Robotic Controllers for Navigation using Reinforcement-Learning

Victor B F Gomes
Supervisor: Eleni Vasilaki

May 2, 2012

This report is submitted in partial fulfilment of the requirement for the degree of Master of Computer Engineer in Computer Science by Victor Borges Ferreira Gomes.
Signed Declaration

All sentences or passages quoted in this report from other people’s work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Victor B F Gomes
Signature:

Date: May 2, 2012
Abstract

Understanding the human brain and its behaviour is the main aim of Neuroscience, therefore forming a model with the objective of imitating a special biological behaviour, like the ability to learn, is a research problem with many potential applications. This thesis aims to present a simulation of the Morris water maze [22] using a robot in order to compare two different Reinforcement Learning techniques; model-based [34], where the transition function between each state is calculated during the interaction with the environment, and model-free [33], without a fully specified model of the environment. It will provide an overview of both principles and algorithms that can be used for Machine Learning in order to simulate animal behaviour, their advantages and disadvantages. A platform will be created to easily determine the best algorithm in both techniques. During the experiment, the robot will produce learning behaviour in order to be able to self-localise in his environment.
Acknowledgements

I am heartily thankful to my supervisor, Eleni Vasilaki, whose encouragement, guidance and support enabled me to develop an understanding of the subject. I would like to thank James Hole for correcting this thesis. I also like to thank my family who always gave the moral support I needed.
## Contents

Signed Declaration  
Abstract  
Acknowledgements  
Contents  
List of Figures  
List of Algorithms  
Listings  

1 Introduction  
   1.1 Aims & Objectives  
   1.2 Structure of the document  

2 Literature Review  
   2.1 Reinforcement Learning Overview  
      2.1.1 Basics Concepts  
      2.1.2 Exploration versus Exploitation  
      2.1.3 Formal Problem  
      2.1.4 Bellman Equations  
      2.1.5 Learning an Optimal Policy  
      2.1.6 Model Free  
      2.1.7 Model Based  
   2.2 Neuroscience and Reinforcement Learning  
      2.2.1 Dopamine Neurons  
      2.2.2 World model  
   2.3 Reinforcement Learning in Robotics  
      2.3.1 Overview  
      2.3.2 POMDP - Partially Observable MDP  
   2.4 Navigation  
      2.4.1 Gridworld  
      2.4.2 Continuous Space  
   2.5 Neuroscience and Navigation  
      2.5.1 Place Cells  
      2.5.2 The Hippocampus Model  
   2.6 Summary  

3 Requirements and Analysis  
   3.1 Requirements  
   3.2 Analysis  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Literature Review</td>
<td>3</td>
</tr>
<tr>
<td>Requirements and Analysis</td>
<td>16</td>
</tr>
</tbody>
</table>
## 4 Reinforcement Learning Framework

4.1 High Level Packages ................................................. 18
  4.1.1 tools package ........................................ 18
  4.1.2 main package ....................................... 19
4.2 k-Armed Bandit Simulation ........................................... 22
4.3 Summary ......................................................... 24

## 5 Algorithms comparison

5.1 Gridworld ......................................................... 25
5.2 Automation ......................................................... 25
5.3 Gathering results .................................................. 27
5.4 Choosing the best parameters ..................................... 28
  5.4.1 Without eligibility traces .................................. 28
  5.4.2 With eligibility traces .................................... 29
  5.4.3 Policy .................................................... 30
5.5 Model-free ......................................................... 32
5.6 Model-based ....................................................... 32
5.7 Stochastic Environment ........................................... 33
5.8 Discussion ......................................................... 34
5.9 Summary ......................................................... 34

## 6 Robotic Controller

6.1 E-Puck, Mobile Robot ................................................ 37
6.2 Webots, Robot Simulator ........................................... 38
6.3 Design .......................................................... 39
  6.3.1 Mapping Morris Experiment onto Reinforcement Learning ..... 39
  6.3.2 Self-Localisation Module ................................... 39
  6.3.3 Odometry Module ......................................... 40
  6.3.4 Tile-Coding Function Approximation .......................... 41
6.4 Implementation .................................................... 42
  6.4.1 Supervisor Controller ...................................... 42
  6.4.2 Finite-State Machine ...................................... 42
  6.4.3 Self-Localisation Module ................................... 43
6.5 Results and Discussion ............................................. 43
6.6 Summary ......................................................... 43

## 7 Conclusion

7.1 Conclusion ......................................................... 45
7.2 Future Work ....................................................... 45

## A Project Plan

A.1 Overview ......................................................... 46
A.2 Gantt chart ....................................................... 46

Bibliography ......................................................... 48
List of Figures

1.1 Water-maze experiment. .................................................. 1
2.1 The reinforcement learning model. ........................................ 3
2.2 The Car on the Hill problem. ............................................ 4
2.3 Dopamine pathways .......................................................... 11
2.4 Examples of environment discretization. .................................. 12
2.5 A gridworld example. ....................................................... 14
3.1 Module diagram for Webots simulation. .................................... 17
4.1 tools package. .................................................................... 18
4.2 The GridState. .................................................................... 19
4.3 Agent class. ....................................................................... 19
4.4 IEnvironment interface. ...................................................... 20
4.5 Developed agents. .............................................................. 20
4.6 Simulator sequence diagram ............................................... 21
4.7 Performance of $\epsilon$-greedy methods on the 10-armed bandit test-bed for 1000 tasks. When $\epsilon = 0.1$, the agent has a better average reward. ......................... 23
5.1 gridworld package. ............................................................. 25
5.2 GridWorld 1. ..................................................................... 27
5.3 Collaboration diagram ......................................................... 27
5.4 GridWorld 2. ..................................................................... 28
5.5 SARSA agent ($\alpha = 0.5$, $\gamma = 0.95$, $\epsilon = 0.1$) using the GridWorld 2 (see Figure 5.4) with 50 episodes averaged by 100 trials. The steady state is reached at the 13th episode. There are a few noises at the steady state. .................................................... 29
5.6 $\mathcal{E}(\alpha,\gamma)$ for SARSA agent. ......................................... 29
5.7 SARSA agent with different values of $\epsilon$-greedy exploration policy. ........................................................... 30
5.8 Results of the experiment using 6 different agents in the GridWorld 2. .......................................................... 31
5.9 Results of the model-free experiment using the GridWorld 1. .......................................................... 32
5.10 Results of model-based experiment. ...................................... 33
5.11 GridWorld 3. ..................................................................... 34
5.12 Results of the stochastic experiment for model-free algorithms. .......................................................... 35
5.13 Results of the stochastic experiment for model-based algorithms. .......................................................... 36
6.1 Sensors and actuators of the e-puck robot. .............................. 37
6.2 Model of an e-puck robot in Webots. .................................... 38
6.3 Module diagram for Webots simulation. ................................. 39
6.4 Diagram showing the self-localisation module. The camera sends the image to the self-localisation module that will calculate the position of the robot in the environment. .................................................. 40
6.5 An example of tile coding with two tilings. .............................. 41
6.6 Finite-State Machine for the Robotic Controller ..................... 42
6.7 Results of the robotic controller experiment. ......................... 44
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TD(0)</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>TD(λ)</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Q-Learning</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>SARSA</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>SARSA(λ)</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Dyna</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Prioritized Sweeping</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>RTDP</td>
<td>10</td>
</tr>
</tbody>
</table>
Listings

4.1 A $k$-armed bandit agent ................................................................. 22
4.2 A $k$-armed bandit environment ...................................................... 22
5.1 Example of XML config file ............................................................... 26
1 Introduction

A universal idea in many sciences, such as Psychology and Neuroscience, is that the brain contains multiple and distinct systems able to make decisions. Although we are far from understanding those systems completely, many experiments have shown that we are able to model specific brain functions related with learning, using simple concepts like Reinforcement Learning (RL) [33], which is essentially learning by interacting with the environment where an agent learns from the consequences of its actions [39].

Some studies have shown evidence that animals can make goal-directed decisions, merely controlled by the possible effects, rewards, of their choices [11]. Those experiments suggest that animals are not condemned to the law of effect, which states that “an action followed by reinforcement is more likely to be repeated in the future” [8]. This has led to two kinds of decision-makers as two different mechanisms for reinforcement learning: model-free [33] - because it avoids the creation of a task model and works just by reinforcing successful actions - and the model-based [34], the goal-directed system which creates a representation of task structure, an internal model, to evaluate possible actions.

Those two RLs methods try to model the learning behaviour of an animal, but which is the best one? Which method allows a machine to learn behaviour as fast as possible? One could expect that the creation of world model might accelerate the learning process, but the higher number of papers using model-free methods indicates otherwise. A model-free algorithm with eligibility traces could be as good as any model-based one. The project purpose is to determinate the best unsupervised reinforcement learning mechanism.

1.1 Aims & Objectives

The aim of this project is to simulate those two decision-making systems in a special and limited case in order to analyse both systems in the robotic navigation problem.

The Morris water maze (Fig 1.1) is widely used in neuroscience to study animal spatial learning behaviour [39]. In this experiment, an animal, usually a rat, is placed in a large circular pool of water and required to escape from water onto a hidden platform whose location cannot be visualized, there are no indication where that platform is located [39]. This experiment is performed many times, always putting the platform in the same place, but changing the animal’s starting position. At the beginning, the animal will need to explore the pool; however, eventually, it will recognize its position and will go directly to the hidden platform only using its spatial memory; it will exploit its knowledge of the particular situation. What happens if we change the platform position? Our animal will need to explore the pool again.

This project will simulate this same experiment using a robot that once in a while needs to go to a specific location in the environment which is initially unknown, such as looking for a power socket in a room to recharge its battery. This experiment can formally be modelled as a Markov Decision Process and therefore can be solved optimally. Our goal is then to be able to answer two questions. How could

Figure 1.1: Water-maze experiment. Axonometric drawing of a typical water maze set-up with overhead video camera and rat swimming to find the hidden platform [39].
we model in a robot the same learning behaviour of an animal? What is the best RL technique and algorithm for this?

Although most research in this area studies model-free algorithms, in this thesis we will explore and compare model-free and model-based techniques. A framework using Java will be created in order to easily choose the best reinforcement learning algorithms for both techniques in this particular case.

Model-based algorithms are more computationally consuming; they are more complex, because they must create a transition function between each state in the environment. This thesis will be able to answer if this complexity is justified. Are they faster at finding an optimal solution than model-free algorithms? Furthermore, analysing both methods, one can think that model-based algorithms are more able to adapt to a stochastic environment. Could a simple model-free algorithm adapt just fine to environmental changes without increasing the computational complexity?

1.2 Structure of the document

The document is organised as follows:

- Chapter 2 will introduce the basic concepts about Reinforcement Learning and the Robot Navigation problem. It includes a discussion about the relationship between Neuroscience and RLs principles.
- Chapter 3 details deeply the aims of each part of the work, explaining how the overall objectives will be achieved. It also states all the requirements of the project and analysis of the expected results.
- Chapter 4 explains the framework created to compare RL algorithms described in previous chapters and, as an example using this platform, the solution of the $k$-armed bandit problem presented in the section 2.1.2.
- Chapter 5 shows the results of a variety of experiments testing both model-free and model-based algorithms.
- Chapter 6 details the design of the robotic controller and the implementation using the best RL algorithm found at the previous chapter. The chapter ends with the result of the simulation.
- Chapter 7 summarises the achievements of the project, suggests some changes which could be made and future work.
2 Literature Review

This chapter intends to introduce the basic concepts of Reinforcement Learning and Robotic Navigation used throughout the project; it also intends to give a background of the most recent developments and studies done in the field. It has a brief discussion about the relationship between RL and neuroscience.

2.1 Reinforcement Learning Overview

According by Sutton and Barto [33], reinforcement learning is learning how to map situations to actions in order to maximise a reward. Inspired by the trial-and-error law-of-effect tradition in psychology, RL is the problem faced by an agent that must learn behaviour through interaction with a dynamic environment [15].

Although RL is a branch of Machine Learning in Computer Science, it remains one of the central concepts in contemporary work on the neurobiological basis of learned, goal-directed behaviour [14].

2.1.1 Basics Concepts

In a reinforcement learning model, an agent is connected to its environment via an input and an output system. On each iteration, the agent receives as input some indication of the current state, $s_i$, of the environment; the agent chooses an action, $a_i$, to generate an output. The action changes the state of the environment and the value of this transition is communicated to the agent as a reward, $r_i$. The agent's behaviour must choose actions to maximise the long-term sum of its rewards. It can learn how to do this over time by trial-and-error. The agent's objective is then to find an optimal policy, $\pi$, mapping states to actions. Therefore, it is necessary for the agent to gather useful experience about the possible system states, actions, transitions and rewards actively to act optimally.

In order to make things more concrete, let's see an example. A very simple and common known experiment is the Car on the Hill problem, which can be solved using reinforcement learning. The problem is defined as that of a stationary car being positioned between two hills (Fig 2.2). The goal of the driver is to drive up the hill on the right to get to the top of it. The situation is such that the car cannot just drive up the hill, the driver must learn how to use momentum to gain enough velocity to climb the hill, because the driver wants to minimise the time it takes to drive the car up to the goal state, therefore maximise his rewards. He learns to choose actions using an optimal strategy for climb the hill.

A RL model usually has the following elements [33]:

![Figure 2.1: The reinforcement learning model. For each iteration, the agent receives the current state, $s_i$, and the reward value of his last action, $r_i$. Using a policy, $\pi_i$, he must choose a new action, $a_{i+1}$.](image-url)
CHAPTER 2. LITERATURE REVIEW

Figure 2.2: The Car on the Hill problem. The agent (the driver) must learn how to use momentum to gain enough velocity to be able to climb the hill.

A State
A state is any relevant information that the agent might take into consideration in making a decision. In our previous example, the state will be the car’s position and velocity.

An Action
An action is any decision an agent might take. It causes the agent to move from one state to another. In the Car on the Hill problem, the agent can choose two different actions in each state: go on the right or go on the left.

The Policy
The agent’s policy is the mapping from possible states to possible actions. In our example, it will be the optimal strategy choosing the best actions for driving the car up the hill.

The Reward
Rewards are scalar variables associated with some states, this association defines the goal of the agent in a given situation. In the finite-horizon model, the agent should optimize its expected reward, $R_e(k)$, for the next $k$ steps:

$$R_e(k) = \frac{r_1 + r_2 + ... + r_k}{k} = E(\sum_{i=1}^{k} r_i) \quad (2.1)$$

The infinite-horizon discounted model takes the long-term reward of the agent into account, however rewards that are received in the future are geometrically discounted according to discount factor $\gamma$, where $0 \leq \gamma \leq 1$:

$$R_e = E(\sum_{i=1}^{\infty} \gamma^i r_i) \quad (2.2)$$

In this model, a reward received $k$ time steps in the future worth only $\gamma^{k-1}$ times what it would be worth if it were received immediately.

For the Car on the Hill problem, the reinforcement function would be $-1$ for all state transitions except the transition to the goal state, where the value zero is returned.

The Value Function
The value function is the agent’s current mapping from the set of possible states (or state-action pairs) to its estimates of the net long term reward to be expected after visiting a state (or a state-action pair) and continuing using the same policy.

The state-value function for policy $\pi$ can be defined formally as

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

where $R_t$ is the total reward receive in the state $s_t$. 
2.1. REINFORCEMENT LEARNING OVERVIEW

Similarly, the action-value function for policy $\pi$ is defined as

$$Q^\pi(s, a) = E_{\pi}(R_t | s_t = s, a_t = a)$$ (2.4)

The Transition Function

$T(s, a, s')$ is the probability of making a transition from state $s$ to state $s'$ using action $a$.

2.1.2 Exploration versus Exploitation

One of the first and simplest reinforcement learning problem studied is the $k$-armed bandit problem [5]. The agent is located in a room with a collection of $k$ gambling machines. In each turn, he can choose one of the arms to pull, after his choice he receive a numerical reward chosen from a stationary probability distribution. His objective is to maximize the total rewards over $n$ pulls. What’s the best strategy?

The agent must explore his environment in order to learn the best policy and then exploit his knowledge.

He might believe that one of the arms has a high pay-off probability, should he choose that arm all the time? Or should he choose another one that he has less information about, but seems to be worse? Answers to these question depend on how many pulls, $n$, the agent has. In a long run, he must explore to avoid prematurely converging on a sub-optimal arm (cf. 4.2).

The literature shows that there are many solutions to those questions, all those strategies usually have in common the greedy behaviour, where the agent would always select the action with highest value - exploitation - except when a random action is taken - exploration. Some of those strategies are:

$\epsilon$-Greedy Method

The best action is selected for a proportion $1 - \epsilon$ of the trials and another arm is randomly selected for a proportion $\epsilon$. Although $\epsilon$-greedy method is effective and popular, when it explores it chooses equally among all actions, i.e. it is as likely to choose the worst action as it is to choose the closest best action. Others variants of this method exists, like the $\epsilon$-decreasing strategy where the value of $\epsilon$ decreases as the experiment progresses, resulting in highly explorative behaviour at the start and highly exploitative behaviour at the finish.

Softmax Action Strategy

In the softmax method, actions are ranked and weighted according to their estimate values. The most common softmax method uses a Gibbs distribution. It chooses action $a$ in the $i$th play (pull) with probability

$$\frac{e^{Q_i(a)/\tau}}{\sum_{k=1}^{n} e^{Q_i(k)/\tau}}$$ (2.5)

where $\tau$ is a positive parameter called the temperature. In the limit as $\tau \to 0$, softmax action selection becomes the same as the greedy method. The best selection between softmax and $\epsilon$-greedy is unclear and depend on the task and on human factors [33].

Optimistic Initial Values

Initial action values can be used as a simple way of encouraging exploration. Whichever actions are initially selected, the reward is less than the starting estimates, therefore the agent, subject to a greedy policy, switches to others actions. The result is that all actions are tried several times before the value estimates converge. It is a quite effective approach on stationary problems.

2.1.3 Formal Problem

A general RL problem is formally modelled as a finite Markov Decision Processes (MDPs) [15]. An MDP consist of a tuple $< S, A, T, R >$.

- a set of states $S$;
- a set of actions $A$;
• a reward function $R: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$;
• a state transition function $T: \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$, where $\Pi(\mathcal{S})$ is a probability distribution over the set $\mathcal{S}$. Therefore $T(s, a, s')$ is the probability of making a transition from state $s$ to state $s'$ using action $a$.

The state transition function $T(s, a, s')$ possesses the Markov property, the conditional probability distribution of futures states of the process depends only upon the present state, not on the sequence of events that preceded it. That is:

$$T(s, a, s') = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$$ (2.6)

### 2.1.4 Bellman Equations

In order to solve a reinforcement learning problem, one must find the optimal policy given a correct model. According to Bellman, for the infinite-horizon discounted model, there exists an optimal deterministic stationary policy \cite{bellman1957rep}. The optimal value of a state is the expected infinite discounted sum of reward that the agent will gain if it starts in a state, $s$, and executes the optimal policy, $\pi$. It is written:

$$V^*(s) = \max_{\pi} V^\pi(s) = \max_{\pi} E \left( \sum_{i=1}^{\infty} \gamma^i r_{t+i+1} | s_t = s \right)$$

The optimal value function is unique and can be defined as the solution to the following equation:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a) + \gamma V^*(s') \right]$$ (2.7)

Similarly, the optimal value of the pair state-action can be defined as:

$$Q^*(s) = \max_{\pi} Q^\pi(s)$$

The optimality equation for the action-value function is:

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a) + \gamma \max_a Q^*(s', a') \right]$$ (2.8)

The Bellman optimality equations have special consistency conditions that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined with relative ease.

### 2.1.5 Learning an Optimal Policy

Reinforcement learning is primarily concerned with how to obtain the optimal policy when a model is not known in advance. The agent must interact with its environment directly to obtain information which can be processed to produce the optimal policy. There are, mainly, two different ways to solve this problem.

- **Model-free**: learning behaviour without creating a model of the environment;
- **Model-based**: learn a model, and use it to produce learning behaviour.

### 2.1.6 Model Free

Algorithms that learn to act in a Markovian domain without a fully specified model of the environment are called model-free methods. They are also the most popular methods for solving MDPs.
2.1. REINFORCEMENT LEARNING OVERVIEW

Temporal Difference Learning

Temporal difference (TD) learning is a prediction method, it learns by sampling the environment according to some policy. TD learning takes into account the fact that predictions are correlated; it adjusts predictions to match other, more accurate, predictions about the future. For this reason it is a bootstrapping method.

The value of a policy is learned using Sutton’s TD(0) algorithm [33] which uses the update rule

\[ V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s)) \]  

(2.9)

Algorithm 1 TD(0)

<table>
<thead>
<tr>
<th>Initialize ( V(s) ) arbitrarily</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
</tr>
<tr>
<td>( a \leftarrow ) action given by policy ( \pi ) in the state ( s )</td>
</tr>
<tr>
<td>Execute ( a ), observe reward ( r ) and new state ( s' )</td>
</tr>
<tr>
<td>( \delta \leftarrow r + \gamma V(s') - V(s) )</td>
</tr>
<tr>
<td>( s \leftarrow s' )</td>
</tr>
<tr>
<td>end loop</td>
</tr>
</tbody>
</table>

Each time that the state \( s \) is visited, its estimated reward value is updated to be closer to \( r + \gamma V(s') \), where \( r \) is the instantaneous reward received and \( V(s') \) is the estimated value of the next state. If the learning rate \( \alpha \) is adjusted properly and the policy is held fixed, TD(0) converge to the optimal value function.

TD(\( \lambda \))

Methods in which the temporal difference extends over \( n \) steps are called \( n \)-step TD methods. The TD(\( \lambda \)) can be understood as one particular way of averaging \( n \)-steps backup. This average contains all the \( n \)-step backups, each weighted proportional to \( \lambda^{n-1} \), where \( 0 \leq \lambda \leq 1 \). The general TD(\( \lambda \)) rule is

\[ V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s)) e(s). \]  

(2.10)

It is applied to every state according to its eligibility trace, \( e(u) \), rather than just to the immediately previous state, \( s \).

In the backward view of TD(\( \lambda \)), the eligibility traces for all state decay by \( \gamma \lambda \) and the eligibility trace for the current state is incremented by 1. The eligibility trace for state \( s \) is

\[ e(s) \leftarrow \begin{cases} 
\gamma \lambda e(s) + 1 & \text{if } s = \text{current state} \\
\gamma \lambda e(s) & \text{otherwise}
\end{cases} \]  

(2.11)

Algorithm 2 TD(\( \lambda \))

<table>
<thead>
<tr>
<th>Initialize ( V(s) ) arbitrarily</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
</tr>
<tr>
<td>( a \leftarrow ) action given by policy ( \pi ) in the state ( s )</td>
</tr>
<tr>
<td>Execute ( a ), observe reward ( r ) and new state ( s' )</td>
</tr>
<tr>
<td>( \delta \leftarrow r + \gamma V(s') - V(s) )</td>
</tr>
<tr>
<td>( e(s) \leftarrow e(s) + 1 )</td>
</tr>
<tr>
<td>for all ( s ) do</td>
</tr>
<tr>
<td>( V(s) \leftarrow V(s) + \alpha \delta e(s) )</td>
</tr>
<tr>
<td>( e(s) \leftarrow \gamma \lambda e(s) )</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>( s \leftarrow s' )</td>
</tr>
<tr>
<td>end loop</td>
</tr>
</tbody>
</table>
Q-Learning

Q-Learning is a reinforcement learning technique developed by Watkins [38] where the agent learns behaviour using the action-value function. It is usually called as an off-policy TD method, because the Q function makes the action explicit, i.e. the action can be chosen just by taking the one with the maximum Q value for the current state.

Its simplest form, Q-Learning applies the following equation to update the Q-value of a state-action pair.

\[ Q(s,a) \leftarrow Q(s,a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s,a)) \] (2.12)

Algorithm 3 Q-Learning

Initialize \( Q(s,a) \) arbitrarily

loop

Choose action \( a \) using policy (e.g., \( \epsilon \)-greedy) on the Q-values of the current state \( s \)

Execute \( a \), observe reward \( r \) and new state \( s' \)

\[ Q(s,a) \leftarrow Q(s,a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s,a)) \]

\( s \leftarrow s' \)

end loop

Q-Learning is the most popular and seems to be the most effective model-free algorithm for learning from delayed reinforcement [15], because the Q values will converge to the optimal values independent of how the agent behaves while the data is being collected. Q-Learning is popular as it is both simple and easy to implement.

Like TD(\( \lambda \)), Q-Learning can be extended to update states that occurred more than one step.

SARSA

A variation on-policy of Q-Learning is SARSA (State-Action-Reward-State-Action) algorithm. SARSA is the same as Q-Learning, except for the Q-values update function:

\[ Q(s,a) \leftarrow Q(s,a) + \alpha (r + \gamma Q(s', a') - Q(s,a)) \] (2.13)

Algorithm 4 SARSA

Initialize \( Q(s,a) \) arbitrarily

loop

Choose action \( a \) using policy (e.g., \( \epsilon \)-greedy) on the Q-values of the current state \( s \)

Execute \( a \), observe reward \( r \) and new state \( s' \)

\[ Q(s,a) \leftarrow Q(s,a) + \alpha (r + \gamma Q(s', a') - Q(s,a)) \]

\( s \leftarrow s' \)

\( a \leftarrow a' \)

end loop

Q-Learning has an infinite exploration that might cause it to take suboptimal actions. While it is usually tolerable to execute suboptimal actions, taking hazardous action may be intolerable. SARSA handles this problem by taking into consideration the exploration policy of the learning algorithm.

However shutting off exploration may be undesirable, because infinite exploration can handle environments that change over the time.

SARSA(\( \lambda \))

SARSA(\( \lambda \)) is the extended SARSA method using eligibility traces. It is one of the most common implementations of reinforcement learning with eligibility traces [29].
Algorithm 5 SARSA(\( \lambda \))

Initialize \( Q(s,a) \) arbitrarily and \( e(s,a) \leftarrow 0 \)

**loop**
- Execute \( a \), observe reward \( r \) and new state \( s' \)
- Choose action \( a \) from \( s' \) using policy from \( Q \) (e.g., \( \epsilon \)-greedy)
- \( \delta \leftarrow r + \gamma Q(s',a') - Q(s,a) \)
- \( e(s,a) \leftarrow e(s,a) + 1 \)
- for all \( s \) do
  - \( Q(s,a) \leftarrow Q(s,a) + \alpha \delta e(s,a) \)
  - \( e(s,a) \leftarrow \gamma \lambda e(s,a) \)
- end for
- \( s \leftarrow s' \)
- \( a \leftarrow a' \)
**end loop**

2.1.7 Model Based

Although model-free methods are guaranteed to find optimal policies, they make inefficient use of the data they collect and for this reason often require a great deal of experience to achieve good performance. In model based algorithms, the agent learns and creates the models \( T(s,a,s') \) and \( R(s,a) \) for the problem. Once the model has been learned, the agent can use a policy computation algorithm to calculate an optimal policy. The main idea of is that planning is “trying things in your head”, using a internal model of the world [32].

**Certainty Equivalence**

In this straightforward method developed by Kumar & Varaiya [16], the agent begins by learn the \( T(s,a,s') \) and \( R(s,a) \) functions exploring the environment and gathering information about the results of each action, and then computes an optimal policy using a model free method.

There are some inconveniences with this naive method. One important advantage of model-free methods is that they keep exploring the world, therefore they can adapt to changes in the environment. **Certainty Equivalence** does not possess this important feature. The second disadvantage is how the agent should explore the world, because random explorations usually are not the best method, it wastes time.

**Dyna**

Dyna architecture [32] is usually more effective than model-free learning and more computationally efficient than the certainty equivalence method [15].

This method, developed by Sutton [32], incrementally learns both the state-action values and the model of the environment, updating the policy and model simultaneously. It keeps all statistics for the state transition and rewards, using them to perform updates.

Dyna algorithm requires \( k \) times the computation of Q-Learning method, but this is typically vastly less than for the naive model based method. A reasonable value of \( k \) can be determined based on the relative speeds of computation and of taking action.

**Prioritized Sweeping**

The Dyna architecture continues to update random state-action pairs even when the goal has been reached or when the agent is stuck in a dead end. To contour these problems, **prioritized sweeping** was developed by Moore [21].

Similar to Dyna algorithm, in the prioritized sweeping, the updates are no longer chosen randomly, and values are associated with states instead of state-action pairs. Each state remembers its **predecessors** and has a priority, initially set to zero. Instead of updating \( k \) random state action pairs, prioritized sweeping updates \( k \) states with the highest priority. For each high priority state, \( s \), it works as explained in 7.
CHAPTER 2. LITERATURE REVIEW

Algorithm 6 Dyna

\begin{algorithm}
\textbf{Initialize} $Q(s, a), T(s, a, s'), R(s, a)$
\textbf{loop}
  \begin{itemize}
  \item Choose action $a$ using a policy and the current state $s$
  \item Execute $a$, observe reward $r$ and new state $s'$
  \item Update the models, $T(s, a, s')$ and $R(s, a)$, incrementing statistics for the transition from $s$ to $s'$ on action $a$ and for receiving reward $r$
  \item $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$
  \end{itemize}
\textbf{for} $i = 0$ to $k$ \textbf{do}
  \begin{itemize}
  \item Choose an arbitrarily state, $s_i$, and action, $a_i$
  \item $Q(s_i, a_i) \leftarrow R(s_i, a_i) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$
  \end{itemize}
\textbf{end for}
\textbf{end loop}
\end{algorithm}

Algorithm 7 Prioritized Sweeping

\begin{algorithm}
\textbf{for all} high priority state $s$ \textbf{do}
  \begin{itemize}
  \item Remember the current value of the state, $V_{old} = V(s)$
  \item $V(s) \leftarrow \max_a (R(s, a) + \gamma \sum_{s'} T(s, a, s')V(s'))$
  \item Set the state’s priority back to zero
  \item Compute the value change $\Delta V = |V_{old} - V(s)|$
  \item Use $\Delta V$ to modify the priorities of the predecessors of $s$
  \end{itemize}
\textbf{end for}
\end{algorithm}

The global behaviour of the algorithm is that when a transition is unexpected, the system propagates this new information back to relevant predecessor states, however when a transition is expected, then computation continues in the most deserving part of the space.

Real-time dynamic programming

The Real Time Dynamic Programming (RTDP) is an asynchronous value iteration algorithm developed by Sarto [3]. It is used only in domain where a set $G$ of goal states and a single start state $s_0$ exist. A state $s$ is called relevant if the agent travelling to a goal state using an optimal policy passes through this state $s$.

RTDP is another model-based method that uses Q-Learning to concentrate computational effort on the areas that the agent is most likely to occupy. Although some states may be never updated, the proof of optimality shows that the relevant states will always be updated.

Like in the others model-based methods, RTDP performs a series of simulations of interaction with the environment.

Algorithm 8 RTDP

\begin{algorithm}
$s \leftarrow s_0$
\textbf{for all} $s \not\in G$ \textbf{do}
  \begin{itemize}
  \item Choose action $a$ using a policy and the current state $s$
  \item $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$
  \item Pick state $s'$ with probability $T(s, a, s')$
  \item $s \leftarrow s'$
  \end{itemize}
\textbf{end for}
\end{algorithm}

RTDP has a good anytime behaviour; it can quickly produce a good policy and then improve it with time. However, its convergence to the optimal solution is quite slow [6].
2.2 Neuroscience and Reinforcement Learning

Behavioural experiments suggest that the animal learning system is driven by changes in the expectations about future events [28]. This matches the main RL principle which is to maximize a reward signal function.

Animals must be able to predict future events, this capacity is crucial to decide the best actions leading to food and survival. One simple prediction is the probable time and magnitude of future rewarding events. *Reward* is defined in Neuroscience as a positive value that a creature attributes to an object or a behavioural act. Just like the reward function in RL, animals can assign different appetitive values to stimulus as a function of their internal states.

2.2.1 Dopamine Neurons

Dopamine is a neurotransmitter present in animals brain. Some research shows that dopamine is involved as the main role in reward learning and animal appetite conditioning [10].

The substance is manufactured in nerve cell bodies, the *dopamine neurons*, located within the VTA and is released in the *nucleus accumbens* and the pre-frontal cortex.

Dopamine neurons are fired when an unexpected reward is presented. These responses transfer to the onset of a conditioned stimulus after repeated pairings with the reward. Further, dopamine neurons are depressed when the expected reward is omitted. Hence, dopamine neurons seem to encode the prediction error of rewarding outcomes.

Dopamine signal is therefore believed to provide a learning system responsible for acquiring new behaviour.

![Dopamine pathways on the human brain](image)

**Figure 2.3**: Dopamine pathways on the human brain [23]. Dopamine plays an important role in the brain system that is responsible for a reward-driven learning.

Many studies on the field show that the temporal difference learning (TD) has been adapted to understand the role of the dopamine signal [10, 13, 24, 28, 36]. It provides a computational model describing how the prediction error of dopamine neurons is used as a learning system.

The ventral tegmental area (VTA) and substantia nigra also have been identified as the rewarding stimuli in motivation and goal-directed behaviour [28].

2.2.2 World model

The dopamine learning system is, generally speaking, related to pure *model-free* reinforcement learning (*cf.* 2.1.6). However, the dopaminergic signal in ventral striatum does not reflect a pure TD model, some experiments suggest a model-based value prediction, this is, a more complicated computational model of the neural substrates of valuation [9].

According to Doya [12], a world model could be used to predict the next state following a candidate action, and that a dopaminergic *reward prediction error* signal with the respect to that projected state could then be used to evaluate whether the action was worth taking. This model-based use of dopamine
for planning would appear to be different in timing and content than the use of dopamine for learning, see 2.2.1.

Those results suggest that a dopaminergic reward prediction error response could be more sophisticated to model than a basic temporal difference theory would predict.

Nevertheless, the connection between an artificial system using Reinforcement Learning and Neuroscience experimental results is strong and can provide an excellent framework for future experiments.

2.3 Reinforcement Learning in Robotics

2.3.1 Overview

In recent years, there has been a lot of robotics and control applications that have used reinforcement learning. According to Mahadevan [19], robot learning behaviour is quite a hard problem, because of input noise, stochastic environment, real-time response and the generally limited amount of training time available.

Smart [30] has summarized some of the challenges of implementing reinforcement learning techniques in a real robot.

The Curse of Dimensionality

Robot input data values are usually discretized into finite sets of states and actions; however those data values are usually continuous. Incorrect discretization can be a limit to learning behaviour. It has the risk of aggregating states that do not belong together or has an unmanageable dimension in the state and action space, the curse of dimensionality.

In his PhD work, Smart [30] gives a very good example of how discretization can be a real problem when using robots. Imagine that our robot is walking in one dimension according to the Figure 2.4. Its goal is a place situated 0.5 \( m \) of the origin. The policy for this situation is quite obvious and optimal.

\[
\pi(s) \leftarrow \begin{cases} 
\text{go right} & \text{if the robot is on the left of the goal position} \\
\text{go left} & \text{if the robot is on the right of the goal position} \\
\text{halt} & \text{if the robot reaches the goal position}
\end{cases}
\]

But if we use a bad discretization, the problem can be impossible to solve; the robot will never reach its goal. For instance, if we discretize its environment into an even number of states, we will
calculate the following policy.

\[
\pi(s) \leftarrow \begin{cases} 
  \text{go right} & \text{if } s = 1 \\
  \text{go right} & \text{if } s = 2 \\
  \text{go left} & \text{if } s = 3 \\
  \text{go left} & \text{if } s = 4 
\end{cases}
\]

In this case, the robot will oscillate between state 2 and 3. A good learning method should cope with continuous state and action spaces without discretizing them.

**Learning right away**

In this project, we are interested in learning on-line, interacting directly with the environment, therefore we cannot wait until we have a large amount of training examples before starting to learn. The learning system must make good predictions based on few trainings. It must use all previous information as possible to be able to generalize similar situations.

**Dumb robot**

A real robot cannot start learning without prior knowledge. In order for the learning system be effective, some bias must be provided to give it an idea how to begin progressing towards the goal state. Unlike simulations, a real robot cannot make a huge amount, millions, of trials during the learning phase, because robot’s experiments are more time consuming. However adding insufficient and incorrect bias can also condemn the learning system to failure.

**Lazy robot**

Although computers are nowadays always becoming faster, robots usually use a simple and limited CPU\(^1\). The robot must be able to make control action choices at an appropriate rate in order to learn on-line. If it takes too long to learn a policy, it would be better to explicitly design and debug a policy with similar performance.

**2.3.2 POMDP - Partially Observable MDP**

RL problems usually assume that the environment can be perfectly observable; however, like it was already discussed (cf. 2.3.1), robot inputs are normally noisy and give limited (sometimes discretized) measurements of the environment, and therefore one cannot be sure that the information gathered is completely accurate.

The general RL problem with partial observability of the environment can be modelled by POMDP - Partially Observable Markov Decision Process. Like MDP (cf. 2.1.3), a POMDP can be formally defined as a tuple \(<\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, \Omega>\). It includes:

- a set of observations \(\mathcal{O}\);
- an observation function \(\Omega : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \rightarrow \Psi(\mathcal{O})\), where \(\Psi(\mathcal{O})\) is a probability distribution over the set \(\mathcal{O}\). Therefore \(\Omega(o, a, s')\) is the probability of observe \(o\) in the state \(s'\) after taking the action \(a\).

Adding some bias of how to interpret useful inputs for the learning system in our limited navigation experiment can allow us to model the problem using just an MDP. The use of POMDP is outside the scope of the project.

**2.4 Navigation**

In order to define the robotic navigation problem, one must first explain how to define a state on the environment. There are two ways to do this, the first one is using a discretization of the environment (a grid world) and the second one is using a continuous space state.

\(^{1}\)For instance, the E-Puck mobile robot uses a dsPIC 30 CPU, running only at 30 MHz.
2.4.1 Gridworld

In 1992, it was unclear if reinforcement learning was able to solve complicated problems, Lin [17] creates an interesting and non-trivial experiment where an agent was located in a grid world and needed to learn how to gather food, avoid enemies and obstacles. He was able to verify and test 8 different RL frameworks. His conclusion confirmed the superiority of Q-Learning over AHC-Learning \(^2\).

A gridworld is a discretization of the environment in a rectangular grid. The cells of the grid correspond to the possible states of the environment. At each cell, the agent usually has four possible actions (north, south, east and west) which cause the agent to move one cell in the corresponding direction on the grid.

![Gridworld Example](image)

Figure 2.5: A gridworld example with a goal state, G, and an agent a which has 8 possible actions.

The gridworld is extensive used in RL experiments as a representation of the environment.

2.4.2 Continuous Space

Robot navigation is a complex system which includes environment self-localisation, dynamical decision making and actions. Using a classical reinforcement learning algorithm usually leads to the problem of dimension explosion, because most RL techniques use finite state and action spaces. To get around this defect, one can bind together RL and Artificial Neural Networks (ANNs) [25].

A neural network is a mathematical model inspired by the circuit of biological neurons on the animal brain. It is usually an adaptive system that changes its structure based on external data; it can model arbitrarily complex functions and can make predictions quite efficiently. We can see examples of RL and ANN together in many earlier papers (Arleo and Gerstner 2000 [1] and 2004 [2]; Carreras 2007 [7]; Ma 2007 [18]; Tan 2008 [35];).

Other techniques are used to overcome this classical difficulty combining reinforcement learning with others techniques, like decision trees [37], CMAC function approximator [26] and memory-based methods [30].

2.5 Neuroscience and Navigation

There are many theories how animals are able to handle the navigation problem. This section presents two simple methods.

2.5.1 Place Cells

The hippocampus\(^3\), has been studied as part of the brain system responsible for spatial memory and navigation. Animals usually create a spatial representation of the environment to be used as a cognitive basis to support navigation. Many studies suggest that Place Cells and Head Direction Cells neurons located in the hippocampus, are responsible for mapping the animals location.

---

\(^2\)Actor-Critic Method Learning

\(^3\)The hippocampus is a major component of the brain which belongs to the limbic systems and plays important roles from short-term memory to long-term memory and spatial navigation.
Place cells are neurons exhibiting a high rate of firing whenever an animal is in a specific location in the environment. Artificial place cells can be constructed using a Hebbian learning algorithm. They are usually used with head direction cells which are neurons firing when the animal’s head is oriented in a specific direction.

2.5.2 The Hippocampus Model

Arleo and Gerstner [1] proposed a computation model of the hippocampus to study its role in spatial cognition and navigation. Using Gabor filters to simplify the raw information, they were able to use visual cues in order to create a hippocampal space representation applying an Unsupervised Hebbian learning algorithm building the neural system incrementally and on-line. The proposed hippocampal architecture incorporates the knowledge about the obstacles and target locations and allows the robot to self-localise within its environment.

A navigational map can be derived applying reinforcement learning methods to map the ensemble activity into goal-oriented behaviour. Synapses from the hippocampal place cells to action cells are modified in order to learn the continuous location-to-action mapping functions, and therefore the robot can be able to associate appropriate actions to spatial positions even if it has never be visited before. In their experiment, Arleo and Gerstner [1], used a linear gradient descent version of Watkins’ Q-Learning algorithm (cf. 2.1.6). They created an action-value function over a continuous location space.

\[
Q(s, a) \leftarrow Q(s, a) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a) - Q(s_t, a)) e_t
\]

The eligibility, \(e_t\), is updated using the following rule

\[
e_t \leftarrow u(s_t) + \begin{cases} 
\gamma \lambda e_{t-1} & \text{if exploiting} \\
0 & \text{if exploring}
\end{cases}
\]

where \(\lambda\) is the trace-decay parameter and \(u(s_t)\) is the hippocampal vector activity calculated using the hippocampal model.

The experiments have been carried using a reward function \(R(s)\) defined by

\[
R(s) \leftarrow \begin{cases} 
1 & \text{if } s = \text{target state} \\
-0.5 & \text{if } s = \text{otherwise} \\
0 & \text{otherwise}
\end{cases}
\]

The result of their project is an ensemble activity of neurons that provides a navigational map [1].

2.6 Summary

This chapter has introduced the basic concepts about Reinforcement Learning and the Robot Navigation problem. It also included a discussion about the relationship between Neuroscience and RLs principles.
3 Requirements and Analysis

3.1 Requirements

The objective of this project is to compare some reinforcement learning algorithms discussed in the previous chapter, both model-free and model-based techniques, simulating a water-maze experiment using a robot that can learn from its experience acquired during the interaction between the environment.

The overall objective was broken down into a few milestones divided in two main parts. The first part concerns the study of both reinforcement learning methods and the investigation of the relationship between RL and the animal way of thinking. Ending by the implementation of some RL algorithms to solve simple problems like the $k$-armed bandit problem, in order to assure that both methods are fully understood. The second part of the project concerns the comparison of those algorithms using the previous framework and the development of a robotic controller for the E-Puck mobile robot using a robot simulation environment, Webots.

- First part
  1. The study of reinforcement learning methods and the investigation of the relationship between RL and neuroscience. This task will also concern the literature review about RL and robotic navigation.
  2. Design and implementation of a framework in Java in order to develop and understand the logic behind reinforcement learning principles. Using this framework, it will be much easier to test and compare algorithms.
  3. Development of the GridWorld as an array of location (states) modelling the robot environment. This will be used to test and compare some algorithms in order to choose the best option for model-free and model-based technique for future tasks. Performance will be measured by the number of interactions required to learn how to find the goal-state. State space can also be modified, making changes to the environment, like changing the goal-state position and adding walls as obstacles.

- Second part
  1. Development of a simple and limited self-localisation algorithm (module) discussed in the previous chapter using special patterns around the environment to simplify the problem. The implementation of a general case is beyond the scope of this thesis, because it demands further research and the development of a neural network, which would be too time consuming.
  2. The experiment will then be simulated using robot modelling software Webots. Both algorithms selected in part 1 will be used. We will then be able to analyse and compare both techniques and extract the best option for this robot navigation problem.
  3. If time permits, the same experiment will be made using a real robot, E-Puck mobile. We will discuss then the difficulties of implementation in a real robot and the differences found between the real experiment and its simulation.

In the first part, we expect to extract the best algorithms for model-free and model-based RL techniques for the robotic navigation problem. The best algorithm will be measured in terms of the number of interactions required to learn the optimal path to find the goal-state and its capability to adapt to changes in the environment.
3.2. ANALYSIS

Subsequently, Webots will be used as a simulator environment for the robot; nevertheless the Framework and Gridworld developed in the first term will be used as a decision maker module. Figure 3.1 shows a diagram explaining the interaction between all modules.

Figure 3.1: Module diagram for Webots simulation. The camera input is sent to the Self-Localisation Module, where the correct position is calculated. The position is then updated at the Odometry Module. Using a function approximator, Tile-Coding Approximation, the current state is then calculated and sent to the RL-Framework that will choose a new action.

3.2 Analysis

Analysing the project, before any experiments, one can expect that SARSA(\(\lambda\)) could be a good example of a model-free algorithm for robotic navigation, because of the simplicity of implementation and a large number of papers using this technique; however, if a stochastic environment is used, a model-based algorithm is more likely to adapt, since it has a model of the world, a function with the probability of making a transition from one state to another using a given action.

Therefore, we expect to find similar results for model-free with eligibility traces and model-based when the environment is fixed and unchangeable, but a better result for model-based in a world plausible to change. Finally, we could say that due to the calculation consuming model-based technique, if the difference in time for this environment adaptation is unimportant, model-free could be the best technique for robotic navigation.
4 Reinforcement Learning Framework

In order to compare different Reinforcement Learning algorithms presented in the previous chapters, an independent framework was created using object oriented principles and patterns. The key principle here was to encapsulate what really changes (the agent and the environment), so later one can alter it without affecting what it does not change (the interaction between the agent and the environment, as well as the verbose of the result of each experiment). The goal was to create a framework that could be used outside the scope of this project. The code in Java and the .jar file are available under GPL\textsuperscript{1} on the Internet at github.com/vborgesfer.

In this project, the framework will be used to implement and analyse the algorithms presented in the previous chapter 2.

4.1 High Level Packages

The framework was designed to be very simple to use. It contains only two packages, the first one contains objects, or tools, that help to develop a reinforcement learning algorithm. The second package contains the core of the framework, such as the interfaces of an agent and an environment.

4.1.1 tools package

The first package, tools, contains all functions that one may need to develop a Reinforcement Learning algorithm, like the value functions, $V(s)$ and $Q(s, a)$; the transition function, $T(s, a, s')$; and the reward function, $R(s, a)$. Those classes model an ordinary mathematical function that uses a State and an Action as input and returns a double as output.

![tools package diagram](image)

Figure 4.1: tools package. $V$ and $Q$ are classes representing value functions. $T$ is the transition function and $R$, the reward function. Those classes uses State and Action objects as input. IValueFunction is an interface that allows the user to create its own value function.

The value functions also contain some strategies to choose an action that can be used when implementing the agent such as:

- Greedy method
- $\epsilon$-Greedy method

\textsuperscript{1}General Public License
4.1. HIGH LEVEL PACKAGES

- Softmax Action Strategy

*State* and *Action* are simple objects which must contain a unique identifier to safely be used as an input for value functions.

One may also want to inherit a *State* to create its own state object. For instance, in the grid world problem, a state can be described by two integers, x and y positions. The same thing might be done with *Action* in order to customize the action object needed to the current problem.

This package also offers an interface to create user’s value functions, *IValueFunction*, easily coupled with the framework.

### 4.1.2 main package

The second package, *main*, contains the keys of this framework, where one has the choice to use an implemented *Agent* or develop its own.

**Interfaces**

The package contains two interfaces, *IAgent* and *IEnvironment*, used to implement an agent and an environment object respectively. One can also use the abstract class *Agent*, where some default functions are already developed.

The majors methods when implementing an agent are:

- *init* is responsible to initialize all variables (such as value functions and environment model estimations) to Agent implementation.
- *chooseAction* is the principal way of interaction with the environment. It receives as parameters the current state and the reward due to the last action and waits in return for the next action.

Some other methods are used to set the possible actions for the agent and returns information about the agent, such as cumulative reward and step count. Others methods are used to gather information of the agent and save in a CSV file. *getInitVerbose* is called just once per experiment after the Agent initialisation. *getVerbose* is called in each iteration after the next action was chosen.

<table>
<thead>
<tr>
<th>rlf::main::Agent</th>
<th>Agent -&gt; IAgent</th>
</tr>
</thead>
<tbody>
<tr>
<td>episode : int</td>
<td>&lt;&lt;realize&gt;&gt;</td>
</tr>
<tr>
<td>step : int</td>
<td>rlf::main::IAgent</td>
</tr>
<tr>
<td>cumulativeReward : double</td>
<td></td>
</tr>
<tr>
<td>actionList : ArrayList</td>
<td></td>
</tr>
<tr>
<td>valueFunction : ValueFunction</td>
<td></td>
</tr>
<tr>
<td>newTrial() : void</td>
<td></td>
</tr>
<tr>
<td>init(initialState : State) : Action</td>
<td></td>
</tr>
<tr>
<td>chooseAction(s : State,r : double) : Action</td>
<td></td>
</tr>
<tr>
<td>end(s : State,r : double) : void</td>
<td></td>
</tr>
<tr>
<td>setActionList(actions : ArrayList) : void</td>
<td></td>
</tr>
<tr>
<td>getReward() : double</td>
<td></td>
</tr>
<tr>
<td>getEpisode() : int</td>
<td></td>
</tr>
<tr>
<td>getStep() : int</td>
<td></td>
</tr>
<tr>
<td>getValueFunction() : IValueFunction</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3: *Agent* class. UML object diagram showing the relationship between an *Agent* and the interface *IAgent*. 

Figure 4.2: The GridState. It is fully represented by two integers indicating the position at the grid.
Similarly, the IEnvironment interface contains the following methods:

- **init** is responsible for initializing the environment and returning the initial state.
- **doAction** receives the action made by the agent, changes its state and returns the corresponding reward.
- **getCurrentState** returns the current state of the agent in the environment.
- **isEpisodeOver** returns a boolean indicating whether or not the agent has arrived at the goal state or the step count has reached its maximum.

### Implemented Agents

The following classes are the implementation of different reinforcement learning algorithms, here called as different agent objects.

- **ModelFree** is an abstract class that models all model-free algorithms, it models the temporal difference equation, where the reinforcement is the difference between the ideal reward prediction and the current prediction.
- **Sarsa** implementation of SARSA algorithm.
- **SarsaLambda** implementation of SARSA algorithm with eligibility traces.
- **QLearning** implementation of QLearning algorithm.
- **QlearningLambda** implementation of QLearning algorithm with eligibility traces.
- **ModelBased** is an abstract class that models all model-based algorithms, it contains a model object of the environment.
- **Dyna** implementation of Dyna Architecture developed by Sutton [32].
• **DynaQ** implementation of Dyna Architecture using QLearning method as value evaluation.

• **PSweeping** similar implementation of Dyna Architecture, but the values updates of each states are chosen by priorities.

### Interaction

Another important class in the *main* package is the *Simulator*. The object instantiated is a thread instance responsible for simulating the interaction between the agent (an `IAgent`) and the environment (an `IEnvironment`). In order to receive information about the simulation, one must set the log’s message queue (a `java.util.concurrent.BlockingQueue` object).

It is also possible, when using the Framework, to create its own simulator; however, one must guarantee, for each trial\(^2\), the sequence of methods calls illustrated in figure 4.6.

\(^2\)Trial, in this context, means a sequence of episodes where the agent is *learning* some behaviour. An episode is a sequence of actions until the goal (or a maximum number of actions.)
4.2 $k$-Armed Bandit Simulation

As an example of the use of the Reinforcement Learning Framework designed in the previous section, the following codes model the $k$-armed bandit problem with $k = 10$ presented in the chapter 2 (cf. 2.1.2).

```
package examples.bandit;
import rlf.main.*/;
import rlf.tools.*/;
public class BanditAgent implements IAgent {
    protected int k; // k value
    protected Q q; // Action−state function
    protected int [] t; // Table of rewards
    protected Action pa; // Previous action

    public BanditAgent(int k) {
        this.k = k;
        a = 0.1;
    }

    @Override
    public void init(State state) {
        q = new Q();
        pa = new Action(0);
        t = new int[k];
        for (int i = 0; i < k; i++) {
            q.set(state, new Action(i), 0);
            t[i] = 0;
        }
    }

    @Override
    public Action chooseAction(State s, double r) {
        // Update value
        double v = q.get(s, pa);
        v = (t[pa.toInt()] * v + r) / (t[pa.toInt()] + 1);
        q.set(s, pa, v);
        t[pa.toInt()]++;

        // Choose action
        pa = q.getGreedyAction(s);
        return pa;
    }
}
```

Listing 4.1: A $k$-armed bandit agent

```
package examples.bandit;
import java.util.Random;
import rlf.main.*/;
import rlf.tools.*/;
public class BanditEnvironment implements IEnvironment {
    protected R r; // Rewards values
    protected State s; // Current state
    protected int k; // k value

    private Random random = new Random();

    public BanditEnvironment(int k) {
        this.k = k;
    }

    @Override
    public State init() {
```

Listing 4.1: A $k$-armed bandit agent
A very simple method was used to estimate the values of actions by averaging the rewards actually received when the action was selected, i.e., if at the $t$th play action $a$ has been chosen $n_a$ times prior to $t$, then its value is estimated to be

$$Q_t(a) = \frac{r_1 + r_2 + ... + r_{n_a}}{n_a}$$

We were able to create a test-bed using a set of 1000 pseudo-randomly generated 10-armed bandit tasks. For each action $a$, the rewards were selected from a normal probability distribution $\mathcal{N}(0,1)$ plus the true value of action $a$, $Q(a)$. Averaging over the 1000 tasks, we can plot the figure 4.7. It compares a greedy method with two $\epsilon$-greedy methods ($\epsilon = 0.01$ and $\epsilon = 0.1$).

The greedy method performance is considerably worse in the long-term, even if it can be slightly faster at the very beginning. It achieved an average reward of 1, compared with 1.3 for $\epsilon = 0.1$ and 1.2 for $\epsilon = 0.01$. Hence, a good action selection rule in this case is to use the 10%-greedy method.

The $\epsilon$-greedy methods usually performs better because they continue to explore, and then to improve their chances to not get stuck in suboptimal actions. It would be possible to reduce $\epsilon$ over time so that the agent explores less when it finds the optimal action.
4.3 Summary

This chapter has presented the framework created in order to develop and test reinforcement learning algorithms. The framework is generic enough to be used outside the scope of this project and it is available under GPL on the Internet at github.com/vborgesfer.
5 Algorithms comparison

5.1 Gridworld

This project aims to compare Reinforcement Learning algorithms in the robotic navigation problem, in order to use our framework to solve that problem, we must implement an IEnvironment, called here as GridWorld.

A GridWorld is a simple environment. It is a 2D maze delineated by grids, where one grid can be either empty or a wall.

A new Java package was designed with the following classes:

- GridState is a child of State representing a position in the grid.
- GridActions contains all possible actions for an agent in this environment.
- GridReward is a static class containing all rewards values that an agent may receive. 100 for goal state, -1 otherwise.
- Maze is the object representing the grid, such as the position of the walls. It contains some methods to create random mazes, like a recursive backtracker DFS \(^1\).
- GridWorld is the implementation of IEnvironment. It contains a Maze object; the start and goal positions; and the \(R(s,a)\) function which uses the values stored in the GridReward class.

5.2 Automation

In order to test each algorithm effectively, one must test it in a variety of different environments, ranging from a simple environment (without walls for instance) to a complex one (e.g., with stochastic goal states). We must also be able to test the same algorithm in the same environment with different parameters, such as the reward discount factor, the learning rate, and so on.

The most effective way of solving this problem is to automate the process. In order to handle this, the markup language XML was used. A DOM\(^2\) parser was created to read from an XML file all parameters for the experiment, such as the algorithm used, its parameters, the size and format of the GridWorld, etc.

\(^1\)Depth-First Search

\(^2\)The Document Object Model (DOM) is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents.
An example of this XML config file can be seen at the listing 5.1, where the Dyna algorithm was chosen with the following parameters:

- Learning rate, $\alpha = 0.1$
- Reward discount factor, $\gamma = 0.95$
- Epsilon parameter for e-greedy policy, $\epsilon = 0.9$
- $k$ (Number of state-action updates for each step) = 50

```
<?xml version="1.0" encoding="utf8"?>
<essay id="1">
  <maxSteps>1000</maxSteps>
  <episodes>50</episodes>
  <trials>30</trials>
  <startState x="0" y="2"/>
  <goalState x="8" y="0"/>
  <agent type="Dyna">
    <alpha>0.1</alpha>
    <gamma>0.95</gamma>
    <epsilon>0.1</epsilon>
    <k>50</k>
  </agent>
  <environment id="2">
    <width>9</width>
    <height>6</height>
    !-- Walls: -->
    <maze>
      E: 1
      N: 2
      W: 4
      S: 8
      -->
      6 2 10 2 2 2 3 7 7
      4 1 7 4 0 0 1 5 5
      4 1 5 4 0 0 1 13 5
      4 1 13 4 0 8 0 2 1
      4 0 2 1 15 4 0 1
      12 8 8 8 10 8 8 9
    </maze>
  </environment>
</essay>
```

Listing 5.1: Example of XML config file

But the file also define the GridWorld and others useful parameters.

- **maxSteps** is the max number of steps allowed per episodes;
- **episodes** is the number of episodes per trial;
- **trials** is the number of trials per experiment, the rewards values per episode are averaged by the number of trials;
- **startState** is the start state of the agent, in this case at the position (0, 2) at the GridWorld;
- **goalState** is the goal state, for instance the position (8, 0).

The figure 5.2 shows the GridWorld generated by the given XML file.
5.3. GATHERING RESULTS

Figure 5.2: Gridworld 1, represented by the XML file from the listing 5.1. The black sphere represents the agent; the red square, the goal state. Each state is represented by a rectangle pointing out to the greedy action according to the value function. The rectangles on the middle of the grid world are obstacles (walls).

5.3 Gathering results

A last package must be created in order to gather information about the simulation of an experiment; however the design of this package is not relevant for the project. It will be created to easily see what’s happening during the simulation and be able to plot a graph that will be used to compare the algorithms.

When using the framework, a `BlockingQueue` must be given to the `Simulator` thread, c.f. 4.1.2. In each message in this queue, we have the necessary information about each step made by the agent along with the reward received. In our application, two processes will be created in order to read the information. One process will read the message and display it in a log window, which can be save it in a file after. The second one will be use it to display the current state of the simulation. It will draw the grid world and show the current position of the agent, along with a representation of its value function, see figure 5.2.

Figure 5.3: Collaboration diagram, showing the 3 threads presented in the application. The `Simulator` puts a new message in the queue indicating the values of each steps, such as: action, reward and state. The `LogWindow` reads the message, stores the values in an array and displays them on the screen. The array can then be used to plot a graph. The `SimulationFrame` reads the value functions status asynchronously from the agent and displays the image representation of the experiment.
CHAPTER 5. ALGORITHMS COMPARISON

In the end, with the information gathered, we will be able to plot a graph showing the number of average steps per episode or the average reward per episode. The application has then three threads and theirs relationships can be seen on the collaboration diagram shown in figure 5.3.

5.4 Choosing the best parameters

Let $E : \mathbb{P} \rightarrow \mathbb{N}$ be a function from the set of algorithm parameters $\mathbb{P}$ to the minimum number of episodes to reach the optimal path, that is, to reach a steady state. In this section, we want to find the minimum $E$, for all parameters $\mathbb{P}$ to a specified agent. In practice, this function works like an error function, the smaller it is, the better it is.

There are many ways to calculate the best parameters of an algorithm or function, such as Direct Line Search, Gradient Method or Genetic Algorithms. However, they usually depend on the initial guess (the seed) and are only able to calculate the local minimum.

Nevertheless, in this project, the parameters are usually bounded between 0 and 1 which permit us to calculate, in brute-force, all possible combinations with a tolerance of 0.05. That is, calculate the function $E(\mathbb{P})$ for all parameters multiple of 0.05.

5.4.1 Without eligibility traces

Let’s suppose a simple case of the grid world problem using a SARSA agent, which has two important parameters:

- $\alpha$ - the learning rate
- $\gamma$ - the reward discount factor

![Graphical representation](image.png)

Figure 5.4: GridWorld 2. Sample maze for testing and choosing optimal parameters. This is the simplest 4x4 grid world, without any obstacles.

Using the labyrinth shown in the picture 5.4, where the black circle is the agent and the red square is the goal state, one can draw the graph 5.5 for $\alpha = 0.5$ and $\gamma = 0.95$. It shows us that the agent attends the steady state close to the 13th episode, $E(0.5, 0.95) = 11$. But how could we be sure that this is the minimum, i.e., the parameters are optimal?

As explained, one could calculate all combinations of $\gamma$ and $\alpha$ for steps of 0.05. Doing that, we can plot the graph 5.6, which shows us that there are 8 minimums (10 episodes) in this case, which are the following pairs $(\gamma, \alpha)$:

$$
\mathcal{S} = \{(0.85, 0.6), (0.75, 0.75), (0.85, 0.75), (0.65, 0.8), (0.7, 0.8),
(0.75, 0.8), (0.85, 0.8), (0.8, 0.85), (0.75, 0.9), (0.65, 0.95)\}
$$

, where $E(\mathcal{S}) = 10$.

As we expected, the reward discount factor must be close to 1, for $\gamma$ less than 0.45, the steady state is never reached independently of $\alpha$. 
5.4. CHOOSING THE BEST PARAMETERS

Figure 5.5: SARSA agent ($\alpha = 0.5, \gamma = 0.95, \epsilon = 0.1$) using the GridWorld 2 (see Figure 5.4) with 50 episodes averaged by 100 trials. The steady state is reached at the 13th episode. There are a few noises at the steady state.

If $\alpha$ and $\gamma$ are both close to 1, once again the steady state is never reached, because the reward discount factor is too high, then the agent never forgets the past and as the learning rate is also high, the past is very important to the present. One bad episode can put the $Q$ values too low and be difficult to forget. After all, the new $Q$ value depends on the product $\alpha \gamma$.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s,a)Q(s,a))$$

5.4.2 With eligibility traces

For SARSA($\lambda$), a new parameter $\lambda$ must be added that can influence the other two parameters. In order to minimize the number of possible triples ($\alpha$, $\gamma$, $\lambda$), one can state the following:

- $\gamma > 0.5$
- $\lambda > 0.5$
CHAPTER 5. ALGORITHMS COMPARISON

After all, $\lambda$ is used to average n-step backups, each weighted proportional to $\lambda^{n-1}$. If $\lambda < 0.5$, the average will be too close to the last value, then the algorithm will work as without eligibility traces.

Unfortunately, it is not possible to plot this 4D graph, but we are still able to find the optimal parameters:

\[ \alpha = 0.45, \gamma = 0.95, \lambda = 0.9 \]

\[ E(0.45, 0.95, 0.9) = 3 \]

Using the same strategy shown in this section, one can calculate the best parameters for all algorithms, which may depends on the environment too. The only parameter that is not bound is $k$, the number of state-action updates for each step using the model of the environment, in Dyna agent. The section ?? shows how it was calculated.

Using the same simple grid presented in the beginning of this section, we are able to find the optimal parameter for different algorithms. The table 5.1 shows the best parameters and the correspondent $E$ value for 6 different algorithms. The figure ?? presents the corresponding results of each experiment.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$\alpha$</th>
<th>$\gamma$</th>
<th>$\lambda$</th>
<th>$k$</th>
<th>$\theta$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SARSA</td>
<td>0.65</td>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>SARSA($\lambda$)</td>
<td>0.45</td>
<td>0.95</td>
<td>0.90</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Q-Learning</td>
<td>0.65</td>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Dyna</td>
<td>0.10</td>
<td>0.95</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Dyna-Q</td>
<td></td>
<td></td>
<td>50</td>
<td></td>
<td>1.0</td>
<td>2</td>
</tr>
<tr>
<td>Prioritized Sweeping</td>
<td>0.40</td>
<td>0.95</td>
<td>50</td>
<td>1.0</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.1: Review of optimal parameters

5.4.3 Policy

As shown in the reinforcement learning review, all algorithms depend on the policy used, which can be an $\epsilon$-greedy method, a Softmax Action Strategy, etc.

In all algorithms used in this project, we have set a policy of $10\%$-greedy, which is perfectly fine for our purpose; the graph 5.7 shows the same experiment (SARSA with $\alpha = 0.5$ and $\gamma = 0.95$) for $\epsilon$ equal $5\%$, $10\%$ and $20\%$.

Figure 5.7: SARSA agent with different values of $\epsilon$-greedy exploration policy. The $\epsilon = 0.10$ was chosen to permit the agent to explore a little bit more the environment without compromising the steady state value.
5.4. CHOOSING THE BEST PARAMETERS

(a) Model-free algorithms

![Graph showing steps vs. episodes for model-free algorithms](image)

(b) Model-based algorithms

![Graph showing steps vs. episodes for model-based algorithms](image)

Figure 5.8: Results of the experiment using 6 different agents in the GridWorld 2 along with their optimal parameters. As expected, model-free algorithms without eligibility traces (SARSA and Q-Learning) have the worst performance. Nonetheless, in this simple environment, the algorithm with eligibility traces, SARSA(\(\lambda\)), had as good a performance as any other model-based algorithm.
5.5 Model-free

In order to choose the best model-free algorithm for our purpose, another labyrinth, more sophisticated was chosen. The picture 5.4 shows the start state, on position (0,2), and the goal state, illustrated as a red square, on (8,0).

Using the same method shown in the previous section to choose the optimal parameters, it was possible to compare 3 different model-free algorithms: SARSA, SARSA(\(\lambda\)) and Q-Learning.

![Graph showing results](image)

Figure 5.9: Results of the model-free experiment using the GridWorld 1 presented at the figure 5.2, where the best parameters are used for each algorithms. SARSA(\(\lambda\)) is by far the fastest algorithm.

The results are shown on the graph 5.9. As expected, SARSA and Q-Learning have approximately the same appearance, however, the latter was a little bit faster than the former. Nevertheless, as predicted, SARSA(\(\lambda\)) is the best mode-free algorithm, due to the eligibility traces.

5.6 Model-based

Using the same labyrinth as in the previous section, three model-based algorithms were tested in this section:

- Dyna
- DynaQ
- Prioritized Sweeping

In order to choose the optimal parameters for those algorithms, it is not possible to use the same method as before, because \(k\) is not bound. However, it is acceptable to assume that the big \(k\) is, the small \(\mathcal{E}(k)\) will be; and \(k\) will not influence the others parameters, since it is just the number of updates before choosing an action.

A reasonable value of \(k\) can be determined based on the relative speeds of computation and of taking action. The picture 5.12a shows different value of \(k\): 0, 5 and 50. When \(k = 0\), we have a model-free algorithm, since there is no update using the model. According to the picture, this is the slowest algorithm. It takes about 25 episodes to reach the steady state, instead of 5 episodes for \(k = 5\) or only 3 episodes for \(k = 50\).

The graph 5.10 shows 3 model-based algorithms for \(k = 50\). They have similar results, with Dyna-Q being slightly worst than the others. Dyna and Prioritized Sweeping have the same \(\mathcal{E}, \mathcal{E} = 5\). Nevertheless, Prioritized Sweeping had a faster decreasing rate at the beginning. In the third episode, the number of steps to reach the goal state was already smaller than 100.
Figure 5.10: Results of model-based experiment. Figure (a) shows Dyna algorithm using 3 different values of $k$; the biggest $k$ is, the fastest the algorithm reaches the steady state. Figure (b) shows the results of the model-based experiment using the GridWorld 2 presented at the figure 5.2, where the best parameters are used for each algorithms. The three algorithms had a similar performance, with Dyna-Q being slightly worst than the others.

### 5.7 Stochastic Environment

This section tries to answer the question: What does happen when the environment works stochastically? Which agent will adapt to a new environment?

The problem caused by an environmental change is illustrated in the picture 5.11. Initially, the goal state is on the right side, marked with a red square, of the maze, but after 50 episodes, it abruptly changes to the left. The agent will have then 2 options, continue to go to the right, trying to reach the first goal state and then accidentally reach the second state or find out that the real optimal path is turning left.

The Figures 5.12 and 5.13 show the results of this experiment. As we can see, all algorithms were able to find the optimal path after a while. The model-based algorithms had, as expected, less problems to find it; it takes the same amount of time (steps) to find the second goal state. Meanwhile, the model-free algorithms without eligibility traces took a little bit more time to re-adapt to the second goal state, 20-30% more time than usual when the value function is empty. This was already expected, since, without
eligibility traces, the information about the goal state is propagated only between neighbours states. Nevertheless, SARSA(\(\lambda\)) adapted better than expected, as fast as any other model-based algorithm.

According to the graph, however, the best algorithm is definitely Dyna. That was not what we were expecting, since Prioritized Sweeping would allow the model to avoid performing dummy updates, only the real changes would be updated. Therefore, we were expecting to have better results.

### 5.8 Discussion

The model-based algorithms were expected to have better results than model-free ones, but we are hoping, due to the number of researches using the latter, that that the former would be just slightly better in a simple environment and much faster in a stochastic one. However, our experiment show that that model-based algorithms can be 100% faster than the best model-free algorithm, SARSA(\(\lambda\)).

The results show that, in the simple environment of the figure 5.2, the number of episodes needed for a model based algorithm to reach the steady state is almost 3 times bigger (12 episodes instead of 4). For a simple SARSA agent, it is 10 times more (42 episodes). Furthermore, we expect that this number can even be bigger than this in a more complex environment, after all in the simplest environment possible, SARSA(\(\lambda\)) had the same performance than a model-based agent, c.f. Figure 5.5.

The stochastic environment experiment shows us that model-based is definitely a better choice to re-adapt to the environment changes, with Dyna agent being the best one.

### 5.9 Summary

This chapter has shown three different experiments in order to find the best reinforcement learning algorithm. The results show how individual parameter changes can alter the learning algorithms, leading to a non-optimal solution. The best algorithm was measured in terms of the number of interactions required to learn the optimal path to find the goal-state and its capability to adapt to changes in the environment. According to those experiments, the model-based algorithm Dyna was chosen as the best one.
Figure 5.12: Results of the stochastic experiment for model-free algorithms. All algorithms were able to find the optimal path to the second goal state. SARSA($\lambda$) was the only one that did not have problems to re-adapt to the new environment, the new goal state. Figure (b) and (c) show a zoom of the interesting parts of graph (a).
Figure 5.13: Results of the stochastic experiment for model-based algorithms. Like expected, all algorithms were able to find the optimal path to the second goal state and none of them had problems to re-adapt to the new environment. Figure (b) and (c) show a zoom of the interesting parts of graph (a).
6 Robotic Controller

This chapter discusses the last part of the project. The development of a robotic controller for an E-Puck mobile robot using the reinforcement learning algorithms shown in the previous chapters, in order to simulate the Morris water maze experiment.

The development faced many challenges with using reinforcement learning methods, such as large state space, uncertain state and self-localisation problem.

The chapter starts with a brief introduction of the tools used: the E-Puck mobile robot and Webots; followed by the design and implementation of the controller. The results and discussion about the experiment ends the chapter.

6.1 E-Puck, Mobile Robot

The E-Puck is a small differential wheeled mobile robot designed by Dr. Francesco Mondala and Michael Bonani for education and research purpose. This robot was chosen for many reasons, such as:

- On-board video camera, which aims to be used for the self-localisation module;
- Open hardware and on-board software open source;
- Machine Learning Laboratory at the University of Sheffield has one E-Puck, which could be used to simulate this experiment using the real robot;
- Its characteristics are predefined at Webots.

The E-Puck robot uses a dsPIC processor which is a micro-controller processor produced by the Microchip company capable of performing efficient signal processing. It has a lot of sensors and actuators.

![Image of E-Puck robot with labels for sensors and actuators]

Figure 6.1: Sensors and actuators of the e-puck robot.

Its technical details are listed at the table 6.1.

As explained before, the experiment aims to use the robot colour camera as a visual input source to be able to self-localise in its environment. However, the infrared proximity will also be used to easily verify the existence of a wall. The others features, such as accelerometer, microphone and loudspeaker will not be used.
CHAPTER 6. ROBOTIC CONTROLLER

Technical information

<table>
<thead>
<tr>
<th>Diameter</th>
<th>70 mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>50 mm</td>
</tr>
<tr>
<td>Weight</td>
<td>200g</td>
</tr>
<tr>
<td>Processor</td>
<td>dsPIC 30F6014A @ 60 MHz (~15 MIPS)</td>
</tr>
<tr>
<td>Memory</td>
<td>8 KB RAM; 144 KB FLASH</td>
</tr>
<tr>
<td>Autonomy</td>
<td>5Wh LiIon battery about 2 hours moving</td>
</tr>
<tr>
<td>Motors</td>
<td>2 step motors</td>
</tr>
<tr>
<td>Max speed</td>
<td>15 cm/s</td>
</tr>
<tr>
<td>IR sensors</td>
<td>8 infrared proximity and light (TCRT1000)</td>
</tr>
<tr>
<td>Camera</td>
<td>VGA colour camera, 640x480</td>
</tr>
<tr>
<td>LEDs</td>
<td>8 LEDs in ring + one body LED + one front LED</td>
</tr>
<tr>
<td>Microphones</td>
<td>3 omni-directional microphones</td>
</tr>
<tr>
<td>Speaker</td>
<td>1 loudspeaker capable of WAV and tone sound playback</td>
</tr>
<tr>
<td>Wireless</td>
<td>Bluetooth for robot-computer wireless communication</td>
</tr>
</tbody>
</table>

Table 6.1: Technical details of the E-Puck mobile robot [20]

6.2 Webots, Robot Simulator

Webots is an efficient development environment (IDE) used to program a robotic controller and simulate it in a defined environment. Developed by the Swiss Federal Institute of Technology in Lausanne, it is now used by over 750 universities and research centres in the world.

![Webots model](image)

Figure 6.2: Model of an e-puck robot in Webots.

With Webots, one can develop different complex controllers and simulate it with several virtual robots in a shared environment/world. The properties of the world and their objects (such as position, orientation, shape, colour, texture, mass, friction, etc) can be defined in a simple VRML \(^1\) file. Those objects are organized in the file as hierarchical structures where objects can contain other objects, therefore it is easily modifiable and efficient.

Webots was used to simulate the experiment for three main reasons:

\(^1\)Virtual Reality Modelling Language
- The controller, which is simply a computer program that controls the robot, can be written in many different languages such as Java, which allows us to use our previous framework.
- The E-Puck mobile robot characteristics, such as shape, sensors and actuators, are already predefined.
- The application is well documented and continuously maintained for over 10 years.

6.3 Design

In principle, reinforcement learning techniques are reasonably well suited to be used in a robot, after all, RL is about decision making and achieving delayed goals handling stochastic environments. However the large state space remains a problem, but can, in principle, be handled using function approximation, such as state aggregation approaches [31].

![Module diagram for Webots simulation.](image)

The following sections will explain how the controller was designed to perform the Morris water maze experiment and how the large state space problem was handled.

6.3.1 Mapping Morris Experiment onto Reinforcement Learning

The Morris water maze experiment copes fairly well with the discrete-time, episodic, reinforcement learning framework developed in the previous chapter.

In our vocabulary, a step is when the robot goes to a state (a special position, c.f. 6.3.4) to another one. Each episode is the realisation of the experiment; that is, the robot is put in its initial position and we wait until he arrives at the goal state/position. A trial is a set of episodes where his memory is maintained. The results of the whole experiments is then averaged by the value gathered of each trial.

In each step, the robot receives a reinforcement value; that is 100 if he arrives at the goal state and -1 otherwise.

6.3.2 Self-Localisation Module

The idea of the self-localisation module is that the robot can be able to self-localise in his environment using only the camera as input.

Although there are many ways to do that, the approach chosen at the beginning was Histogram Radial Basis Response technique. Histogram comparison requires less computation and memory than many alternative methods [27].
In each state, a picture is taken; the image is pre-processed using a filter, like the Gabor-Fields filter; and then a histogram is created, for example using the RGB colours. The histogram taken from the current view is then compared with others from previously visited states.

Nevertheless, due to the complexity of this module, it could not be finished on time. In order to not harm the whole project, another technique, less fancy, was chosen. The idea was taken from RoboCup\textsuperscript{2}, instead of using the robot camera, another camera is fixed on the ceiling, pointing down to the field and the image is broadcasted for all robots.

In the case of this project, the camera is located on the top of the maze, pointing down to the labyrinth; the image is then sent to our \textit{Self-Localisation Module} that can easily calculate the robots position by comparing it with the previous image. The position is then sent to the robot to be used to update the odometry module. This is explained in the section 6.4.2, Finite-State Machine.

![Diagram showing the self-localisation module. The camera sends the image to the self-localisation module that will calculate the position of the robot in the environment.](image)

6.3.3 Odometry Module

Odometry is the use of data from moving sensors to estimate the current position over time. The major problem of this method is error accumulation. At each step, the position update will involve some error. Over short distances, however, the odometry can provide good precise results.

In this project, odometry is used to estimate the position when the robot is walking from one \textit{RL state} to another. Instead of using images from the visual input to calculate the current position every time, the robot estimates its position using odometry and updates it from the self-localisation module when arrives at the next state. As the distance between each step is quite short, the error accumulation is insignificant.

Odometry is a form of Dead Reckoning\textsuperscript{3}; each new position is based off of the previous position. The robot position can be given by \(x\), \(y\), and \(\theta\)\textsuperscript{20}; where \(\theta\) is the orientation of the robot’s wheels (or head).

\[
p = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}
\]

To find the robot current position,

\[
p' = p + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix}
\]

, the robot must estimate the values \(\Delta x\), \(\Delta y\) and \(\Delta \theta\) by looking at the difference in the encoder values: \(\Delta s_r\) and \(\Delta s_l\) (right and left wheel). With \(d\) being the distance between the wheels and expressing \(\Delta s = \frac{\Delta s_r + \Delta s_l}{2}\), the current position can be calculated using the following equations:

\textsuperscript{2}RoboCup is a contraction of the competition’s full name, Robot Soccer World Cup. It aims to promote robotics and AI research by a formidable challenge.

\textsuperscript{3}Dead Reckoning is the process of calculate the current position by using a previously determined position and estimating that position based upon known speeds over elapsed time.
6.3. DESIGN

\[ \Delta \theta = \frac{\Delta s_r - \Delta s_l}{d} \]
\[ \Delta x = \Delta s \cos(\theta + \frac{\Delta \theta}{2}) \]
\[ \Delta y = \Delta s \sin(\theta + \frac{\Delta \theta}{2}) \]

The previous equations can be summarized at [20]:

\[
p = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \frac{\Delta s_r + \Delta s_l}{2} \cos(\theta + \frac{\Delta s_r - \Delta s_l}{2d}) \\ \frac{\Delta s_r + \Delta s_l}{2} \sin(\theta + \frac{\Delta s_r - \Delta s_l}{2d}) \\ \frac{\Delta s_r - \Delta s_l}{d} \end{bmatrix}
\] (6.1)

In order to have good results with this method, it is important to calibrate the robot. Negligible differences in wheel diameter will result in high error accumulation. As this project uses a virtual robot, the parameters needed were given by the vendor’s website:

- Increments per rotation, that is the number of increments per wheel rotation, is equal to 1000;
- Axis wheel ratio, that is the ratio between the axis length and the mean wheel diameter, is equal to 1.4134;
- The diameter of the left wheel is 4.16 cm;
- The diameter of the right wheel is 4.04 cm.

6.3.4 Tile-Coding Function Approximation

In order to handle the large state space problem and noise-actuators, the linear tile-coding function approximation will be used.

The basic reinforcement learning algorithms (such as SARSA and Q-Learning) assumes that each action can be tried in each state infinitely often so to fully and accurately populate the value function. An important challenge for applying RL in environments with large state spaces is to be able to generalize the state representation in order to make learning work in practice, despite a relatively sparse sample of the state space. The value function must be approximated using some representation with fewer states, a technique commonly known as function approximation.

There are many different function approximation techniques [33]. The tile-coding function [26] was chosen to be used in this experiment. It allows us to take arbitrary groups of continuous state variables and lay infinite, axis-parallel tilings over them.

---

4For an tutorial how to calibrate the robot see Cyberbotics Robot Curriculum [20]
6.4 Implementation

This section shows the overview and difficulties encountered during the implementation of the robotic controller. It also shows the Finite-State Machine created to control the robot.

6.4.1 Supervisor Controller

The supervisor controller is a class that inherits the Webots’ Robot class with some special functions that can be used to control the simulation process and modify the world scene. It replaces human actions, such as moving the robot back to its initial position.

In this project, the supervisor will be used for three purposes:

- Moving the robot back to its initial position when it has arrived at the goal state;
- Send the reward value to the robot for each step;
- Record a video of the simulation.

6.4.2 Finite-State Machine

The overall process can be modelled using a simple finite-state automaton. This automaton, that models an episode, can be seen at the figure 6.6.

![Finite-State Machine Diagram](image)

Figure 6.6: Finite-State Machine for the Robotic Controller. The state THINK is the real implementation of the reinforcement learning algorithm. The others states are responsible for moving the robot or gather data from the environment.

Here, a brief description of each state:

**START**
- Initialize new episode.
WAIT
Wait for the supervisor signal. If reward received is bigger than zero then the episode is finished and it goes to state END. Otherwise, it goes to state UPDATE. The signal is sent between the two processes by a message queue from Webots framework.

UPDATE
It gets the visual input. If input is known, it updates the odometry position; otherwise, the self-localisation module is updated. The current state is then calculated using the tile coding approximation function.

THINK
Knowing the current state and the last reward signal, it calculates the next action using an Agent class from the Reinforcement Learning Framework developed in the previous chapter.

ROTATE
The robot only has 4 different actions (RIGHT, UP, LEFT, DOWN), therefore in this state, the robot rotates if necessary.

SENSOR
Using the infrared proximity, the robot verifies if there is a wall in front of him. If positive, it goes to UPDATE and the reward value is -1. Otherwise it goes to WALK.

WALK
The robot walks to the next tile state.

END
End of the current episode.

6.4.3 Self-Localisation Module
The self-localisation module was implemented as general as possible. As a matter of fact, the UPDATE state, in the automaton, also gets the robot’s visual input (robot’s camera) and sends it to the the Self-Localisation Module. This module, in the time being, doesn’t use that information, but it would be interesting to continue the development in order to let the robot be independent of any other camera. That would simulate the eyes of an animal.

On the Webots, the supervisor controller simulates the camera located on the ceiling. In fact, the supervisor can easily see the whole scene and get the robot’s position in a Cartesian coordinates.

6.5 Results and Discussion
The robotic controller development was a little bit harder than I was expecting, due my inexperience with robots. Although sometimes the robot lost track of his position and did unexpected actions, it was still possible to do the experiment.

The figure 6.7a shows the labyrinth used for this experiment. The results of 10 trials of 20 episodes using Dyna algorithm, choosen as the best algorithm in the previous chapter, are shown in the graph 6.7b.

As expected, the results of this experiment are similar to the ones found in the first part of the project, after all, the problem and agent (algorithm used) are exactly the same. The only differences here are the issues related with the robotic development, such as getting the real position, converting this position into a grid world and so on.

6.6 Summary
This chapter explained the design and the implementation of a robotic controller for an E-Puck mobile robot when using the reinforcement learning algorithms showed in the Chapter 2. The development had many challenges using reinforcement learning methods, such as large state spaces, uncertain states and self-localisation problems. The results of the water-maze experiment simulated using this controller are similar to those using the framework developed in the previous chapters.
Figure 6.7: Results of the robotic controller experiment. Figure (a) shows the labyrinth used. Figure (b) shows the result using Dyna algorithm averaged by 10 trials.
7 Conclusion

7.1 Conclusion

The overall objective of this project is to compare both reinforcement learning methods, model-free and model-based, in the robotic navigation problem simulating the water-maze experiment. In order to do that, a deep literature review was made during the first weeks of the project explaining the main RL principles used in both methods and their relationship with neuroscience. It was also researched other experiments using robotic navigation.

The project’s initial focus as presented in the requirement chapter (Chapter 3) was the development of a framework (Chapter 4) to be able to easily develop reinforcement learning algorithms presented in the literature review (Chapter 2). Java language was chose to produce a general library (.jar file) available in internet at github.com/vborgesfer.

Using this framework, it was possible to compare the algorithms (Chapter 5) in a grid world. As expected, SARSA(\(\lambda\)) is the best model-free algorithm tested. The model-based algorithms are better than the model-free ones, especially in a stochastic environment. I was expecting to have Prioritized Sweeping as the best algorithm for the model-based family, however Dyna agent was slightly better to readapt of the environment change.

A simplistic controller for Webots was designed in the last chapter (Chapter 6). Unfortunately, due the time constraints, the controller didn’t have all the features conceived. However, it was still possible to simulate the water-maze experiment using an E-Puck mobile robot. The data collected using a Dyna agent corroborates with the result found on the first part of the project.

Due to my difficulties with the English language, the overall progress during the project was quite slow and a lot of work was done just to write this thesis.

7.2 Future Work

This project has shown that model-free based algorithms with eligibitly traces have comparisons to model-based ones, but it did not compare SARSA(\(\lambda\)) and QLearning(\(\lambda\)). Some research [29] demonstrates that QLearning(\(\lambda\)) can have worse results, because it has an infinite exploration that might cause it to take suboptimal actions. But in our GridWorld 1, the QLearning algorithm had better performance than SARSA. Could be also the case of QLearning(\(\lambda\))? In which case would one have better results than the other?

The robotic controller developed in Chapter 6 uses a camera on the ceiling to calculate the robot’s position. It would be interesting, as already mentioned, to develop a better self-localisation module using only the robot camera as input and test it in a real robot instead of an environment simulator.
A Project Plan

A.1 Overview

As discussed in the section 3, the project was divided into two major parts. The first part, during the autumn term, it concerned the study of both reinforcement learning methods and the investigation of the relationship between RL and the animal way of thinking. Ending by the implementation of some RL algorithms to solve simple problems like the $k$-armed bandit problem, in order to assure that both methods are fully understood; followed by the implementation in a 2D-space environment, grid world. It’s important to point out that even if the robot is in a 3D-space, it is always attached to the ground, and therefore it is restricted to two dimensions.

The second part of the project was the comparison of those algorithms using the previous framework. Ending by the development of a robotic controller for the E-Puck mobile robot using a robot simulation environment, Webots.

Unfortunately, due to the time constraints, the experiment using a real robot was not possible. However, it would be interesting to see, after development of a better tile-coding function approximation and a self-localisation module, this experiment in a real robot.

A.2 Gantt chart

### Autumn Term

<table>
<thead>
<tr>
<th>Week</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>C.B. ²</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

¹ Model-free and model-based methods.
² Christmas break
### Spring Term

<table>
<thead>
<tr>
<th>Week</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>E.B.</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explore self-localisation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explore Webots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulate experiment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis of results</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-up of final thesis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deadline for final thesis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create a poster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poster session</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Easter break
Bibliography


