INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de
Electrónica e Telecomunicações e de Computadores

Dynamic Equilibrium through Reinforcement Learning

Paulo Fernando Pinho Faustino

Dissertation of scientific nature executed for the achievement of the degree of
Master in Computer and Information Technology Engineering

Jury
President
Professor Coordenador Fernando Sousa, ISEL – DEETC
Members
Examiner: Professor Adjunto Mestre Paulo Araújo, ISEL – DEETC
Supervisor: Professor Adjunto Doutor Luís Morgado, ISEL – DEETC

September 2011

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de
Electrónica e Telecomunicações e de Computadores

Equilíbrio Dinâmico via Aprendizagem por Reforço

Paulo Fernando Pinho Faustino
(Bacharel)

Dissertação de natureza científica realizada para obtenção do grau de
Mestre em Engenharia Informática e de Computadores

Júri
Presidente
Professor Coordenador Fernando Sousa, ISEL – DEETC
Vogais
Arguente: Professor Adjunto Mestre Paulo Araújo, ISEL – DEETC
Orientador: Professor Adjunto Doutor Luís Morgado, ISEL – DEETC

Setembro 2011

“Intelligence is what you use when you don’t know what to do”

Jean Piaget

Abstract

Reinforcement Learning is an area of Machine Learning that deals with how an *agent* should take *actions* in an *environment* such as to maximize the notion of accumulated *reward*. This type of learning is inspired by the way humans learn and has led to the creation of various algorithms for reinforcement learning. These algorithms focus on the way in which an agent's behaviour can be improved, assuming independence as to their surroundings.

The current work studies the application of reinforcement learning methods to solve the inverted pendulum problem. The importance of the variability of the environment (factors that are external to the agent) on the execution of reinforcement learning agents is studied by using a model that seeks to obtain equilibrium (stability) through dynamism – a Cart-Pole system or inverted pendulum. We sought to improve the behaviour of the autonomous agents by changing the information passed to them, while maintaining the agent's internal parameters constant (learning rate, discount factors, decay rate, etc.), instead of the classical approach of tuning the agent's internal parameters. The influence of changes on the *state* set and the *action* set on an agent's capability to solve the Cart-pole problem was studied.

We have studied typical behaviour of reinforcement learning agents applied to the classic BOXES model and a new form of characterizing the environment was proposed using the notion of convergence towards a reference value. We demonstrate the gain in performance of this new method applied to a Q-Learning agent.

Keywords: Dynamic Equilibrium, Reinforcement Learning, Autonomous Agents, Inverted Pendulum

Resumo

A Aprendizagem por Reforço é uma área da Aprendizagem Automática que se preocupa com a forma como um *agente* deve tomar *acções* num *ambiente* de modo a maximizar a noção de *recompensa* acumulada. Esta forma de aprendizagem é inspirada na forma como os humanos aprendem e tem levado à criação de diversos algoritmos de aprendizagem por reforço. Estes algoritmos focam a forma de melhorar o comportamento do agente, assumindo uma independência em relação ao meio que os rodeia.

O presente trabalho estuda a aplicação de métodos de aprendizagem por reforço na resolução do problema do pêndulo invertido. Neste contexto é estudado a importância da variabilidade do ambiente (factores externos ao agente) na execução de agentes de aprendizagem por reforço utilizando um modelo que tenta obter equilíbrio (estabilidade) através de dinamismo – o sistema *Cart-Pole* ou pêndulo invertido. Procurou-se melhorar o comportamento dos agentes autónomos alterando a informação passada a estes, mantendo constantes os parâmetros internos dos agentes (ritmo ou taxa de aprendizagem, factores de desconto, ritmo ou taxa de decaimento, etc.), em vez da vertente clássica de afinar os parâmetros internos dos agentes. Estudaram-se as influências nas alterações no conjunto de *estados* e no conjunto de *acções* na capacidade de um agente de resolver o problema do pêndulo invertido.

Estudou-se o comportamento típico dos agentes de aprendizagem por reforço aplicado ao modelo clássico BOXES, sendo proposto uma nova forma de caracterizar o ambiente utilizando a noção de convergência para um valor de referência. Demonstrou-se o ganho em desempenho deste novo método aplicado a um agente Q-Learning.

Palavras-Chave: Equilíbrio Dinâmico, Aprendizagem por Reforço, Agentes Autónomos, Pêndulo Invertido

Acknowledgments

This dissertation is dedicated to my family, with a special mention to my mother Deolinda. The sacrifice made by my parents to allow me to complete my first level of academic education was decisive in making me the person I am today. The continued support of my mother, 22 years past from the completion of that first major step, is one of the main driving forces to compel me to finish this most recent achievement in my life.

A special word has to be said to my wife Margarida and my children André and Pedro for all they have put up with in these past couple of years. Their unconditional love has been a major support during the difficult hours. I would also like to thank them for permanently reminding me that there is more to life than this dissertation. If I have not already gone completely mad, it is because of them.

ISEL – Instituto Superior de Engenharia de Lisboa, will always occupy a special place in my heart. It was where I started my academic education and it is where I am finishing this stage of that same education. I would like to thank the teachers that I had back in the period 1984-1989 and the teachers I have had now, in this period 2009-2011 (some are still the same after all these years!), for giving me the basis of a "superior education". I consider their knowledge, skills and devotion to teaching (even with the difficult times for the Institution) one of the foundation stones of making me a better person.

I would like to thank my colleagues, both past and present, from work and at university for all I have learned with them and for having put up with my ramblings. A special word of appreciation has to be made to João Pessoa, Nuno Gabriel and António Borga for all we have gone through together to reach this point. Your hard work and friendship have been important vehicles to achieve success.

To Prof. Dr. Luís Morgado, a very special Thank you, both for your guidance and counselling and also for being a friend during all these years. If it were not for you, I would certainly not be writing *this* dissertation.

The acknowledgements would not be complete without mentioning Richard Sutton and Andrew Barto, the authors of the book "Reinforcement Learning: An Introduction". Their work was essential for moulding my ideas about this dissertation; not only with what they wrote but mainly for what they had the insight not to write.

Contents

Table of Figures

Table of Tables

Table of Listings

1 Introduction

This chapter provides an introduction to the work presented in this dissertation. It defines the scope of the work done, as well as demonstrates the motivation behind doing the work. One other aspect mentioned in this chapter is the objectives we wish to accomplish with this work. Finally, the organization of this document is also explained in this chapter.

1.1 Scope

Reinforcement Learning is an area of Machine Learning (a branch of Artificial Intelligence) that cares about how an *agent* takes *actions* in an *environment* so as maximizing the notion of accumulated *reward*. The idea behind this principle is that an agent perceives his situation (*state*), chooses an action to execute and, after the execution of the chosen action, is informed of the reward corresponding to the action taken. The reward may be delayed in time making the agents perception of which action originated the reward a complicated or impossible task.

This form of learning is inspired by analogy to human and animal learning. When a human is subject to a new task that he must learn, he starts by analysing the situation (determining the state) then deciding to take the action that he believes will bring him the most compensation. The comparison is not as farfetched as one might think, since many of the roots of artificial intelligence come from the humanitarian sciences (psychology, philosophy, etc.).

Based on the observations of human and animal behaviour, different approaches have been proposed to reinforcement learning, leading to the creation of different reinforcement learning algorithms. Many of the studies in this area focus on the optimization of these algorithms, or the creation of new, better ones, based upon the knowledge obtained from the study of existing ones. Without doubting about the usefulness of this approach, we question ourselves about the possible improvement of the behaviour of an agent by changing factors external to the agent's learning mechanism.

With the intent of studying the importance of the external factors to the central algorithm in the execution of reinforcement learning agents, we chose to study the achievement of obtaining equilibrium by dynamic means through the use of reinforcement learning. The choice fell on this environment because of its proximity to real world physical situations with which we can identify (see the example of Segway, 2011), as well as allowing a diversity of alternatives to parameter functions which do not seem to be biased in advance for a specific solution (as could be the case of a cellular environment such as Tileworld (Pollack, 1990) where the choices of states and actions are much more obvious).

1.2 Motivation

As a direct consequence of the aims previously mentioned, we expect to present a study based upon reinforcement learning that solves the inverted pendulum problem using a Cart-Pole environment, allowing us to study the inherent difficulties of solving problems of this nature. At the same time that we attempt to improve the performance of the agent, implemented as a prototype to support the study, we will try to obtain answers for two main questions:

a) In what way does the definition of the set of states change the agent's capacity to reach its goal?

b) In what way does the definition of the set of actions change the agent's capacity to reach its goal?

The interest in the previous questions lay in the fact that in a "real" world, the environment never behaves as in the ideal theoretical conditions. Most of the studies made on reinforcement learning agents that we have reviewed dedicate themselves to the study of the algorithms used internally by agents, with the ultimate goal of optimizing them.

The motivation that has led us to choose this subject is the fact that it is very rare to encounter a study that checks the influence of external factors on the agent's efficiency, this being, in our opinion, a critical factor in a good performance of an agent.

1.3 Objectives

The study of reinforcement learning can be assessed by experiences performed on environments designed for this purpose.

One such example is the Tileworld (Pollack, 1990) that consists of a "world" composed of a rectangular matrix of cells in which targets, obstacles, intelligent agents, and other objects can be placed. One of the characteristics of this environment that makes it interesting for the study of intelligent agents is the direct and simple form by which states and actions can be defined (the actual cells of the matrix and the events that provoke the passage from cell to cell, respectively).

Another example of a propitious environment for the study of reinforcement learning is the so-called Cart-Pole (Sutton & Barto, 1998) (also known as the "inverted pendulum"), which consists in maintaining a pole or pendulum in vertical equilibrium by laterally shifting a cart on which the pole is resting. Due to the physical nature of this model, it is appropriate for studying non-linear problems where the state and action space are continuous values (non-discrete).

This dissertation studies the variation of behaviour of an agent that is placed inside a non-linear environment (Cart-Pole), for different state and action representations, in the context of three reinforcement learning algorithms (AHC, Q-Learning and Dyna-Q), representing main classes of reinforcement learning methods, namely, model free value function based methods, model free action/value function based methods and model based methods.

To that effect, we seek to use reinforcement learning agents, a model that simulates the physics of an inverted pendulum and an interface with the environment that allows gathering of sufficient information about the behaviour of the agent in that environment so as to evaluate its performance. By analysing the gathered data, we will try to interpret the information suggesting alternative action and state sets and, if appropriate, alternative reward functions (all external to the agent's central algorithm) in order to optimize the agent's behaviour.

Before starting any discussion on the resolution of the Cart-Pole problem by using a Reinforcement Learning Agent, we must state that the inverted pendulum problem is not exclusive to this type of solution. The inverted pendulum problem can be (and is) solved by a number of other types of solutions, being Control-theory one of the most common. Likewise, the

Reinforcement Learning approach is not "better" or "faster" than any other, so then why use it? The point here is to study the possible solution of the problem through Reinforcement Learning, and not the optimal way of solving the problem in itself.

1.4 Document organization

This document is comprised of the current introduction and six other chapters, which consist of the following information:

- **Behaviour Learning in Autonomous Agents** – In this chapter we define the autonomous agent concept and discuss the main support architectures. We explain how autonomous agents can be classified regarding architectures as well as explain how they can be directed through motivation. Further in this chapter we examine the way an agent interacts with the environment and how agents and environment are coupled one to the other through perceptions and actions. We also take a first look at how an agent can learn as a result of its interaction with the environment.

- **Reinforcement Learning** – In this chapter we explain how learning can be achieved through the use of rewards. Another topic introduced in this chapter is the concept of temporal difference learning in which predictions are used to guide the agent towards its goal. An introduction to both model based and model free reinforcement learning architectures is made. The former is explained by studying Suttons Dyna architecture in the variant associated to Q-Learning – Dyna-Q. The latter is explained by studying both a value function method, Adaptive Heuristic Critic – AHC, and an action/value function, Q-Learning.

- **Dynamic Equilibrium in a Continuous Environment** – The previous chapter explored the issue of how an agent can solve a problem by using reinforcement learning. This chapter presents the problem we are solving by introducing the Cart-Pole inverted pendulum as a means for studying reinforcement learning with both a logical and physical view of the cart-pole system being presented. This chapter also introduces a framework for experimenting with reinforcement learning agents, the RL-Glue framework, which was used to approach the cart-pole problem.

- **A Classical Model for the Cart-Pole Problem based on the BOXES Algorithm** – In this chapter we present a model, based upon the BOXES algorithm that has become a benchmark for evaluating agents using the Cart-Pole inverted pendulum dynamics. We explain how the state set, action set and reward function are defined for this model. Further in this chapter we examine the way three different agent architectures (AHC, Q-Learning and Dyna-Q) are set up to solve the inverted pendulum problem using this model. We also take a look at the experimental results obtained by these three agents.

- **An Alternative Model for the Cart-Pole Problem** – The previous chapter discussed the classical BOXES model for solving the Cart-Pole problem. In this chapter we define an alternative solution for the Cart-Pole problem by defining an alternative State and Action set. The implications of these changes are discussed as well as the changes needed to the AHC, Q-Learning and Dyna-Q agents, due to the alternative state/action sets. The experimental results of changing action semantics and number of states are also presented in this chapter.

- **Conclusion** – This final chapter presents the conclusions reached resulting from the research made on the theory of reinforcement learning and the Cart-Pole inverted pendulum problem, and also the analysis of results obtained from the experiments performed. A discussion on directions for future work is also presented.

2 Behaviour Learning in Autonomous Agents

In this chapter we define the autonomous agent concept and discuss the main support architectures. We explain how autonomous agents can be classified regarding architectures as well as explain how they can be directed through motivational processes. Further in this chapter we examine the way an agent interacts with the environment and how agents and environment are coupled one to the other through perceptions and actions. We also take a first look at how an agent can learn as a result of its interaction with the environment.

2.1 Autonomous Agents

In the scope of artificial intelligence, an autonomous agent is a computational entity that presents three base properties: *autonomy*, *reactivity* and *pro-activity*. In some cases a fourth property complements the previous three, *sociability*; that is defined as the capacity for an agent to coordinate his activities with other agents, and possibly with humans, so that he may reach his goals and, if it may be the case, to help other agents reach their goals.

By *autonomy*, what is meant is the capacity for an agent to act without the direct intervention of humans or other agents, having control over its internal state and over the actions it performs. By *reactivity*, what is meant is the capacity of an agent to detect changes in the surrounding environment and react timely to those changes. By *pro-activity*, what is meant is the capacity for an agent to react not only to stimuli from the surrounding environment, but also in an oriented manner so as to achieve its goals, taking the initiative whenever appropriate. (Jennings & Woolridge, 1998)

2.1.1 Agent Architectures

An *architecture*, in computer science, is a blueprint for the development of a system defining the arrangement of its components and the forms of interactions of those same components. An *agent architecture* is a blueprint for software agents and intelligent control systems.

Three main types of agent architectures can be identified, reactive architecture, deliberative architecture and hybrid architecture. Next, we will address these different types of architectures.

2.1.1.1 Reactive Architecture

In the world of artificial intelligence, the simplest architecture one could find associated to autonomous agents is what is known as reactive architecture. In a reactive architecture there is a coupling between perceptions and actions through strong reactive mechanisms such as stimulus-response rules, typically organized in modular behaviours as shown in the following figure.

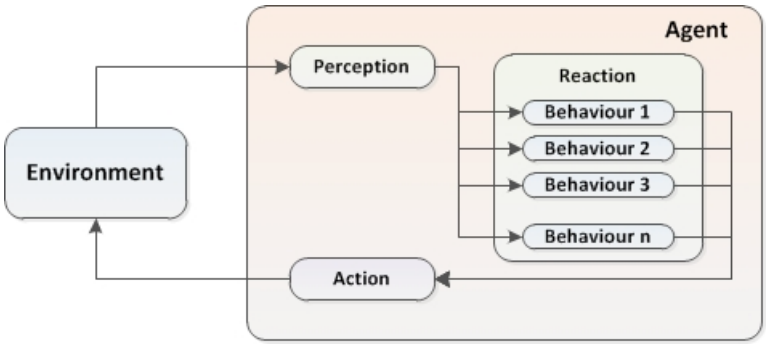

Figure 1 - Reactive Agent Architecture

In this category, Braitenberg Vehicles, so named after the German/Italian researcher Valentino Braitenberg who developed the concept, are a prime example.

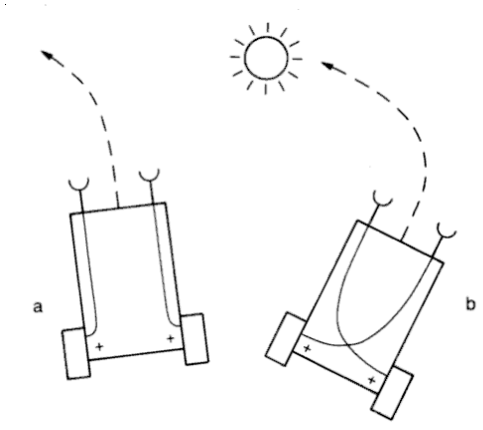

Figure 2 - Braitenberg vehicles (Braitenberg, 1984)

Braitenberg vehicles consist of sensors of varying types connected directly to motors. Whenever a sensor produces a signal, the corresponding motor to which it is connected will start to function. By changing the wiring

in the agent (vehicle), the behaviour can be modified from the goal of avoiding a light source, as on side "**a**" of the previous figure to a new goal of approaching it, as on side "**b**"; this assuming that the sensors used are sensitive to light. These very simple agents display emergent behaviour that starts getting more complex as the number of simultaneous sources of stimuli increase (be it light sensors or any other type of sensors).

The previously mentioned agents were very simple, both in their conception as well as in their pursuit of goals. It is however possible to build more complex reactive agents using more elaborate reactive architectures. Different approaches have been proposed to that aim, having in common the notion of behavioural modularity. The *subsumption* architecture, developed by Rodney Brooks (Brooks, 1989, 1990), is one such example.

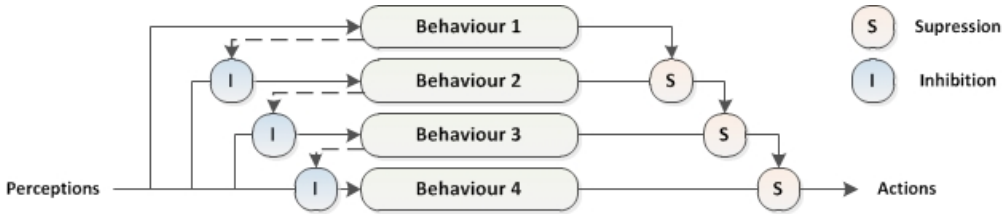

Figure 3 - Subsumption Architecture (adapted from Brooks, 1989)

The *subsumption* architecture consists of building agents in parallel "layers", each of which simultaneously receive information from sensors (the perceptions) and emit commands to actuators (the actions). Each layer is built in such a way as to produce behaviour on its own, such as avoid obstacles, search for objects, etc. The uppermost layers may inhibit the inputs of lower layers (stop them from receiving the input) and may subsume outputs of those lower layers (override the commands). In this way, it is possible to create complex behaviour using simple functions (simple behaviour encoded in each layer). This approach to building agents, with parallel execution of blocks in layers, contrasts with the more traditional *horizontal* or *pipeline* approach in which the blocks are executed in a serial fashion.

One of the characteristics of this architecture is that there is no need for explicit knowledge representation of the environment (*i.e.* a model). Thus, as Brooks says, "*The world is its own best model*" (Brooks, 1990). Another characteristic of this architecture is that all responses to stimuli are reflexive – the perception-action sequence is not modulated by cognitive deliberation.

2.1.1.2 Deliberative Architecture

Another main type of agent architecture is the deliberative architecture. This architecture displays a form of logical reasoning based upon pattern matching and symbolic representation for decision-making. One of the drawbacks of this architecture is the need for the world model (of symbolic nature) to be as complete as possible, posing problems in achieving the correct internal representation from a time-wise varying, dynamic real world (the complexity of the real world entities and processes may be hard to model).

A deliberative architecture is generally characterized by the following serial flow: *perception* (which helps construct the internal model of the world as well as defining the goals for the agent), *deliberation* (the agent deliberates on how to achieve the perceived goal by systematic exploration and comparison of alternative courses of action) and *action* (the result of the deliberation is a representation of the action to be performed). A deliberation is based upon the agent's internal model of the world that represents the current state of the real world. A goal is a representation of a future desired state that the agent plans to achieve by executing a series of actions. In this way, goals determine the internal processing of a deliberative agent. The following figure illustrates a deliberative architecture.

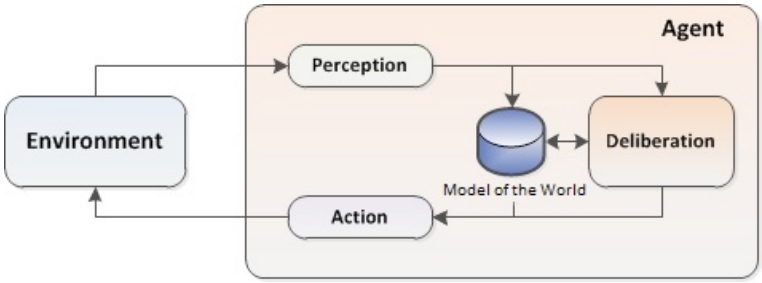

Figure 4 - Deliberative Agent Architecture

This type of architecture is useful when the penalties for incorrect actions are high (for example in a hazardous environment) and also for solving a family or class of problems of similar nature instead of a single instance of a problem. Because the agent will, for each move, plan for the best move before actually executing it, wrong moves will be avoided. As mentioned, the drawbacks are the need for an accurate world model and also the computational resources and time it may take to plan an action may not be compatible with real-time situations (as opposed to a Reactive architecture in which the decision of the action to take is always immediate or near-so).

One example of Deliberative Architecture is the BDI Architecture in which the agent deliberates by using three "mental" attitudes – Beliefs, Desires and Intentions. The following figure illustrates the base BDI architectural model.

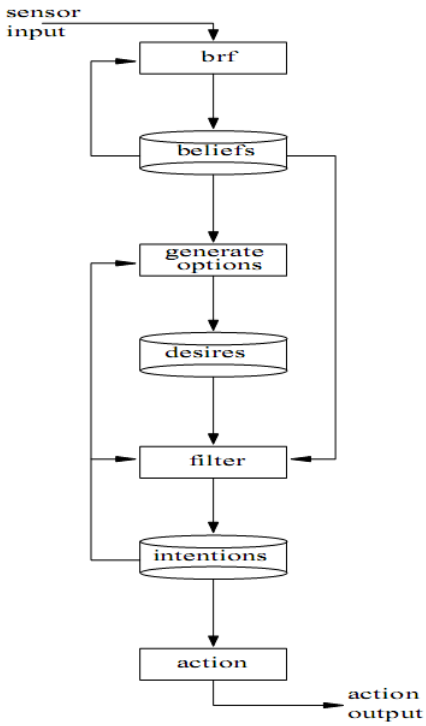

Figure 5 - BDI Agent Architecture (Wooldridge, 2002)

In the BDI model, beliefs represent the information available to the agent, desires represent motivational states driving agent behaviour and intentions represent deliberative states of the agent. The beliefs are related to the current model and perceptions and can be used to derive state through a belief revision function as illustrated in the previous figure. The desires represent the options available to the agent to achieve its goals that are filtered to generate the intentions that correspond to plans to achieve agent goals.

2.1.1.3 Hybrid Architecture

A third type of architecture is the hybrid architecture. This architecture is a conjunction of the two previous architectures. Given the requirement that an agent be capable of reactive and proactive behaviour, a possible decomposition involves creating separate subsystems to deal with these different types of behaviours (Woolridge, 2009). A hybrid agent is an agent that is capable of reactive response to stimuli but at the same time, it can also plan ahead if necessary.

2.1.2 Motivated Behaviour and Autonomy

The behaviours displayed by an agent are generated by internal processes in order to achieve specific states of the world. In this way, these states of the world constitute motivational states that drive agent behaviour. These states can be explicitly represented as goals, in the case of deliberative agents, or exist in an implicit form, such as stimulus-response patterns in reactive agents. However, in both kinds of architectures, from the dynamics driven by those motivational states, behavioural patterns are generated that do not depend on any external source of behaviour, human or other, that is, the agent is autonomous in its behaviour. Motivated behaviour is, in this way, the source of agent autonomy, determining agent perceived states and generated actions, therefore playing a key role in reinforcement learning agents.

2.2 Agent-Environment Interaction and Coupling

Just as motivation to achieving goals is an important part of an agent, observation and acting upon an environment is an equally important part. Recall that the environment is everything outside of the agent's direct control. The agent perceives the environment and based upon the information gathered in that perception (and eventually other information that the agent may already possess – depending on the agent architecture) acts upon that same environment to try to achieve its goals.

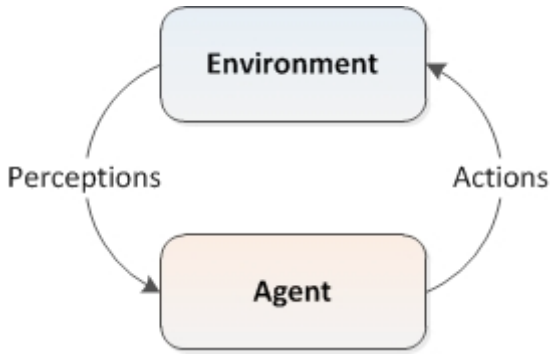

Figure 6 - Agent - Environment Interaction cycle

For each different problem being treated by the agent, the goals may differ. Likewise the set of perceptions available to the agent, as well as the set

of possible actions to be taken, may also differ. Even if those sets are exactly the same, the context of the problem may differ. What this illustrates is that each problem treated by the agent is unique. It is not possible to devise an agent that solves one particular problem and then apply it to another completely different type of problem and expect the same outcome. The agent and the environment are coupled by more than just the interactions between them. On one hand, the support for perception and action (as is the case of sensors and actuators) is specific for some environmental characteristics. On the other hand, agent-environment interaction may affect the internal structure of the agent in order to adapt the agent's behaviour to specific environment conditions. This is the case of the reinforcement learning agents.

2.2.1 Perceiving the Environment

Even the simplest of autonomous agents need to perceive their environment. If they are not able to perceive their surroundings, they are not able to devise a worthwhile internal state. Imagine a very powerful computer capable of executing enormous quantities of calculations in an extremely short period of time. What good would this computer be if it did not present the result of the calculations in some form? Even if it did present its calculations, would it be any good if we could not tell it what we wanted it to perform the calculations on? Could it be considered an autonomous agent if it just sat there without any type of interaction to the outside world?

As was seen in the definition of an agent, interaction with the environment is an important part of the concept of an agent. The agent's ability to obtain information about its surroundings (the environment) is essential for its display of intelligent behaviour. The form through which the agent obtains the information is not of particular relevance[1]. What is of major relevance is what the information represents to the agent and at a smaller scale what the agent does with that information.

Except for the cases of reactive agents (Braitenberg vehicles and some subsumption based agents), most agent architectures implement a software algorithm of one sort or another. Some of these algorithms assume that there exists an internal model representing the world. Others do not. In either case, the agent makes decisions based upon the information it possesses. The agent

[1] The discussion about whether it's the agent that obtains information from the environment or alternatively if the agent is handed information by the environment is not one we wish to pursue. The boundary between the agent and the environment is a matter of design of the architecture and does not detract to our argument over perceptions so we will assume that either form is equivalent and whatever mention we make to one form is applicable to the other.

updates this information by making perceptions of the environment, analysing the information perceived, making some sort of computation based upon its internal algorithm and finally producing an output as feedback to the environment. This cycle is repeated again and again until the agent reaches a point where it decides to stop (typically when it reaches a terminal state). Due to this cyclic nature, for each cycle or step the agent is said to be in a certain *state* and it expects that the action it produces as output will lead it to another state. There is a very close and important relation between the agent's state and its perceptions. The amount of perceptions available to the agent *should* be enough (and not more than that) to unambiguously define the agent's state.

A problem known as **perception aliasing** may occur whenever the amount of different perceptions available to the agent is not enough to clearly define the agent's state. This problem is defined by when the agent is presented with some values from the perceptions, it cannot clearly define the exact state it is in and consequentially will not be able to decide on the correct course of action to take. The following figure illustrates this problem.

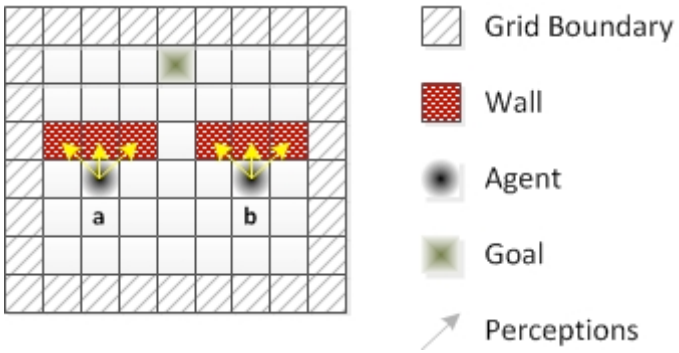

Figure 7 - Perception Aliasing problem

For understanding the perception aliasing problem presented in the previous figure, let us assume that the agent only has sensors allowing it to perceive forward, forward-left and forward-right. Furthermore, the agent does not have any mechanism that allows it to find its coordinates on the grid (which would be an additional perception). The agent can only move forward or rotate to the right or to the left. It cannot move out of the grid or into a wall. When the agent is present in the position marked as "**a**", all three of its sensors indicate a wall. Likewise, in position "**b**" all three sensors also indicate a wall. For the agent, both of these readings (perceptions) tend to indicate the same state (for the agent does not possess other information which might distinguish one from the other). It becomes obvious that the correct action to take in position "**a**" is to rotate right and then advance as moving to the left will trap it against the wall. It is also obvious that in

position "**b**" the correct action is to rotate left and then advance since rotating right will also trap it into the wall. If the agent is in a state in which all three sensors read "wall", what is the correct course of action to take, left or right?

One other interesting aspect of the perceived values is that these may not always be discrete. In our previous example we talked about discrete values for the perceptions (wall, open space, grid boundary and goal), but what if the sensor gave us a continuous reading? Depending on the agent, this may be a problem or it may not. If, for example, we consider a simple Braitenberg vehicle in which the sensors give us a continuous reading but these are connected directly to the outputs (motor drivers), then the fact that a discrete finite set of possible values does not exist is not relevant. On the other hand, these same sensors applied to an agent that needs to discretize the values for elaborating the state-set, may lead to a huge problem as it may generate an enormous amount of discrete values that may be too much for the agent to handle. In this way, state representation plays a key role in autonomous agents.

2.2.2 State Representation

When it is said that the agent perceives the environment, this means that the agent will receive information, in general through sensors, about properties of a physical or virtual environment. When this information is taken from the sensors to be presented to the agent, there is no guarantee that the information is in a format directly useable by the agent's internal processes and so some form of transformation may have to be applied to the values read from the sensors. One possible transformation could be to apply a filter to eliminate jitter from the values being read from the sensor. Another possibility could be to transform a continuous reading into ranges to limit the amount of information that the agent has to process. Establishment of the correct values to use for the ranges is a problem in itself and depends heavily on the nature of the problem at hand. One more possible transformation could be to combine the readings of various sensors into a single value that could have enough meaning to represent the set of sensors whose values were grouped. The amount and types of transformations applied are always a function of the problem at hand. The agent's *state* is a representation taken at any given moment of the combination of all the perceptions after these have been transformed into a useable form for the agent.

To minimize the impact of the perception aliasing problem mentioned in the previous section, one would think that the answer would be to increase as much as possible the amount of perceptions given to the agent. Though this

approach might reduce the perception aliasing problem, it could bring other problems. The set of states that the agent distinguishes, the agent *state-space*, has a direct correspondence to the total amount of perceptions available to the agent (from both external and internal environments). If we assume that a given perception is related to a given sensor and that the sensor may produce a given quantity of different readings, then the state will correspond to a function of the actual values being read in each of the sensors. If we have an agent that takes perceptions of the environment from, for example, 5 different sensors and each of these sensors can assume 4 different values, then the number of states in the state-space (the number of different combinations of variables) will be $4^5 = 1024$ states. Even though this may not seem much, if we were to have the same 5 sensors and each of them perceiving not 4 but 5 different values, then the state-space would now be 3125. If the fifth value were an unnecessary value, just included to have all possible perception values – even if unused, we would have an additional 2101 states which would be occupying resources and adding to the overall complexity of the system. In a "real-world" environment we may have much more than 5 sensors and much more than 5 values per sensor so the state-space may be significantly larger. This in itself may bring problems when the agent has reduced resources (memory, processing capacity, etc.).

2.2.3 Acting over the Environment

In the agent-environment interaction, the perceptions served to give the agent a view of the environment. For the interaction to be complete, the agent needs to act over the environment in order to promote the intended changes. This completes the two-way interchange of information between the two parts.

Actions, like perceptions, can be of two types: discrete and continuous. Continuous actions are not very common though they may be useful in some types of agents. Recall the Braitenberg vehicle in which the sensors are directly coupled to the motor drivers. Since the sensors produce a continuous value and the agent directly returns this value, it is continuous. Even though it is possible to have continuous action values in other types of agents, the usefulness of such actions has to be contemplated.

Discrete actions are usually limited in terms of number of distinct values. Even though the higher the number of distinct values the more commands the agent can transmit to the environment, the trade-off between the resources occupied by a larger number of actions and the gain in performance

should be studied case-by-case. The number of distinct actions that an agent can execute is referred to as the Degrees of Freedom (DOF) (Mataric, 2007).

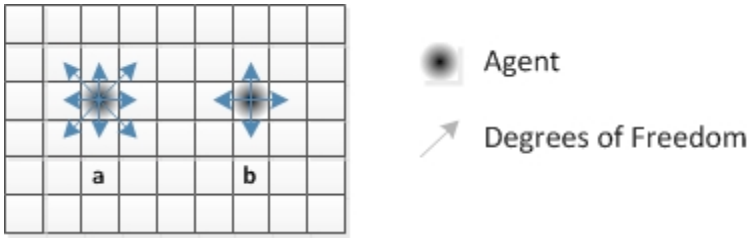

Figure 8 - Varying degrees of freedom

In the previous figure, the agent on the left is allowed 8 degrees of freedom (he can issue 8 commands or actions and move to any of the adjacent cells). The agent on the right is only allowed 4 degrees of freedom: move left, right, up or down. Both agents can move to the upper-right cell of their current location; the agent on the left will take one step while the agent on the right takes two steps (for example, right then up) to get there. The choice of degrees of freedom allowed for the action set is a function of the problem to solve, namely in what concerns agent capacities and environment properties.

The number of degrees of freedom per action can be important in terms of computational resources in the sense that an increase in the degrees of freedom leads to an increase of the computational resources needed by internal processes to generate agent behaviour in order to achieve its goals.

2.2.4 The Role of Action Representation

Just as sensory data is internally processed to generate state representation, a reciprocal process occurs concerning action representation. That is, action representations are converted into actuator data to operate over the environment. In this way, action representation and the corresponding action processes express the semantic grounding that relates internal processing and representations to the external environment.

The set of actions that an agent is able to execute over the environment, the agent *action-set,* is important in terms of computational resources required for internal processing, since an increase in the degrees of freedom associated to the agent action-set leads to an increase of the computational resources needed by internal processes to generate agent behaviour in order to achieve it's goals.

2.3 Learning From Agent-Environment Interaction

The definition of learning as taken from the Merriam-Webster dictionary is: *"learning" – modification of a behavioural tendency by experience.* In the midst of other definitions provided, this is one particularly suitable for intelligent agents.

We can apply this definition to agents by stating that an agent learns when it uses both past experiences and current state to produce actions, updating its experiences on every step. Each new step (action outputted) will not only be a function of the current state, but also of the results of changes produced on the environment by previous actions taken – modification of behaviour. This definition assumes that the agent possesses some form of keeping track of the implications on the environment as a consequence of the actions taken (accumulated experience).

By these standards, even the simplest of reactive agents, such as a Braitenberg vehicle, is capable of demonstrating an ability to learn if, for example, we add a capacitor somewhere between the sensor and the motor driver[2]. For a certain period of time (the time it takes for the capacitor to charge and/or discharge), the agent's actions are modified based upon previous experiences (the charge in the capacitor). From an "outside" point of view, the agent has learned a new behaviour by modifying its outputs as a function of past experience.

2.3.1 Relating Perception, Action and Motivation

In our previous example, the one about the Braitenberg vehicle with the capacitor, there is no doubt that according to the definition we can state that the agent's behaviour has changed. But does this new behaviour produce any improvement in the agent's capability to reach its goal? If not, then the real value of what the agent has retained is debatable. If we recall the definition of an intelligent autonomous agent, we will see that it needs to work towards achieving a goal. The previously mentioned process of changing the behavioural pattern effectively becomes learning if the new behaviour demonstrated leads toward achieving the agent's goals. In our previous example, the agent may adapt to approach a light source quicker than before due to the fact that he has previously experienced an interaction with a light

[2] This is just a simple example of what could be the logical circuit for this vehicle. A "real" physical electronic circuit would probably have to comprise more components to work properly.

source. One might say that the agent's adaption to the environment has increased due to his past experience.

At this moment we possess all the necessary information to define an intelligent autonomous agent. This agent should perceive the environment in which it operates and using those perceptions (along with whatever other information it may have) should be able to select the best action to perform in order to achieve one or more of its goals, that is, to achieve its motivations. After the execution of the selected action, possibly there will be changes produced in the environment due to this action. The agent now needs to reassess the perceptions making any change to whatever knowledge it may have of the world as a consequence of the action it took. This relation between perception, action and motivation is the base of behaviour learning from experience. This process will be repeated over and over again until a goal is reached.

2.3.2 Behaviour Learning from Experience

As we have previously seen, behaviour learning from experience is a process that involves making decisions, taking actions relating to those decisions and evaluating the consequences. If the consequences are positive, one can choose to repeat them the next time a similar situation arises. If the consequences are negative, one can choose to try a new alternative action to try and improve the situation. This is only possible when there is a memory of past actions and the consequences thereof. If an action is taken and its results are forgotten then the next time a similar situation arises, there is a possibility of repeating a previous bad decision, or not being able to replicate a previous good decision.

Just as this holds true for humans, it also holds true for intelligent agents. The process of learning cannot be dissociated from past experiences, both good and bad. The following chapter investigates one such process of learning from experience, designated reinforcement learning, which is our main research focus.

3 Reinforcement Learning

In this chapter we explain how behaviour learning from experience can be achieved through the use of rewards. We present the concept of temporal difference learning in which predictions about changes in rewards during the course of time are used to guide the agent towards its goal. An introduction to both model based and model free reinforcement learning architectures is made. The former is explained by studying Suttons Dyna-Q architecture. The latter is explained by studying both a value function method, Adaptive Heuristic Critic – AHC, and an action/value function, Q-Learning.

3.1 Learning through Rewards

"Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward, but also the next situation and, through that, all subsequent rewards. These two characteristics – trial-and-error search and delayed reward – are the two most important distinguishing features of reinforcement learning." (Sutton & Barto, 1998)

A reinforcement learning (RL) agent distinguishes itself from a non-reinforcement learning agent by the two above enumerated characteristics: trial-and-error search and delayed rewards. Unlike the other agents we have seen previously, a reinforcement learning agent does not need to have an *a priori* representation of the world to deliberate and make a plan to reach its objective. When in a given situation – state – the agent will select an action based on information retained from past experience and execute that action over the environment observing its effects, repeating this process until a goal state is reached. From this process a relation is formed between states and actions that determine the agent behaviour. That relation is therefore designated the agent control *policy*. Formally, a policy π represents a mapping relating states and actions, defined as follows:

$$\pi : S \rightarrow A(s); s \in S, a \in A \tag{3.1}$$

where S is the set of possible states and $A(s)$ is the set of admissible actions for some state s. That is, a policy π defines which action a should be taken in which state s.

In a reinforcement learning agent, the control policy is dynamically generated based on the experience that results from agent-environment interaction along time, relating state, action and reward. A reward is a reinforcement signal that indicates how effective the transition from state s using action a effectively is in order to achieve the agent's goals. The agent should find a policy π, such that it displays behaviour of action selection that tends to maximize the long run sum of the reinforcement signals (rewards). Repeated iterations of executing the policies according to adequate internal processes, evaluating received rewards and readjusting actions will lead to finding the policy that maximizes this long-term sum of rewards. The policy that maximizes the long-term sum of rewards is called the *optimal policy* π^*.

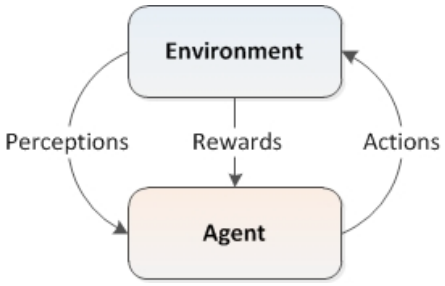

Figure 9 - Agent / Environment coupling with reward signal

The reward signal is fundamental in reinforcement learning, being a distinguishing feature of reinforcement learning. In supervised learning, during the learning phase, after the agent takes an action it will be informed if the action taken was correct or not. It learns to relate stimuli and answers by examples given during the learning phase with the objective of minimizing the error resulting in the difference between the produced result and the desired result (Norvig & Russel, 2003). This is not so in reinforcement learning as it is never informed if the action taken is correct or not. It is just told the immediate reward and the next subsequent state. It learns behaviour as a result of the effects of actions taken while trying to maximize the return reward (Norvig & Russel, 2003). The reinforcement learning agent will have to gather experience about the possible system states, actions, transitions and rewards and learn what the best action to take is that will maximize its long-term interests. The trial-and-error nature of reinforcement learning also distinguishes it from supervised learning; the evaluation of the system is concurrent with the learning process.

A key aspect of reinforcement learning is the notion of *delayed rewards*. As can be observed in the previous figure, upon executing an action the agent can assess from the environment the subsequent state and the reward for that particular interaction. A significant reward attribution may only happen upon reaching a terminal state while relatively insignificant rewards may be received in the meanwhile. This makes the agent's learning process even more difficult because it will have to back propagate the significant reward value to all the states it passed to reach the terminal state. This is the reason why the rewards are called delayed rewards. The agent has to be able to transform the immediate rewards it is receiving into a sum that can be maximized in the long-term, sometimes by sacrificing immediate rewards to receive larger long-term rewards. The trade-off between exploring (searching for new actions that *may* provide larger rewards) and exploiting (using the action that is producing the largest *known* reward) has to be carefully considered. Three main models of reward accumulation are defined: finite-horizon model, infinite-horizon discounted model and average-reward model.

The finite-horizon model states that, at any given moment of time, the agent should optimize its expected reward in a determined temporal scope. The next n steps, being n the distance of the horizon, can be used to define the finite horizon. This model is useful when the time length of the agent's activity is known in advance.

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_{t+n} = \sum_{i=0}^{n} r_{t+i} \tag{3.2}$$

The average-reward model states that the long run average is to be maximized (instead of a sum as in the other models). The problem with this model is the difficulty in distinguishing policies that have higher gains at the start from those having higher gains at the end.

$$R_t = \lim_{n\to\infty}\left(\frac{1}{n}\sum_{i=1}^{n} r_{t+i}\right) \tag{3.3}$$

The infinite-horizon discounted model states that the long run reward is accounted for, but rewards that are received in the future are geometrically discounted as of the factor γ, (where $0 < \gamma < 1$).

$$R_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \ldots = \sum_{i=1}^{\infty} \gamma^{i-1} \cdot r_{t+i} \tag{3.4}$$

The infinite-horizon discounted model does not impose a finite time horizon on agent activity, however in this case the discounted factor is needed in order to guarantee a definite limited result for the potentially infinite sum of rewards, considering that each individual reward *r* is bounded.

In the context of an autonomous agent architecture, the reward signal expresses the motivational base for agent behaviour, therefore it is internally generated based on the relation between the observed state and the motivations of the agent.

3.2 Temporal Difference Learning

Kaelbling *et al* (1996) have stated, "*The biggest problem facing a reinforcement-learning agent is* temporal credit assignment. *How do we know whether the action just taken is a good one, when it might have far-reaching effects?*" If we consider the notion of delayed rewards, in which only at the end of the run we may have a significant reward value, how can we classify the intermediate actions taken? To face this challenge Sutton (1988) formalized the notion of *learning to predict*, that is, of using past experience with an incompletely known system to predict its future behaviour. This method of prediction was named Temporal Difference Learning (TD).

Temporal difference learning, as described by Sutton, differs from conventional prediction learning by the way the prediction error is considered. In conventional prediction learning, the difference (error) between the predicted value and the actual outcome is relevant. In temporal difference learning, the difference between the predicted value and the next temporarily successive predicted value is relevant. In this way, learning occurs whenever there is a change in prediction over time.

According to Sutton (1988), this approach to prediction learning has two kinds of advantages over conventional prediction learning: first, it is more incremental and therefore easier to compute. Second, it tends to make more efficient use of experience thus converging faster and producing better predictions. An additional benefit is that fewer resources are required as there is no need to store all the values since the prediction up until the actual outcome, but instead only the previous prediction must be stored, so there is no need for a model. The actual outcome may be available on the next time step as it may also be available after quite a few (undetermined) number of time steps. In some extreme cases, it may never even be known (the terminal state may never be reached). The distinguishing mark of temporal difference methods is their reaction to changes in successive predictions instead of

reacting to the overall error between predictions and the final outcome. TD is said to *bootstrap* due to the fact that it is learning an estimate in part on the basis of other estimates, without having to wait for a final outcome (it learns a guess from a guess).

The TD method uses experience to solve the prediction problem. Given some experience following policy π, the method updates its estimate V of V^{π}. If a non-terminal state s_t is visited at time t, then the method updates its estimate $V(s_t)$ based on what happens after that visit. At time $t+1$ it immediately forms a target and makes a useful update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$ (Sutton & Barto, 1998). The simplest TD method, known as TD(0), is characterized by the following update rule:

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \tag{3.5}$$

From the previous expression, the update target is:

$$r_{t+1} + \gamma V(s_{t+1}) \tag{3.6}$$

The TD(0) algorithm can be described as follows.

1. Initialize $V(s)$ arbitrarily, initialize π to the policy to be evaluated
2. Repeat (for each episode)
   a) initialize s
   b) Repeat (for each step of the episode)
      i. $a \leftarrow$ given by π for s
      ii. Take action a, observe reward r and next state s'
      iii. $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$
      iv. $s \leftarrow s'$
   c) Until s is terminal

Listing 1 - TD(0) algorithm for estimating V^{π}

3.3 Learning without Environment Models

Reinforcement learning agents may be implemented using either model-free or model based algorithms. The difference between them is the existence (or non-existence) of internal models of the environment that are used to aid in obtaining an optimal policy. A model of the environment is anything that an

agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. Models can be used to mimic or simulate the environment and produce *simulated experience*, (Sutton & Barto, 1998).

A model-free reinforcement-learning algorithm is one that manages to obtain an optimal policy without recurring to internal models. Due to the lack of the models, and hence the lack of maintenance on those same models, very little computation time is needed per experience compared to model-based algorithms (Kaelbling *et al*, 1996). However, the data gathered by the agent is used in an extremely inefficient manner and a great deal of experience is required to achieve good performance. This is in part due to the fact that all the states have to be visited numerous times for the policy to start converging. These algorithms will find the optimal policy eventually but will take a long time doing so (Kaelbling *et al*, 1996).

Model-free algorithms are best suited for agents who possess few resources (computational and/or memory) or in situations in which computation is costlier than real world experiences.

3.3.1 Learning a Value Function – AHC

Almost all reinforcement learning algorithms are based on estimating *value functions* – functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular policies (Sutton & Barto, 1998). One example of a value function is the Adaptive Heuristic Critic (AHC).

The Adaptive Heuristic Critic structure is an abstraction of Barto's Adaptive Critic Element (ACE) & Associative Search Element (ASE) framework. A function approximator (AHC) is used to estimate the value function instead of calculating it by solving the set of linear equations in policy iteration (Alavala, 2008). The basic blocks of the framework are an Associative Search Element that uses a stochastic method to determine the correct relation between input and output and an Adaptive Critic Element which learns to give a correct prediction of future reward or punishment (Yang, 2008).

The Adaptive Heuristic Critic (AHC) was the first method proposed for model-free learning using Temporal Difference based techniques. Barto *et al* (1983) used this technique to solve the cart-pole problem, drawing interest to the TD approach of machine learning (Ribeiro, 1996). The rationale behind AHC is to combine a policy evaluation step with an on-line action selection strategy, providing a model-free Policy Iteration mechanism.

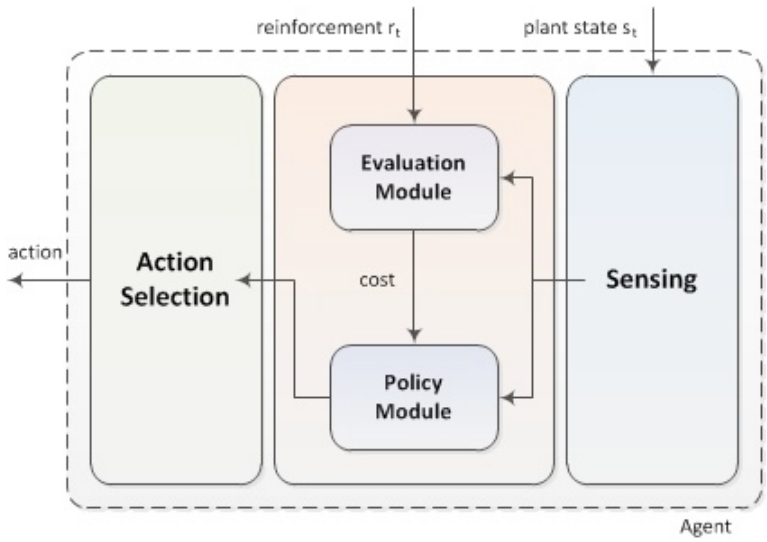

Figure 10 - Adaptive Heuristic Critic (AHC) agent (Ribeiro, 1996)

The AHC architecture in the previous figure contains two main components: an Evaluation Module and a Policy Module. The two secondary components are the State Sensing and Action Selection. The meanings of each of the components and the associations between them are described as follows:

- State Sensing – Given the environment information, this module will compute the current state s_t (also known as the input state);

- Evaluation Module – Responsible for computing an approximation of the value function $V(s)$ for every input state. This is performed by updating the current parameters by comparing the current output $V(s_t)$ with the expected cost $r(s_t, a_t) + \gamma V(s_{t+1})$. The evaluation module will not provide the action for s_{t+1} but instead will provide information to the policy module for calculating this action;

- Policy Module – Responsible for iteratively estimating an optimal action policy $\pi^*(s)$. The parameters of this module are updated according to a gradient method which encourages or discourages

the current action, depending if the updated $V'(s_t)$ is smaller or larger than the former $V(s_t)$ from the previous time step (Ribeiro 1996). As it deals with cost, if the new cost is smaller than the previous one then the action should be encouraged as it decreases the expected cost of the visited state. The policy network must not be updated with respect to other actions, because their merits are not known through the current experience;

- Action Selection – Selects an action to execute from the action policy. Initially, for exploration, random actions are generated stochastically.

While learning, both the evaluation and policy modules are adjusted. Setting up the learning parameters in AHC can be very difficult as the evaluation and policy modules perform tasks that are conflicting (Ribeiro, 1996). For the evaluation module to perform its task it needs plenty of *exploitation* to obtain the relevant feedback, while the policy module needs as much *exploration* of the state space as possible to correctly map the actions that lead to the best states for the optimal action policy. However, the actual assessment of the actions depends on good cost evaluation that must be provided by the evaluation module.

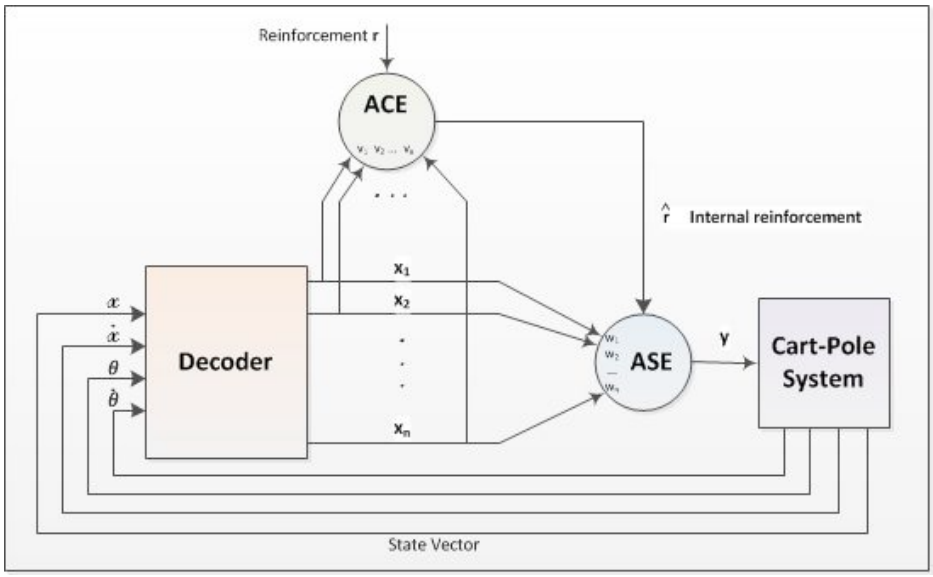

Figure 11 - ASE / ACE for pole-balancing task (Barto *et al*, 1983)

The previous figure shows the AHC method that was used in Barto *et al*, (1983) to solve the Cart-pole problem. In this figure, the block labelled Decoder corresponds to the State Sensing module of the generic block diagram. In this case, instead of generating a single output signalling the current input state, it activates one of its outputs $x_1 \ldots x_n$. This indicator of

the current input state is fed into both the evaluation module (ACE) and the policy module (ASE). Besides the indication of the current state, the ACE also receives information on the current reinforcement value ***r***. It uses both these values to produce a cost value $\hat{r}$ that is the internal reinforcement signal that feeds the ASE to obtain the actual action policy. In this model the ASE also provides directly the action signal ***y*** acting as the Action Selection module of the generic architecture.

Due to the fact that the State Sensing only supplies an indication of the current state, it is debatable if it could reside outside of the agent (in the environment).

3.3.2 Learning an Action/Value Function – Q-Learning

Reinforcement learning methods attempt to obtain an optimal *policy* for the agent to follow. Watkins (1989) presented a method that he called Q-Learning that attempted to learn the optimal policy without the use of a model of the environment. This method uses a simple structure, the *action-value function*, an extension of the *value function*, represented as *Q(s, a)*, that stores the value of executing action *a* in state *s*. The action-value that corresponds to the best policy is represented as *Q*(s, a)* allowing the optimal behavior to be obtained.

The optimal policy can be directly obtained from the Q* action-value function by selecting the optimal action from each state, the one with the highest value (this corresponds to a greedy approach). The relation between the optimal policy and the action-value function is:

$$\pi^*(s) = \text{argmax}_a Q^*(s,a) \tag{3.7}$$

We have seen the relationship between the action-value function and the policy, but we still have not seen how to obtain the actual action-value function. This can be achieved by converging the values, through iterative approximations, using the following update rule:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a')) \tag{3.8}$$

In the previous expression, the α factor is the learning rate that can have values ranging from 0 to 1. This learning rate factor influences the amount by which the Q-values are updated making for faster (nearer to 1) or slower (nearer to 0) convergence to the optimal values. Watkins has shown that if each action is executed an infinite number of times in each state and

α is decreased appropriately, then the Q-values will converge to Q* with probability 1, and the optimal policy (3.7) will have been found (Watkins, 1989). Q-Learning is exploration insensitive which means that convergence to optimal guarantee holds, independent of the exploration method used during the learning phase. Sutton (1990) explains this in greater detail when describing the Dyna architecture.

The relation between definitions (3.8) and (3.5) should be noted. The update of the Q-values *Q(s,a)* is in effect a temporal difference method. It is updated based upon the (estimated) value existing in *Q(s',a')*.

The Q-Learning algorithm can be described as follows.

3. Initialize $Q(s,a)$ arbitrarily
4. Repeat for each *step* in the learning process
   d) Observe the environment state s
   e) Choose an action a, based upon some exploration strategy
   f) Take action a, receive the reward r and observe the new state s'
   g) Make a backup of the Q-table using the rule:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

Listing 2 - Q-Learning algorithm

3.4 Learning with Environment Models

We have previously indicated that model-free algorithms need a great deal of experience to achieve good performance. We have also indicated that this is due to the fact that all the states have to be visited numerous times for the policy to start converging. Model-based algorithms were created to overcome this limitation by allowing the agent to "visit" states more frequently thus allowing for faster convergence of policies.

Similar to their model-free counterparts, model-based algorithms seek to obtain the optimal policy, but they do so relying on a transition model and a reward model. The transition model **T(s, a, s')**, as the name suggests, keeps information on the transition made from state ***s*** to state ***s'*** by executing action ***a*** from state ***s***. The reward model **R(s, a)** keeps track of reward information associated to executing action ***a*** in state ***s***.

At the start, the agent does not know anything about the world, and hence the models are empty, but it will build up the information in the models as it goes along.

Unlike the model-free algorithms that only obtain a real world experience per each step, model-based algorithms will execute a series of simulated experiences along with the real experience in each step. The real experience is used to update the models that are then used for the execution of multiple simulated experiences. In general, this form of proceeding speeds up considerably the act of obtaining the optimal policy.

Model-based algorithms are best suited for agents who are not lacking of resources, when computation is cheap compared to costlier real world experiences.

3.4.1 Dyna-Q

In Q-Learning, the agent learns (obtaining information via the reward signal) when transiting between states due to taking an action, *i.e.* it acts over the environment observing the results (an execution step). If a long sequence of steps is executed but only the last one provides a meaningful reward, all the previous steps will have been in vain as far as learning is concerned. The Dyna architecture proposed by Sutton (1990) is an extension to other algorithms (for example Q-Learning) that takes into account such a fact. This architecture mixes planning with direct reinforcement learning (by adding an internal model to the agent for the planning) and becomes advantageous whenever the costs of real world interactions supersede the cost of the excess computational power needed for the planning. The Dyna architecture applied to the Q-Learning algorithm produced what Sutton called the Dyna-Q algorithm (Sutton, 1990).

In the Dyna-Q architecture an internal model has been introduced, as shown in the following figure, so that the agent can perform additional steps (not directly resulting in externally executed actions) that allow a more rapid convergence of the Q-values for each "real" *step* taken. The model is updated every time the agent is informed of the consequences of the real action produced (once per real *step*). In general, this allows the agent to learn faster than with the traditional Q-Learning algorithm.

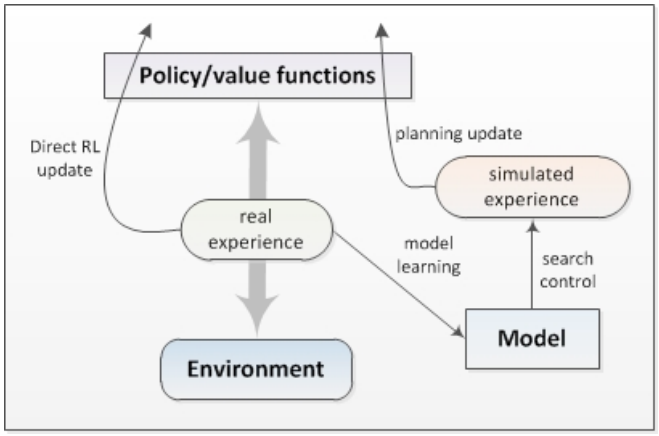

Figure 12 - Dyna-Q Architecture (Sutton & Barto, 1998)

When the environment in which an agent executes is deterministic, repeating an estimation (internal or simulated step) that has been known to produce a good result will almost certainly also produce a good result; it will reinforce the probability of executing a good action. On the other hand, if our environment is stochastic and there is only a certain probability that executing an action will lead to a certain state (it may lead to other states depending on the probability distribution function), then repeating an estimation over and over again (to reinforce the Q-values) may be misleading as we are fictionally inflating the Q-values when there is a probability that this might not be the most efficient action to take in that particular state.

The Dyna-Q algorithm uses the agent experiences to simultaneously build an internal representation of the world or model, and to adjust the policy for choosing the best action to take. The internal model for this purpose consists of:

$$Model(s,a) = \begin{cases} \hat{T}(s,a,s') \\ \hat{R}(s,a) \\ SA \end{cases} \tag{3.9}$$

In the previous definition $\hat{T}(s,a,s')$ represents an estimate of the state transition function (the probability of transiting to state ***s´*** from state ***s*** by way of action ***a***), $\hat{R}(s,a)$ represents an estimate of the reward function (the reward associated to taking action ***a*** in state ***s***) and *SA* represent a set of state-action pairs.

This is the differentiation point between Q-Learning and Dyna-Q. When interpreting an experience tuple, a Q-Learning agent would *only* update the action-value function (3.8) from actual experience while a Dyna-Q agent

would also update the action-value function by choosing ***n*** random state-action pairs from SA and creating some "pseudo experiences" from the model and using this to perform ***n*** additional updates to the action-value function *Q(s,a)*. If ***n* = 0** then the Dyna-Q algorithm degenerates to the Q-Learning algorithm. The higher the value of ***n*** the more processing power will be used between "real" observed experiences. This may bring serious limitations in some applications of the algorithm.

The general formulation for estimating the model $\hat{R}$ and $\hat{T}$ are:

$$\hat{R}(s,a) = \frac{\text{Sum of rewards received by taking action a in state s}}{\text{Number of (s, a) experiences}} \tag{3.10}$$

The estimated reward $\hat{R}(s,a)$ is the average value of the sum of rewards received when transiting out of state ***s*** by way of action ***a***, and

$$\hat{T}(s,a,s') = \frac{\text{Number of (s, a, s') experiences}}{\text{Number of (s, a) experiences}} \tag{3.11}$$

The probability of reaching state ***s'*** from state ***s*** by means of action ***a*** is the number of times that this state is actually reached divided by the total number of times state ***s*** is exited by means of action ***a***.

The values of $\hat{R}$ and $\hat{T}$ are updated at each "real" world experience and then used for calculating the updates to perform on the action-value function in the "pseudo experiences".

In the last step of the Dyna-Q algorithm, presented in listing 3, the update of the action-value function *Q(s,a)* is similar (but not equal to) the one from Q-Learning (3.7). The difference is the use of the estimated rewards and transitions instead of the "real" rewards and transitions. These estimated values would be propagated throughout the action-values bringing a faster convergence to the "real" final values. These additional updates to the action-value function are referred to as "pseudo experiences" for they are not based on interactions with the "real" environment.

The algorithm for Dyna-Q is described as follows.

1. Initialize $Q(s,a)$ arbitrarily
2. Initialize the internal $Model(s,a)$
3. Repeat for each *step* in the learning process
   a) Observe the environment state s
   b) Choose an action a, based upon some exploration strategy
   c) Take action a, receive the reward r and observe the new state s'
   d) Use the experience tuple (s,a,r,s') to update the internal $Model(s,a)$
   e) Make a backup of the Q-table using the rule:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

   f) Choose n additional state-action pairs from SA, and make n backups using the same rule as before:

$$Q(s,a) \leftarrow \hat{R}(s,a) + \gamma \sum_{s' \in S} \hat{T}(s,a,s') \max_{a'} Q(s',a')$$

Listing 3 - Dyna-Q algorithm

Now that we have given an overview of three main types of reinforcement learning methods, both with and without internal models, we will now proceed to examine the environment associated to our study, the Cart-Pole system.

4 Dynamic Equilibrium in a Continuous Environment

The previous chapter explored the issue of how an agent can solve a problem by using reinforcement learning. This chapter presents the problem we are solving by introducing the Cart-Pole inverted pendulum as a means for studying reinforcement learning with both a logical and physical view of the cart-pole system being presented. This chapter also introduces a framework for experimenting with reinforcement learning agents, the RL-Glue framework, which was used to approach the cart-pole problem.

4.1 The Cart-Pole Inverted Pendulum Problem

What is there so special about balancing a pole on top of a cart? Not much really. Then why is this so commonly used for demonstrating the capabilities and limitations of a reinforcement learning agent? (Michie & Chambers, 1968; Barto *et al*, 1983; Sammut, 1994; Ribeiro, 1996; Blynel, 2000). The answer might be found if we consider that reinforcement learning tries to solve the problem by simple trial and error. We could compare this to a human being handed a pole and told to balance it, without being given any instructions or any formal notion of physics. A similar situation is placed to the reinforcement learning agent; it should "learn" to balance the pole without any prior knowledge of the subject and without any other form of feedback except for the reward signal produced each time the agent tries to do something to balance the pole by executing an action.

The inverted pendulum system consists of a pole (which may or may not have an added weight at the end opposite the base) that is "fixed" at its base to a cart that can move freely in one dimension (horizontally with a left – right movement). The pole, while being "fixed" at the base, can oscillate freely along the same plane as the plane of the cart movement. Under these circumstances, the pole and weight component (normally simply referred to as the "pole") will behave like an inverted pendulum (thus the other more frequent name by which this system is known). The following figure illustrates what we have just described.

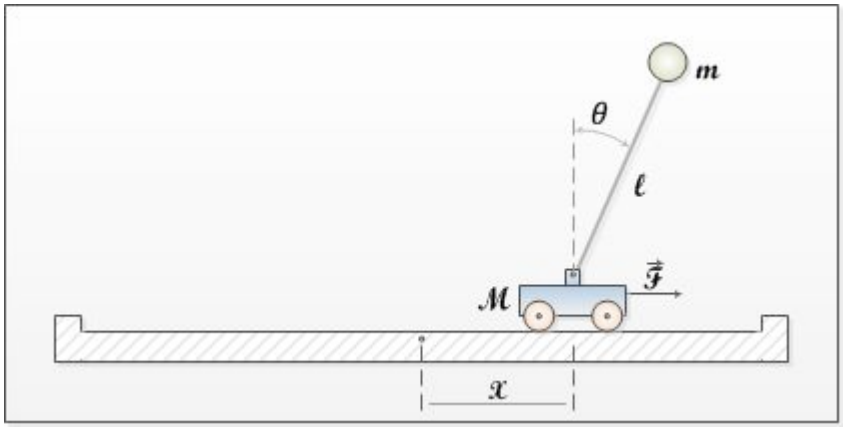

Figure 13 - Representation of a Cart-Pole (Inverted Pendulum)

The problem of "balancing" consists of finding an appropriate control policy that keeps the cart from reaching the ends of the track while at the same time keeping the pole from falling over (leaning more than a specific pre-defined angle).

The cart-pole problem is of historical importance to reinforcement learning because it was one of the first successful applications of an algorithm based on model-free action policy estimation, as described in 1983 by Andrew Barto and Richard Sutton in their pioneering paper (Barto *et al*, 1983; Ribeiro, 1996).

As the agent has no other knowledge of the system besides that which it discovers by direct execution of actions, "learning through direct experimentation in this case means that the agent must try to control the system a number of times but always receiving a reinforcement only when it fails to do so, until a satisfactory action policy is found" (Blynel, 2000). The most significant problem to be overcome in designing a learning algorithm for this task is that of credit assignment. The control system may make an incorrect choice as to whether to push the cart left or right. However, the consequences of that incorrect choice may not be noticed for some time, when the system finally fails. Many actions may have been taken between the incorrect choice and failure. So how can the learner decide which of those actions was truly incorrect? (Sammut, 1994).

In the classical study of this system (described in the next chapter), the perceptions given to the agent are composed of four variables X, $\dot{\mathrm{X}}$, Θ, $\dot{\Theta}$. These variables represent the position of the cart relative to the centre of the track, the linear velocity of the cart, the angle of the pole with respect to the vertical axis of the system and the angular velocity.

Once again, in the classical study, the actions that the agent is allowed to execute consist of applying a force of value $+F$ or $-F$ to the cart (known as *bang-bang* control). For each action executed, the agent will receive a reward value of minus one (-1) if the terminating conditions are met (pole falling past a certain angle or the cart going further than a certain distance from the centre of the system) or zero (0) for all other occasions. By receiving a negative reward, the agent is being "punished" for letting the pole fall down (or go out of range) while in normal circumstances (non-terminal states) he receives a "neutral" reward that is actually of higher value than the punishment.

4.1.1 The Physical Model

The current dissertation uses an abstract mathematical model to emulate the cart-pole inverted pendulum. We have not built, and/or performed any tests on anything other than a mathematical model; so all the results produced can only be discussed in light of this fact. Furthermore, the model we have adopted is the same as the one presented by Barto *et al,* (1983) in their work on Adaptive Elements due to the fact that we execute experiments on their original code (rewritten by us in the Java Language for the purpose of the experiments). Since we wish to compare the results obtained in our experiments with their results (used as a reference), we use the same model for all experiments.

The general model defined by Barto *et al* (1983) uses the following non-linear differential equations that are derived from the physics of an inverted pendulum:

$$\ddot{\Theta}_t = \frac{g \sin\Theta_t + \cos\Theta_t \left[\frac{-F_t - ml\dot{\Theta}_t^2 \sin\Theta_t + \mu_c \operatorname{sgn}(\dot{x}_t)}{m_c + m} \right] - \frac{\mu_p \dot{\Theta}_t}{ml}}{l\left[\frac{4}{3} - \frac{m\cos^2\Theta_t}{m_c + m} \right]} \tag{4.1}$$

and

$$\ddot{x}_t = \frac{F_t + ml\left[\dot{\Theta}_t^{\,2} \sin\Theta_t - \ddot{\Theta}_t \cos\Theta_t\right] - \mu_c \operatorname{sgn}(\dot{x}_t)}{m_c + m} \tag{4.2}$$

In these previous equations:

g = -9.8 m / s^2, acceleration due to gravity,

m_c = 1.0 kg, mass of the cart,

m = 0.1 kg, mass of the pole.

l = 0.5 m, half-pole length,

F_t = +/- 10.0 newtons, force applied to cart's centre of mass at time t.

μ_c = 0.0005, coefficient of friction of cart on track,

μ_p = 0.000002, coefficient of friction of pole on cart.

Even though the general formulas have been presented, it is possible to make some slight simplifications that do not affect the overall system too much. We can assume that the values of the friction coefficients μ_c and μ_p are sufficiently small that they can be reduced to zero (0). This simplification was made by Sutton & Barto (1998) in their code and followed by us in this work. Another such simplification made by these authors was to use Eulers method for calculating the evolution of the other variables ($\dot{\Theta}_t$, Θ_t, $\dot{x}_t$ and x_t) with a standard time step t of 0.02s. The final equations for all variables will thus be:

$$\Theta_{t+1} = \Theta_t + \dot{\Theta}_t t \tag{4.3}$$

$$\dot{\Theta}_{t+1} = \dot{\Theta}_t + \ddot{\Theta}_t t \tag{4.4}$$

$$\ddot{\Theta}_t = \frac{g \sin \Theta_t + \cos \Theta_t \left[\frac{-F_t - ml\dot{\Theta}_t^{\,2} \sin \Theta_t}{m_c + m} \right]}{l \left[\frac{4}{3} - \frac{m \cos^2 \Theta_t}{m_c + m} \right]} \tag{4.5}$$

$$x_{t+1} = x_t + \dot{x}_t t \tag{4.6}$$

$$\dot{x}_{t+1} = \dot{x}_t + \ddot{x}_t t \tag{4.7}$$

$$\ddot{x}_t = \frac{F_t + ml\left[\dot{\Theta}_t^{\,2} \sin \Theta_t - \ddot{\Theta}_t \cos \Theta_t\right]}{m_c + m} \tag{4.8}$$

4.2 Experimental Platform

In our quest to solve the Cart-Pole problem by using a Reinforcement Learning Agent, we used an existing framework for creating a prototype to execute all of the tests performed in this dissertation. We opted to use the RL-Glue framework, mainly due to the following aspects:

- It is tailored for use of Reinforcement Learning;
- The flexibility of the design – the actual design of the framework fits in perfectly with our requirements allowing repeatability and traceability;
- It's independence of programming languages allows for the use of whichever of the supported languages preferred (or even a mix of languages);
- The existence of an "optimized" mode of use when used solely in Java (which we will explain later);
- The existence of code already developed which we could use to assert our findings;
- The fact that it was developed and used in an academic environment (the University of Alberta) by one of the references in this area, Richard Sutton.

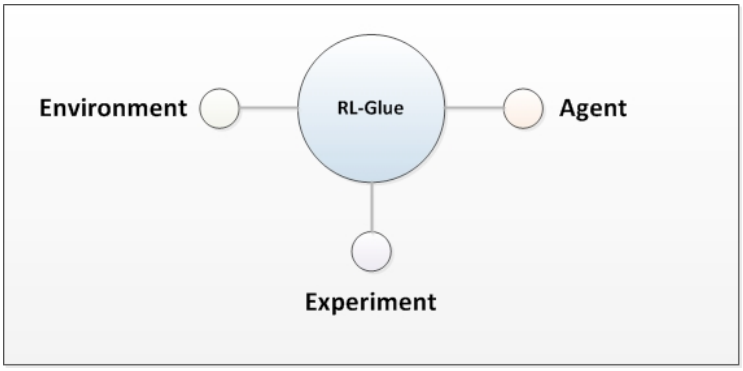

Figure 14 - RL-Glue Architecture

RL-Glue implements a centralized core or "hub and spoke" model. In the centre of the framework architecture we can find a messaging switch (or router), which is normally accessed by connections via sockets. The developers of the framework have provided libraries for various programming languages (C/C++ and Java in internal mode – agent, environment and experiment all linked into a single thread of execution – and C/C++, Java, Python, Lisp and Matlab in external mode – messages are passed between separate agent, environment and experiment threads of execution), which allow the use of the framework hiding most of the complexities of the underlying message system. It is due to the fact that the framework is based upon the exchange of text messages that it allows the use of multiple programming languages in simultaneous. Though this is an important benefit in using the framework for investigation, it is a major drawback if we were to

use it in an actual "production" environment. The overhead associated to having to "translate" all the messages from their object representation to text and back again could make the use of such a framework in a real-time architecture inadequate.

Further to this centralized messaging system we have the various components that hook-up to the messaging system. These are the implementation of an *Agent*, the implementation of an *Environment* and an *Experiment* that controls the flow of execution. At this point it is worth mentioning that this framework is meant for a single agent (does not support multiple agents), which does not pose a problem for us since we are studying the implications of environment changes on the behaviour of a single agent anyway.

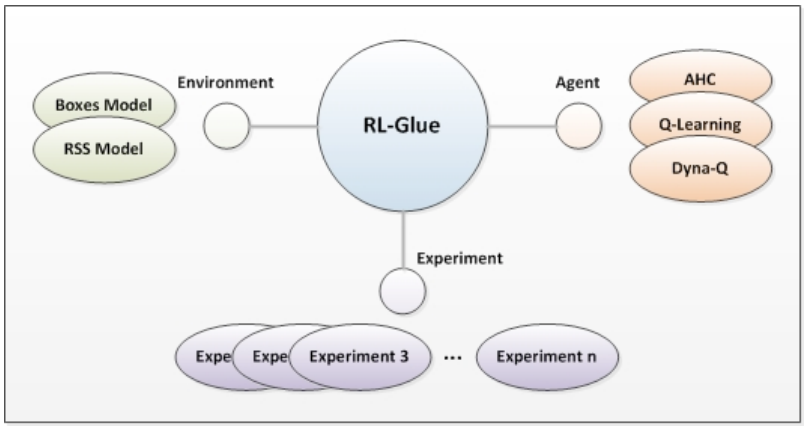

Figure 15 - RL-Glue as used in the tests

After a quick overview of how the framework is comprised, we now explain how it was used in the work presented in this dissertation, but before we do so, there is an important aspect of the framework that we have not talked about up until now. This is the fact that if all the code involved is Java, it is possible to use the framework in the exact same manner as has been described, except that the messaging interfaces are done directly and not through the central "hub and spoke" socket mechanism. In this case, all the involved components (agent, environment, experiment) run in the same Java thread[3]. However, for what we are concerned, this has no implication. It might only be an issue if we were to act upon the information in real time, which is not the case as the elapsed time is a controlled variable, and even so it would only benefit us as the execution is somewhat faster than through the normal messaged interface. If we were to implement a "real" robot that had to

[3] This is called the internal mode as opposed to various programs executing in their own threads communicating by text messages through sockets, which is called external mode.

act in real-time, then the choice of framework, or even the mode of operation of the framework, would certainly be very important.

In our prototype, we created a testing platform based upon the RL-Glue framework. We used Java because of the possibility given by RL-Glue of simplifying the execution model, which contributes to the reduction of the complexity of the system. Overall, for this prototype we implemented two distinct environments: one which reproduces the Barto *et al* investigations on the BOXES model and another of our own design in which we use a reduced state set for the model. Besides the two environments, we also created three different agents: an implementation of Barto *et al* AHC agent, a Q-Learning agent and a Dyna-Q agent. We also created various experiments to test various hypotheses for improving the agent's performance.

5 A Classical Model for the Cart-Pole Problem based on the BOXES Algorithm

In this chapter we present a model, based upon the BOXES algorithm that has become a benchmark for evaluating agents using the Cart-Pole inverted pendulum dynamics. We explain how the state set, action set and reward function are defined for this model. Further in this chapter we examine the way three different agent architectures (AHC, Q-Learning and Dyna-Q) are set up to solve the inverted pendulum problem using this model. We also take a look at the experimental results obtained by these three agents.

5.1 State/Action/Reward Representation

The BOXES algorithm, first described by Michie & Chambers (1968), was developed for learning to control dynamic systems. It has been successfully applied as a benchmark for many experiments in control tasks such as pole balancing by trial and error (Sammut, 1994).

5.1.1 Representing State

The Cart-Pole inverted pendulum can be determined by four variables: the pole angle Θ, the pole's angular velocity $\dot{\Theta}$, the horizontal displacement of the cart X, and finally the horizontal velocity of the cart $\dot{X}$. We thus can state that the system can be determined by a four-dimensional state space.

The BOXES algorithm derives its name from the way in which it partitions the state space. The state space is partitioned into regions (boxes) by dividing the range of each dimension into intervals (Sammut, 1994). Barto *et al* (1983), in their study of this problem, have used a similar approach to obtaining the state space by using the combination of the ranges of the four linear variables to calculate a single discrete value that is then used as the indicator of the current state.

We can break down the state space into two categories: terminal states and non-terminal states. The terminal state (only one in this case) is obtained

whenever the X value exceeds the length of the track (2.4 metres in either direction counting from the middle of the system) or whenever the pole angle Θ is greater than twelve degrees (could be $+12^o$ or -12^o, *i.e.* to either side of the vertical). All other situations will translate to a non-terminal state corresponding to a specific combination of values of the four-dimension variables. The ranges for the variables are obtained as follows:

- For the position of the cart X:
  - –2.4 m <= X < –0.8 m
  - –0.8 m <= X < 0.8 m
  - 0.8 m <= X <= 2.4 m

- For the cart velocity $\dot{X}$:
  - –infinity m/s < $\dot{X}$ < –0.5 m/s
  - –0,5 m/s <= $\dot{X}$ < 0.5 m/s
  - 0.5 m/s <= $\dot{X}$ < +infinity m/s

- For the pole angle Θ:
  - -90^o <= Θ < -6^o
  - -6^o <= Θ < -1^o
  - -1^o <= Θ < 0^o
  - 0^o <= Θ < 1^o
  - 1^o <= Θ < 6^o
  - 6^o <= Θ <= $+90^o$

- For the pole angle angular velocity $\dot{\Theta}$:
  - –infinity deg/s < $\dot{\Theta}$ < –50 deg/s
  - –50 deg/s <= $\dot{\Theta}$ < 50 deg/s
  - 50 deg/s <= $\dot{\Theta}$ < +infinity deg/s

The combinations of the previously defined ranges give a total number of 162 distinct states.

It should be noted that this is just one possibility of defining ranges for dividing the state space for this problem. We have used this particular division because it is the same division of state space used by Barto *et al* (1983).

5.1.2 Representing Action

The BOXES algorithm, the work from Barto *et al* (1983), as well as all of the Cart-Pole implementation we have analysed, comprised an action space of

two values: apply a left oriented force and apply a right oriented force. This two-action approach is commonly known as a "bang-bang" controller. Whenever the agent returned an action consisting of "apply a left force", a force was applied to the base of the Cart-Pole oriented to the left (corresponding to a push to the left). Whenever the agent returned an action of "apply a right force", a force was applied to the base of the Cart-Pole oriented to the right (corresponding to a push to the right). We will study the implications of this definition of actions in subsequent sections of this document as well as propose different semantics for these same actions.

5.1.3 Representing Reward

In the BOXES algorithm, the reward function associated to the Cart-Pole problem is a very simple one. A reward of value 0 (zero) is given to the agent whenever any non-terminal state is reached and a reward of value -1 (minus one) is returned when the agent reaches the terminal state. We may consider that this is not actually a reward for keeping the pole in an upright position, but more of a "punishment" for dropping the pole. It should be duly noted that this reward mechanism provides absolutely no knowledge to the agent about how he should balance the pole, and also no knowledge about what the best states are.

As has been stated elsewhere (about the state distribution), this is one possible form of defining the reward values. Other value distributions are possible. However, due to the fact that we wish to evaluate implications of producing changes in this distribution, the original values have been used to determine a base reference.

5.2 Agent Implementations

For the experiments used to evaluate the BOXES model, we have developed three different agents based upon three different algorithms: AHC (Adaptive Heuristic Critic), Q-Learning and Dyna-Q. We have tested these agents in accordance with the parameter set defined by Barto *et al* (1983) in their approach to solving the Cart-Pole problem (see the Experimental Results section).

5.2.1 AHC

This agent was developed by first porting to java the original Barto *et al.* code for the Cart-Pole problem and later adapting this code to work with the RL-Glue framework. The output produced by executing the original "C" language code agent and the output produced by the original Java translation of that code can be found in Appendix B. These outputs can be seen graphically in the following figure.

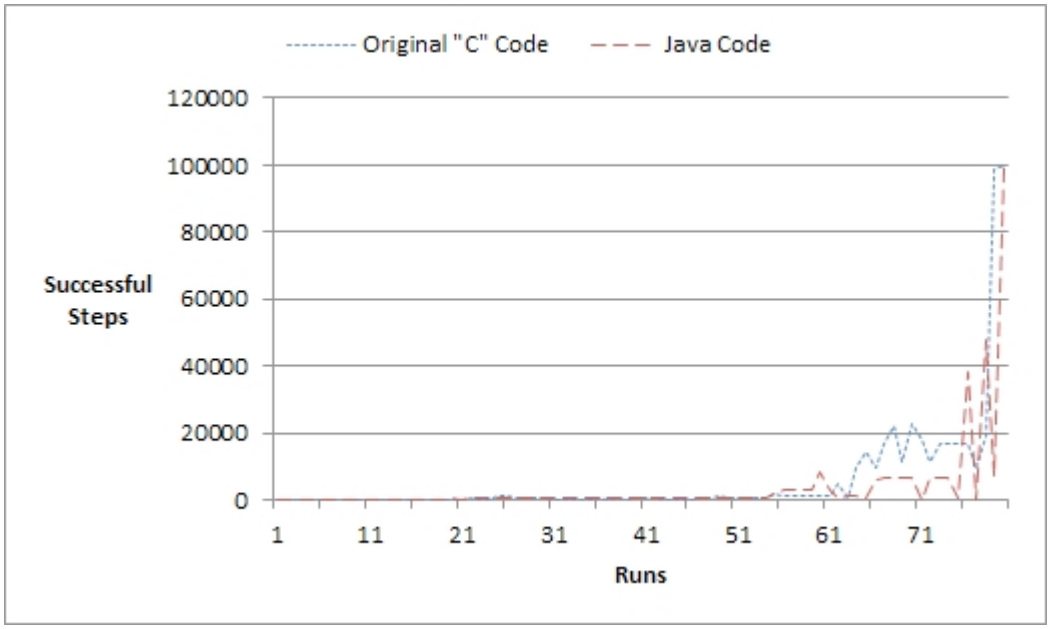

Figure 16 - Comparison of original AHC code with Java version

The global results are the same. The minor differences are a result of the random nature of the exploratory component of learning mechanisms.

During the execution of the experiments, the following values were used for the internal parameters of the AHC agent.

Table 1 - Values for internal parameters for the AHC agent

Item	Definition	Value
α	Learning rate for action weights	1000
β	Learning rate for critic weights	0.5
γ	Discount factor for critic	0.95
λw	Decay rate for $\boldsymbol{w}$ eligibility trace	0.9
λv	Decay rate for $\boldsymbol{v}$ eligibility trace	0.8

In all of the execution of code using this algorithm (whether with the original "C" code, with the Java equivalent or with the RL-Glue agent

version), we did not change the values of the internal parameters and always assumed the existing ones. This was done purposefully to be able to establish any comparison with the original values published as well as allow any other future research to compare with these values.

5.2.2 Q-Learning

The Q-Learning agent used in the experiments was developed from the theoretical algorithm presented in section 3.3.2 and converted for use with the RL-Glue framework.

The values for the internal parameters were obtained experimentally and correspond to the best behaviour observed.

Table 2 - Values for internal parameters for the Q-Learning agent

Item	Definition	Value
α	Learning rate	0.5
γ	Discount factor for future reinforcements	0.999
policy	The estimation policy	greedy

5.2.3 Dyna-Q

The Dyna-Q algorithm enhances the Q-Learning algorithm by using the agent experiences to simultaneously build an internal representation of the world or model, and to adjust the policy for choosing the best action to take.

During the course of our experiments, we verified that the "normal" form of constructing the internal model was incompatible with the highly dynamic and non-deterministic environment associated with the Cart-Pole. Under normal circumstances (in a deterministic environment), the model would only account for the sum of the received rewards in a given state-action pair along with indication of which state followed when taking the indicated action from a given state (3.9).

When the agent is faced with a problem in which the next state traversed to from a given state-action pair is not constant, the model has to be adapted to support a possible transition from the given pair to any of the existing states in the model (including the originating state and also the terminal state). Even though this model (3.9) uses the information on transitions to all the possible end states from a given start state, in our findings it did not

produce correct results. This is because the rewards kept in the model did not take into account the frequency with which the changes occur, that is, if a given tuple (s_1, a_1, s_2) occurs 90% of the time and another such tuple (s_1, a_1, s_3) occurs only 10% of the time, then the reward should reflect this distribution and should be discounted proportionally (the states and action mentioned are just for exemplification purposes).

The following alternative model allows the previously mentioned situation to be covered:

$$Model(s,a) = \begin{cases} \hat{T}(s,a,s') \\ \hat{R}(s,a,s') \\ SA \end{cases} \tag{5.1}$$

The change produced to the model is the substitution of $\hat{R}(s,a)$ by $\hat{R}(s,a,s')$ which describes accurately a proportional distribution of the accumulated reward of a tuple taking into account the frequency in which this tuple occurs. This produces changes in the Dyna-Q algorithm in the last step presented. The new formula for the last step should now read:

$$Q(s,a) := \hat{R}(s,a,s') + \gamma \sum_{s' \in S} \hat{T}(s,a,s') \max_{a'} Q(s',a') \tag{5.2}$$

As was done in the Q-Learning algorithm, the best possible observed results for the internal parameters are as follows:

Table 3 - Values for internal parameters for the Dyna-Q agent

Item	Definition	Value
α	Learning rate	0.5
γ	Discount factor for future reinforcements	0.999
NSIMUL	Number of simulated steps per "real" observed step	50
policy	The estimation policy	greedy

5.3 Experimental Results

In order to set a benchmark for all the tests performed during the elaboration of this dissertation, three RL agent architectures were run against a set of default parameter values for the Cart-Pole problem. The three types of agent algorithms were: AHC, Q-Learning and Dyna-Q. The relevant parameters for

each of the agents have already been previously presented. Here we enumerate the remaining global parameters used in the experiments.

The following table shows information that is common to all of the tests performed, *i.e.* the default parameter set.

Table 4 - Common values used in the tests

Item	Definition	Value
Vertical Axis	Number of successful steps achieved by the agent (limited by code to a maximum value to prevent "infinite" runs)	100.000 (maximum)
Horizontal Axis	Number of runs repeated per episode	600
Episodes	Number of repeated episodes per experiment (in the end the values per run are averaged over all the episodes)	20
Window size	The window size (number of values) considered when averaging a single value to be output (after the previous averaging of episodes is completed). Each value output is an average of its own value plus the 19 previous to it.	20
τ	Physics time-tick, the delay or time elapsed between successive calls to agent *step* code.	0.02 seconds
States	Number of states as defined by (Sutton & Barto, 1998). There are actually 162 "normal" states and 1 terminal state. See the relevant section of this document for further information on state definition.	163 states, {-1, [0…161]}
Actions	0 – produces a "Left Force" of 10 N, 1 – produces a "Right Force" of 10 N,	{0, 1}
Reward Function	A reward of -1 (minus one) on terminal states, A reward of 0 (zero) on all non-terminal states	{-1, 0}

All of the results presented (in this chapter and also in the following chapter as well) are a result of executing a series of 20 episodes (Episodes parameter) with 600 runs each (Horizontal Axis parameter). The results of these episodes are stored in a matrix (of 20 by 600 values) and at the end of the series of episodes, the values are averaged over the runs. This results in a single episode composed of 600 "averaged" values (one per each run).

The results of executing the three agent types against the default parameter set are presented in the figure below.

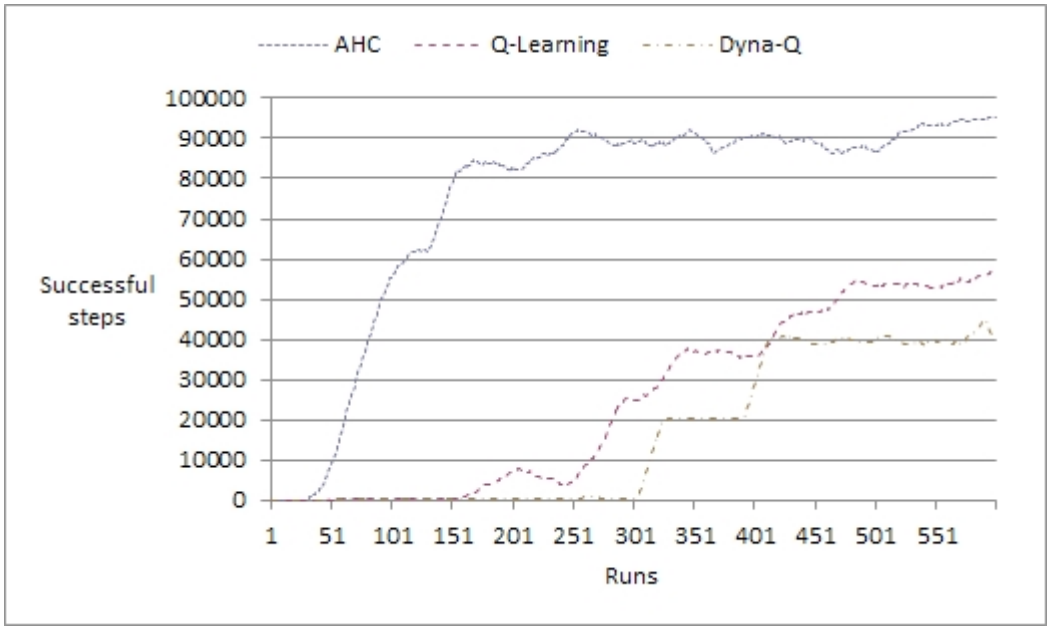

Figure 17 - Reference step count values using default parameters

From the analysis of the results we can conclude that the AHC agent performed as was to be expected by comparison to the pattern of values published by Sutton and Barto (1998) (refer to Figure 16).

The difference between the graph shown here and the graph in Figure 16 is the number of runs. Whilst in the former (Figure 16) the original C language code executed only once (single episode) and stopped executing as soon as it reached the 100.000 step mark (at around run 80), in the latter (the figure shown here) the code was run for 20 episodes (with the values averaged out amongst those 20 episodes), and each run was ended if and when it reached the 100.000 step mark, starting a new run until achieving a total of 600 runs. The original C code was not modified to execute for 600 runs (aborting each run at the 100.000 step mark) to guarantee that the results produced were not influenced in any way allowing for comparison to the results published by Sutton and Barto.

The Q-Learning agent had a hard time of learning in this "harsh" environment. This behaviour was to be expected as Sutton and Barto had provided an agent (AHC) that was better adapted to what they called "extreme conditions" implemented in the test environment. In their work, Sutton and Barto defined a series of initial parameters (presented partially in the common values table and the AHC agent parameters) that they concluded

to be on the limits for an agent of such type to be able to solve the inverted pendulum problem.

The results of the Dyna-Q agent were lower than our expectations as we expected them to be at least as good as the equivalent Q-Learning algorithm. If we consider that a Dyna-Q agent is a Q-Learning agent that does some internal planning, then we would expect that the Dyna-Q agent perform better than the Q-Learning agent. The observed results are due to the fact that the environment is very dynamic and non-deterministic, these characteristics pose more of a problem to the Dyna-Q agent than to the simple Q-Learning agent, due to the reliance on learning from simulated experience based on the internal model.

5.4 Analysis of Results

Experiments were conducted with each of the three types of agents and the resulting values were gathered. For the AHC agent, there were no changes made to internal parameters, as we wanted to use the same ones as those published by Sutton & Barto (1998). For the other two agents, the internal parameters were tuned until the best performance was observed. After this, the parameter values were registered and used for all other experiments made with these agents.

Using the environment conditions as specified by Sutton and Barto, all three agents were subject to a final run for information gathering. The information gathered was used to produce the graph of Figure 17. From these results, we concluded that the AHC agent performed as Sutton and Barto had indicated being capable of completely solving the Cart-Pole problem.

The Q-Learning agent was not capable of reaching the imposed upper limit of successful steps (100.000). Though the results show a constant increase in number of successful steps, we verified that the upper limit stabilized between 50.000 and 60.000 steps after 500 runs (we executed a total of 1000 runs several times and the 500 run limit was common in all executions). For this reason (result stabilization after 500 runs) we decided to only show the first 600 runs of all executions as from there on the values did not show signs of changing tendencies.

The Dyna-Q agent performed worse than the Q-Learning counterpart, which in our opinion is due to the non-determinism of the environment. Convergence to the optimal policy should have happened sooner than in

Q-Learning, but this did not happen because the state transitions were very dynamic, reducing the effectiveness of using an internal model.

6 An Alternative Model for the Cart-Pole Problem

The previous chapter discussed the classical BOXES model for solving the Cart-Pole problem. In this chapter we define an alternative solution for the Cart-Pole problem by defining an alternative State and Action set. The implications of these changes are discussed as well as the changes needed to the AHC, Q-Learning and Dyna-Q agents, due to the alternative state/action sets. The experimental results of changing action semantics and changing the number of states are also presented in this chapter.

6.1 State/Action Representation

The BOXES model has been a good base for the study of the Cart-Pole inverted pendulum problem. Many researchers have used this model to test their hypothesis on how to solve this classical problem. However, it has been pointed out by some of these researchers (for example Sammut, 1994) that this model can be improved. In this chapter an alternative model for the study of the Cart-Pole problem is proposed and its effects on the defined reinforcement learning mechanisms are studied.

6.1.1 Representing State

Watkins and Dayan (1992) proved that Q-Learning will converge to the optimal policy when all the state/action pairs have been visited an "infinite" number of times, so it stands to reason that the fewer the number of states, the faster this convergence will be obtained. With that argument in mind, we set about to discover a way of reducing the state set from 162 non-terminal states to a lower cardinality of non-terminal states. We managed to obtain this reduction by doing away with ranges for the four variables (X, $\dot{X}$, Θ and $\dot{\Theta}$) and only considering their convergence (or divergence) to zero (in absolute values). Convergence was measured by checking if the current value is less than (in absolute terms) the previous value for any given perception. If so, then convergence is true and a value of 0 is assumed. If not, the values are diverging and a value of 1 is used. This allowed us to go from the original 162 ($3 \times 3 \times 6 \times 3$) states down to a reduced set of 16 ($2 \times 2 \times 2 \times 2 = 2^4$) states.

The new perception variables are:

$$\delta X = \begin{cases} 1 : |X_{t-1} < X_t| \\ 0 : |X_{t-1} \geq X_t| \end{cases} \tag{6.1}$$

$$\delta \dot{X} = \begin{cases} 1 : |\dot{X}_{t-1} < \dot{X}_t| \\ 0 : |\dot{X}_{t-1} \geq \dot{X}_t| \end{cases} \tag{6.2}$$

$$\delta \Theta = \begin{cases} 1 : |\Theta_{t-1} < \Theta_t| \\ 0 : |\Theta_{t-1} \geq \Theta_t| \end{cases} \tag{6.3}$$

$$\delta \dot{\Theta} = \begin{cases} 1 : |\dot{\Theta}_{t-1} < \dot{\Theta}_t| \\ 0 : |\dot{\Theta}_{t-1} \geq \dot{\Theta}_t| \end{cases} \tag{6.4}$$

This change has an added benefit of removing dependencies from the values used for the limits in the ranges by only considering the tendency to converge to zero (or diverge from it). Because state definitions are now independent of the actual values of the variables, we have eliminated any problems that may have existed due to the use of continuous variables rather than discrete ones.

6.1.2 Representing Action

In our study of the Cart-Pole problem, associated to the new reduced state set (16 non-terminal states plus 1 terminal state), we decided to change the semantics of the two existing actions from the FORCE-LEFT, FORCE-RIGHT meanings to CONVERGE and DIVERGE.

A typical resolution of Cart-Pole problem uses a two-action *bang-bang* model for the actions. This has been explained as resulting in producing a force either pushing the pole to the right or to the left depending on the action indicated.

Please recall that the change of state set to a reduced state set as discussed in the previous section should (without applying any other changes) result in a mass failure for all agents. An interpretation of the actions as producing a left oriented force and a right oriented force should fail miserably if there is no laterality information present. The *bang-bang* actions worked

fine when we could associate a state as belonging to either side of the pendulum (the state definition was symmetrical as of the vertical) or the cart being on either side of the centre of the track. Due to the new 16 state model, we now have a lack of laterality information and as such, we need to adopt a new semantics for the actions: CONVERGE (produce a force oriented to making the pendulum vertical or produce a force making the cart centred) and DIVERGE (produce a force sending the pendulum away from the vertical or away from the centre of the track). The actual decision of whether a variable is converging or diverging depends on a comparison of its actual value against its previous value. This still leaves the problem of deciding if it should be a positive or negative force, when compared to the horizontal axis of the pendulum or to the centre of the track. Each of the possibilities were considered and tested.

A preliminary analysis of the variables breaks them down into two categories, those that are independent and those that are derived. The X variable is the distance measured from the centre of the cart to the centre of the system and the Θ variable is the angle of the pole relating to vertical. Both of these variables are independent of any of the other variables (they just depend on the physics).

The $\dot{X}$ variable and the $\dot{\Theta}$ variable are both derivations of their corresponding distances (linear and angular respectively) and are thus velocities. They are, as such, dependent on the values (more accurately on the changes occurring in those values) of the respective distances. Furthermore, the fact that these two variables are velocities makes them vectors, *i.e.* their speed is measured as the difference in space over a period of time (and this is always positive or zero) and the direction depends on the way it is being measured. It is possible to use these variables for choosing between Convergence and Divergence but it is not possible to infer the direction to know if we should apply a leftward force or a rightward one. It is also possible to have a moving object with a constant velocity. This means that the difference between two consecutive measurements will produce zero and be inconclusive regarding convergence/divergence and/or direction. The experimental results presented in the next section confirm this analysis.

Excluded the velocities, we are left with the two independent variables. Each of them can be used for determining both the convergence/divergence and also the direction of the force to apply (based upon the sign of the last reading of the variable). One of them concentrates on keeping the cart near the centre of the system, while the other concentrates on keeping the pole vertical. While both seem to produce useable results, one of them produces better, more consistent, results than the other. The best result should be

obtained by using the X variable due to the applying of a force to the cart-pole system resulting in a direct change in the cart's position. On the other hand, the Θ variable does not depend only on the force applied to the cart, but also on the effect of gravity.

6.2 Agent Implementations

The agents used in this section have suffered changes to reflect the new state and action representation. The internal parameters of all the agents have been purposefully maintained at their previous values. This was done to assess the degree of improvement obtained (if any) on the agent's behaviour by the changes made solely in the environment representation (the state and action set). Any change to these parameters at this time may influence the behaviour and as such corrupt the assessment of the impact of the changes being made.

In what concerns the reinforcement learning mechanisms, no changes were made to any of the agents in order to study the effect of changes in the internal representation independently of the learning mechanisms.

6.3 Experimental Results

The following sections present the experimental results obtained by testing the various hypotheses mentioned in previous sections.

6.3.1 Alternative Reduced State Set

The alternative reduced state set, as expected due to the lack of laterality information, did not obtain a performance gain in any of the agents. The results for the agents averaged 90 steps per failure for the AHC agent and averaged 30 steps per failure for the other two agents (Q-Learning and Dyna-Q). The curve could be considered stable after the 600 runs for the AHC agent but it was gradually climbing for the other two agents.

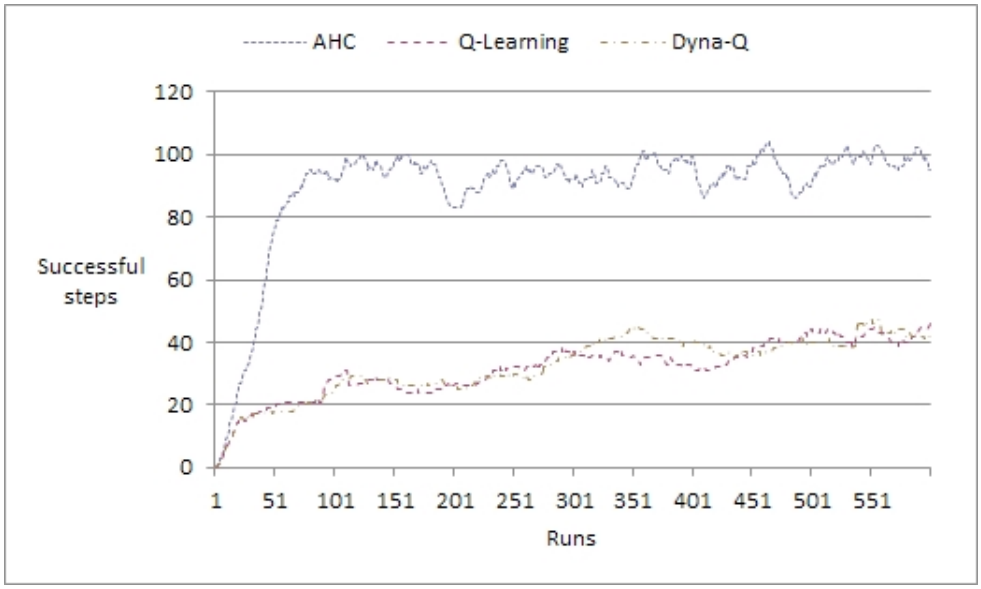

Figure 18 - Reduced State Set

These results are not an indication of failure, mainly because they were expected. In the previous scheme for obtaining the state set, there was information about whether or not the cart and/or the pole was to the left of the origin or to the right; there was information on laterality. In the new state set scheme we simply look for convergence but we keep absolutely no information as to whether the cart/pole is converging from the left or converging from the right. If we consider that our agents produce *bang-bang* information of push-left and push-right, we are now faced with a problem of the agents not knowing when to apply a left-force or when to apply a right-force. The solution to this problem is to change the semantics of the actions produced from Left-Right to Converge-Diverge.

6.3.2 Alternative Action Semantics

As predicted in a previous section, the two derived variables, $\dot{\Theta}$ and $\dot{X}$, did not perform well as the variable for testing convergence. They managed average values rounding the low hundreds of steps before reaching a terminal state (the pole either falls down or the cart leaves the boundaries). The following two graphs illustrate these findings.

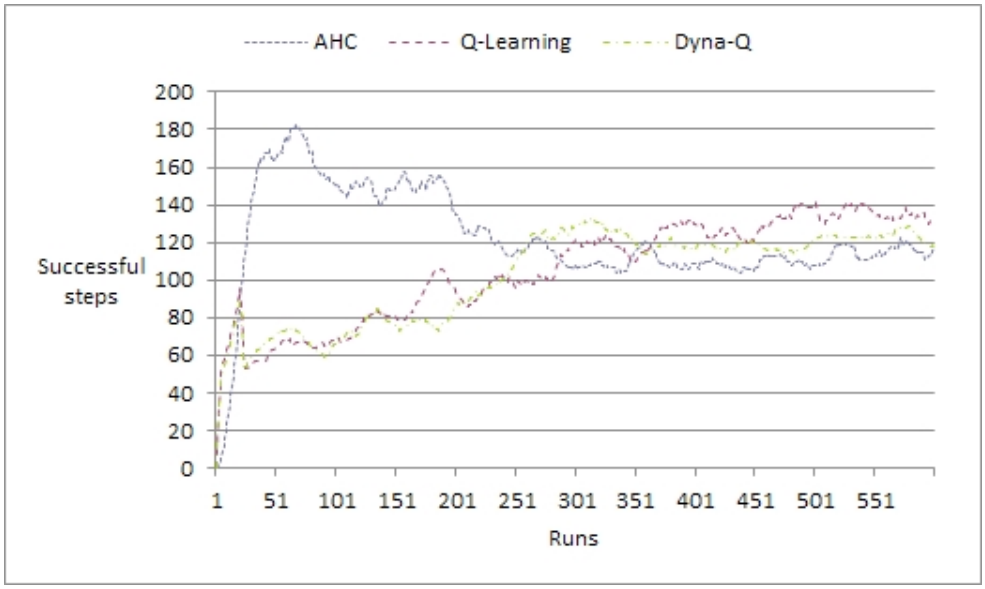

Figure 19 - Use of $\dot{\Theta}$ as the action convergence variable

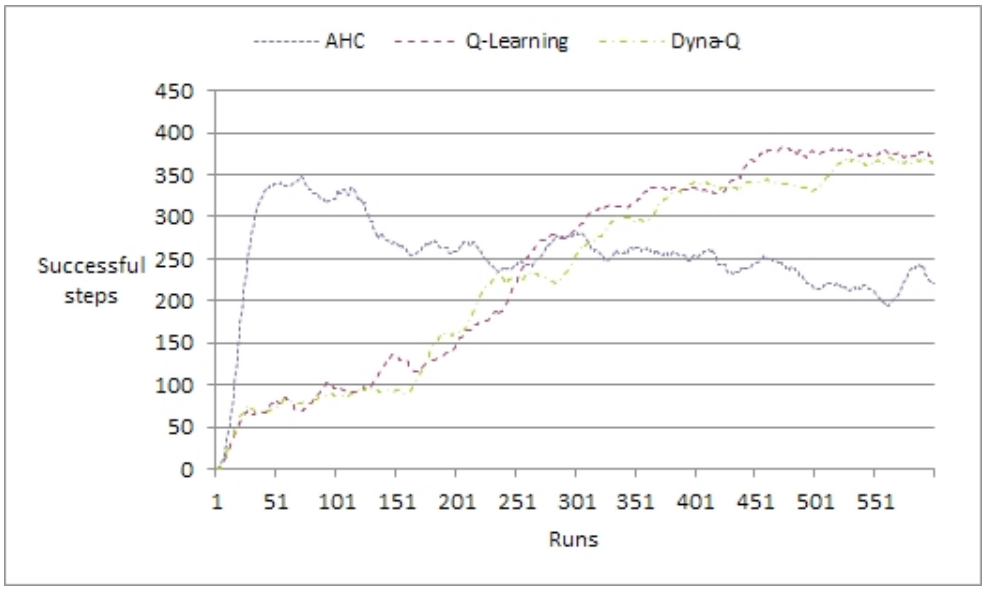

Figure 20 - Use of $\dot{X}$ as the action convergence variable

One curious aspect of these tests is the fact that even though both do badly, $\dot{X}$ manages to do better than $\dot{\Theta}$. This is a consistent behaviour that we will see again in the tests made using the independent variables X and Θ.

Tests made with the Θ variable produce moderately adequate results. The pole is kept balanced for an average of 50.000 steps using any one of the three types of RL agent algorithms.

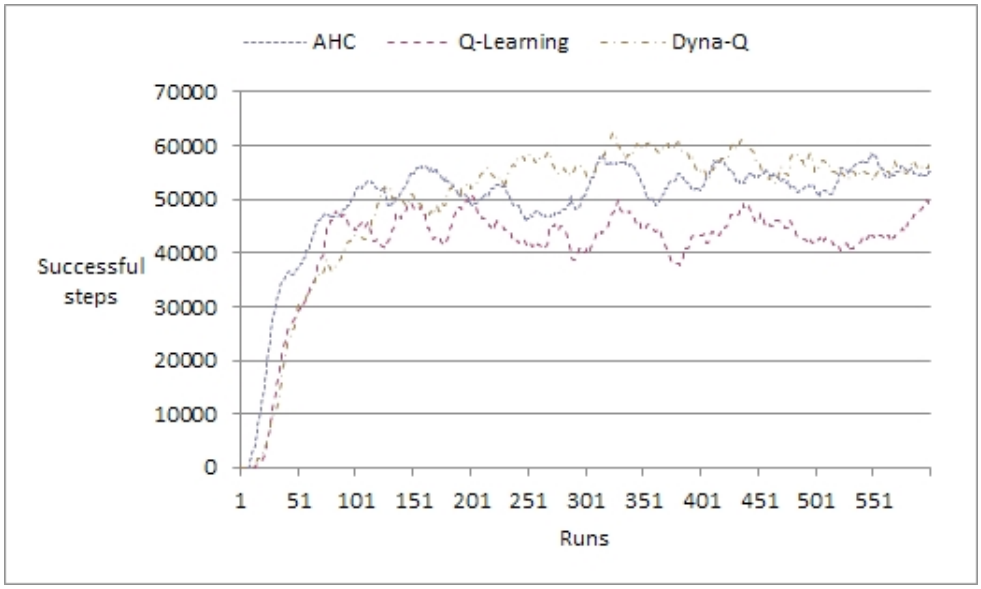

Figure 21 - Use of Θ as the action convergence variable

The main reason for failure in this case is the drift produced when stabilizing the pole. Because the agents are paying more attention to the pole angle than the actual distance from the system centre, they start to let the cart slide away from the centre eventually ending in a terminating condition corresponding to the ends of the track being reached.

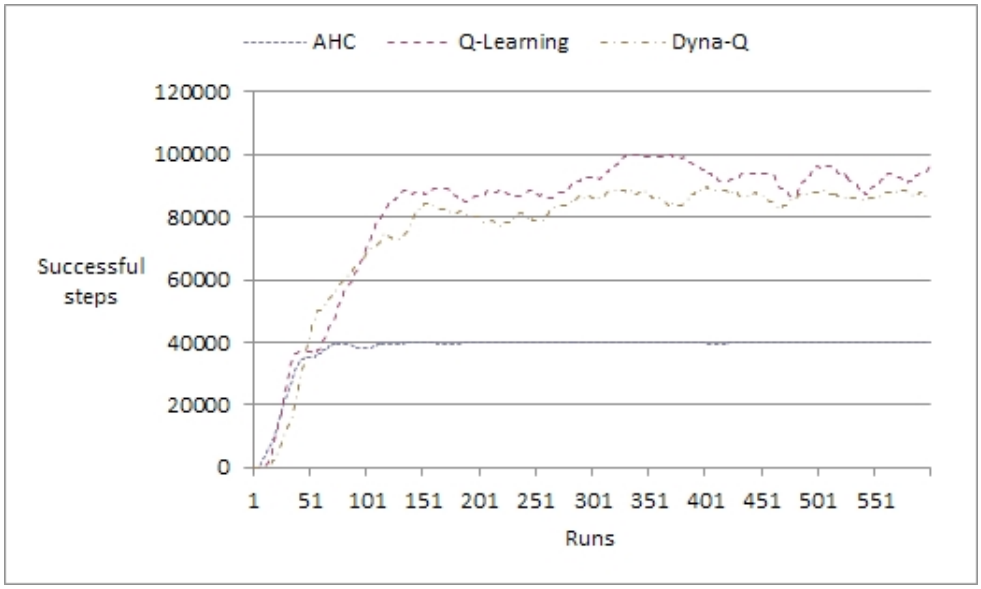

Figure 22 - Use of X as the action convergence variable

The best performer using the new action semantics is through the use of the X variable as the convergence variable, as can be seen in the previous figure. In this case, the new semantics for the two actions produced a massive improvement in performance on the Q-Learning and Dyna-Q algorithms. These two algorithms achieved an average of circa 90.000 steps before failure.

This is even more impressive if we consider that neither of the algorithms was changed or tuned in any way. The AHC agent was very consistent (at around 40.000 steps per run) but clearly performed worse than in the reference scenario.

Our suspicions about the results of the AHC agent have to do with the changes made to the environment representation (state and action sets) without having updated the internal parameters of the algorithm. Kaelbling *et al* (1996) have suggested that the AHC algorithm is prone to be too dependent on the correct definition of the internal parameters for any given problem. The fact that we have changed the environment settings without updating the internal parameters of the agent seems to support this theory.

In order to support this theory, we have changed the value of the β parameter (learning rate for critic weights) from the default 0.5 to a new value of 0.7. We have verified that with this new value the average results are no longer of 40.000 but 60.000. Further tuning to the variables would most certainly produce other significant changes to the values. However, no further changes were produced and tested, as it was not our intention to evaluate performance based upon changes to the internal parameters.

6.4 Analysis of Results

In attempting to improve the performance of our agents, we have created a new set of perceptions based upon the convergence to a reference value of the previous perception variables (0 meters for the position of the Cart and 90º, or vertical, for the position of the pole). Due to this change, we managed to reduce the agent state space from 162 states (from the BOXES model) to just 16 states (our reduced state set model) by using the new δX, $\delta \dot{X}$, $\delta \Theta$ and $\delta \dot{\Theta}$ variables. However, the mere reduction of states was not enough for the agents to perform better. In fact, by just changing the state set the agents all performed much worse (on average, none were able to balance the pole farther than 100 steps). This behaviour of the agents was due to the lack of information regarding the correct direction in which to apply the force (laterality information).

The solution to this second problem was to change the action set (just the semantics as the number of actions remained the same). The meaning of the actions, for the environment, passed from “apply a left or right oriented force” to “apply a converging or diverging force”. While this solution eliminated the laterality problem, it introduced a new problem related to finding whether to converge or diverge.

The answer to this new problem came by analysing one of the original perception variables to decide whether to apply a converging force (in the direction of making the Cart go nearer the centre of the system) or a diverging one. The position variable X was chosen due to being the only one providing a direct relation between the force applied and the resulting motion of the system. This new form of applying forces to the Cart resulted in huge improvements to the Q-Learning and Dyna-Q agents but not to the AHC agent.

The Q-Learning agent was now able to solve the Cart-Pole problem by averaging 90.000 successful steps per run (after convergence of the action-values function after 100 runs). The Dyna-Q agent also averaged 90.000 successful steps and also managed to converge faster than the Q-Learning agent from around run 60 to run 90.

By definition (of achieving 100.000 successful steps) the AHC agent did not manage to solve the Cart-Pole problem with this new state set. However, we discovered that this situation was due to our not having altered the internal parameters of the agents for the new conditions (which was one of our initial assumptions; just change the environment and not the internal parameters of the agents).

The results obtained from the execution of these experiments have demonstrated the importance of state and action representations in reinforcement learning. We have shown that changes made to the way states and actions are represented / interpreted produce dramatic changes to the results obtained by the agents in solving the Cart-Pole problem.

7 Conclusion

This final chapter presents the conclusions reached resulting from the research made on the theory of reinforcement learning and the Cart-Pole inverted pendulum problem, and also the analysis of results obtained from the experiments performed. A discussion on possible future work is also presented.

7.1 Conclusions

Since Michie and Chambers first decided to use the Cart-Pole inverted pendulum problem for presenting their BOXES algorithm, it has served as a test platform for many researchers. We too have used the Cart-Pole as a base for evaluating our investigation of the impacts on reinforcement learning agents due to changes made in the state and action sets.

For the course of our investigations, we have started by using the environment conditions as specified by Sutton and Barto, for gathering information on the performance of the three types of reinforcement learning agents we are using, namely Adaptive Heuristic Critic (AHC), Q-Learning and Dyna-Q. This information serves as reference for subsequent experiments. With the initial conditions, AHC successfully solves the Cart-Pole problem but Q-Learning and Dyna-Q do not.

With the purpose of improving the performance of our agents, we have created a new set of perceptions based upon the convergence to a reference value of the previous perception variables thus creating a new reduced agent state set. Likewise, we have changed the semantics of the agent action set and used the original perceptions to obtain laterality information in the environment.

With these new conditions both the Q-Learning and Dyna-Q agents were now able to solve the Cart-Pole problem averaging 90.000 successful steps per run. However with these new conditions the AHC agent, who previously was able to solve the problem, was no longer able to solve the problem. These values were obtained with no changes performed to any of the internal parameters for any of the agents.

With the results presented in this dissertation, we believe that we have accomplished our initial objectives of having managed to solve the Cart-Pole problem with a reinforcement learning agent. We have implemented a new way of interpreting the perception variables allowing a simple agent (as is the case of Q-Learning) to successfully solve this problem. Furthermore, we have obtained the answers to our initial questions:

In what way does the definition of the set of states change the agent's capacity to reach its goal? For the Q-Learning and Dyna-Q agents the reduction in the state set was fundamental in the process of solving the problem. For the AHC agent, the change in state set (without adapting the internal parameters) was enough for the agent to stop solving the problem. This suggests that the AHC algorithm is more sensitive to external changes than both the Q-Learning and Dyna-Q algorithms.

In what way does the definition of the set of actions change the agent's capacity to reach its goal? The correct semantics for the actions is important when solving the Cart-Pole problem. The fact that a change in semantics was necessary after changing the state set shows the tight coupling that exists between the agent and the environment through the state and action set. On the other hand, a change in the number of actions available to the agents did not bring an improvement in performance but did increase the time needed for the value functions to converge.

Likewise we have also formulated and answered other, secondary questions:

Does the number of states influence the agent's behaviour? Yes, the number of states influences significantly the agent's behaviour. Q-Learning and Dyna-Q directly benefited from the reduction in number of states. For AHC it is not clear if the reduction of states would be beneficial or not as changes to the internal parameters would also be necessary for this type of agent.

Does the number of actions influence the agent's behaviour? No, there was no significant change in agent's behaviour (for all three agents) due to an increase in the number of actions. The added action did not help in solving the Cart-Pole problem.

Having presented our conclusions, we now present some topics for future work.

7.2 Future Work

Any work of scientific nature is never finished; there is always something more that can be done. In this dissertation we believe to have accomplished our initial objectives, but we feel that more can be done. We leave some suggestions as to possible future work.

7.2.1 Using the Reduced State Set to solve other Problems

In this dissertation we have proposed an alternative solution to the BOXES algorithm for solving the Cart-Pole inverted pendulum problem. This solution passes through the use of testing the perception values for convergence instead of dividing the continuous values into ranges. One future work of interest would be to check if this solution is applicable to other problems of similar or even different nature.

7.2.2 Virtual versus Physical

This dissertation is based upon a theoretical model and not upon a "real" physical model. We are certain that the same experiments performed on a real physical platform would bring new problems, new solutions and new results. With current technology, any smartphone running the Android operating system or an Apple iPhone should have enough power to solve the Cart-Pole problem, with the added advantage that most of these phones have built in accelerometers and tilt sensors. Thus, it should be a relatively easy job to implement a hardware prototype capable of repeating our experiments.

7.2.3 Reliability

Reliability is an issue that has not been approached in this document. Under the definition provided by Sammut (1994) in which he states that an agent that has learned to control the Cart-Pole in a particular run has not necessarily learnt to control the Cart-Pole in general, we assume that the agents used in our work may be "not reliable". By Sammut's definition, a reliable controller is one that can control the Cart-Pole starting from any state of the state set.

In the case of the BOXES model, because any given state in the state set is always associated to a range of possibilities in the physical variables (X, $\dot{X}$, Θ and $\dot{\Theta}$), it is difficult, if not impossible, to obtain the exact values of the variables from any given state (*i.e.* reverse the operation of assigning a

state from the current values of these variables). It is possible to find an infinite number of different combinations of the variables that will result in the same state. Furthermore, depending on the actual values chosen, the next resulting state when applied one of the actions will be variable and inconstant depending exactly on the values of the variables at the initial moment (when the values are closer or further from the boundaries of the ranges).

8 Bibliography

Alavala C. R. (2008), *Fuzzy Logic and Neural Networks: Basic concepts and applications*, New Age International Publisher, 2008

Barto A. G., Sutton R. S., Anderson C. W. (1983), *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems*. IEEE Transaction on Systems, Man, and Cybernetics, Vol. SMC-13, No. 5, September/October

Blynel J. (2000), *Reinforcement Learning on Real Robots*. University of Aarhus

Braitenberg V. (1984), *Vehicles: Experiments in Synthetic Psychology*, Bradford Book, MIT Press, 1984

Brooks R. (1989), *A Robot that walks: Emergent behaviour form a carefully evolved network*. Neural Computation, 1(2) Summer 1989 pp. 253-262, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, USA

Brooks R. (1990), *Elephants don't play chess*, Robotics and Autonomous Systems 6 (1990) 3-15, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, USA

van Hasselt H., Wiering M. A. (2009), *Using continuous action spaces to solve discrete problems*. In: Neural Networks, 2009. IJCNN 2009. International Joint Conference on. 2009. pp. 1149–1156.

Hyman S. E., Malenka R. C., Nestler E. J. (2006), *Neural Mechanismsof Addiction: The Role of Reward-Related Learning and Memory*. The Annual Review of Neuroscience. Doi: 10.1146/annurev.neuro.29.051605.113009, 29: pp. 565-598

Jennings N., Wooldridge M. (1998), *Applications of Intelligent Agents*, In N. Jennings, M. Wooldridge, (Eds.), Agent Technology - Foundations, Applications, and Markets, Springer-Verlag, 1998

Kaelbling L. P., Littman M. L., Moore A. W. (1996), *Reinforcement learning: A survey*. Arxiv preprint cs/9605103

Kobori N., Suzuki K,, Hartono P., Hashimoto S. (2003), *Learning to control a joint driven double inverted pendulum using nested actor/critic algorithm*. In: Neural Information Processing, 2002. ICONIP'02. Proceedings of the 9th International Conference on. 2003. pp. 2610–2614.

Mataric M. J. (2007), *The Robotics Primer*, MIT Press, Cambridge 2007

Michie D., Chambers R. A. (1968), *Boxes: An experiment in adaptive control*. In E. Dale and D. Michie (Eds.), *Machine Intelligence 2*. Edinburgh: Oliver and Boyd

Morgado L. G. (2010), *Agentes Inteligentes: Introdução*, Mestrado em Engenharia Informática e de Computadores, ISEL-DEETC, 2010-2011

Morgado L. G. (2011), *Aprendizagem por Reforço*, Mestrado em Engenharia Informática e de Computadores, ISEL-DEETC, 2010-2011

Morgado L. G. (2008), *Complementos de Inteligência Artificial: Aprendizagem por Reforço*, Mestrado em Engenharia Informática e de Computadores, ISEL-DEETC, 2008-2009

Norvig P., Russell S. (2003), *Artificial Intelligence, A Modern Approach*. 2nd ed. Prentice Hall

Pollack M. E., Ringuette M. (1990), *Introducing the Tileworld: Experimentally Evaluating Agent Architectures*. National Conference on Artificial Intelligence – AAAI, pp. 183-189

Reinforcement learning (2011) – Wikipedia, the free encyclopedia [Internet]. [cited 2011 Jan 20]; Available from: http://en.wikipedia.org/wiki/Reinforcement_learning

Ribeiro C. H. (1999), *A tutorial on reinforcement learning techniques*, Conference Paper, Supervised Learning track tutorials of the 1999 International Joint Conference on Neuronal Networks, 1999

Sammut C. A. (1994), *Recent Progress with BOXES*. In K. Furakawa, Michie, D. & S. Muggleton (Eds.), Machine Intelligence 13. Oxford: The Clarendon Press, OUP, pp. 363-384

Segway (2011) – The leader in personal, green transportation [Internet]. [cited 2011 Jan 20]; Available from: http://www.segway.com/

Singh S. (2011), Large state spaces are hard for RL [Internet]. [cited 2011 May 21]; Available from: http://umichrl.pbworks.com/w/page/7597584/Large-state-spaces-are-hard-for-RL

Stang J. (2005), *The Inverted Pendulum*. Design Project, Cornell University

Sutton R. S. (1988), *Learning to predict by the methods of temporal differences*. Machine Learning 3: pp. 9-44, erratum p.377

Sutton R. S. (1990), *Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming*. Proceedings of the Seventh International Conference on Machine Learning, pp.216-224, Morgan Kaufmann

Sutton R. S., Barto A. G. (1998), *Reinforcement Learning: An Introduction*. MIT Press/Bradford Books, Cambridge, MA

Sutton R. S. (1999), *Reinforcement Learning*, University of Alberta Webdocs; Available from http://webdocs.cs.ualberta.ca/~sutton/papers/Sutton-99-MITECS.pdf

Tesauro G. (1992), *Practical Issues in Temporal Difference Learning*, Machine Learning 8, pp. 257-277, Kluwer Academic Publisher, 1992

Watkins C. J. (1989), *Learning from delayed rewards*. PhD Thesis, University of Cambridge, England

Watkins C. J., Dayan P. (1992), *Technical note: Q-Learning*, Machine Learning 8:279-292

Wooldridge M.(2002), *Multiagent Systems*, John Wiley & Sons, 2002

Woolridge M. (2009), *An Introduction to Multiagent Systems*, John Wiley and Sons, 2009

Yang L. (2008), *Understanding and analyzing approximate dynamic programming with gradient-based framework and direct heuristic dynamic programming*, Doctoral Thesis, Arizona State University, 2008

Appendix A - Glossary

Action	A *Reinforcement Learning Agent* produces as output a signal for the *Environment* to decode and act upon. The set of all the signals produced by the *Agent* is called the Action Set and any individual member of this set is called an *action*. Actions are the way the *Agent* tells the *Environment* what to do. See also: Reinforcement Learning, Agent, Environment
Agent	An Agent is anything that can be viewed as perceiving its *Environment* through sensors and acting upon the environment through effectors. (Norvig & Russel, 2003) See also: Environment
Artificial Intelligence	The theory and development of computer systems able to perform tasks that normally require human intelligence. See also: Agent
Cart Pole	An inverted pendulum is a pendulum which has its mass above its pivot point. It is often implemented with the pivot point mounted on a cart that can move horizontally and may be called a Cart and Pole. Whereas a normal pendulum is stable when hanging downwards, an inverted pendulum is inherently unstable, and must be actively balanced in order to remain upright, either by applying a torque at the pivot point or by moving the pivot point horizontally as part of a feedback system. See also: Environment
Dyna-Q	Dyna-Q is a *Reinforcement Learning* technique that extends the Q-Learning algorithm by creating a model for performing Planning by simulating State-Action transitions. The “pseudo experiences” are used to speed up convergence of the Action-Values compared to “normal” Q-Learning. See also: Reinforcement Learning, Agent, Q-Learning
Equilibrium through	The definition of Equilibrium is the state of a body

Dynamism	or physical system at rest or in un-accelerated motion in which the resultant of all forces acting on it is zero and the sum of all torques about any axis is zero. Our intent on Equilibrium through Dynamism is the maintenance of Equilibrium, as previously mentioned, through the use of movement (dynamics) on a *Cart-Pole* system governed by an Intelligent *Reinforcement Learning Agent*. See also: Reinforcement Learning, Agent, Environment
Environment	In *Artificial Intelligence* everything outside of the *Agent* and/or its direct control can be considered as the Environment. Typically this will include the physical space (which may be a virtualized physical space) in which the Agent acts. The frontier between the *Agent* and the Environment is a highly debatable subject but in this work we will assume that any sensors that may provide information to an *Agent* (such as those used for determining the current *State*) are considered as part of the Environment. See also: Artificial Intelligence, Agent, State
Experiment	In RL-Glue terminology, an Experiment is the code that links an *Agent* to an *Environment*, establishing the initial parameters, defining the number of *Steps* or time limit for execution of interactions of the *Agent* in the *Environment* and finally establishing the termination of the Agent-Environment interactions. See also: RL-Glue, Agent, Environment
Machine Learning	Machine learning, a branch of *Artificial Intelligence*, is a scientific discipline concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data, such as from sensor data or databases. See also: Artificial Intelligence
Model	In such *Reinforcement Learning* algorithms as Dyna-Q where Planning is an important part of achieving the best *Policy*, a model contains information on State-Action pairs such as statistics

	of occurrence, Rewards received and Transitions occurring. After updating the model with "real" sampled information, "pseudo experiences" can be generated to speed up convergence of the Action-Values (and hence the Policy). See also: Reinforcement Learning, Dyna-Q, Policy, Reward
Policy	In Reinforcement Learning a Policy is sometimes called a "universal plan" and it consists of a mapping from *States* to *Actions*, or to probability distributions over actions. (Sutton, 1999) See also: Action, State
Q Learning	Q-learning is a *Reinforcement Learning* technique that works by learning an Action-Value function that gives the expected utility of taking a given *Action* in a given *State* and following a fixed policy thereafter, all without requiring a model of the environment. See also: Reinforcement Learning, Action, State, Model, Environment
Reinforcement Learning	Reinforcement Learning is an area of *Machine Learning* in computer science, concerned with how an *Agent* ought to take *Actions* in an *Environment* so as to maximize some notion of cumulative *Reward*. See also: Machine Learning, Agent, Environment, Reward
Reward	A *Reinforcement Learning Agent* learns by interacting with the *Environment* and observing the results of the interaction. The quantification of the results of the interaction is what is called the Reward and it typically corresponds to a scalar value. It is then used by the *Agent* to establish the *Policy*. See also: Reinforcement Learning, Agent, Environment, Policy
RL-Glue	A generic framework for executing *Experiments* on *Reinforcement Learning Agents* in an *Environment*. It allows the use of heterogeneous programming

	languages and physical machines (by communicating through messages) to execute such experiments. It also allows the sharing of Agents, Environments and Experiments by different investigators (due to the common base of execution). See also: Reinforcement Learning, Agent, Environment, Experiment
State	A *Reinforcement Learning Agent* may receive as input a signal from the *Environment* indicating the status or "state" of the world. The set of all the signals of this nature received by the *Agent* is called the State Set and any individual member of this set is called a *state*. See also: Reinforcement Learning, Agent, Environment
Step	A step is a single iteration in the process of gathering information from the *Environment* (sampling the variables) and producing an adequate response (*action*) by the *Agent*. See also: Agent, Environment, Action
Tileworld	The Tileworld is an abstract test-bed system designed to support experimentation with agent architectures in dynamic and unpredictable environments. The *Environment* is a two-dimensional grid on which are located different kinds of objects. See also: Environment
Θ	The angle which exists between the Pole on the Cart and the absolute vertical.
$\dot{\Theta}$	The first derivative of Θ corresponding to the angular velocity of the Pole (in relation to the absolute vertical).
X	The distance measured from the centre of the system to the centre of the Cart – the position of the Cart relating to the system.
$\dot{X}$	The first derivative of X corresponding to the horizontal velocity at which the Cart is travelling.
$\delta\dot{\Theta}$	The pole angle convergence perception derived from

	comparison of the current angle with its previous value.
$\delta\dot{\Theta}$	The pole angular velocity convergence perception derived from comparison of the current pole angular velocity with its previous value.
$\delta\dot{X}$	The distance convergence perception derived from comparison of the current distance with its previous value.
$\delta\dot{X}$	The cart velocity convergence perception derived from comparison of the current cart velocity with its previous value.

Appendix B – Output from AHC code

Here follows the original output of the Barto *et al* "C" language code as published in Sutton & Barto (1998). The second set of values is the one produced by our Java translation of the original code.

Please note that due to differences in the initial random seed as well as differences in the implementation of the "random" library code, the results will never be exactly the same in both sets of code. We have however verified that the output of the java code is always consistent with the pattern presented.

```
/*--------------------------------------------------------------------
  Result of: cc -o pole pole.c -lm   (assuming this file is pole.c)
             Pole
--------------------------------------------------------------------*/
/*
Trial 1 was 21 steps.
Trial 2 was 12 steps.
Trial 3 was 28 steps.
Trial 4 was 44 steps.
Trial 5 was 15 steps.
Trial 6 was 9 steps.
Trial 7 was 10 steps.
Trial 8 was 16 steps.
Trial 9 was 59 steps.
Trial 10 was 25 steps.
Trial 11 was 86 steps.
Trial 12 was 118 steps.
Trial 13 was 218 steps.
Trial 14 was 290 steps.
Trial 15 was 19 steps.
Trial 16 was 180 steps.
Trial 17 was 109 steps.
Trial 18 was 38 steps.
Trial 19 was 13 steps.
Trial 20 was 144 steps.
Trial 21 was 41 steps.
Trial 22 was 323 steps.
Trial 23 was 172 steps.
Trial 24 was 33 steps.
Trial 25 was 1166 steps.
Trial 26 was 905 steps.
Trial 27 was 874 steps.
Trial 28 was 758 steps.
Trial 29 was 758 steps.
Trial 30 was 756 steps.
Trial 31 was 165 steps.
Trial 32 was 176 steps.
Trial 33 was 216 steps.
Trial 34 was 176 steps.
Trial 35 was 185 steps.
Trial 36 was 368 steps.
Trial 37 was 274 steps.
Trial 38 was 323 steps.
Trial 39 was 244 steps.
Trial 40 was 352 steps.
Trial 41 was 366 steps.
Trial 42 was 622 steps.
Trial 43 was 236 steps.
Trial 44 was 241 steps.
Trial 45 was 245 steps.
Trial 46 was 250 steps.
Trial 47 was 346 steps.
Trial 48 was 384 steps.
Trial 49 was 961 steps.
Trial 50 was 526 steps.
```

```
Trial 51 was 500 steps.
Trial 52 was 321 steps.
Trial 53 was 455 steps.
Trial 54 was 646 steps.
Trial 55 was 1579 steps.
Trial 56 was 1131 steps.
Trial 57 was 1055 steps.
Trial 58 was 967 steps.
Trial 59 was 1061 steps.
Trial 60 was 1009 steps.
Trial 61 was 1050 steps.
Trial 62 was 4815 steps.
Trial 63 was 863 steps.
Trial 64 was 9748 steps.
Trial 65 was 14073 steps.
Trial 66 was 9697 steps.
Trial 67 was 16815 steps.
Trial 68 was 21896 steps.
Trial 69 was 11566 steps.
Trial 70 was 22968 steps.
Trial 71 was 17811 steps.
Trial 72 was 11580 steps.
Trial 73 was 16805 steps.
Trial 74 was 16825 steps.
Trial 75 was 16872 steps.
Trial 76 was 16827 steps.
Trial 77 was 9777 steps.
Trial 78 was 19185 steps.
Trial 79 was 98799 steps.
Pole balanced successfully for at least 100001 steps */


/**
  Result of executing Java translation of the Barto et al code
**/
Trial 1 was 33 steps.
Trial 2 was 42 steps.
Trial 3 was 10 steps.
Trial 4 was 38 steps.
Trial 5 was 9 steps.
Trial 6 was 25 steps.
Trial 7 was 14 steps.
Trial 8 was 13 steps.
Trial 9 was 25 steps.
Trial 10 was 52 steps.
Trial 11 was 45 steps.
Trial 12 was 166 steps.
Trial 13 was 189 steps.
Trial 14 was 200 steps.
Trial 15 was 83 steps.
Trial 16 was 54 steps.
Trial 17 was 246 steps.
Trial 18 was 277 steps.
Trial 19 was 252 steps.
Trial 20 was 335 steps.
Trial 21 was 237 steps.
Trial 22 was 453 steps.
Trial 23 was 319 steps.
Trial 24 was 320 steps.
Trial 25 was 328 steps.
Trial 26 was 308 steps.
Trial 27 was 362 steps.
Trial 28 was 344 steps.
Trial 29 was 303 steps.
Trial 30 was 561 steps.
Trial 31 was 480 steps.
Trial 32 was 472 steps.
Trial 33 was 511 steps.
Trial 34 was 496 steps.
Trial 35 was 571 steps.
Trial 36 was 635 steps.
Trial 37 was 604 steps.
Trial 38 was 548 steps.
Trial 39 was 519 steps.
Trial 40 was 548 steps.
Trial 41 was 472 steps.
Trial 42 was 499 steps.
```

```
Trial 43 was 473 steps.
Trial 44 was 470 steps.
Trial 45 was 465 steps.
Trial 46 was 471 steps.
Trial 47 was 671 steps.
Trial 48 was 726 steps.
Trial 49 was 705 steps.
Trial 50 was 718 steps.
Trial 51 was 727 steps.
Trial 52 was 708 steps.
Trial 53 was 756 steps.
Trial 54 was 711 steps.
Trial 55 was 1509 steps.
Trial 56 was 2897 steps.
Trial 57 was 3001 steps.
Trial 58 was 3001 steps.
Trial 59 was 3001 steps.
Trial 60 was 8184 steps.
Trial 61 was 3001 steps.
Trial 62 was 602 steps.
Trial 63 was 1004 steps.
Trial 64 was 1004 steps.
Trial 65 was 602 steps.
Trial 66 was 5719 steps.
Trial 67 was 6445 steps.
Trial 68 was 6464 steps.
Trial 69 was 6325 steps.
Trial 70 was 6339 steps.
Trial 71 was 291 steps.
Trial 72 was 6321 steps.
Trial 73 was 6337 steps.
Trial 74 was 6337 steps.
Trial 75 was 328 steps.
Trial 76 was 38250 steps.
Trial 77 was 291 steps.
Trial 78 was 47755 steps.
Trial 79 was 6892 steps.
Pole balanced successfully for at least 100001 steps.
```

Appendix C – Code Bias

A topic that we have not found discussed in any of the literature, but that we consider being very relevant to the performance of the agents is code bias. Most of the documents we reviewed present an algorithm for developing a reinforcement learning agent and also statistics of execution of that agent; practically none of them provide concrete code for implementing the agent. It could be argued that the algorithm is enough and any implementation of the algorithm should produce the same results. During the execution of our experiments, we have observed that the way the code is written can have impact on the final outcome of those experiments.

A concrete example of code bias was discovered when experimenting with a third "NO-FORCE" action (even though the problem can be observed with just the two *bang-bang* actions). The algorithm for a greedy Q-Learning agent states that when more than one action produces the same action-value (i.e. more than one action produce the same utility for the agent), then any one of them can be randomly chosen. In practical terms the following code snippet could be used for this purpose:

```
…
  int theAction = 0;
  for (int i = 1; i < nActions; i++) {
    if (qVal[state][i - 1] <= qVal[state][i]) {
      theAction = i;
    }
  }
…
```

The previous code chooses the action that has the biggest action-value from all possible values, but if they are equal the last one will always be chosen. Likewise if we change the less-than-or-equal sign (<=) to a simple less-than (<), then in case of equal values, the first one will always be chosen. In practical terms, both of these variants of code induce a lop-sided behaviour of the agent (always tends to go for the same side whenever the actions are equally possible). A more conforming implementation would be to store all the actions that produce equal values in an auxiliary structure and then choose randomly from there. The exact influence of this code bias was noted in finding the convergence of the action-values. A random choice of the action produced a slightly quicker convergence than a lop-sided biased choice. Even though in this case the issue was not critically important, other forms of code bias could easily creep in to the experiments and influence the results in a more impacting way.

The author:

(Paulo Fernando Pinho Faustino)

The supervisor:

(Luís Filipe Graça Morgado)