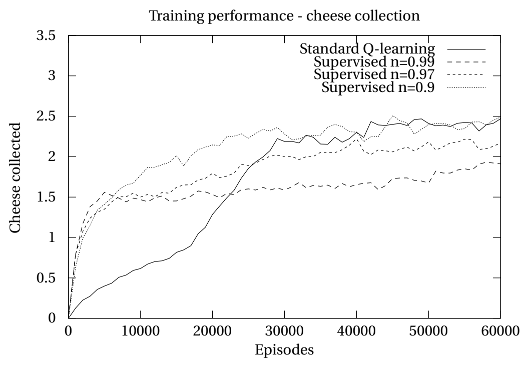
Death reward $r = -100$

Move reward $r = -1$

The supervisor consisted of a policy trained with the same parameters but training was stopped after 20000 episodes. This policy was saved including the $Q(s,a)$ values in order to make them available to the learning agent. Testing this policy alone yielded a $Deaths/CollectedCheese$ ratio of 56%. This policy was used to train three agents using the different functions for the k parameter depending on n. The training ran for 100000 episodes and the online learning performance can be seen in figures 6.5(a) and 6.5(b).
It is seen that all the supervised algorithms initially perform better than the standard Q-learning. The algorithms that uses the higher values for n, 0.99 and 0.97 are slow to converge after the initial ascend whereas the one with n=0.9 converges on par with the standard Q-learning. This is because higher values for n causes the learning to depend on the supervisor



(a) The y-axis is the amount of cheese collected in an episode.



(b) The y-axis is the number of steps the mouse survived in an episode.

Figure 6.5: The online learning performance of the mouse on *Map1* with a supervisor trained on *Map1*. The values are smoothed using a low-pass filter with a smoothing factor of 0.9993.

for too long. Having trained 20000 episodes, the usefulness of the supervisor becomes obsolete at some point and the k parameter must be used to balance this carefully.

This experiment was carried out in order to show that having a supervisor can increase the learning performance of the agent. It might seem a bit odd to use almost the same algorithm to train the supervisor and the agent, but a main observation is that even a supervisor with a suboptimal policy can greatly speed up the learning performance in the beginning of the training. If the k parameter is tuned correcly, it is possible for the agent to depend less and less on the supervisor and obtain a better policy than the supervisor.

The algorithm with n=0.9 ended up with a policy that yielded a $Deaths/CollectedCheese$ ratio of 8% which is comparable with the 7% ratio of the standard Q-learning. It makes sense that these values are close since the supervised learning algorithm basically turns into a normal Q-learning algorithm when the k parameter approaches 1.

Another thing to notice is that the supervisor does not need to be trained with the same algorithm as the learning agent. The only thing required of the supervisor is that it can output actions and a value function that is compatible with the agents own. The supervisor could have been trained using a different algorithm or even created from examples of a humans selecting actions. A handcoded script could also be used to define a deterministic policy that could be used by the supervisor. Again, according to our experiments, this scripted behaviour does not need to reflect the optimal behaviour in order to assist the learning agent.

6.3 Supervised Q-learning Between Environments

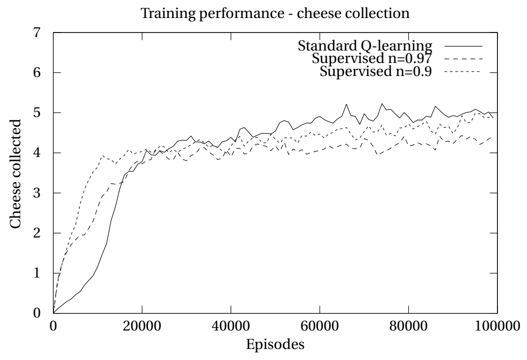
This section elaborates on experiments done where the supervisor is trained in one environment and supervises a learning agent in another environment. The idea is that if these environments are similar, the supervisor can still speed up the learning of the agent. Even though that the policy of the supervisor is suboptimal for the agents environment, there might still be many states of the environment where the knowledge of the supervisor is useful.

To illustrate this, multiple experiments were run which are elaborated in this chapter.

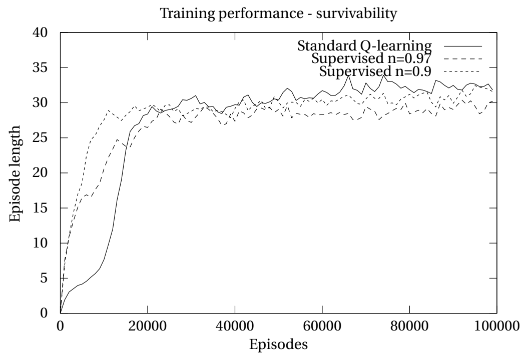
6.3.1 Experiment 1

In the first experiment, the supervisor was trained on the simple *Map1* without teleports and supposed to supervise the learning agent train on *Map2* with teleports. The thing to notice about the maps, is that the state space is exactly the same. The mouse, cat and cheese can obtain all the same positions. But because of the teleports, the transition function is different for *Map2*. The supervisor have not experienced the transition that happens when the mouse enters a teleport entrance and is moved to the teleport exit. The state-action pair of going through a teleport will in the supervisors policy not have the correct Q value and the learning agent will need to learn this value by itself. This should be possible because the agent will select a random action with probability $k\epsilon$.

The supervisor was trained on *Map1* with standard Q-learning for 20000 episodes. The learning agent was trained on *Map2* using supervised Q-learning for 100000 episodes. The learning algorithm parameters was identical to those of the supervised Q-learning experiment. There is no uncertainty in relation to actions. Online learning performance is shown in figure 6.6(a) for cheese collection and in figure 6.6(b) for mouse survival.



(a) The y-axis is the amount of cheese collected in an episode.



(b) The y-axis is the number of steps the mouse survived in an episode.

Figure 6.6: The results of experiment 1 shows the online learning performance of the mouse on *Map2* with a supervisor trained on *Map1*. The values are smoothed using a low-pass filter with a smoothing factor of 0.9993.

The figures clearly shows that training speed is increased in the beginning of training. $n = 0.9$ still seems like the best value for the $k(s)$ function. It is important to point out that both the standard Q-learning and the supervised Q-learning ends up with a policy where the mouse collects cheese continuously and is never caught by the cat.

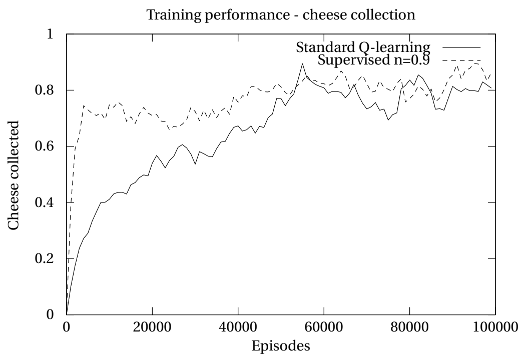
This shows that the two algorithms in the experiments eventually converge to very similar policies, but that the online performance of the supervised Q-learning is superior to the standard Q-learning. The amount of collected cheese for the algorithms after 100000 episodes is about 5 per episode. This seems like a lot less than the offline amount, but the reason that online performance is much worse than offline is that online uses the ϵ-greedy version of the learned policy. If ϵ were decreased over time, the online learning will at some point use the deterministic policy and cause the mouse to never get caught. Our implementation does not include this in order to avoid the algorithm running for a very long time because the episodes never ends.

This experiment also indicates that it is faster to learn the policy where the mouse is never caught for *Map2* than *Map1*. This makes sense because the mouse clearly has an advantage over the cat by using the teleports to move multiple squares in one action.

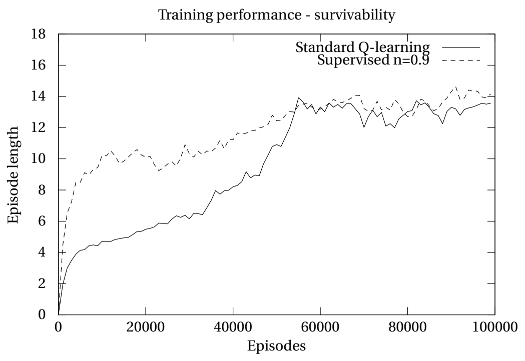
6.3.2 Experiment 2

This experiment is conducted with both the supervisor and the agent training on *Map1*. The difference is that the probabilities of the transition function for the agent is different. Uncertainty is introduced, meaning that the actions does no longer have a deterministic outcome. As described in section 1.3 the agent will with probability 0.8 move in the selected direction, with 0.1 move to the right and with 0.1 move to the left in relation to the selected direction.

The supervisor was trained on *Map1* with standard Q-learning for 20000 episodes. The learning agent was trained on *Map1* with action uncertainty using supervised Q-learning for 100000 episodes. Again, the learning was done using the same parameters as in the other experiments. The graphs in figure 6.7(a) and figure 6.7(b) shows the results.



(a) The y-axis is the amount of cheese collected in an episode.



(b) The y-axis is the number of steps the mouse survived in an episode.

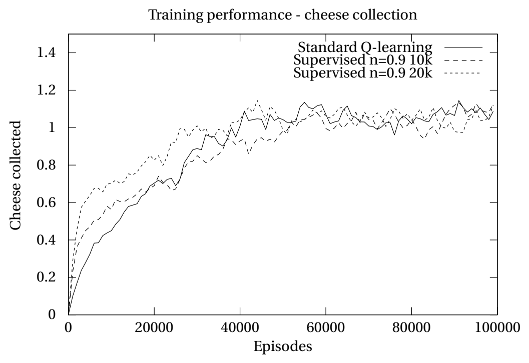
Figure 6.7: The results of experiment 2 shows the online learning performance of the mouse on *Map1* with action uncertainty using a supervisor trained on *Map1* without action uncertainty. The values are smoothed using a low-pass filter with a smoothing factor of 0.9993.

Again, the learning performance is increased initially after which both algorithms converge to about the same online performance. The offline performance for standard Q-learning is 88% in comparison with 78% of the supervised Q-learning. It seems that introducing uncertainty in relation to actions makes it a lot harder for the mouse to collect cheese. This makes sense because all positions of the map have an adjecent wall which means that most actions will have a significant probability of going into the wall. Doing this will result in the mouse not being moved while the cat is moved. With no teleports it is fairly hard for the mouse to avoid the cat.

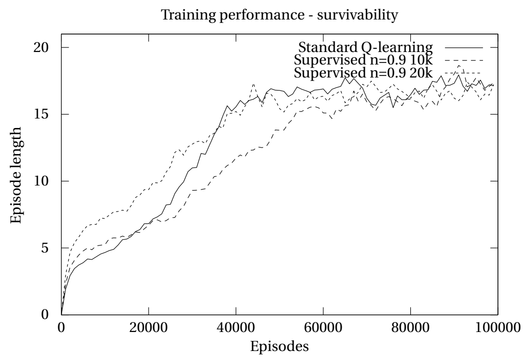
6.3.3 Experiment 3

This experiment illustrates the difference in performance depending on how good the supervisor is. The supervisors was trained on *Map1* whereas the agent was training on *Map2*. In both cases action uncertainty was included. The only difference between the maps is that *Map2* has teleports.

Two supervisors were used. One trained for 10000 episodes and one trained for 20000 episodes. The learning agent and the standard Q-learning agent trained for 100000 episodes. The results of training with these two supervisors are shown in figures 6.8(a) and 6.8(b).



(a) The y-axis is the amount of cheese collected in an episode.



(b) The y-axis is the number of steps the mouse survived in an episode.

Figure 6.8: The results of experiment 3 shows the online learning performance of the mouse on *Map2* with action uncertainty using a supervisor trained on *Map1* with action uncertainty. The two agents are trained for 10000 and 20000 episodes respectively. The values are smoothed using a low-pass filter with a smoothing factor of 0.9993.

The graphs shows that the supervisor trained for 20000 episodes speeds up the learning more than the supervisor that trained only 10000 episodes. This makes sense, since it must be assumed that the less trained supervisor has a worse policy. All of the trained agents end up with an offline performance of 73% This indicates that with uncertainty, *Map2* is easier than *Map1* because of the teleports.

6.4 Summary

This section elaborated on experiments with generalisation and supervised learning. It was shown that using generalisation with radial basis functions it is possible to approximate a value function by updating the weights of the basis functions. This makes it possible to generalise between states and makes it possible to reason about the value of a state that have never been encountered before. The idea is that it is not required to visit all states many times in order to learn a value function which is an advantage for tasks with large state spaces.

The experiments with supervised learning showed that it is possible to speed up the learning by using a supervisor that only have limited training experience. If the parameters of the supervised learning algorithm is tuned correctly, it is possible to achieve this boost in online learning performance as well as being able to converge on par with the normal Q-learning. Since the supervisor have a suboptimal policy it is important reduce the supervisor influence over time in order to surpass the performance of the supervisor.

It was shown that supervised learning could be used to transfer knowledge between two environments. Training with the supervisor that were trained in different environment than the learning agent itself could still boost the learning performance of the learning agent. This was because the environments had many things in common. For example the map with and without teleports have the same layout.

This was also the case when the difference between the environments was in relation to action uncertainty. A supervisor trained in a an ideal environment without uncertainty could speed up the learning performance of the agent in a environment with uncertainty. Because the two environments are still somewhat similar, knowledge from the supervisor is still able to be carried over and used by the learning agent.

The increase in online performance is good in relation to real world environments. Because the learning performance curves have a higher initial assend it should indicate that the agent is performing better than its nonsupervised counterpart. In real world scenarios, receiving negative rewards might be associated with some kind of failure. In, for example, a robot domain this could have a certain risk or cost affiliated with it. A robot will not take as many risky explorative actions if it is supervised which indicates that the risk or costs could be reduced by using a supervised learning algorithm.

Chapter 7

Recapitulation

This chapter concludes the project and discusses our efforts done during this semester. We explored different techniques and developed prototype frameworks in order to experiment with the reinforcement learning theory. The discussion elaborates on this and provides ideas for how future work could be carried out.

7.1 Discussion

During this project another prototype simulator were developed than the one described in this report. It aimed to more closely model the Kinect framework and robot environment. The framework did not use a gridlike model of the floor but supported the full 320 × 240 resolution of the Kinect depth field. The actions of the robot had variable length to make it possible to model movement at different velocities.

In an attempt to reduce the state space and thereby the needed Q value table we implemented a state representation based on observations. The idea was to limit the vision of the learning agent to a field around itself. In our approach we created 16 view zones around the agent as seen in figure 7.1.

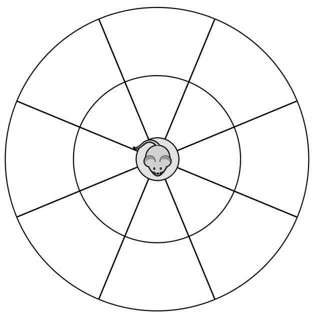


Figure 7.1: The observation field around the agent divided into 16 zones.

Each of these zones could be occupied by the different obstacles in the environment. This included the walls and other agents in the environment. Limiting the perception of the agent by only allowing it to receive an observation is a way to make the problem into a partial observable MDP. This issue is discussed in [Perkins, 2002] where it is pointed out that a simple solution is to treat the observations of the POMDP as states for an MDP. An MDP based on observations instead of the full state represtiation has a much smaller state space and should be easier to handle in a reinforcement learning framework. However, learning approaches can sometimes fail to converge on problems with partial observabilty. The use of gradient descent methods can however be shown to be able to converge to at least a local optimal policy.

Experimenting with the MDP based on observations in a simple tag game proved to be quite problematic. The agents could only learn a very simple strategy where they just moved in the opposite direction of the pursuer. We tried different tuning parameters and different versions of the observations with limited success.

Another approach we attempted was to represent the state using a number of state variables. Our idea was to use radial basis functions to approximate the value functions based on features of these state variables. The state variables were, for example, the euclidian distance between the agent and the pursuer and between the agent and the surrounding walls. In [Stone et al., 2006] the state is represented with a number of distances and angles between players in a simulated soccer game. Our idea was to generalise between similar states using radial basis functions to approximate the values of state-action pairs.

Designing the radial basis function features proved to be a complex task however. Often the weights for the functions converged towards infinity or towards zero and it was impossible to learn anything. Another big issue was the increase in computation time because of the need to update all the RBF features. In a six dimensional state representation with five RBF features for each state variable there are around 15000 distinct features that needs to have their weights updated at each time step. Even if many of the features are only present to a very small degree, the calculation still needs to be done.

These were the main reason why we decided to implement a simple learning framework for a gridworld based cat and mouse game. Here, it was possible to implement a supervised learning algorithm and show how it can be used to increase learning performance for a learning agent that trains in a slightly different environment. In a simple environment we showed that it is possible to use radial basis functions to approximate value functions in order to generalise between similar states.

This gives us some reassuring that it should be possible to use these techniques in a real world reinforcement learning problem. The vision is that supervised learning and generalisation can be used for robots in the Kinect environment. By developing a simulator with uncertainty that aims to model that of the real world robots, it should be possible to ensure that the simulated environment is somewhat similar to the real world environment. A supervisor could be trained in the simulator and then used to aid the training of an agent interacting with the real world environment. The supervisor, having learned its policy under simulated uncertainty, should be useful for the learning agent in the initial phase of the training. As training progresses and the learning agent becomes less dependent on the supervisor, it should start to shift its policy to take the uncertainty of the real world environment into consideration.

Futhermore if the real world environment was changed in some way it would be faster to change the simulator as well and do supervised learning instead of learning from scratch in the changed environment. Introducing teleports might not be possible, but the obstacles could be changed to create new possible paths for the robots to take. This would create a similar environment where the supervisor could be used to speed up the learning performance.

The supervised learning would still require a significant amount of episodes in order to train well. Here, generalisation could be used to reduce the number of training episodes needed in order to obtain a good policy. Generalisation between states could make it possible to be able to reason about the values of actions in non-visited states. Using a carefully designed radial basis function approximation with features based on the state representation could be used to achive generalisation. The state could be represented by the positions of the different agents and possible objects but also on compound variables like used in the soccer game [Stone et al., 2006]. For example the angle and distance between the robots, the distance and direction of the nearest wall or the distance to some goal. The design of these parameters have a high influence over how good and robust the learning can be.

If generalisation could be used in such a way that it would be possible to obtain a policy based on under maybe 100 episodes it would be possible to use a human supervisor. The human does not need to know the optimal policy but could most certainly have some idea of what action could be worth exploring. This could be used to speed up the learning performance of the agent and maybe even learn to behave like the human instead of learning the optimal policy.

7.2 Conclusion

The purpose of this semester was to gain knowledge on model-free reinforcement learning and examine how it could potentially be used in a real world robot domain where it is hard to obtain a complete model for the environment. We studied different reinforcement learning methods including dynamic programming, Monte Carlo and temporal difference approaches. Based on this, we elaborated on a supervised learning algorithm where a supervisor is used to speed up the learning of an agent. To cope with larger state spaces and to allow generalisation between similar states, techniques like coarse and tile coding as well as radial basis functions can be used. These feature based approximation techniques made the learning more complex, but makes it possible to solve more advanced reinforcement learning problems.

The use of a simulator for the environment can be an advantage even though it is impossible to obtain an exact model of the environment. We developed a reinforcement learning framework and a simple simulator for the cat and mouse game to show the usefulness of having a simulator. This framework was used to conduct small scale experiments with reinforcement learning theory. We showed that radial basis functions can be used to approximate value functions and allow generalisation between states. Supervised learning was proven to be able to speed up the online learning performance of the mouse agent in the game. This could be achieved using a limited trained supervisor and also when the supervisor were trained in a different environment than that of the learning agent. For example, when uncertainty was introduced, the learning agent could still benefit from a supervisor that was trained without uncertainty.

A supervisor can be trained on a simulator that aims to model the dynamics of the real world environment. This supervisor can then be used to aid the training of an agent that interacts directly with the environment. If this technique is combined with well designed generalisation, there should be potential for realising supervised learning in real world robot domains.

Bibliography

[Eden et al., 2012] Eden, T., Knittel, A., and van Uffelen, R. (2012). Reinforcement Learning. `http://www.cse.unsw.edu.au/~cs9417ml/RL1/`.

[Fowler, 2012] Fowler, M. (2012). Model view controller. `http://www.martinfowler.com/eaaDev/uiArchs.html#ModelViewController`.

[Freeman et al., 2004] Freeman, E., Robson, E., Bates, B., and Sierra, K. (2004). *Head First Design Patterns.* OŔeilly Media, 1st edition.

[Højlund and Madsen, 2012] Højlund, F. and Madsen, K. R. (2012). Kinect Positioning System for LEGO® Robot Planning.

[Jensen et al., 2010] Jensen, B., Arroyo, D. O., Cortés, N. C., and Rodríguez-Henríquez, F. (2010). Supervised Reinforcement Learning in Discrete Environment Domains.

[Microsoft, 2012] Microsoft (2012). Layered Application. `http://msdn.microsoft.com/en-us/library/ff650258.aspx`.

[Papierok et al., 2008] Papierok, S., Noglik, A., and Pauli, J. (2008). Application of Reinforcement Learning in a Real Environment Using an RBF Network.

[Perkins, 2002] Perkins, T. J. (2002). Reinforcement Learning for POMDPs based on Action Values and Stochastic Optimization.

[Rosenstein and Barto, 2004] Rosenstein, M. T. and Barto, A. G. (2004). Supervised Actor-Critic Reinforcement Learning.

[Rummery and Niranjan, 1994] Rummery, G. A. and Niranjan, M. (1994). On-Line Q-Learning Using Connectionist Systems.

[Russell and Norvig, 2006] Russell, S. and Norvig, P. (2006). *Artificial Intelligence: A Modern Approach.* MIT Press, 3rd edition.

[Stone et al., 2006] Stone, P., Kuhlmann, G., Taylor, M. E., and Liu, Y. (2006). Keepaway Soccer: From Machine Learning Testbed to Benchmark.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, 1st edition.

[Watkins, 1989] Watkins, C. J. C. H. (1989). Learning from Delayed Rewards.

Appendix A

Summary

The topic for this project was to explore reinforcement learning techniques in relation to implementation in real world learning environments. In the preceding project, a robot framework that uses a Microsoft Kinect® as a positioning system for LEGO® robots was developed. In this project we aimed to study how existing reinforcement learning techniques potentially could be used in an environment like this.

We explored different methods for solving reinforcement learning tasks. This included dynamic programming, Monte Carlo and temporal difference methods. These methods build on the foundation of Markov decision processes which lay the groundwork for sequential decision making. Furthermore, theory about generalisation and value function approximation was elaborated to cope with more complex learning tasks. A supervised learning algorithm was presented based on related work.

We defined a toy-like cat and mouse game and developed a reinforcement learning framework that included a simulator. The learning framework makes it possible to implement the different learning algorithms and train agents with different learning parameters.

By using the developed learning framework, we showed that radial basis functions can be used for generalisation and value function approximation. Other experiments indicated that the supervised learning algorithm could increase the learning speed of an agent as well as provide a way to transfer knowledge from one domain to another.

This page is left blank for the purpose of containing the attached CD-ROM.