BUILDING AN ARTIFICIAL CEREBELLUM
USING A SYSTEM OF DISTRIBUTED Q-LEARNING AGENTS

by

Miguel Ángel Soto Santibáñez

_____

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2010

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Miguel Ángel Soto Santibánez entitled Building an Artificial Cerebellum Using a System of Distributed Q-Learning Agents and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

_____ Date: December 22, 2010
Dr. François E. Cellier

_____ Date: December 22, 2010
Dr. Salim Hariri

_____ Date: December 22, 2010
Dr. Bernard P. Zeigler

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: December 22, 2010
Dissertation Co-Director: Dr. François E. Cellier

_____ Date: December 22, 2010
Dissertation Co-Director: Dr. Salim Hariri

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Miguel Ángel Soto Santibánez

ACKNOWLEDGMENTS

I would like to take this opportunity to thank Dr. Cellier and Dr Hariri for all their support and patience throughout all these years, in spite of their very busy schedules. In addition, I would like to thank them for giving me complete freedom to select my own line of research. Limiting this freedom would have surely made their lives easier.

I would also like to thank Dr. Zeigler for his valuable feedback and for taking time away from his busy schedule to be part of my dissertation defense.

Finally, I would like to thank Tami Whelan. She always welcomed my questions; even when I interrupted her lunch or it was obvious she was very busy.

To my grandmother …

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

ABSTRACT

About 400 million years ago sharks developed a separate co-processor in their brains that not only made them faster but also more precisely coordinated. This co-processor, which is nowadays called cerebellum, allowed sharks to outperform their peers and survive as one of the fittest. For the last 40 years or so, researchers have been attempting to provide robots and other machines with this type of capability. This thesis discusses currently used methods to create artificial cerebellums and points out two main shortcomings: 1) framework usability issues and 2) building blocks incompatibility issues. This research argues that the framework usability issues hinder the production of good quality artificial cerebellums for a large number of applications. Furthermore, this study argues that the building blocks incompatibility issues make artificial cerebellums less efficient that they could be, given our current technology. To tackle the framework usability issues, this thesis research proposes the use of a new framework, which formalizes the task of creating artificial cerebellums and offers a list of simple steps to accomplish this task. Furthermore, to tackle the building blocks incompatibility issues, this research proposes thinking of artificial cerebellums as a set of cooperating q-learning agents, which utilize a new technique called Moving Prototypes to make better use of the available memory and computational resources. Furthermore, this work describes a set of general guidelines that can be applied to accelerate the training of this type of system. Simulation is used to show examples of the performance improvements resulting from the use of these guidelines. To illustrate the theory developed in this dissertation, this paper implements a cerebellum for a real life application, namely, a cerebellum capable of controlling a type of mining equipment called front-end loader. Finally, this thesis proposes the creation of a development tool based on

this formalization. This research argues that such a development tool would allow engineers, scientists and technicians to quickly build customized cerebellums for a wide range of applications without the need of becoming experts on the area of Artificial Intelligence, Neuroscience or Machine Learning.

CHAPTER 1: INTRODUCTION

1.1 Overview

One of the goals of this chapter is to provide an introduction to the topic of biological cerebellums, also called natural cerebellums. The current development of artificial cerebellums is tightly related to biological cerebellums. Therefore, I believe it is important to be aware of some of the details associated with natural cerebellums in order to better understand the current literature on artificial cerebellums.

In addition to helping us prepare for analyzing the current literature on artificial cerebellums, this chapter also provides a preview of what will be developed more in depth on the rest of this dissertation. I start with a quick overview of the main shortcomings found in the development of current artificial cerebellums. Based on this analysis I come up with a problem statement for my dissertation. After this, I mention how I plan to tackle this problem and discuss what each one of the following chapters in this document will do to accomplish this plan. I finish this discussion by mentioning a list of contributions I hope to provide with this research.

1.2 History of the Cerebellum

The cerebellum has been recognized as a distinct division of the brain since Herophilus, more than two thousand years ago. However, it wasn't until a few hundred years ago that people started to comprehend some of the functions of this important part of the brain. In 1664, Thomas Willis suggested that the cerebellum presided over the involuntary movements of the body. Observations made in the 18th century on patients with cerebellar damage and experiments made

on animals in the 19th century corroborated Thomas Willis' theory [Lar67, Fin02]. The picture below shows the cerebellum within the brain:

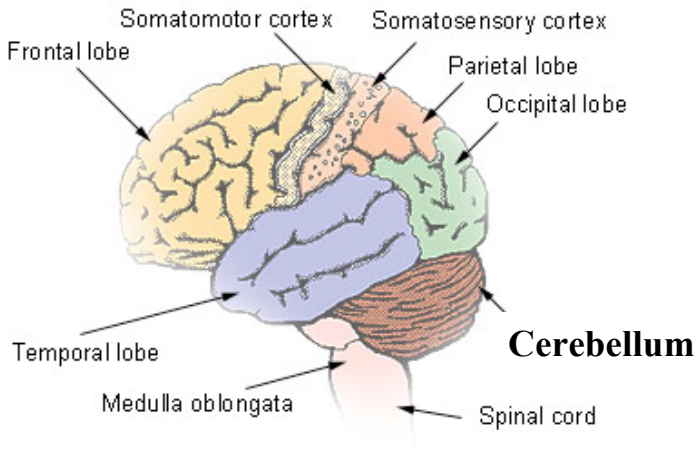

Figure 1.1. The Cerebellum within the Brain

By the 1900's scientist would already speak of particular functions of the cerebellum. For instance, in 1924, C.J. Herrick would write that the cerebellum controlled posture, regulated and coordinated all movements of precision of the skeletal musculature, and maintained muscular tone. Furthermore he compared the cerebellum's stabilizing influence with the action of a gyroscope on a large steamship, which is responsible for keeping the vessel on its course in spite of winds and waves. He thought of the cerebellum as the "Proprioceptive Adjustor" (i.e. an adjustor associated with the perception of movement and other sensations within one's own body) [Her24].

The study of the cerebellum has been particularly prolific in the past few decades. A vast amount of literature on the cerebellum has emerged and continues to appear, at what seems to be an ever-increasing rate [Bar02].

1.3 Current Understanding of the Cerebellum

Nowadays, the cerebellum is believed to play an important role in the integration of sensory perception, coordination and motor control. To accomplish this, the cerebellum processes signals coming from the spinocerebellar tract (i.e. set of axonal fibers originating in the spinal cord and terminating in the cerebellum) and sends signals to the cerebral motor cortex. The picture below shows the four primary tracts of the cerebral motor cortex:

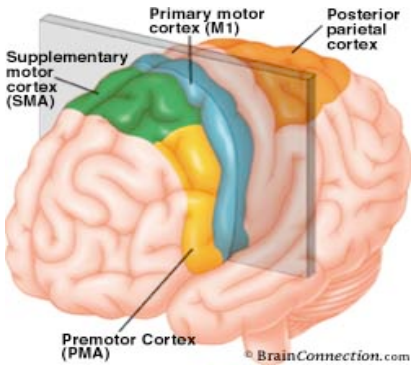

Figure 1.2. The Four Primary Tracts of the Cerebral Motor Cortex

The spinocerebellar tract provides propioceptive feedback on the position of the body in space, while the cerebral motor cortex is able to send information to the muscles causing them to move [Fin02].

In other words, the spinocerebellar tract allows the cerebellum to have an idea of the current state of the environment and the cerebral motor cortex allows the cerebellum to react to this state of the environment and in some situations try to change it.

For instance, at some point in time, the cerebellum in a human may perceive that the body is standing but losing its equilibrium (by receiving signals from the spinocerebellar tract). At this time the cerebellum would be able to make the needed adjustments to the tone of certain muscles (by sending signals to the cerebral motor cortex) so that the body regains its equilibrium.

Observations made in the 18th century on patients with cerebellar damage and experiments made on animals in the 19th century confirmed that the cerebellum is a motor control structure [Fin02]. However, modern research has also shown that the cerebellum has a role in a number of cognitive functions such as the processing of language and music [Rap01].

1.4 Theories about the Cerebellar Function on Motor Coordination

There are several theories on how the cerebellum accomplishes its functions. However, if I focus on the motor coordination function, I will find that there are currently two main theories that explain this phenomenon.

One theory states that the cerebellum is the part of the body that regulates the timing of movements. This theory emerged from studies of patients with disrupted timed movements. [Ivr88].

A second theory, called Tensor Network Theory, uses Artificial Neural Networks to provide a mathematical model of transformation of sensory space-time coordinates into motor coordinates [Pel80, Pel85].

Both theories have evidence to support them. For instance, studies of motor learning in eye blink conditioning show that the cerebellum encodes the timing and amplitude of learned movements [Boy04]. Furthermore, experiments have also shown that the cerebellum is able to do sensory-motor transformations which support the Tensor Network Theory [Pel82, Gie86].

1.5 Artificial Cerebellums and their Current Limitation

The theories mentioned in the previous section have provided great insight to the area of Artificial Intelligence. This has led to the creation of artificial cerebellums, such as Cerebellatron

[Hin06], Cerebellar Model Associative Memory (also called CMAC networks) and SpikeFORCE for robotic movement control [Spi06].

Regrettably, current approaches to the development of artificial cerebellums seem to fail at breaking away from the mold created by cerebellums found in nature. It is not my intention to undermine the millions of years of evolution that has taken these cerebellums to get to their current level of sophistication. However, if we are to accelerate the creation of useful and practical artificial cerebellums, we need to recognize that we are not using the same components found in nature. For instance, a human cerebellum has hundreds of millions of neurons working in parallel, each one interacting with hundreds and even thousands of other neurons. In the case of artificial cerebellums, the best basic components we have available are computers with a handful of processors. Each one of these processors is able to run at several megahertz but processes information sequentially (as opposed to in parallel, as groups of neurons do in biological cerebellums).

I believe current attempts to creating artificial cerebellums are similar to men's first attempts at flying. The first designs of artificial birds included flapping wings just as in real birds. However, we eventually realized that other designs worked better for artificial birds made of wood, cloth and metal (as opposed to muscles, bones, tendons and feathers).

Current artificial cerebellums, such as the ones developed by SpikeFORCE make use of Artificial Neural Networks (ANNs) in an attempt to do things in a way very similar to the way natural cerebellums do. Unfortunately, currently available computers are not well suited to efficiently implement huge sets of neurons, which are highly parallelized entities. I believe there are other methodologies out there that can be used to implement artificial cerebellums more efficiently in highly sequential devices such as computers.

There exists promising technology, such as Graphics Processing Units (GPUs), which may eventually allow us to reach the level of parallelism observed in the human cerebellum. [Gpu10, Sut05, Thi09]. Nevertheless, until this day comes we will be better off making use of algorithms that run more efficiently under limited parallelism.

Another observed issue is that current techniques used to build artificial cerebellums tend to require highly specialized skills, which most engineers do not have, such as the ability to analyze biological systems and extract from them their underlying mechanisms. At the end of the day, these techniques restrict so much the number of people that can actually use them that I expect to either see a slow production of new artificial cerebellums or a high production of low quality artificial cerebellums.

1.6 Problem Statement

This dissertation should try to solve several related issues. For starters, it should describe a more efficient method to implement artificial cerebellums in currently available computers (i.e. computers with a limited number of processors) as compared to other methods in the literature.

Additionally, this dissertation should address the fact that the methods currently used to implement artificial cerebellums require a relatively high level of expertise on the part of the user. This study should try to come up with an easy to follow development framework that formalizes and facilitates the task of creating customized artificial cerebellums. This formalization should allow engineers, scientists and technicians to build customized cerebellums for a wide range of real life applications without the need of becoming an expert in the area of Artificial Intelligence, Machine Learning or Neuroscience.

Furthermore, this framework should provide simple rules that allow the automatic

simplification of the designed “artificial cerebellums” by identifying and modifying redundant sections of the system.

In addition, I should come up with a real life application that illustrates how all the theory and methods I have developed so far are used.

Finally, this study should consider the possibility of using a distributed system to improve the learning efficiency of each agent even further. Several methods should be proposed and analyzed. This dissertation should not only explain why this improved efficiency would happen theoretically, but it should also exhibit the improved efficiency by running simulations.

1.7 Plan of Dissertation

In chapter two I will talk about other methods that have been used to implement artificial cerebellums. After some analysis, I will identify some of the perceived shortcomings of these methods and will come up with a list of things I would like to improve on. Finally, I will mention in large terms how I propose to tackle these issues. The details associated with the proposed solution will be discussed more in depth in chapters three and four.

In chapter three I will focus on showing why I believe we would be better off using the method Moving Prototypes to implement artificial cerebellums for real life problems using modern computers. This analysis will require describing some of the details associated with Moving Prototypes and also other methods in the literature such as CMAC (Cerebellar Model Articulation Controller), LWPR (Locally Weighted Projection Regression). I will also mention some of the details associated with a new controller called FOX, which uses a modified form of Albus’s CMAC neural network.

In chapter four I will describe the new framework mentioned in chapter two, which

formalizes and facilitates the task of creating customized cerebellums. Furthermore, I will argue that this framework could be used to build a development tool that could allow people, with little or no expertise on the area of A.I., Machine Learning or Neuroscience, to quickly design and implement customized cerebellums for a wide range of applications.

In chapter five, I will illustrate how the theory described in previous chapters can be used to implement an artificial cerebellum, which is able to solve a real life problem. In this case, the problem at hand will be coming up with an artificial cerebellum that will allow a front-end loader (a type of mining equipment) to automatically load a hauling truck (another type of mining equipment) in a way as to optimize a set of goals. Among the possible goals would be minimizing the amount of fuel used, avoiding dangerous collisions and minimizing the wearing of its expensive tires by avoiding unnecessary skidding.

In chapter six I will propose the use of distributed systems to implement very complex agents. I will do an overall analysis of the proposed method and try to come up with a set of general guidelines that can make this method run even faster and more efficiently. Furthermore, I will use simulation results using DEVS Java to show examples of the performance improvements that can be achieved by using these guidelines.

In chapter seven I will mention the highlights discussed in this document and will attempt to provide a guide for future researchers by mentioning what areas still need further improvement and what related problems still need to be solved.

1.8 Summary of Contributions

This dissertation describes a new methodology that allows building artificial cerebellums. This study compares and contrasts this new methodology with other proposed methods. This

paper explains why currently available computers (with a limited number of processors) are ill suited to implement currently proposed methods to implement artificial cerebellums and why the proposed method runs more efficiently and scales up better under this environment. Additionally, I argue that a development tool could be implemented based on this formalization. This tool would allow engineers, scientists and technicians to quickly build customized cerebellums for a wide range of applications without the need of becoming an expert in the area of Artificial Intelligence, Neuroscience, or Machine Learning. As part of this formalization, I provide a set of rules that allow the automatic simplification of the designed artificial cerebellums. In addition, this research implements a cerebellum for a real life application that illustrates the advantages and usefulness of the theory developed in this dissertation. Finally, this work describes a set of general guidelines that can be applied to the proposed method to make further performance improvements. I show that after applying these guidelines, the resulting artificial cerebellum is able to learn more efficiently and, therefore, can be trained in shorter periods of time. Simulation is used to show examples of the performance improvements resulting from the use of these guidelines.

1.9 Review

This chapter began by providing a short history of what I have learned so far about natural cerebellums. Furthermore, this chapter mentioned some of the current theories on how biological cerebellums work. I believe this information will help us understand the current literature on artificial cerebellums, which is tightly related to biological cerebellums. After this introduction I briefly mentioned some limitations on the previously proposed methods used to create artificial cerebellums. Furthermore, I stated the general problem I want to tackle and

mentioned what is my plan for this dissertation. Finally I summarized the contributions I hope to provide with this research.

CHAPTER 2: RELATED WORK

2.1 Overview

This chapter starts by providing a short introduction to the topic of biological cerebellums and mentions how they have inspired engineers and scientists to try to emulate the capabilities observed by this part of the brain. After this, I mention some of the most important work on the development of artificial cerebellums followed by a review, which focuses on gathering advantages and disadvantages associated with these methods. With the help of this analysis I try to determine in what cases previously proposed methods seem to be a good choice and where there is room for improvement. In cases where there is room for improvement I propose general techniques, which may help fill the gap. I postpone mentioning the details associated with these techniques until chapters three and four.

2.2 Introduction

At the earliest evolutionary stages, sensorimotor neural networks consisted of direct connections of receptors to motor executors. However, around 400 million years ago, at the time of the emergence of sharks, a separate co-processor appeared between sensory and motor systems. This co-processor, which we now call the cerebellum, performed a transformation that made the sensorimotor system of the shark distinctly better coordinated than that of other aquatic animals. This not only made sharks faster but also made their movements more precisely synchronized allowing them to outperform their peers and allowing them to survive as one of the fittest [Hil66]. For the last 40 years of so, researchers have been attempting to provide robots and

other machines with this capability.

The following subsection describes previous research associated with the creation of artificial cerebellums, such as Cerebellatron [Hin06], CMAC networks [Alb75], FOX [Smi98], SpikeFORCE [Spi06] and SENSOPAC [Sen09]. This review will allow us to identify some of the significant features and perceived shortcomings of this previous work.

2.3 History on Artificial Cerebellum

A project called **Cerebellatron** seems to be one of the first attempts at constructing artificial cerebellums. David Marr, James Albus and Andres Pellionez worked on this project between the years 1969 and 1982. Cerebellatron was used to control motor action of a robotic arm. One of its advantages was its capacity to blend commands with different weights for smoothness. On the other hand, one of its biggest drawbacks was that it required a very complicated control input.

In 1975, James Albus, one of the creators of Cerebellatron, introduced a new method called **CMAC** (Cerebellar Model Articulation Controller) [Alb75, Albu75, Alm03, Lee92, Lee99, Lee02]. This method has become a widely known neural network and has been used for robotics, control, digital communication and many other applications requiring a method for function approximation with fast convergence. A CMAC is essentially a lookup table algorithm that maps an input space into an output space. In this method, the input space is quantized using a set of overlapping tiles organized in rows of tiles called tilings. The picture below depicts the case in which the input in the CMAC is one-dimensional and we have three tilings:

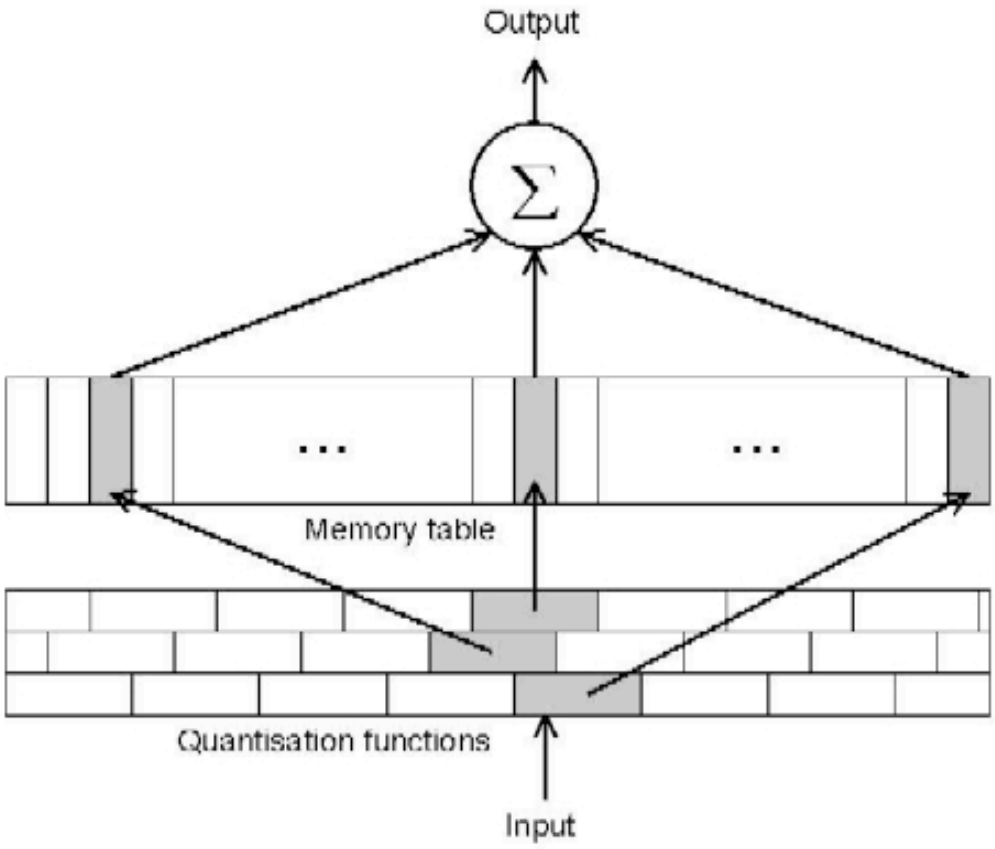

Figure 2.1. A One-dimensional Input CMAC with Three Tilings.

In CMAC, a change of value of the input vector results in a change in the set of activated tiles. The picture above shows the activated tiles, in gray, associated with a given input. Notice that one tile comes from each one of the three tilings. The CMAC output is stored in a distributed fashion, such that the output corresponding to any point in input is derived from the value associated with the activated tiles. Notice that the memory size required by a CMAC depends on the number of tilings and the size of the tiles. If the tilings are few and the tiles are large less memory will be required. However the CMAC may not be able to properly represent the desired function. On the other hand, if we use many tilings and make the tiles smaller we have a better chance of being able to represent the desired function. Unfortunately, the CMAC memory

requirements will increase. So far we have only seen the case in which the input for the CMAC is one-dimensional. If we have more dimensions, the tiles form hypercubes.

In 1998, Russell Smith described a new approach for adaptive optimal control. The new controller named **FOX** used a modified form of the Albus's CMAC neural network. This system was trained to generate control signals that minimized the performance error of a system. Fox was tested using four control problems: 1) controlling a simulated linear system, 2) controlling a model gantry crane, 3) balancing an inverted pendulum on a cart and 4) making a wheeled robot follow a path. Fox was successful on each one of these environments as long as the training was comprehensive enough. Fox was different to Albus's CMAC in that it used eligibility vectors, not just scalars. In both CMAC and FOX eligibilities control how the weights associated with the tiles are updated. Unfortunately, FOX has the same problem as CMAC when it comes to its memory requirements. I will discuss this issue in the following section.

Another study associated with building artificial cerebellums was named **SpikeFORCE** (Real-time spiking networks for robot control). The SpikeFORCE consortium was a multidisciplinary group of scientists, which started in 2002 and ended in 2005. The aim of this group was to adapt biological strategies distilled from studying the cerebellum to develop spiking neural networks that would be useful in learning and controlling real-time tasks, such as robot motor control. One of the main goals of SpikeFORCE was to provide machines with the finesse of movement seen in animals [pre07]. This consortium constructed both hardware and simulated networks in order to evaluate the control abilities of real-time spiking networks based on these strategies [Bru04, Iso04, Dan04, Bez04, Phi04, Gal05, Dan05]. One of the main advantages of this type of approach is that it may someday lead to a better understanding of how biological cerebellums work. Unfortunately, elucidating biological strategies from natural

cerebellums is a very difficult task and requires skills that very few people have. This seems to indicate that this research will be slow to come up with artificial cerebellums that can be used to solve a large range of real life problems.

SENSOPAC, which stands for "SENSOrimotor structuring of Perception and Action for emergent Cognition" [Sen09], was the continuation of the project SpikeFORCE [Ros06]. The SENSOPAC project is still under way and it is still analyzing and expanding on the work of several other researchers on the subject of ANN simulation and robotics [Car06, Car07, Ing08, Pan08 and Sma08].

Some researchers associated with SENSOPAC, such as Dr. Daniel Wolpert from the University of Cambridge in the United Kingdom are attempting to understand how humans and animals integrate sensory information to form beliefs about the world, how this information is modulated by the active exploration process and how these processes form a framework for active perception. To accomplish this task this research uses techniques such as recording information associated with whisking cells in living animals such as rats. This research hopes to elucidate neural coding for active sensing in rodents. This type of research is similar to the one done by SpikeFORCE in the sense that they both try to elucidate processes based on observation of natural systems. This task is also very difficult and requires a set of skills not often found in scientists, let alone engineers or technicians. I suspect this approach will eventually lead to a better understanding of natural neural systems but it will take a while before it produces solutions to real life problems.

In the meantime, other researchers such as Dr Sethu Vijayakumar from the University of Edinburgh, United Kingdom, are working on addressing the question as to how we can automatically detect and represent structure in high dimensional sensorimotor maps under

dynamically changing contexts. They argue that the inverse dynamics mapping (from the current state and the desired change to the necessary motor commands) has a high input dimensionality, so many standard regression techniques designed for rather low dimensionalities fail [Sen09]. Their solution to this problem is to use a new technique called Locally Weighted Projection Regression (LWPR) algorithm. This method approximates the global regression function by a mixture of multiple linear models, each of which is responsible in a certain region as determined by a center and an automatically adapted distance metric. The local regression models are learned using partial least squares. Leave-one-out cross-validation is utilized for ongoing assessment and complexity control of the model. Local models can be added and purged as required during training [Sen09]. LWPR achieves nonlinear function approximation in high dimensional spaces with redundant and irrelevant input dimensions [Vij00]. In a nutshell, LWPR is able to handle high dimensional spaces by automatically identifying redundant and irrelevant input dimensions and performing regression on a much smaller number of relevant dimensions. This method is a step forward toward the goal of handling the obvious issue of dimensionality in Artificial Neural Networks. Unfortunately, this method is only a good choice in cases when the function we are trying to represent contains only a few non-redundant and non-irrelevant input dimensions. I believe that cerebellums used in real life applications will have to handle cases in which the function will contain a large number of non-redundant and non-irrelevant input dimensions.

SENSOPAC also has researchers, such as Dr. Henrik Jörntell from the University of Lund, Sweden, who are attempting to understand the biological mechanisms in the cerebellum [Jor08, Ben08]. They hope this understanding will allow them to build artificial bio-inspired haptic cognitive systems, which are able to scale to the complexity of the real world. They hope

this research will allow them to elucidate network and temporal coding in microcomplexes and be able to simulate large networks of microcomplexes. The picture below shows an example of a cerebellar microcircuitry within an area of the cerebellum:

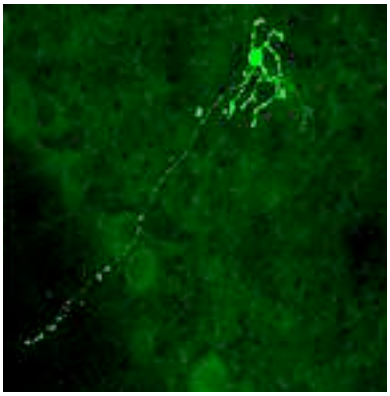

Figure 2.2. Picture of a Cerebellar Microcircuit.

Once a cerebellar microcircuit is discovered, a model of this circuit is designed. The picture below shows an example of a model of a cerebellar microcircuit:

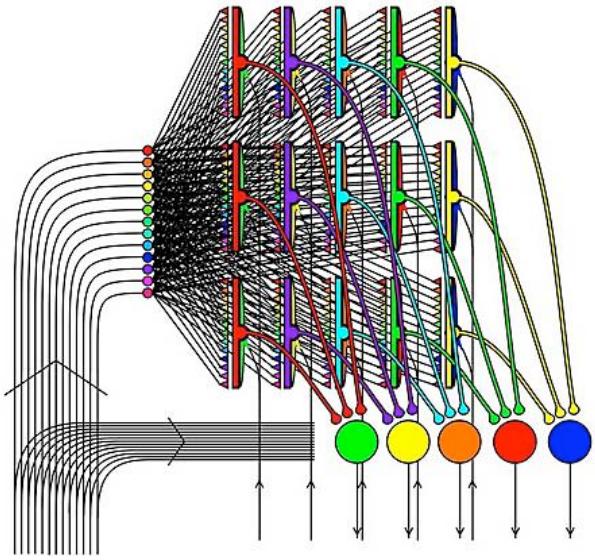

Figure 2.3. Model of a Cerebellar Microcircuit.

This type of research is done by performing electrophysiological recordings on biological cerebellums from animals such as cats. This SENSOPAC group focuses on microcircuitry analysis and temporary coding to elucidate network and temporal coding in microcomplexes. As you may remember from chapter 1, network and temporal coding are the two main theories that try to explain how natural cerebellums accomplish their motor coordination functions.

Unfortunately, performing electrophysiological recordings on live animals, extracting cerebellar microcircuits and elucidating network and temporal coding out of them are not common human skills.

The following section analyses all the methods I have talked about and tries to summarize their advantages and disadvantages.

2.4 Previous Work Analysis

After reading the literature on artificial cerebellums, the reader can easily start to see some recurring themes.

One of the recurring themes seems to be: *We should try to achieve the dexterity attained by biological cerebellums.* For instance, methods such as Cerebellatron and SpikeFORCE have tried to smooth out robot movements in an attempt to provide them with the finesse of movement seen in humans and other animals. I believe this is a good goal to have since it is obvious that, currently, natural systems appear to have the ability to move much more gracefully than our current robots.

However, this seems to lead researchers to this other theme: *Biological systems are the ultimate model of how a cerebellum should work, and therefore we should focus on understanding and imitating their natural mechanisms*. An example of this can be found in the

literature associated with SpikeFORCE and SENSOPAC. Both consortiums focus on understanding the neural principles that give an organism the ability to learn in real-time and to recreate this ability in real-time spiking neural networks. To accomplish this, researchers such as Dr. Henrik Jörntell perform electrophysiological recording on biological cerebellums from animals such as cats. Dr. Daniel Wolpert does something similar but this time on whisking cells in animals such as rats. In the case of Dr. Henrick Jörntell's, his recordings are used to try to elucidate network and temporal coding in microcomplexes and then try to simulate them.

I believe this approach has some advantages and disadvantages. The main advantage being that this methodology may eventually allow us to better understand how biological cerebellums work. This may lead to cures to ailments associated with the nervous system and it may lead to the discovery of some very interesting biological mechanisms. However, this approach has also its drawbacks. For starters, focusing on understanding and imitating natural mechanisms requires skills such as performing electrophysiological recording on animals. This skill is most often not found in the typical engineer or technician who may work on the development of new artificial cerebellums. Requiring this level of specialization will strongly limit the number of people able to work on the development of new artificial cerebellums, which will either limit the quality or quantity of artificial cerebellums that can be produced for different applications. Furthermore, this approach forgets that the basic components available for the constructions of artificial cerebellums are very different to the basic components found in biological cerebellums. On one hand, artificial cerebellums make use of computers with a very limited number of very fast sequential processors as their basic components while biological cerebellums are composed of huge networks of neurons, which are highly parallelized systems. To illustrate this issue we can take SpikeFORCE as an example. One of the main aims of this

consortium was to adapt biological strategies distilled from studying the cerebellum to develop spiking neural networks that would be useful in learning and controlling real-time tasks, such as robot motor control. Unfortunately, the artificial neural networks generated by this consortium could not be implemented efficiently in modern computers as these networks had to be emulated. A. K. Dewdney, mentioned this issue in one of his Scientific American columns in 1997 by writing the following: "Although neural nets do solve a few toy problems, their powers of computation are so limited that I'm surprised anyone takes them seriously as a general problem-solving tool" [Dew97]. These words may sound too harsh to many people. However, they reflect the frustration of trying to implement a natural mechanism using building blocks, which are very different to the ones found in nature, and realizing that emulating these natural mechanisms using modern computers is highly inefficient and requires excessive memory and computation resources.

Other examples of methods trying to imitate nature in order to implement artificial cerebellums can be found in the methods CMAC and FOX. These two methods try to mimic the functionality of the mammalian cerebellum by implementing sparse coarse-coded associative memory as observed in nature. These methods have been used as function modelers for robotic control for several simple problems, such as balancing an inverted pendulum on a cart. However, the fact that these techniques do not scale well has limited their use in bigger control problems.

Another common theme found in the literature is the following: *Cerebellums, both natural and artificial, often have a large number of input dimensions, however, many of these dimensions are often redundant and/or irrelevant*. This assumption has inspired methods such as LWPR (Locally Weighted Projection Regression). In a nutshell, LWPR is able to handle high dimensional spaces by automatically identifying redundant and irrelevant input dimensions and

performing regression on a much smaller number of relevant dimensions. Unfortunately, while it is true that input dimensions can be simplified, it does not mean that the resulting number of relevant dimensions will be small. On the contrary, many real life cerebellums will have to handle a large number of relevant dimensions. Perhaps there is another method, which is more efficient than regression, when used on modern computers.

In spite of this criticism, previously proposed methods do have several advantages. For a starter, the method Cerebellatron was able to improve movement smoothness in robots. Furthermore, CMAC provided a method for function approximation with fast convergence and FOX made further improvements on the way CMAC updates its tiles. In addition, the SpikeFORCE and SENSOPAC approaches to exploring natural neural networks may eventually allow us to understand how biological cerebellums work. Finally, the method LWPR, used by SENSOPAC, is a step in the right direction towards tackling the scalability issue found in artificial neural networks. Furthermore, LWPR can be a very good choice in cases where the function we want to represent has lots of input dimensions but most of them are redundant and irrelevant.

However, previously proposed methods also have several shortcomings. For instance, methods such as Cerebellatron are difficult to use and require very complicated control input. Furthermore, methods such as CMAC and FOX depend on fixed sized tiles and therefore do not scale well. In addition, methods such as SpikeFORCE and SENSOPAC require skills that very few scientists have. This limits the construction of good quality cerebellums for a large number of applications. Finally, the method LWPR which is used by SENSOPAC to handle the issue of high input dimensionality is only a good choice if the function to be represented has very few non-redundant and non-irrelevant dimensions. Unfortunately, I believe that cerebellums used in

real life applications will have to handle cases in which the function will contain a large number of non-redundant and non-irrelevant input dimensions. In these cases LWPR is not a good option.

2.5 Perceived Shortcomings of Previous Work

I believe there are two broad areas in which previous work seems to show shortcomings. The first one is on the topic of framework usability and the second one is related to the building blocks currently available to create artificial cerebellums.

2.5.1 Framework Usability

In the previous section we saw several examples showing that some proposed methods were difficult to use. For instance, Cerebellatron, one of the earliest methods to construct artificial cerebellum required a very complicated control input. Furthermore, other methods, such as the ones utilized by several members of SpikeFORCE and SENSOPAC, seem to make use of numerous and complex concepts that are so specialized that very few people have the skills and knowledge to grasp and properly use them. For instance, one of the methods used by SENSOPAC involves researching and discovering real life biological mechanisms and then implement them using modern computers. This type of research is usually beyond the grasp of the average engineer and most likely only doable by very specialized and experienced scientists.

I believe that a framework that can only be used by a few very experienced and skilled people will have to sacrifice either productivity or quality. If only a few people can grasp the complexities of the framework, very few people will be able to peer review the finished product and, most likely, many issues will be left undiscovered, decreasing the quality of the resulting

artificial cerebellum. Furthermore, if fewer people are capable on working on new artificial cerebellums, this will, most likely, also mean that fewer artificial cerebellums will be produced. This is particularly unfortunate as artificial cerebellums have potentially very many applications.

2.5.2 Building Blocks Incompatibility

Previous work, such as CMAC, SpikeFORCE and SENSOPAC are based on different types of Artificial Neural Networks (ANNs). For instance, CMAC uses one in which memory is sparsely coarse-coded associated, FOX uses one in which eligibilities are vectors instead of scalars and SENSOPAC uses one in which regression is locally weighted. It should not come as a surprise that all these methods are based on ANNs as they use biological systems as their model of how an artificial cerebellum should work. However, these methods seem to pay little attention to the fact that biological and artificial cerebellums make use of different types of building blocks. On one hand, biological cerebellums are based on neurons. On the other hand, modern artificial cerebellums make use of computers as their building blocks.

Unfortunately, neural networks are highly parallel entities and they perform poorly when implemented using highly serial modern computers. Even a specialized neural network such as SPANN (Scalable Parallel Artificial Neural Network), which runs on massively parallel computers, is not a practical option. For instance, SPANN has been used to identify character sets. Only the neurons on the edges of the domains were involved in communication, in order to reduce the communication costs and maintain scalability. The back-propagation algorithm was used to train the network. SPANN used 500 Intel Itanium processors to implement 25,000 neurons [Lon08]. Now consider the fact that human cerebellums have hundreds of millions of neurons. Even without taking into account communication issues, this may scale up to a system

that requires millions of processors. Such a system may be realizable but does not look like a very practical option, especially when considering its cost.

Furthermore, Both CMAC and FOX make use of sparse coarse-coded associative memory as observed in nature. However, this type of memory as implemented by CMAC and FOX does not scale well when implemented in modern computers. The main reason for this is that memory requirements grow exponentially as we add more input dimensions.

Finally, SENSOPAC tries to tackle the scalability issue by making use of a method called Locally Weighted Projection Regression (LWPR). As I mentioned before, LWPR is able to handle high dimensional spaces by automatically identifying redundant and irrelevant input dimensions and performing regression on a much smaller number of dimensions. Unfortunately, while it is true that input dimensions can, usually, be simplified, it does not mean that the resulting number of relevant dimensions will be small. On the contrary, many real life cerebellums will have to handle a large number of relevant dimensions. In these cases using regression may end up being highly inefficient when used in currently available computers, which have a very limited number of processors.

2.6 Proposed Solutions to Shortcomings

As mentioned in the previous section I perceive two areas in which previous work seems to show some shortcomings. In this section I briefly describe a new framework and enumerate some of the ways it may be able to tackle some of these shortcomings.

2.6.1 Addressing Framework Usability

In order to tackle the usability issue, I propose the use of a new framework, which makes

real effort on simplifying things. For starters, this framework minimizes as much as possible the number of concepts that the user has to learn and remember.

Furthermore, the proposed framework automatizes as many steps as possible and tries to make the other steps very intuitive. I hope that by offering a simple to use framework, more and more people will be able to get involved in the creation of new artificial cerebellums. This will not only allow more peer reviews and, therefore, better quality of the resulting artificial cerebellums but also a much larger production of this type of systems in a large variety of environments.

2.6.2 Addressing Building Blocks Incompatibility

In the preceding section I mentioned that previous work focuses on the use of different types of ANNs to implement artificial cerebellums. I also mentioned that ANNs seemed to be incompatible with modern serial computers, which are, nowadays, the building blocks of artificial cerebellums. However, this does not mean I believe this research is without merit. This type of research may some day allow us to understand how the human cerebellum works. Furthermore, as computers become more and more powerful, I believe the day will come when even very large ANNs (with as many neurons as in the human cerebellum) will be implemented using computers in a practical manner. However, even when this day comes, we will still be better off using methods that are more compatible with the building blocks available at the time, as they will allow us to tackle bigger and more complex problems.

There seems to be lots of similarities between what is happening with the development of artificial cerebellums and the development of the first flying machines. At first, flying machines looked very similar to birds, early models even had wings, which flapped. However, these early

attempts ended in a failure as birds and flying machines have very different building blocks. For birds, equipped with large and flexible muscles, it made perfect sense to flap wings. However, for a flying machine made of wood, metal and in some cases some type of engine, flapping wings made little sense. But that did not mean that a flying machine was inherently at a disadvantage when compared to a bird. For instance, our modern airplanes are able to fly further, faster and higher than any bird. In the same way I believe it is possible to create artificial cerebellums that are, in certain aspects, superior to human cerebellums as long as we are conscious of the building blocks we are using and try to take advantage of their best properties.

I suggest that, instead of considering biological systems as the ultimate goal we should strive for, maybe we should start thinking outside the box and develop new ways to construct artificial cerebellums, which may be more compatible with the building blocks we currently have at hand. The framework proposed in this document is one attempt at doing just that.

For instance, instead of using methods such as CMAC, FOX or LWPR, I suggest using a method called Moving Prototypes, which is an enhancement of a method called Q-learning.

In a nutshell, I believe that Moving Prototypes scales up better than CMAC or FOX. Furthermore, I believe that Moving Prototypes run more efficiently than LWPR in problems where the number of non-redundant and non-irrelevant dimensions is large.

In the next chapter I will describe the proposed method Moving Prototypes and will explain more in depth why I believe using this method is more appropriate than CMAC, FOX or LWPR when tackling real life problems and implementing artificial cerebellums using modern computers.

2.7 Review

This chapter explained what is the motivation behind the creation of artificial cerebellums and provided an overview of the most important work related to this topic. In addition, this chapter analyzed these methods and extracted some of their most important advantages and shortcomings. Furthermore, the shortcomings were grouped together into two broad issues: *framework usability* and *building blocks incompatibility*. Finally, a new framework based on the method Moving Prototypes was proposed. It was argued that this framework would tackle these two issues. The details associated with this new framework and an in depth comparison with previous work is deferred to the next two chapters.

CHAPTER 3: PREVIOUS WORK VS MOVING PROTOTYPES

3.1 Overview

As I mentioned in chapter 2, previously proposed methods used to build artificial cerebellums show shortcomings in two main areas: 1) framework usability and 2) building blocks incompatibility. Furthermore, I proposed using a new framework based on Moving Prototypes, which tackled these two issues. However I never described in detail the method Moving Prototypes and showed how it was better than previously proposed methods used to build artificial cerebellums. This chapter will do just that. I will star by describing the method Q-learning, which is the basis for the creation of Moving Prototypes. I will continue by describing some of the key features of Moving Prototypes and then I will explain why I believe Moving Prototypes is a better choice than CMAC and FOX when implementing an artificial cerebellum using modern computers. I will end the chapter by describing why I believe we may be better off using Moving Prototypes than LWPR to implement artificial cerebellums that can be used in real life applications.

3.2 Q-learning

Before I can describe Moving Prototypes, which is at the center of my proposed framework, I need to explain its parent formalism, Q-learning. However, if you are already familiar with Q-learning you may want to go ahead and skip this section.

Q-learning is an off-policy Temporal-Difference (TD) control algorithm and it is believed to be one of the most important breakthroughs in reinforcement learning [Wat89]. When I say that Q-learning is an off-policy control algorithm I mean that it is able to learn the optimal policy

while following a slightly randomized policy that ensures exploration. Furthermore, when I say that Q-learning is a Temporal-Difference control algorithm, I mean that it approximates its current estimate based on previous learned estimates. This process is also called bootstrapping. The fact that Q-learning is an off-policy method has greatly simplified the analysis of the algorithm and has allowed proving that, under certain conditions, the algorithm converges to an optimal policy in both deterministic and nondeterministic Markov Decisions Processes (MDPs) [Mit97]. An MDPs can be thought of as a problem setting in which the outcome of applying any action to any state depends only on this action and state and not on preceding actions or states. Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems.

Q-learning has one more advantage in that it can be applied online. Online learning occurs when the agent learns by moving about the real environment and observing results. In contrast, in offline learning the agent learns by making use of typically large sets of training examples. Once the training session is over, an offline learning agent no longer changes its approximation to the target function. In other words, post-training queries to the system have no effect on the system itself, and thus the same query to the system will always produce the same result. By using an offline learning method we are betting that this set of training examples is all we need to handle any situation the system may encounter in the future. The main disadvantage with offline learning is that it is generally unable to provide good local approximations to the target function. The most important characteristic of online learning is that soon after the prediction is made, the true label of the instance is discovered. This information can then be used to refine the prediction hypothesis used by the algorithm. Online learning has several advantages. For starters, online learning tends to require less memory resources than offline learning, since

offline learning often processes training examples in batches, which can be very numerous. Furthermore, an offline learning system will not be able to adapt to changes of the environment unless a new training session takes place. For instance, the system may be in charge of controlling a robot. Eventually, some of the parts in the robot may wear out and the actuators response may be slightly different. An online learning system may be able to automatically adapt to this change while an offline learning system may not.

In a nutshell, Q-learning is a method for learning to estimate the long-term expected reward for any given state-action pair [Sri00]. The set of all these long-term expected rewards is what is called the Q-function, also called value function.

The ultimate goal of a Q-learning agent is to come up with a control policy. This policy consists of a group of state-action pairs. In its most basic form, this policy has a state-action pair for each possible state. The action associated with each state represents the “best” action that can be taken from that state. By “best” action we mean the action that is expected to provide the largest reward in the long run. In a more sophisticated form, this policy has only some examples of state-action pairs and the other pairs are inferred from these examples. Once Q-learning generates a Q-function, it is very simple to come up with its associated control policy. All that needs to be done is to analyze each state in the Q-function and select the action that maximizes the expected reward.

The Q-learning algorithm is shown in procedural form below:

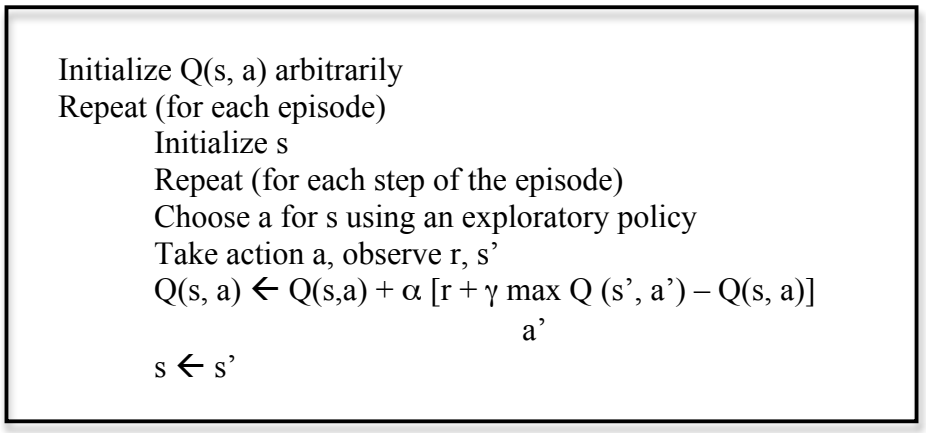

Figure 3.1. The Q-learning Algorithm in Procedural Form.

In the figure above, **s** represents the current state, **a** represents the selected action given that state **s**, **r** represents the immediate reward of applying action **a** given state **s** and **s'** represents the resulting immediate state. Furthermore, **Q(s, a)** represents the value in the value function that is associated with applying action **a** while in state **s**. The symbol γ is typically called the **discount rate** while the symbol α is called the **step size**. The symbol γ determines the present value of future rewards while the symbol α is used to control how fast **Q(s,a)** converges to the recently found **Q(s, a)**. An **episode** is simply a group of iterations with the environment. Each one of these iterations is what is called a **step** of an episode.

In its most basic form Q-learning uses a table with an entry for every state-action tuple. Unfortunately, in many problems we have an infinite number of state-action tuples and, therefore, using a table is no longer feasible. Other methods such as Tile coding and Kanerva coding have been proposed to alleviate this problem. However, these methods distribute memory resources more or less evenly throughout the state-action space. This approach runs the risk of

underutilizing memory resources in certain areas of the state-action space while the allocated memory resources in others areas may end up being insufficient to represent the desired function. This problem was the inspiration for the method Moving Prototypes, which will be explained in the next section.

3.3 Moving Prototypes

As I mentioned before, previous implementations of the Q-learning method do not make good use of the available memory resources. These methods run the risk of allocating too much memory resources in areas of the function that are very simple and not allocating enough resources in other areas where the function is more complex. The reason for that is because they allocate memory resources more or less uniformly throughout the space of the function they are trying to represent.

The reason why they do this is because methods such as Tiling and Kenerva coding allocate memory resources before Q-learning has a good idea of what the final function will look like and these memory resources are not re-arranged as this function is converging to its final shape.

To solve this shortcoming, I came up with a new algorithm called Moving Prototypes. This method uses a tree-like data structure to store the value function in a piecewise manner. The leaves of this tree represent certain zones of the function we are trying to represent. These sections of the function can be generated out of a few values, also called prototypes. As you may expect, leaves associated with areas in which the function is very simple will cover large areas of the state space. On the other hand, areas where the function is more complex will be represented by leaves, which cover small areas of the state space of the function. The picture below shows a

sample Moving Prototypes tree used to represent a Value function.

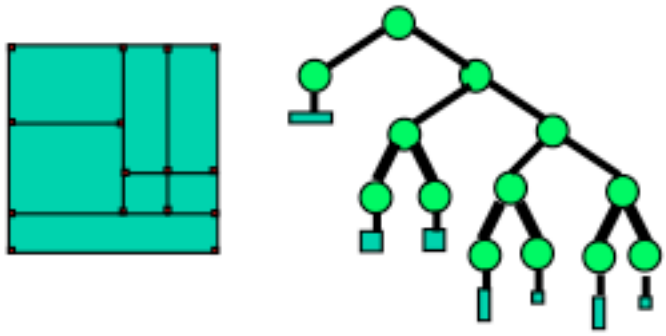

Figure 3.2. A Value Function and its Associated Moving Prototypes Tree.

The main advantage of using Moving Prototypes is that this algorithm automatically moves prototypes around the state space of the function to areas where they are needed the most. In other words, this algorithm automatically takes prototypes away from areas where the function is simpler to areas where the function is more complex. This allows this algorithm to make a better use of memory resources and therefore be able to tackle bigger problems than Tile or Kanerva coding. More details on this algorithm can be found in my Master Thesis [Sot04].

The following section compares Moving Prototypes with two of the main methods currently used to build artificial cerebellums, CMAC and FOX.

3.4 CMAC and FOX vs. Moving Prototypes

The main issue with CMAC and FOX is that their memory requirements grow exponentially as we add more and more input dimensions. To illustrate this point let us consider a problem in which we have a single input dimension and we decided that having a single tiling

and 100 tiles in this tiling makes sense. In this case the memory requirements associated with the tiles in this function is 100 memory units. Now let us consider the case in which the input has two dimensions and we keep the tiling and tile size the same. In this case we would need to allocate 10,000 memory units for the 10,000 tiles we now need. If we now go further and consider a case in which the input has 5 dimensions, we would need 10,000,000,000 memory units to implement the 10,000,000,000 tiles we now require. You can see how this method does not scale well.

The reason for this shortcoming is that both, CMAC and FOX, decide how many tilings will be used and how big each tile will be before they have any idea what the final function will look like. In other words these methods allocate memory to describe their function before their algorithms are applied. Since at this time they have no idea what the final function may look like, they need to be conservative and use relatively small tiles for each input dimension. Otherwise CMAC or FOX may be unable to describe the desired function. This inefficiency is normally not an issue when the input has only a few dimensions. However as we increase the dimensionality of the input the memory requirements for these two methods increase exponentially.

On the other hand, Moving Prototypes does not have this problem. Memory requirements for Moving Prototypes only grow with the complexity of the function it needs to represent not with its dimensionality. In this method memory resources are automatically shifted from areas where the function is simpler to areas where the function is more complex. This is especially important since the learning process often goes through a series of very different functions before convergence. Knowing the shape of the final function may help methods such as CMAC and FOX to allocate resources efficiently to represent this function. However, this allocation may be inappropriate to represent intermediate functions and therefore learning may be slowed down.

Furthermore, Moving Prototypes has the added benefit that it can be allocated a fixed amount of memory resources and can be asked to do the best it can with these resources. This last feature is especially beneficial, as the memory requirements will never grow beyond what the system can provide.

To summarize, I argue that CMAC and FOX do poorly solving problems where the input space has lots of dimensions. I believe artificial cerebellums used in real life applications will need to handle large numbers of input dimensions and, therefore, I advise against using these methods. Instead, I suggest using a method called Moving Prototypes, which handles high dimensionality much better, at least when used in currently available computers. Moving prototypes handles high dimensionality better by automatically reallocating memory resources from areas of the function that are simpler to areas of the function that are more complex. This characteristic of Moving Prototypes has the added benefit that it is also better suited to represent the intermediate functions the algorithm will encounter as it converges. Since Q-learning bootstraps a new function from its previous function, I believe that Moving Prototypes may be able to decrease the time it takes to converge to the final function. In other words I believe Moving Prototypes may be able to learn faster by doing a better job at representing its intermediate functions.

In the next section I will again compare Moving Prototypes but this time against the LWPR algorithm, used by SENSOPAC.

3.5 LWPR vs. Moving Prototypes

As I mentioned SENSOPAC uses a method called LWPR to handle high input dimensionality. Unfortunately, LWPR only works efficiently if the artificial neural network has

only a handful of non-redundant and non-irrelevant input dimensions. Once the number of non-redundant and non-irrelevant input dimensions is large, the regression algorithm used by LWPR becomes too inefficient to be used.

Before I go ahead and explain why I believe Moving Prototypes is a better choice when implementing artificial cerebellums for real life applications, let me introduce the concept of artificial neural networks and the regression algorithm. The next subsection will do just that. However, if you are already familiar with these two concepts you may want to skip this next subsection.

3.5.1 Artificial Neural Networks and the Regression Concept

Wikipedia defines an artificial neural network (ANN) as a mathematical model or computational model that tries to simulate the structure and/or functional aspects of biological neural networks. However, Wikipedia also mentions that there is no precise agreed-upon definition among researchers as to what a neural network is. Nevertheless, most researchers agree that it involves a network of simple processing elements (neurons), which can exhibit complex global behavior, determined by the connections between the processing elements and element parameters.

The central nervous system was the original inspiration for ANNs. The processing elements are analogous to biological neurons and the connections between these processing elements are analogous to axons and dendrites.

ANNs have been used as function approximators in several studies. For instance, a program called TD-GAMMON used neural networks to play backgammon very successfully [Tes95]. Furthermore, Zhang and Dietterich used ANNs to solve job-shop scheduling tasks

[Die96]. Crites and Barto used an ANN to solve an elevator-scheduling task [Bar96]. Also, Sebastian Thrun used a neural network based approach to Q-learning to learn basic control procedures for a mobile robot with sonar and camera sensors [Thr96].

In the late 1950's a computer scientist named Frank Rosenblatt proposed one of the first learning networks. He called this learning network a "perceptron". The main goal of this perceptron was to discover a set of connection weights, which correctly classify a set of binary input vectors. The picture below shows a typical perceptron in which we have some input nodes, one or more output nodes and some arcs between them. Each one of these arcs has an associated weight:

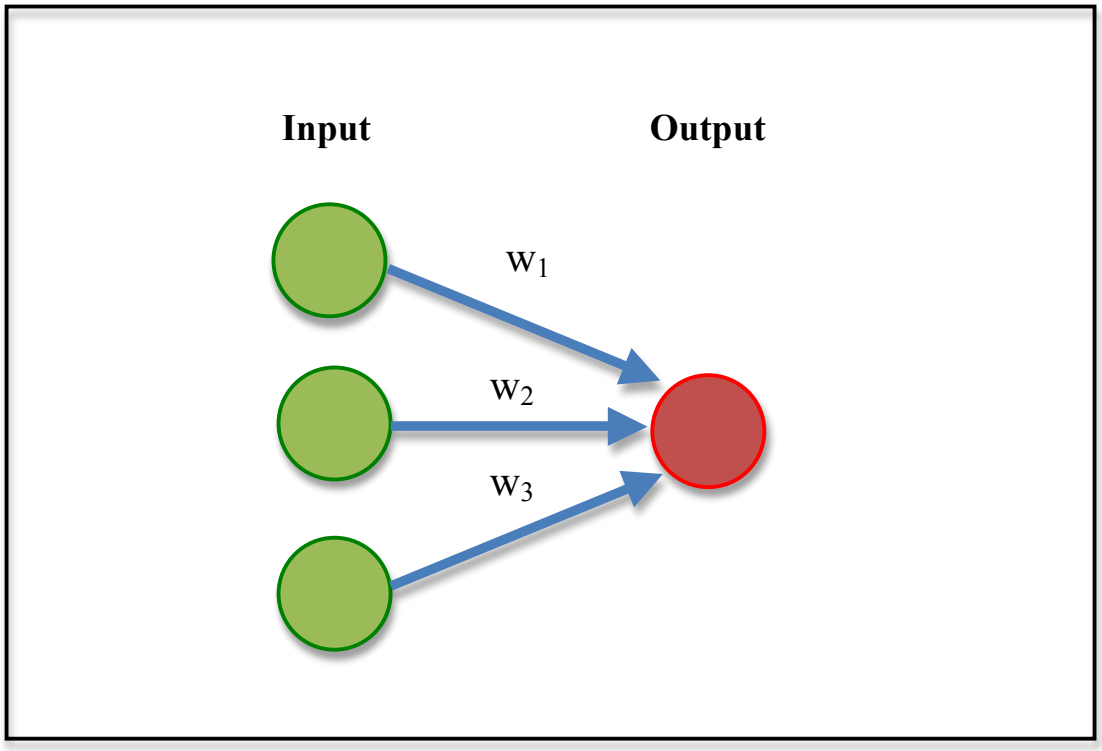

Figure 3.3. A Rosenblatt's Perceptron

The output for this system can be calculated by using the algorithm below:

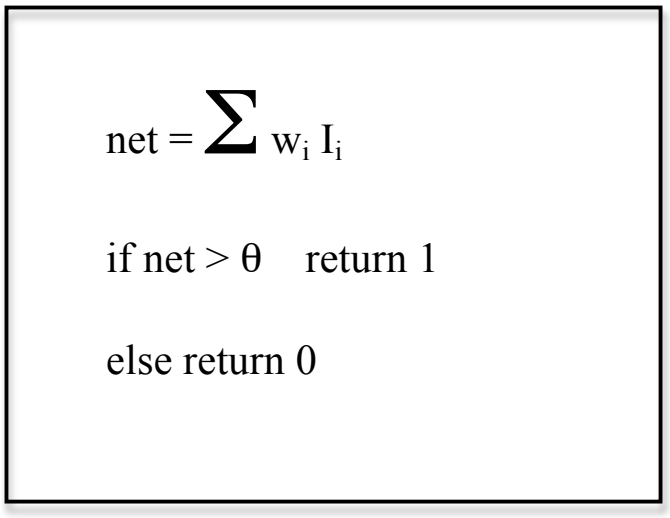

Figure 3.4. How to Calculate the Output in a Perceptron

In the algorithm above, w_i is the weight associated with the ith arc, I_i is the input associated with this ith arc and θ is the perceptron's threshold.

It is probably easier to understand how a perceptron works if we go through an example. For instance, let us assume we want to create a perceptron that implements an AND logic gate. The table below describes the desired behavior:

First Input	Second Input	Desired Output
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3.5. Desired Behavior for an AND Logic Gate

The figure below shows the perceptron we want to train so that it behave as this AND logic gate:

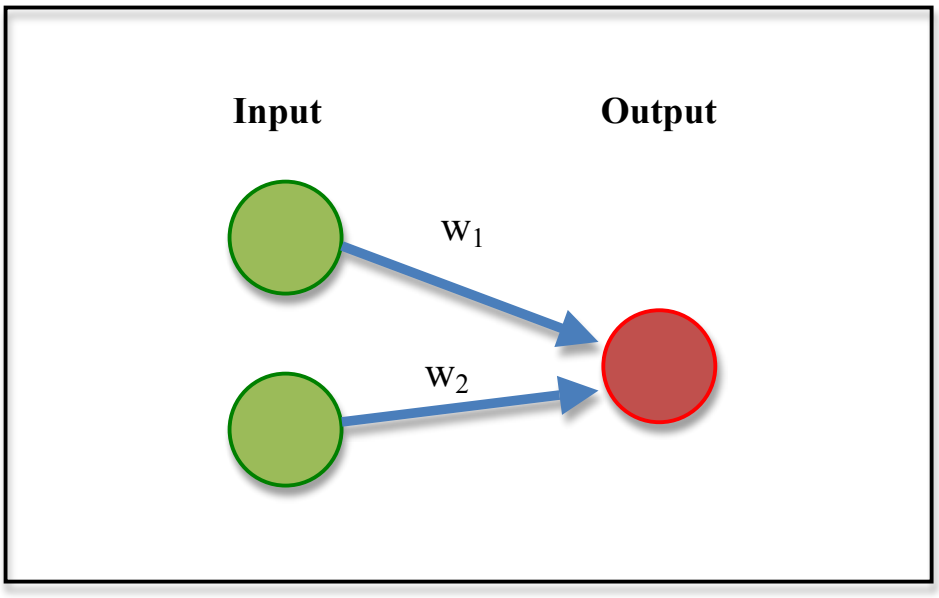

Figure 3.6. Perceptron used to implement an AND Logic Gate

Notice that to accomplish this, what we need to do is to find a set of appropriate values for the perceptron's weights and threshold that make the system behave as an AND logic gate. This concept of discovering a mathematical relationship between variables using sample data is often referred to as **regression**. Fortunately, Rosenblatt came up with a simple regression method that allows us to adjust these values. All he required to do was to adjust the weights and the perceptron's threshold using the two formulas below:

$$\Delta w_i = (t - o)\, I_i$$

$$\Delta \theta = -\,(t - o)$$

Figure 3.7. Formulas Used to Update the Weights and Threshold in a Perceptron

Notice that **t** is the target or desired output and **o** is the actual output produced by the perceptron using its current weights and threshold. The table below shows the resulting weights and threshold after using 4 training examples and these two formulas:

Input 1	Input 2	Target	Output	W_1	W_2	θ
1	1	1	0	-0.5	0.5	1.5
1	0	0	0	0.5	1.5	0.5
0	1	0	1	0.5	1.5	0.5
0	0	0	0	0.5	1.5	1.5

Figure 3.8. Resulting Weights after Using each one of Four Training Examples

After this training session, our perceptron ended up with a first weight, $\mathbf{w_1}$, equal to 0.5, a second weight, $\mathbf{w_2}$, equal to 1.5 and a perceptron's threshold, $\boldsymbol{\theta}$, equal to 1.5. It is easy to check that now our perceptron behaves just as an AND logic gate by plugging in each one of the four possible combinations of inputs. However, notice that these values are not the only way we can make our perceptron behave as an AND logic gate. Values such as 0.6, 1.8 and 2.1 would also do the trick.

It has been proven that this method is always able to discover a set of weights that correctly classifies its inputs, given that the set of weights exists. This important result, which is also known as the Perceptron Convergence Theorem, also extends to multiple output networks.

Unfortunately, there are certain types of problems that cannot be solved by using a perceptron. A simple example of this type of problems is an XOR logic gate. Below is the proof that it is not possible to implement an XOR logic gate using a perceptron:

If we use the first one of the four training examples, then we know that:

$(0*w1) + (0*w2) < \theta$

This implies that:

$0 < \theta$

If we do the same with the other three training examples we notice that:

$(0*w1) + (1*w2) > \theta \implies w2 > \theta$

$(1*w1) + (0*w2) > \theta \implies w1 > \theta$

$(1*w1) + (1*w2) < \theta \implies w1 + w2 < \theta$

If we apply the first finding on the other three we get:

$0 < \theta \implies w2 > 0$

$0 < \theta \implies w1 > 0$

$0 < \theta \implies w1 + w2 < 0$

However notice that this is impossible because we cannot add two positive numbers and end up with a negative number.

A way to address this problem is by adding an extra (hidden) layer between the inputs and the outputs. It has been proven that given a sufficient number of hidden units, it is possible to model any given function. The picture below shows an example of an ANN with a hidden layer.

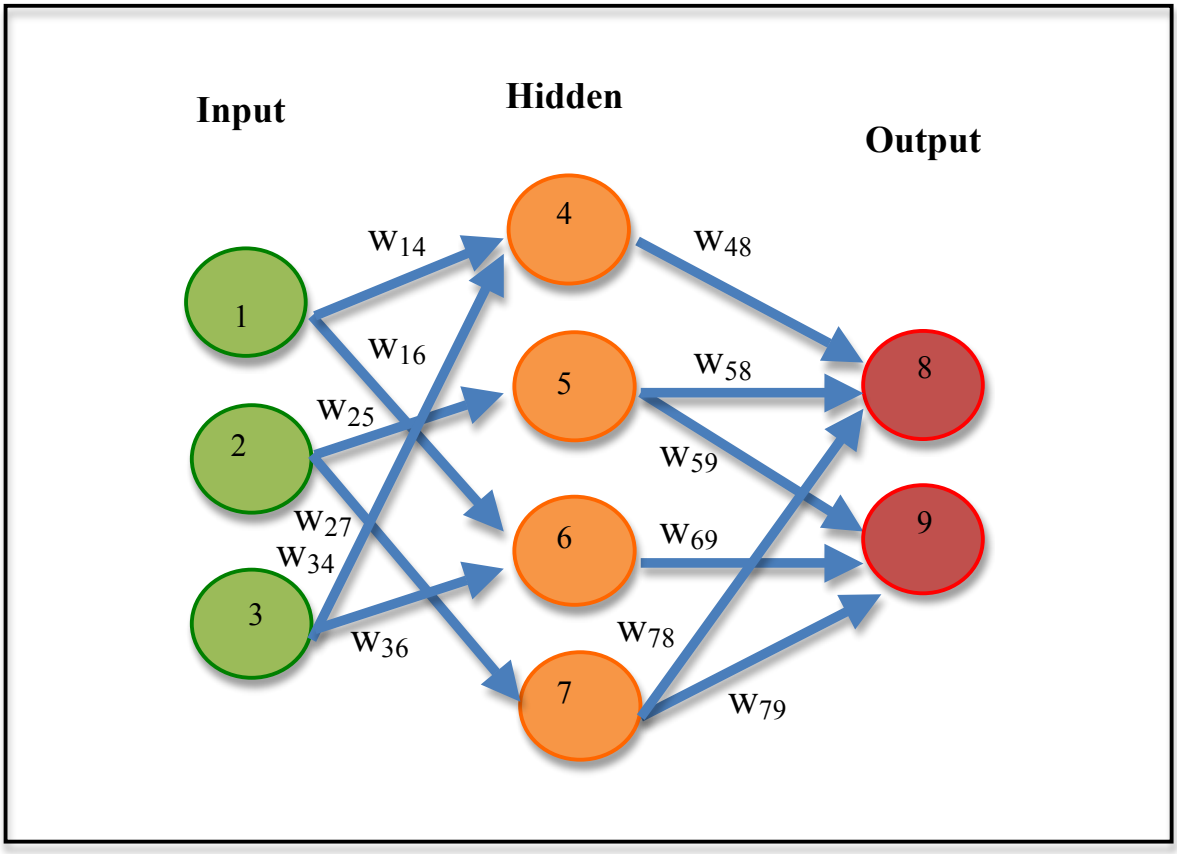

Figure 3.9. A Neural Network with a Hidden Layer.

In the case of a perceptron, we know what the second layer should output (since the training examples provide this information). Unfortunately, we have no ideas what the hidden layer should output. Therefore, we cannot any longer use the perceptron formulas to update the weights in this three-layer network.

Fortunately, in the 1970's Paul Werbos came up with a new regression algorithm called **backpropagation,** which can be used to train an ANN with a hidden layer. Learning using backpropagation occurs in two steps.

During the first step, a pattern or set of patterns is presented to the network and propagated forward to the output. Second, a method called gradient descent is used to minimize

the total error. In gradient descent, weights are changed in proportion to the negative of an error derivative with respect to each weight. In other words, we use the formula below to update each one of the weights:

$$\Delta w_{ji} = -\varepsilon \, [\delta E / \delta w_{ji}]$$

Figure 3.10. Equation Used by Backpropagation to Update the Weights in the Network.

In this equation **ε** is the **learning rate** and **E** is the **total error** as defined by the equation below. Notice that this equation accounts for the possibility of having more than one output signal:

$$E = \tfrac{1}{2} \sum (t_m - o_m)^2$$

Figure 3.11. Total Error Equation, where m is the Index for Each One of the Outputs.

Notice that we may require a large number of adjustments on the set of weights in order to minimize the total error below a certain acceptable threshold. Furthermore, it is even possible for this algorithm to enter an infinite loop. Therefore, it may be a good idea to limit the number of iterations to a reasonable number. Let us call A the maximum number of iterations we will allow. If n is the number of weights in the ANN, the computational complexity of updating the function given a single pattern could be described as A*O(n) or more appropriately O(n).

3.5.2 Why Updating The Moving Prototypes Function Is More Efficient

Before we can compare regression on ANNs and Moving Prototypes we need to recognize that these two methods work very differently. Therefore, it is impossible to compare

their performance apples to apples. However, both methods have a single type of parameter they modify in order to describe their function. In the case of ANNs this type of parameter is their weights. In the case of Moving Prototypes, the parameters are the prototypes themselves. Weights and prototypes are not so different. They are both associated with a specific section of their basic data structure, both can be implemented as a simple floating number and both of them are used to generalize from encountered learning experiences to new experiences not yet encountered. If we allow ourselves to compare ANNs and Moving Prototypes in the way they interact with these weights and prototypes I believe that Moving Prototypes is a more efficient method, when run in a modern serial computer.

To illustrate this point, let us compare an ANN whose function is described using n weights and a Moving Prototypes setup whose function is described by n prototypes. Furthermore, consider a problem in which none of the input dimensions are redundant. Now let us assume the artificial neural network is asked to update its function given a training example. In this case, since all the input dimensions are relevant, the regression algorithm will need to be applied to all the n weights of the network. Therefore updating the function will require a computational complexity of O(n) in a modern serial computer. On the other hand, Moving Prototypes keeps its prototypes in a tree like structure. Therefore, updating this function will require a computational complexity of just O(log n). This may not seem to be a big difference on performance if we talk about a few hundreds weights or prototypes. However, the human cerebellum has hundreds of millions of neurons and each one of these neurons may interact with hundreds or even thousands of other neurons. At this scale, a performance difference between O(log n) and O(n) can be huge. Just for illustration purposes, let us assume that we have one system with one trillion weights and another system with one trillion prototypes. Updating the

ANN would require interacting with one trillion weights while updating the Moving Prototypes may only require interacting with a few dozen prototypes.

There are at least two reasons why regression applied to ANNs with a large number of non-redundant dimensions is so inefficient when implemented in a modern serial computer. The first reason is that, even though ANNs are highly parallelized entities, modern serial computers are only able to emulate this network by simulating each neuron one at time. The second reason is that in the case where all the input dimensions are non-redundant, all the weights in the network are used to calculate the output and, therefore, regression needs to be applied to all the weights. In the case of Moving Prototypes, a function's update only affects at most a few nearby prototypes. Furthermore, thanks to the tree-like data structure Moving Prototypes is able to access the prototypes very efficiently.

3.5.3 Other Reasons for Choosing Moving Prototypes over LWPR

Our main reason for suggesting the use of Moving prototypes over LWPR is mostly related to efficiency. However, there are other disadvantages related to the use of ANNs, which is at the core of LWPR.

Even though neural networks are robust to errors in the training data and often find reasonably good policies while requiring relatively small memory resources they often require excessively long training times ([Tes00], [How98], [Kae96], and [Die99]). This is a major drawback in some problems, in which agents must adapt quickly to an ever-changing environment.

Furthermore, ANNs require some level of calibration. For instance, we need to decide how many elements we want to put in the hidden layer and which connections (i.e. arcs) we want

to have between the elements in all the layers. If we add too few hidden elements or too few arcs, we may not be able to properly describe the function. If we add too many hidden elements or too many arcs, learning may take longer that it needs to. Unfortunately, we have no way to know apriori what the function will look like. Therefore, the best we can do is experiment with several networks and see which one seems to work better. This adds an extra level of difficulty to its use, which may limit the number of people who may be qualified to use this method. Furthermore, the fact that we do not know what the final function may look like at the time we start learning, implies that we may have to go through multiple iterations on the topography of this network. This may delay the ultimate goal of creating a new artificial cerebellum.

3.6 Review

In this chapter I described some of the most important details related to the method Moving Prototypes. Furthermore, I explained why we believe this method is a better choice than other methods used to build artificial cerebellums such as CMAC, FOX and LWPR when modern computers are used. In this chapter I argued that artificial cerebellums used in many real life applications would need to handle a large number of non-irrelevant inputs. Furthermore, I mentioned that CMAC and FOX would do a poor job in these cases because these two methods do not scale up well. In addition, I mentioned that LWPR would update its function with a computational complexity of O(n). On the other hand, Moving Prototypes presented a computational complexity of just O(log n). Finally, I mentioned that methods using ANNs such as LWPR presented further issues. For example, I mentioned that the literature is full of examples of people complaining that ANNs require excessive training examples ([Tes00], [How98], [Kae96], and [Die99]). I suspect that at least in some cases, slow learning may be due

to poor representation of the intermediate functions. In addition to this issue, ANNs requires the user to perform complex calibration, which is pretty much impossible to automatize. The calibration depends on the function the ANNs will eventually learn. Unfortunately, this is a vicious circle: proper calibration requires knowing the function but the function cannot be learned until the calibration takes place. What often happens with ANNs is that the user experiments with several networks to see which one seems to work better. This adds an extra level of difficulty to its use, which may limit the number of people who may be qualified to use this method. Furthermore, the fact that we do not know what the function looks like at the time we start learning, implies that we may require multiple iterations on the topography of this network. This may slow down the process of creating new artificial cerebellums.

In the previous chapter I mentioned that previously proposed methods used to build artificial cerebellums show shortcomings in two main areas: 1) framework usability and 2) building blocks incompatibility. In this chapter I mostly addressed the second issue and mentioned why I believe Moving Prototypes is more compatible with modern serial computers, which are, currently, the main building blocks for Artificial Cerebellums. At this point the reader may be wondering what about massively parallel computers. Couldn't we use these to implement artificial cerebellums using ANNs? As I described in Chapter 2 using massively parallel computers, such as SPANN, has been attempted. Unfortunately this technology is still not practical to solve real life complex problems mostly because of cost. In the following chapter I will describe a newly proposed framework, which tackles the framework usability issue. I believe this framework provides clear and easy to use rules to build artificial cerebellums for a large number of applications.

CHAPTER 4: A NEW FORMALISM

4.1 Overview

In this chapter I describe a new framework that formalizes and facilitates the task of creating customized cerebellums. This methodology tackles the framework usability issue found in previous methods used to develop artificial cerebellums. In addition, I argue that this framework could be used to build a development tool that would give people, with little or no expertise in the area of A.I. or Machine Learning, the ability to quickly design and implement customized cerebellums for a wide range of applications.

4.2 Basic Components

In this section I will discuss the basic components used in my formalism. Each one of these components is associated with a unique symbol. For instance, we have a ***Master Learning Agent*** (LA) and its input/output signals:

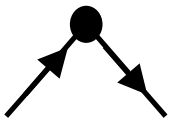

Figure 4.1. Symbol Used to Represent a Master LA and its I/O Signals

In this symbol, the dot represents the *master learning agent*. At this time you are probably wondering: Why do we call this LA a *master LA*? The answer is simple: Under this formalism, we may have many learning agents working together and organized in one or more trees. At the top of each one of these trees we will always have a learning agent, which "uses" the other learning agents in this tree to accomplish its task. We call this LA a *master LA*.

Using a similar logic, the other learning agents in the tree are called ***Slave Learning Agents***. The reason why I decided to keep track of this hierarchical organization is that, ultimately, a *master LA* will not achieve an optimal and stable policy until its *slave learning agents* have done so. By identifying the *master LA*, we also recognize the order in which the learning agents will be able to converge. This is important because this will help us direct our training efforts in a more efficient way. In other words, we will make sure *slave LAs* have converged to their optimal policies before we attempt to do the same with the *master LAs*.

Going back to the symbol used to represent a *master learning agent,* notice the two arrows coming in and out of this learning agent. The incoming arrow represents the ***input signal*** and the outgoing arrow represents the ***output signal***. Under this formalism we only have one input and one output signal associated with each LA. This assumption simplifies this formalism but does not make it less powerful, since we can always use encoders and decoders to describe learning agents with multiple inputs and outputs. The length of the incoming or outgoing arrow has no significance. The input signal represents the current state of the environment as seen by the Learning Agent and the output signal represents the action the LA wants to take at this time. The angle made by this arrow is also not important.

Another basic component is a ***sensor***. A sensor is a device that measures a physical quantity and converts it into a signal. Sensors allow us to discover the current state of our environment. An example of a sensor is a GPS receiver, which allows us to know the position of a certain component of our system. Other examples of sensors are devices that detect the humidity in the air or devices that detect the wind speed and direction. Below is the symbol associated with a sensor and its output signal:

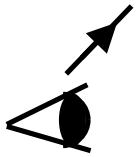

Figure 4.2. Symbol Associated with a Sensor and its Output Signal.

A sensor is represented using a symbol that resembles an eye. The arrow points away from this symbol because the signal is produced at the sensor and used somewhere else.

Another basic component is the ***actuator***. An actuator, in the context of an autonomous system, is typically a mechanism that allows it to change the state of its environment. An example of an actuator may be something like a turbine in an airplane. This turbine would allow the autonomous airplane to accelerate or decelerate. Other examples of actuators are the engine of a land rover or the steering system in a hauling truck.

Below is the symbol associated with an actuator and its input signal:

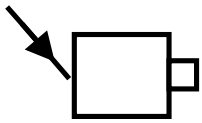

Figure 4.3. Symbol Associated with an Actuator and its Input Signal.

Notice that an actuator is represented using a symbol that resembles an electrical motor. The arrow points toward this symbol because the signal is produced somewhere else and consumed at this actuator.

As I mentioned before, we have *master learning agents* and we have *slave learning agents*. Among the *slave learning agents* we have two types: ***actuator learning agents*** and ***sensor learning agents***. *Sensor learning agents* take a signal and convert it into a hopefully more useful signal. For instance, a *sensor learning agent* may take in a signal that represents the pixels

in a photo of a face and may output a signal that specifies whether the person in the picture seems to be “happy”, “angry”, “sad”, etc.

The symbol of a *sensor learning agent* and its associated input/output signals is shown below:

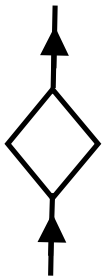

Figure 4.4. Symbol Associated with a Sensor LA and its I/O Signals.

An *actuator learning agent* takes a command from a “higher level” learning agent and creates a hopefully more useful signal. For example, a “higher level” learning agent may send the order to accelerate a vehicle by X m/s^2. In this case, the actuator LA may have the responsibility of finding how many centimeters it needs to press the accelerator pedal in the vehicle to reach this acceleration. In other words, it converts a signal that represents a desired acceleration into a signal that represents by how may centimeters we need to press the accelerator to achieve this goal. The symbol associated with an Actuator Learning Agent and its input/output signals is shown below:

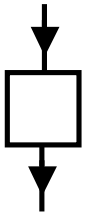

Figure 4.5. Symbol Associated with an Actuator LA and its I/O Signals.

As I mentioned before, I made the simplification in the master and slave learning agents that each one of them has a single signal coming in and a single signal going out. Clearly, we will find many situations in which a learning agent will depend on multiple signals. In these cases we will make use of a new basic component we call an ***encoder***. The picture below shows the symbol used to describe an encoder that takes three signals and converts them into a single signal:

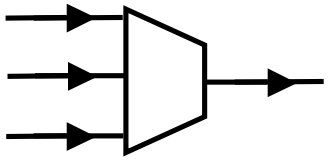

Figure 4.6. Symbol Associated with an Encoder, its 3 Input Signals and Output Signal.

This formalism does not put many constraints on how this encoder is implemented. The only constraint is that each combination of signals produces a unique output signal. To illustrate this, let us assume that we have two input signals I_1 and I_2. Each one of these signals can have integer values between 0 and 9. We also have an output signal called O. A way to implement this encoder would be to use the formula below:

$$O = (I_1 * 10) + I_2$$

Figure 4.7. Possible Implementation of an Encoder with Two Input Signals I_1 and I_2.

In a similar way, there will be cases in which we may want to take a signal and break it up into two or more simpler signals. For instance, a master learning agent may be in charge of driving a simple vehicle with an acceleration pedal and a steering wheel. By convention the LA would only have a single signal coming out of it. But this signal could be broken up into two

signals: one signal going into a stepper motor that controls the acceleration pedal and another into a stepper motor that controls the steering wheel. The picture below shows the symbol used to describe a ***decoder***, which takes a single signal and converts it into three simpler signals:

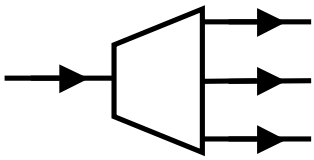

Figure 4.8. Symbol Associated With a Decoder, its Input Signal and Three Output Signals.

This formalism does not put any constraint on how this decoder is implemented. The only constraint is that each input signal produces a unique combination of output signals. To illustrate this, let us assume that we have an input signal I, which can have values from 0 to 99, and two output signals O_1 and O_2. A way to implement a decoder, using C++ notation, is shown below:

$$O_1 = I / 10$$

$$O_2 = I \% 10$$

Figure 4.9. Possible Implementation of Decoder with Two Output signals O_1 and O_2.

In this section I have talked about seven of the basic components in this formalism: 1) the master LA, 2) the sensor, 3) the actuator, 4) the sensor LA, 5) the actuator LA, 6) the encoder and 7) the decoder. In the next section we will see some examples of the types of distributed Q-learning systems we can create with these basic components.

4.3 Sample Distributed Q-learning Systems

A Distributed Q-learning system consists of at least one tree. Furthermore, each tree must

have at least one *master LA* and at least one *sensor* and one *actuator*. Therefore, the simplest system we can build with these components is the one shown below:

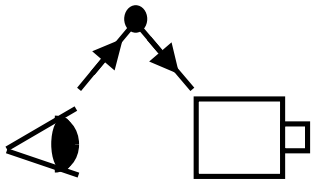

Figure 4.10. A Master LA with a Single Sensor and a Single Actuator.

If our *master LA* uses more than one *sensor*, we will need to insert an *encoder* between the *sensors* and the *master LA* as shown in the picture below:

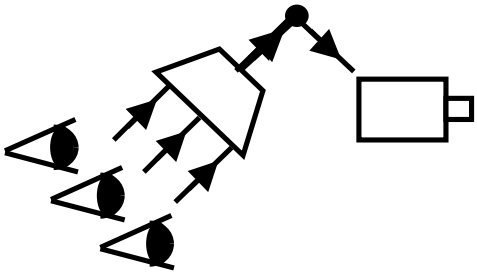

Figure 4.11. A Master LA with Multiple Sensors and a Single Actuator.

In the same way, if our LA controls multiple *actuators*, we will need to insert a *decoder* between the LA and the *actuators* as shown in the picture below:

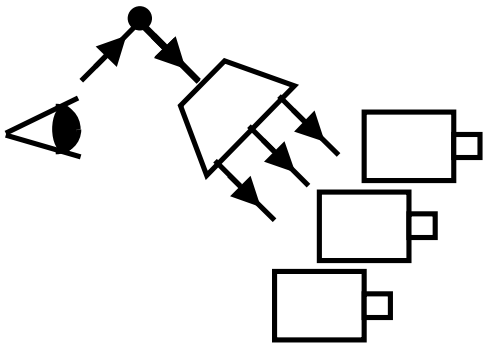

Figure 4.12. A Master LA with a Single Sensor and Multiple Actuators.

In some cases we may want to do some "pre-processing" on a complex input signal before feeding it to the *master LA*. For instance, let us assume our system is responsible for driving a car on a paved road. In this case our sensor may be a camera, which is responsible for recording the road ahead. In this scenario we may want to first create a simpler signal that describes how far ahead the next curve is. Then, we may want to feed this simpler information to a Master LA in charge or steering the vehicle. This distributed system is shown in the picture below:

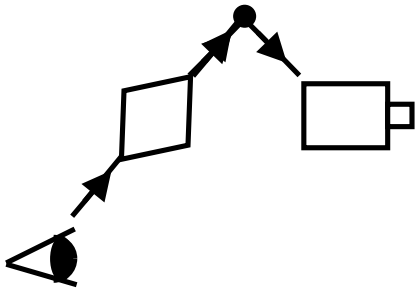

Figure 4.13. A Master LA with a Slave Sensor LA and a Single Actuator.

In other cases we may want to have the master LA be responsible for giving a high level command and let a *slave actuator LA* be responsible for executing this order. For example, the *master LA* may be responsible for coming up with the appropriate acceleration for a vehicle given the current traffic. However it would be the responsibility of a *slave actuator LA* to learn how many centimeters to press the accelerator pedal in order to reach a certain acceleration. This system is shown in the picture below:

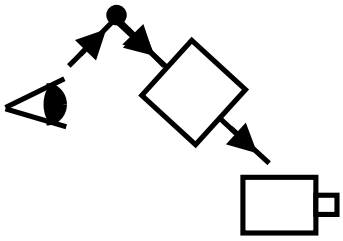

Figure 4.14. A Master LA with a Single Sensor and a Slave Actuator LA.

A system with a *master LA* and a *slave sensor LA* may use more than one *sensor*. The picture below shows such a system that uses three *sensors*:

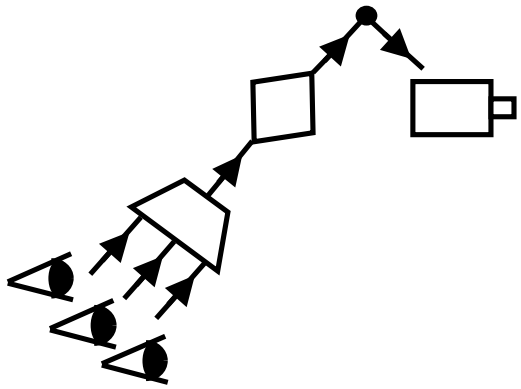

Figure 4.15. Multiple Sensors Feeding a Slave Sensor LA.

The picture below shows a distributed system with a *master LA* and a *slave actuator LA* that uses three *actuators* instead of one:

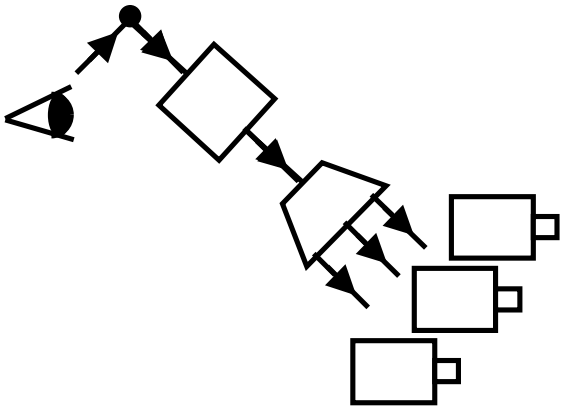

Figure 4.16. Multiple Actuators Controlled by Slave Actuator LA.

In some cases, several *master learning agents* may use the same *slave sensor LA*. This type of system is shown in the picture below:

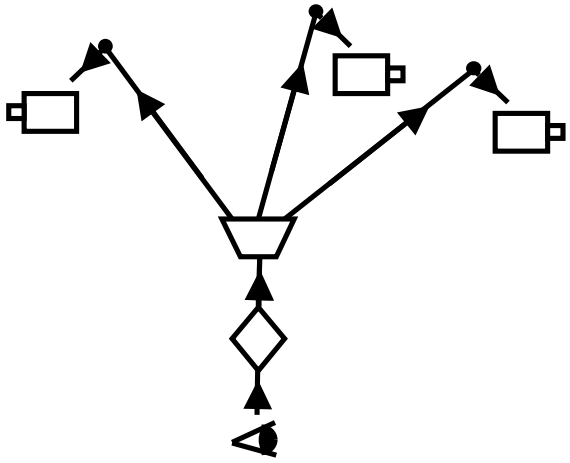

Figure 4.17. A Single Slave Sensor LA Feeding Multiple Master LAs.

Also, it is possible to have several *master learning agents* controlling a single *slave actuator LA* as shown in the picture below:

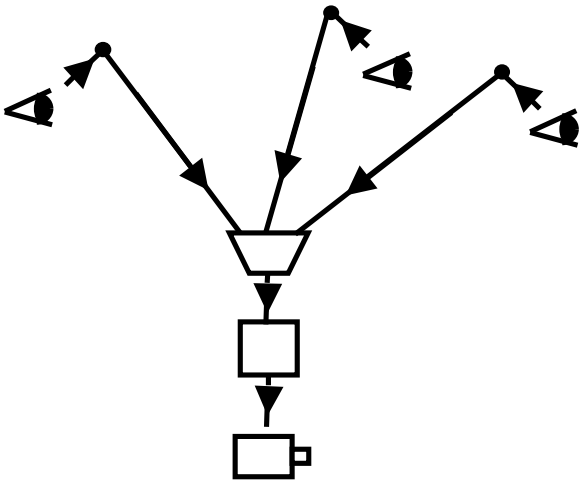

Figure 4.18. Multiple Master LAs Controlling a Single Slave Actuator LA.

Below are two pictures describing the two previous systems but this time with several *sensors* and several *actuators*:

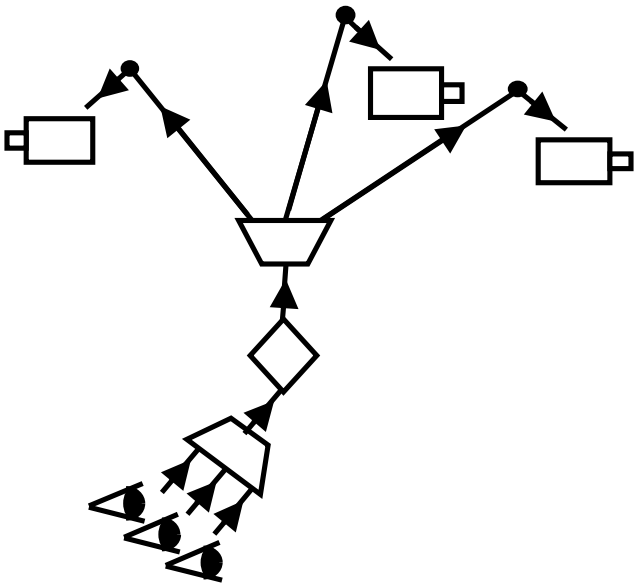

Figure 4.19. Multiple Sensors Feeding a Slave Sensor LA Used by Many Master LAs.

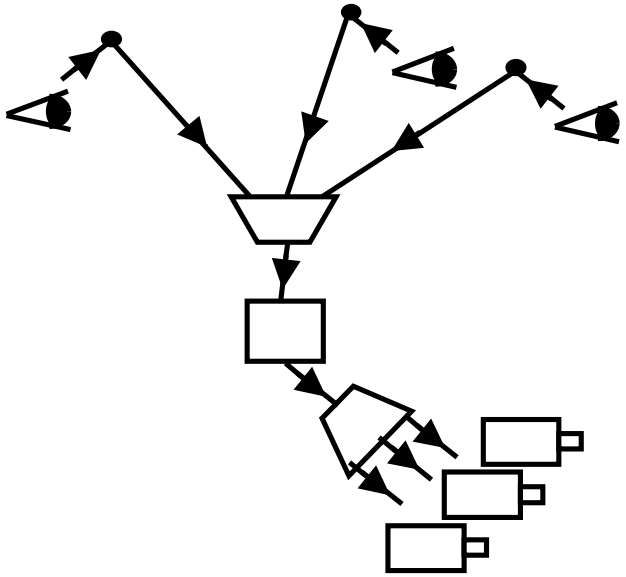

Figure 4.20. Many Master LAs Controlling a Slave Actuator LA with Many Sensors.

It is also possible to have a distributed system in which we have a set of *master learning agents* that share a single *slave sensor LA* and a single *slave actuator LA*. The picture below shows such a system:

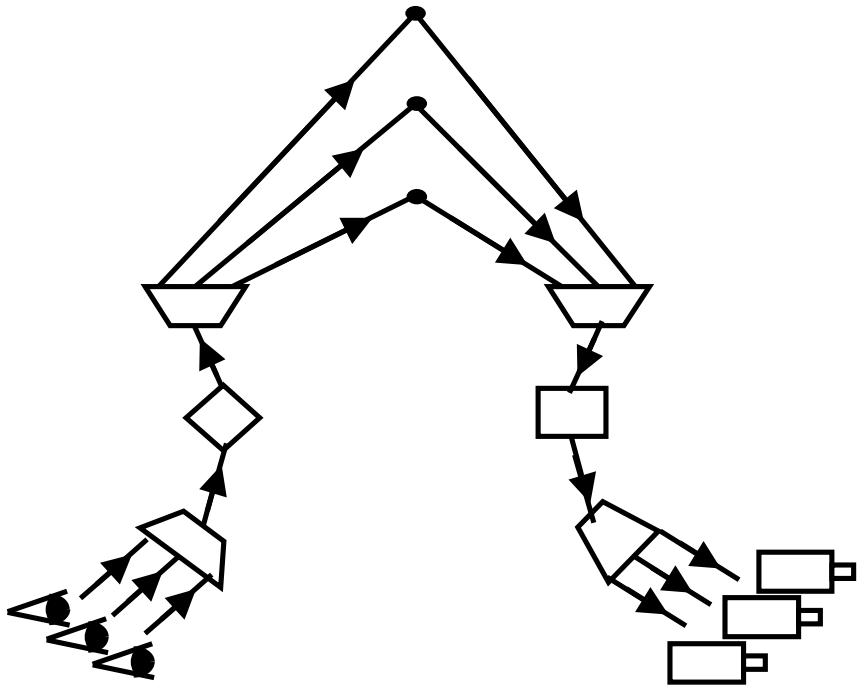

Figure 4.21. Master LAs Using a Single Slave Sensor and a Single Slave Actuator LA.

A system is not always described using a single tree. The picture below shows a system formed by three separate trees, each one using several *sensors*:

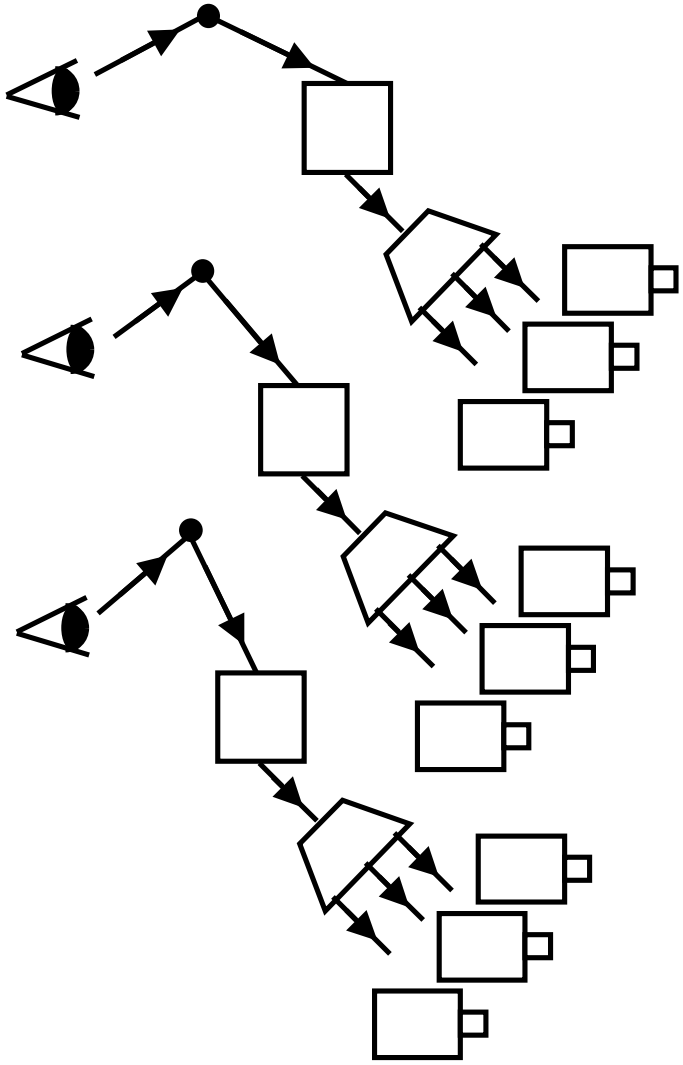

Figure 4.22. System Composed of Three Separate Trees.

The picture below shows a system composed of three trees, each one making use of several *actuators*:

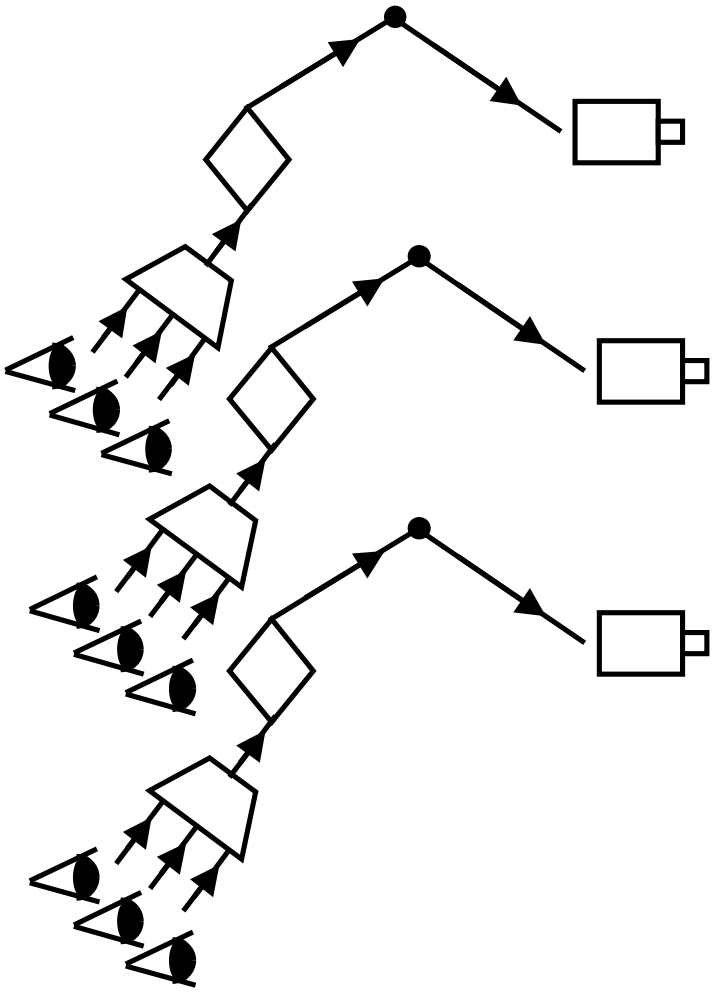

Figure 4.23. System with Three Trees Using more Complex Slave Actuators LAs.

In some cases we may have a system where several multi-input *sensor LAs* service several *master learning agents* and these *master LAs* control a single multi-output *actuator LA*, as shown below:

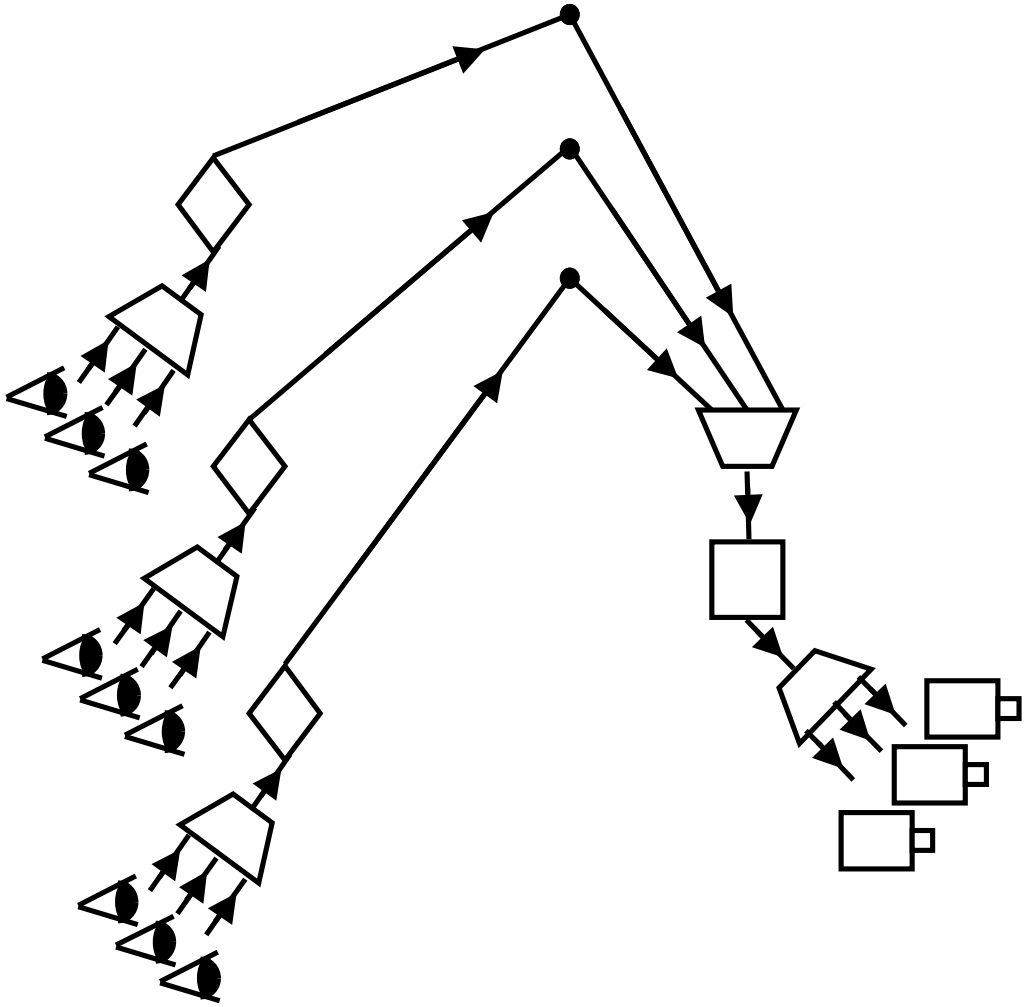

Figure 4.24. Master LAs with Separate Slave Sensor LAs but Common Slave Actuator LA.

It would also be possible to have a system in which a single multi-input *sensor LA* services multiple *master LAs* and these *master LAs* control their own separate multi-output *actuator learning agents*. This system is shown in the picture below:

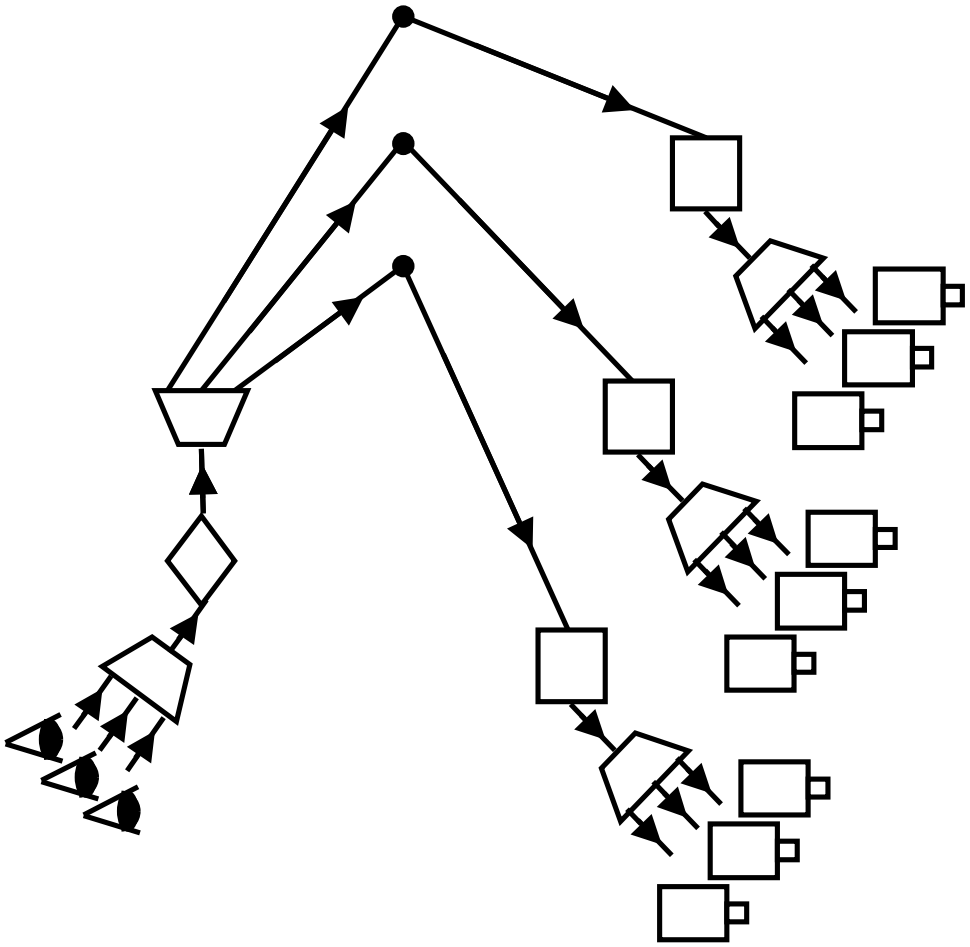

Figure 4.25. Master LAs with Common Slave Sensor LA but Separate Slave Actuator LAs.

Another system we may encounter is one composed of several trees, each one making use of their own multi-input *sensor LA* and multi-output *actuator LA*. This is shown in the picture below:

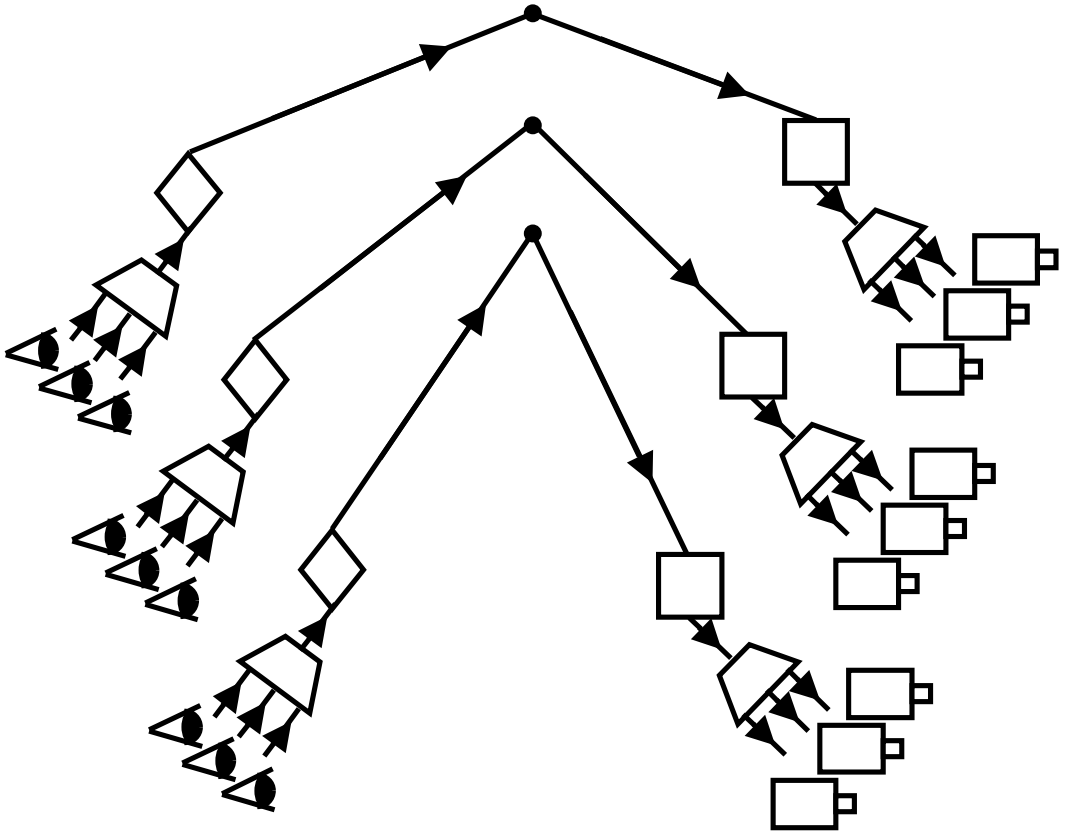

Figure 4.26. System with Three Complex Trees.

It may also be possible to have a system in which we have a set of *slave sensor learning agents* connected in a daisy chain as shown below:

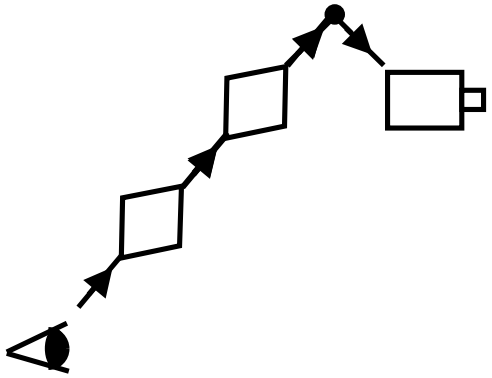

Figure 4.27. Slave Sensor LA Servicing Another Slave Sensor LA.

Furthermore, we could have several slave *actuator learning agents* controlling each other in a daisy chain as shown below:

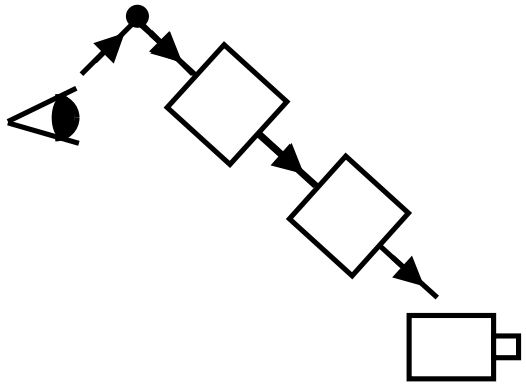

Figure 4.28. Slave Actuator LA Controlled by Another Slave Actuator LA.

These daisy-chained systems may have several *sensors* or several *actuators* as show in the two pictures below:

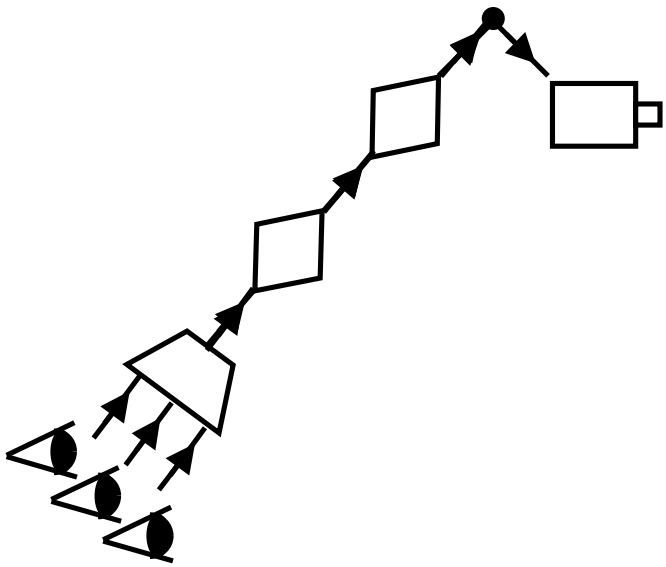

Figure 4.29. Complex Sensor LA Feeding Another Slave LA.

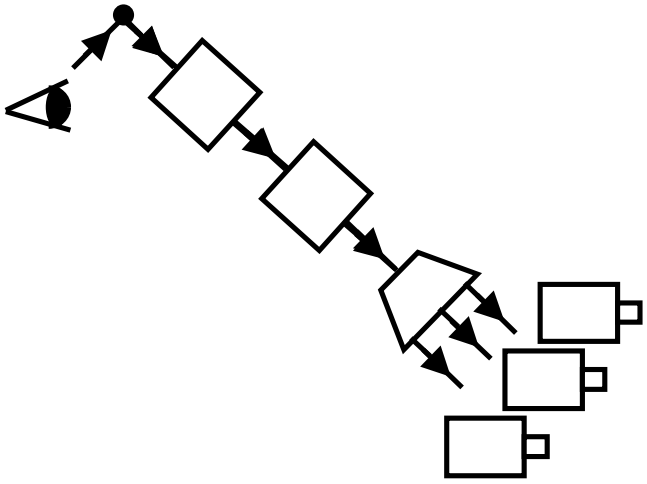

Figure 4.30. Complex Actuator LA Being Controlled by Another Slave Actuator LA.

Sometimes we may have several multi-input *sensor learning agents* servicing a multi-input *sensor LA*, which serves a simple *master LA* as shown below:

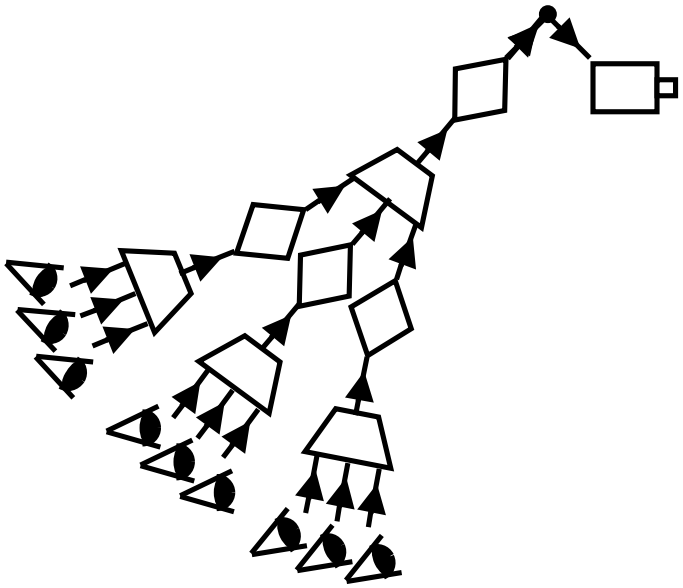

Figure 4.31. Complex Sensor Learning Agents Feeding Single Slave Sensor LA.

We could also have a system in which several multi-output *actuator LAs* are being controlled by a multi-output LA which is controlled by a simple *master LA* as shown below:

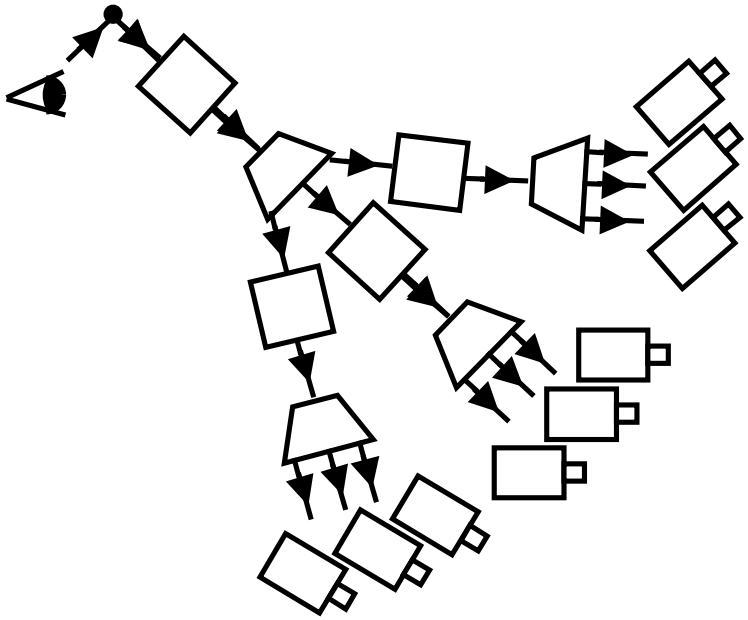

Figure 4.32. Complex Actuator LAs Controlled by Single Slave Actuator LA.

A more complex system would be one in which several multi-input *sensor learning agents* service a multi-input *sensor LA,* which services a *master LA*. This *master LA* controls a multi-output *actuator LA*, which control several multi-output *actuator learning agents*. The picture below shows this system:

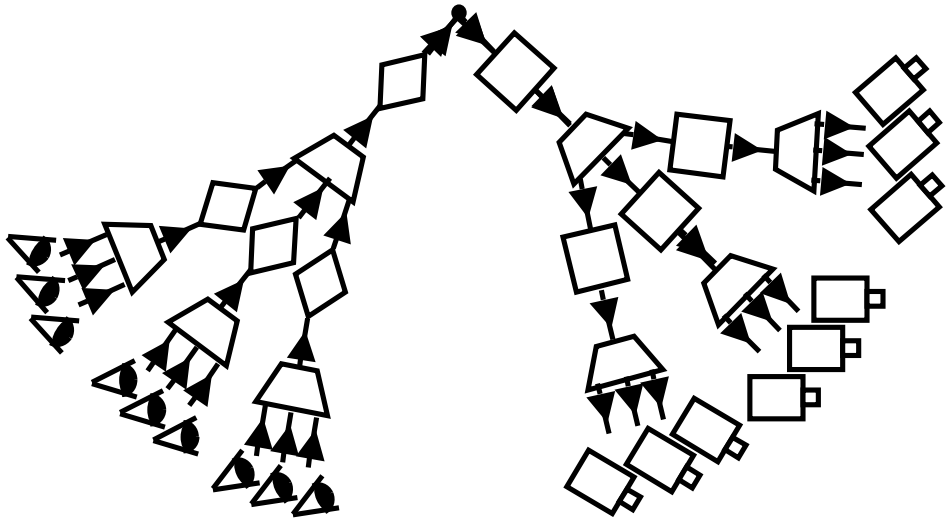

Figure 4.33. Complex Daisy Chained Sensor and Actuator Learning Agents.

We could continue creating more and more complex and exotic systems but these examples are probably enough as an illustration to what can be described using this formalism.

4.4 Sanity Points

As I mentioned in the previous chapter, a learning agent using Q-learning is very good at finding optimal policies. However, this type of LA does have several disadvantages.

The first disadvantage has to do with safety. It is not practical to track and make sense of all the steps used by a Q-learning LA in order to come up with an "optimal policy". Therefore, we can never be sure that we have achieved a policy that is optimal and safe in every single case. Furthermore, even if a policy has been safe to use for a long period of time, it does not mean it will continue being safe in the future. The environment may eventually change and the current policy may no longer be appropriate. Unfortunately, a system using a single Q-learner does not provide explicit safeguards to blow the whistle if dangerous actions are ever selected by this policy.

A second disadvantage is that using a single Q-learner in our system limits the level of parallelism we can apply during the training of this agent. We would much rather have a system composed of simpler LAs, each learning in parallel, possibly by running on a separate processor or a separate computer.

This is where the concept of ***sanity point*** comes about. The main idea behind *sanity points* is being able to open windows at different points in the system. These windows can be used to check things and add alarms. *Sanity points* allow us to detect if a policy is not behaving as it should. An added advantage of *sanity points* is that they allow us to break a big black box system into smaller and simpler components that are easier to comprehend, manage and eventually reuse. Furthermore, these simpler components can be trained in parallel and often completely independently of each other.

Our proposed formalism uses the symbol below to describe a *sanity point*:

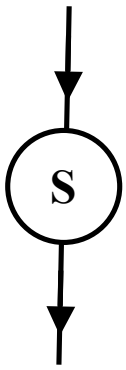

Figure 4.34. The Sanity Point Symbol and its Input and Output signals.

A *sanity point* can be placed at any signal that can be corroborated independently. For instance, let us assume that we have a system that is in charge of accelerating or slowing down a spacecraft so that it is not hit by meteorites moving perpendicular to its trajectory. Let us assume that our system looks like this:

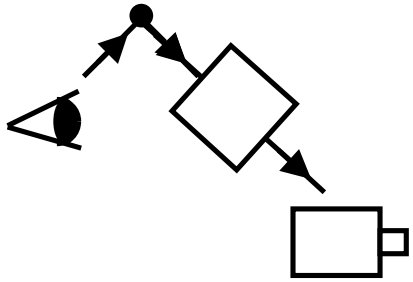

Figure 4.35. Meteorite Anti-collision System.

In this case, the *sensor* indicates where and when the meteorite will intersect the trajectory of the spacecraft. The *master LA* is in charge of indicating the desired acceleration one second later. The *actuator LA* is in charge of sending the appropriate signal to the propulsion system so that the desired acceleration is achieved one second later.

Since acceleration of the spacecraft one second later can be corroborated independently of the system (e.g. by making use of an accelerometer), we could very well put a *sanity point* at this signal. Therefore, our system would look like this:

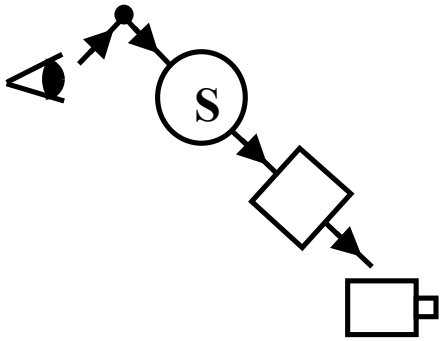

Figure 4.36. A Safer Meteorite Anti-collision System.

In this case, the *sanity point* would be in charge of comparing the achieved acceleration, after one second, to the acceleration ordered by the *master LA*. If these two accelerations differ by more than a certain amount and this happens very often, this may tell us that the policy

learned by the *slave actuator LA* is either not very good or that the environment has changed too much. In either case, we may be better off stopping this system and retraining it, before it causes some damage to the spacecraft.

Furthermore, the system is now broken into two systems, which can be trained independently of each other. The *master LA* can be trained assuming that the desired acceleration can and will be achieved every time. Furthermore, the *slave actuator LA* can be trained to just achieve the desired acceleration without having to worry where the meteorites will intersect our trajectory or when.

In other words, after using a *sanity point*, we end up with two much simpler systems, which are not just easier to understand and test but can also be trained independently. Furthermore, this *sanity point* makes our system more reliable.

The use of *sanity points* in our formalism is completely optional. However, I strongly suggest its use. I believe that *sanity points* cannot only make the system safer to use and faster to train (by allowing parallel learning), but they also make the system simpler, which facilitates the task of finding flaws and improving the system.

4.5 Abstract vs. Specific Sensors and Actuators

So far, when I have mentioned *sensors* and *actuators* in our sample systems, I have done a little bit of *hand waving* and have not elaborated how the *sensors* are able obtain their signal and how *actuators* are able to accomplish their task. In other words, I have been really talking about ***abstract sensors*** and ***abstract actuators***, as opposed to ***specific sensors*** and ***specific actuators***. It is fine to do this when we are beginning to design a new system. However, before we can really implement our system, we need to replace all our *abstract sensors* and *abstract*

actuators with *specific sensors* and *specific actuators*. For example, let us assume we have a simple system like the one below:

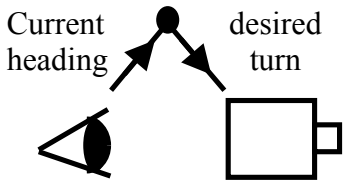

Figure 4.37. North Driver System with Abstract Sensor and Actuator.

In this case, the *master LA* receives a current heading and turns the vehicle so that it always drives north. This is a straightforward system. However, we have not yet specified how we get the current heading and how we turn the vehicle left or right.

For illustration purposes let us assume we decide that the heading will be calculated using an electronic compass and that the vehicle will be turned left or right by a stepper motor attached to the steering wheel. Furthermore, we specify that the electronic compass will provide 360 different signals (one for each degree). In addition, we assume the stepper motor will accepts 101 different command signals (where -50 orders the motor to turn the steering wheel 50 degrees to the left, 0 orders the motor to keep the steering pointing straight ahead and 50 orders the motor to turn the steering wheel 50 degrees to the right). After this description we now have specific sensors and actuators and our system looks like this:

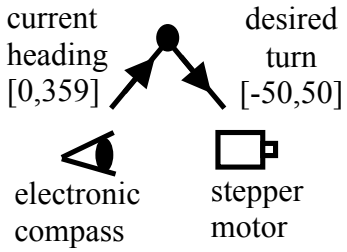

Figure 4.38. North Driver System with Specific Sensor and Actuator.

As I mentioned before, it is ok to use abstract sensors and actuators at the first stages of our design. However, we need to specify all the sensors and actuators in the system before we can implement it.

4.6 Dynamic vs. Static Sensors

When we talk about *sensors*, we need to differentiate between ***dynamic sensors*** and ***static sensors***. The difference is that, while the value associated with a *dynamic sensor* may constantly change, the value in a *static sensor* never changes, or at least not significantly. An example of a *dynamic sensor* is a speedometer. An example of a *static sensor* is a balance mounted at an electrical autonomous vehicle shock system that is able to tell the weight of the vehicle (no tires and shock system included of course). Notice that in this last case, we may weigh the vehicle every day but the weight would be the same (except for minor wear and tear on the vehicle). If this is the case, we may be better of categorizing this *sensor* as *static*, and assume the reading will never change. The significance of *static sensors* is that we may be able to do without installing this *sensor* in the equipment at all and just measure this value during installation.

Some *sensors* can be very expensive and may malfunction in extreme conditions. If a *sensor* can be categorized as a *static sensor* we probably should do so. This would not only make the system cheaper to produce and maintain but and also more robust.

4.7 Simplified Distributed Q-learning Systems

Now let us talk about systems that can be simplified. For example, let us suppose we have a system that has two *slave sensor learning agents* (i.e. A and B) setup in series as shown below:

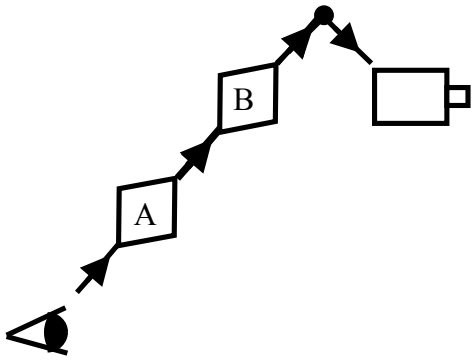

Figure 4.39. System with Slave Sensor Learning Agents A and B in Series

If we do not intent to have a *sanity point* between the two *sensor LAs*, the system below will also be able to learn the control task at hand:

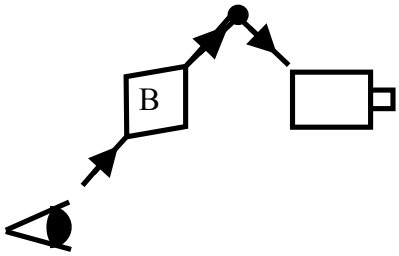

Figure 4.40. System with Single Slave Sensor LA.

The reason is the following. The first single-input *sensor LA* (labeled as A) is mapping a set of signals *a* to a set of signals *b*. Then, the second single-input *sensor LA* (labeled as B) is mapping a set of signals *b* to a set of signals *c*. Since a *sensor LA* is able to map any set of input signals into any other set of output signals, the second sensor (labeled as B) will be able to map the set of signals a directly to the set of signals c. Therefore, the second system (the one with a single *slave sensor LA* is equivalent to the first system, the one with two *slave sensor learning agents* in series).

Furthermore, if we do not require having a *sanity point* between the *sensor LA* and the *master LA*, we can use the same logic to simplify the system into:

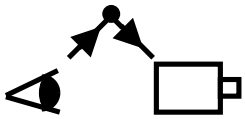

Figure 4.41. System with no Slave Sensor LA.

The same thing could be done with a system that has a set of chained *actuator learning agents* like the one below, as long as we do not wish to have *sanity points* between the learning agents. The picture below shows how we can simplify a system with several *slave actuators learning agents* into one without any *slave actuator LA*:

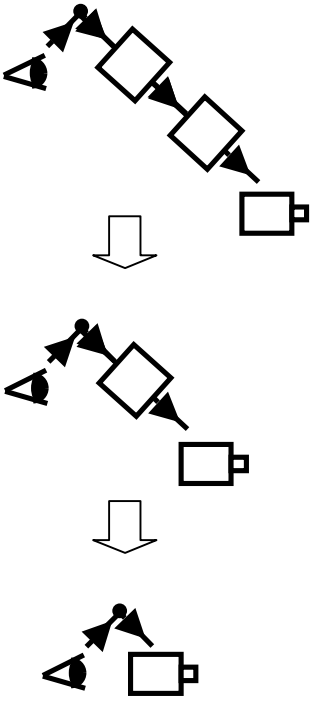

Figure 4.42. Sequential Simplifications on Serial Actuator Learning Agents.

If after some simplification we end up with a system that looks like this:

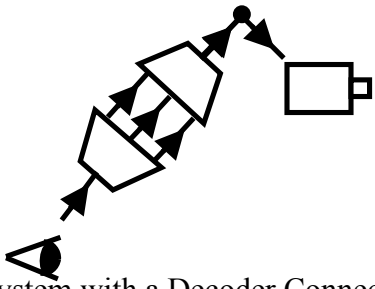

Figure 4.43. System with a Decoder Connected Directly to an Encoder.

We could simplify this system into an equivalent one that looks like this:

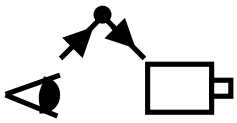

Figure 4.44. Equivalent System Without the Encoder and Decoder.

The reason why we can do this type of simplification is that a *decoder* converts a signal into several signals $x_1, x_2, x_3, \ldots x_n$. The combination $x_1, x_2, x_3, \ldots x_n$ is unique for each unique signal *a*. Furthermore, an *encoder* takes a set of signals $x_1, x_2, x_3, \ldots x_n$ and converts them into a signal *b*. In this case the signal *b* is unique for each $x_1, x_2, x_3, \ldots x_n$ combination. Therefore we will have a unique value for the signal *b* for each unique value in the signal *a*. A learning agent is able to find the optimized policy given any signal, as long as different states are represented using different values. Therefore, if a learning agent is able to find an optimized policy using signal *b*, then a learning agent will be able to do the same using signal *a*.

The same would be true if the *decoder* and *encoder* were on the right side of the master LA, as shown in the picture below:

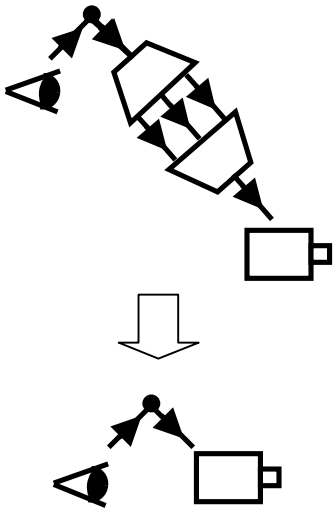

Figure 4.45. Simplification of a System with a Decoder Next to an Encoder.

Notice that in this case, the second system would simply end up generating the same signal generated by the *decoder* and *encoder* in the first system in order to maximize its reward. So in either way we will end up with the same signal being passed to the *actuator* for every different signal detected by the *sensor*. So both systems will be equivalent from the point of view of *sensors* and *actuators*. Therefore, both systems will generate the same value function and, hence, both systems will be equivalent.

Furthermore, if we start with a system that looks like this:

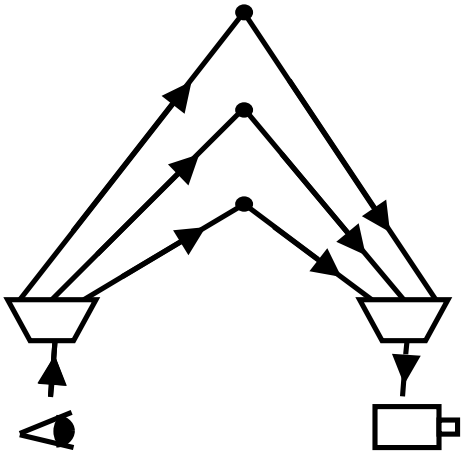

Figure 4.46. System with Several Master LAs Sharing the Same Sensor and Actuator.

In this case, we could be better off converting this system into one that looks like this:

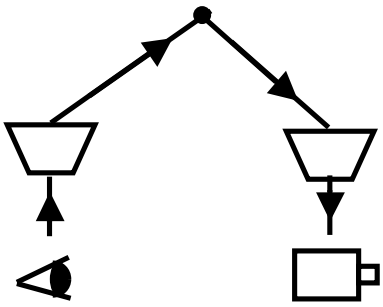

Figure 4.47. Faster Learning System Using a Single Master LA.

The reason why we may be better off with the second system is that the optimal policy for a master LA in the first system will be affected by the policy used by the other master LAs in this system. Since the policies used by the other LAs are not static, the *environment* from the point of view of any of the *master LAs* is changing constantly.

Convergence to an optimal policy in an environment, where the environment is also converging, may not only take a long time but it may never happen [Mit97].

In this case, the two systems are not equivalent. However, they both strive for the same results from the point of view of the user (i.e. maximize the reward provided by the reward function defined by the user, given a set of *sensors* and a set of *actuators*). Nevertheless, the second system will converge faster to the optimal policy and is, therefore, the system of choice.

Furthermore, it does not make much sense to have a single signal going out from a *decoder* or a single signal going into an *encoder*. Therefore one more simplification would be to get rid of both the *decoder* and *encoder* in this system. Therefore, we end up with a system that looks like this:

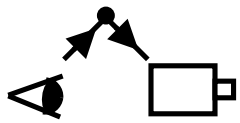

Figure 4.48. Equivalent System Using a Single Master LA.

4.8 Complexity Reduction Rules

The previous section mentioned some simplifications we can make on a system. Below is a more formal list of simplification rules:

1) A simplification may only occur as long as no *sanity point* is destroyed.
2) Two *sensor learning agents* in series can be merged into a single *sensor learning agent*.
3) Two *actuator learning agents* in series can be merged into a single *actuator learning agent*.
4) If we have a *decoder* and an *encoder* in series and the output signals for the *decoder* are exactly the same signals as the input signals in the *encoder*, we can simplify the system by deleting the *Decoder* and the *Encoder* and by connecting the input signal of the *decoder* to the output signal of the *encoder*.
5) If several LAs receive signals from a single *decoder* and pass their signals to a single *encoder*, the system can be made to learn faster by replacing all the LAs by a single LA.
6) If a *decoder* has a single output signal, we can delete this *decoder* and connect its input signal to its single output signal.
7) If an *encoder* has a single input signal, we can delete the *encoder* and connect its single input signal to its output signal.

4.9 Formalization on How to Build a System

In this chapter I have introduced many concepts to try to simplify the task of creating an artificial cerebellum for an autonomous system. However, I have not yet mentioned how all those concepts work together to create a recipe to build these autonomous systems.

Coming up with a formal recipe or method to build artificial cerebellums has many advantages. One of them is that if we have a simple method, more and more people will be able to get involved in the design of the artificial cerebellum and therefore more flaws will be detected early on in the design process. Furthermore, if we have a simple to follow method, the task of creating a complex artificial cerebellum will become less risky. The reason for that is because more people will be able to get involved and, therefore, someone leaving the development team will be less devastating to the project. In other words, nobody in the team will be indispensable, as it happens in projects in which only one person has the expertise to carry out a certain task. Finally, a huge advantage of having a formal method to design artificial cerebellums is that it will allow the creation of tools that automatize many of the tasks in this process and will allow the reuse of well-tested components. All this will accelerate the task of designing new artificial cerebellums and will make the project less risky and less costly.

This section describes a proposed formalized method to design artificial cerebellums for autonomous systems. This method is broken up into eight steps:

The ***first step*** in the proposed methodology is to become familiar with the system we want to design. What is the goal of the system? What is the system's environment like? What other systems interact with it and how? For instance, if the goal for our artificial cerebellum is to drive a vehicle, we need to ask ourselves, what type of vehicle is it? Does it have tires? Does it fly? Does it move below water as a submarine? Does it float on water as boats do? What type of

propulsion system does it have? Is it a turbine? Is it an internal combustion engine? How does this vehicle control its heading? What type of tasks is this vehicle responsible for doing? Does this vehicle transport people? Does this vehicle transport cargo? What is the normal environment for this vehicle? Does it drive in a city? Does it drive in the desert? Should we expect to have people or animals nearby the vehicle? Gaining a lot of application specific knowledge is not absolutely necessary (the nice thing about systems using Q-learning agents is that they do not have to have a model of the environment in order to find an optimal policy). However, it is typically a good idea to try to gain as much application specific knowledge as possible, as such knowledge may allow us to make significant simplifications to the system.

The ***second step*** of the design formalism is to enumerate all the factors we think that may be significant to the system we have at hand. For instance, if our system is in charge of moving large amounts of material from one place to another, one important factor may be the direction of the wheels, the position of the equipment, its total weight or its velocity. However, things like color of the material or exact temperature of the material may not be really critical information and may have such a small effect on the achievement of the goal of the system that we may as well disregard them. The main goal of this step is to come up with the minimum set of factors that are significant to the system we want to design.

The reader may be thinking right now…but what if I get this list wrong? What if I forget to add some important factor and the whole system underperforms? What if it turns out that some factor is not really that important after all? The good news is that the proposed methodology does not require coming up with the optimal set of factors at the very first try. The proposed design methodology is highly iterative and allows for the improvement of the system by trial and error.

The ***third step*** of the design formalism is to look at the list of factors and categorize them

as either *actuators* or *sensors*. Notice that it is possible for a factor to be associated with both, a *sensor* and an *actuator*. For instance, if we talk about the factor *equipment heading*, we may associate this factor with both a *sensor* (which tells us what the current heading of the equipment is) and with an *actuator* (such as a steering system that allows us to change the heading of the equipment). Here is a more formal description of what we mean by *actuators* and *sensors:*

a) Actuators

An *actuator* is typically a mechanism that allows a system to change its state in its environment. An example of an *actuator* may be something like a turbine in an airplane. This turbine would allow the autonomous airplane to accelerate or decelerates. Other examples of *actuators* are the engine of a land rover or the steering system in a hauling truck.

b) Sensors

A *sensor* is a device that measures a physical quantity and converts it into a signal. *Sensors* allow us to discover the current state of our system and its environment. An example of a *sensor* is a GPS receiver, which allows us to know the position of our system. Other examples of *sensors* are things like a device that detects the humidity in the air or a device that detects the wind speed and direction. There are two types of *sensors*:

i. ***dynamic sensors***: The reading coming from this type of *sensor* may be constantly changing (an example of this type of *sensor* is a speedometer).
ii. ***static sensors***: Their values may be measured only once as they are not expected to change significantly during the life of the system we are

designing. An example of a *static sensor* would be something like the weight of a component of the system. There may be wear and tear on this part and its weight may change with time, but this change is expected to be negligible. Therefore, we may decide to consider this *sensor* to be *static*. The significance of *static sensors* is that we may be able to do without installing this ***sensor*** in the system and just measure these values during manufacturing. Some *sensors* can be very expensive and may malfunction in extreme conditions (such as the ones often found in mining). If a factor can be categorized as a fixed sensor we probably should do so. This would not only make the system cheaper to produce and maintain but also more robust.

The ***fourth step*** is to specify what *sanity points* we want to have. However, this step is optional. If we decide not to have any *sanity point* then what we end up with is a single LA that has all the sensors and actuators specified in the previous step. Something like this:

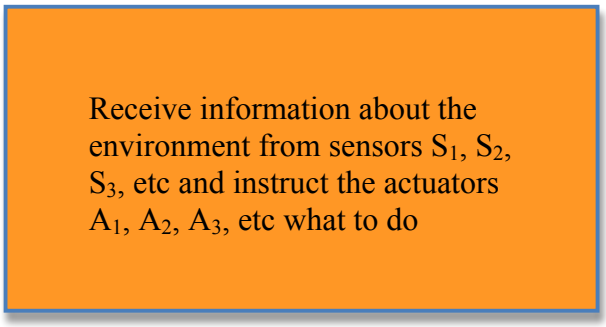

Figure 4.49. A System Without any Sanity Point.

If on the other hand we decide we want to insert a sanity point we need to find a signal, which can be verified independently of the system. For instance, let us assume that our system is

in charge of driving a vehicle and one of the *actuators* in this system is able to increase its acceleration. Here we have identified a signal (i.e. acceleration) that can be verified independently of the system by making use of something like an electronic accelerometer. In this case, the system may come up with a desired acceleration given the current state of the environment and the sanity point could verify that this acceleration is accomplished after a certain period of time. If this is not accomplished it could sound an alarm warning us that the system is either not mature enough or the environment has changed too much. In either case, the system is probably not safe to use and, most likely, it is better to stop using it until it has gone through more training and its policies have become more optimal and more robust. After the introduction of this sanity point we would end up with a system that looks like this:

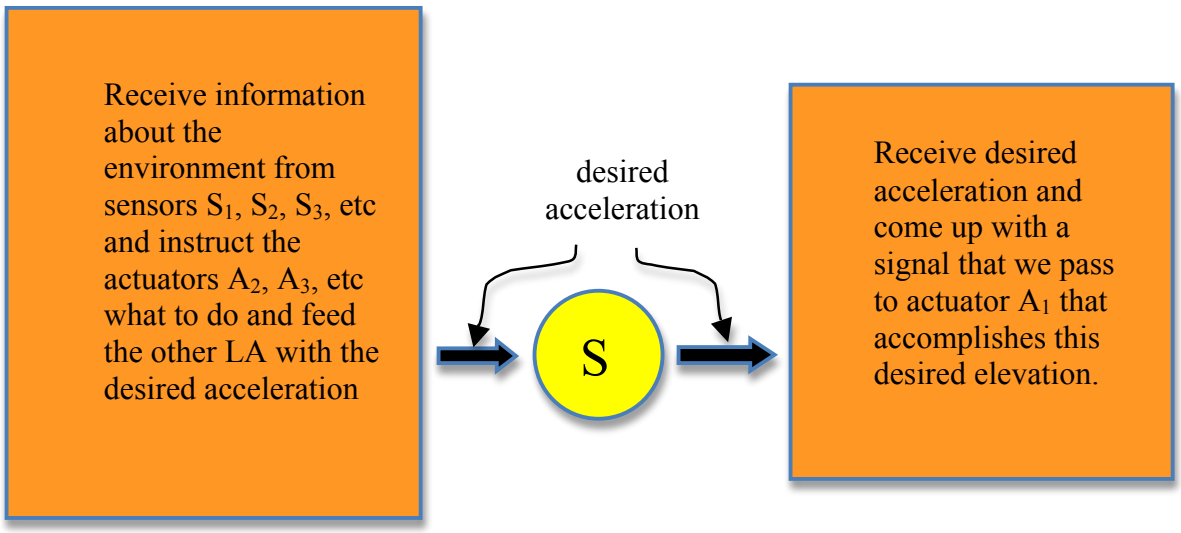

Figure 4.50. A System with One Sanity Point.

We can repeat this process as many times as we want to insert more sanity points. Remember that the main purpose of having sanity points is to make the system trustworthier and less complex. In other words, sanity points make it easier to determine if our system is not

mature enough or if the environment has changed so much that our current control policies are no longer effective. If we determine that this is the case, we may decide that it is no longer safe to continue the operation of this autonomous system and may request further rounds of learning experiences before it is deployed again. Furthermore, sanity points may allow us to break a very complex system into simpler components. This may not only simplify the design of the system and allow the reuse of robust components but it may also shorten the overall training time since it may be possible to train some components in parallel. In the case above, the insertion of a new sanity point created two simpler learning agents out of a more complex single LA. Furthermore, it made the system more robust by being able to identify when the system cannot accomplish the optimal acceleration any longer for the current state of the environment.

The ***fifth step*** is to look for overloaded learning agents in the system and break them into simpler and more manageable LAs. In other words, review the responsibility given to each one of the learning agents. If you think one of them may be overloaded, break this LA into two or more LAs. For instance, you may discover an LA in your system that is in charge of driving a car on the freeway. The sensor for this learning agent is a stream of 100x100 pixels coming from a camera and the actuators are the accelerator and the steering wheel. You may compare this task to the tasks given to the other LAs in this system and realize that this task is much more difficult to achieve. For instance, other learning agents in the system may be responsible for simple tasks such as turning on the head lamps depending on the time of the day or turning on the wipers in front of the camera if it rains. These are way simpler tasks than driving a vehicle on a freeway. Therefore we may decide to break the task of driving the vehicle on a freeway into five simpler tasks: 1) determine from the stream of 100x100 pixels, what is the distance to the closest vehicle in front of us, 2) determine how far is the next curve by using the stream of 100x100 pixels, 3)

take in the distance to the next vehicle and the distance to the next curve to determine the desired heading and desired acceleration, 4) achieve desired heading, 5) achieve desired acceleration. In this case, we may have started with three LAs, one of them very overloaded. After this exercise we would end up with seven LAs, each of them responsible for a simple task.

The ***sixth step*** is to use the symbols introduced in this chapter to describe the *learning agents*, *sensors*, *actuators* and *sanity points* in our system, and their relationship.

The ***seventh step*** is to apply the simplification rules explained in the previous section to the current description of the system. However, remember that a simplification rule should not be applied if it results in the deletion of a sanity point. You may also decide not to apply a simplification rule if this generates a learning agent, which has a task that is way more complex than the tasks given to the other agents in the system. If we have several agents in our system and one of them if way more complex, the whole system will not mature until the most complex system matures. We may be better off having a system in which all its agents mature at about the same time.

The ***eighth step*** is to come up with a reward function for each LA. Keep in mind that a reward function does not only indicate what we want the LA to accomplish but also what we want it to avoid doing.

These eight steps can be applied as many times as desired on the system. Once you come up with a system (i.e. a set of learning agents sometimes joined together by sanity points and each one with their own reward function) you need to train it. Training can take place using a simulator or using a real life environment or both. I strongly advice to use a simulator first, not only because that would most likely be more cost effective but also because learning probably

will occur faster. If a given system does not perform as desired, we should apply the eight steps described above one more time. If we suspect that a sensor or an actuator is not really helping with optimizing the task at hand, just remove it and let the system learn again from scratch. If performance does not decrease, this sensor or actuator is most likely redundant.

Perhaps the best way to explain this methodology is by going through an example. In the next chapter, I will illustrate how the theory described in this and previous chapters can be used to implement an artificial cerebellum, which is able to solve a real life problem. In this case, the problem at hand will be coming up with an artificial cerebellum that will allow a front-end loader (a type of mining equipment) to automatically load a hauling truck (another type of mining equipment) in a way as to optimize a set of goals. Among the possible goals would be minimizing the amount of fuel used, avoiding dangerous collisions and minimizing the wearing of its expensive tires by avoiding unnecessary skidding.

4.10 Review

In this chapter I described a new framework that formalizes and facilitates the task of creating customized cerebellums. This methodology tackles the framework usability issue found in previous methods used to develop artificial cerebellums. I believe that this framework could be used to build a development tool that would give people, with little or no expertise in the area of A.I. or Machine Learning, the ability to quickly design and implement customized cerebellums for a wide range of applications. The following chapter provides an example of how this framework can be used to solve a real life problem.

CHAPTER 5: AUTONOMOUS FRONT-END LOADER

5.1 Overview

In the previous chapter I introduced a newly proposed framework to build artificial cerebellums. However, a theoretical framework is sometimes difficult to grasp without putting it to work. This chapter does just that by implementing an artificial cerebellum for a mining piece of equipment named front-end loader. This chapter starts by describing the motivation behind building such an artificial cerebellum. It continues by pointing out the challenges of building this system. After that I provide a quick review of the steps required to build an artificial cerebellum as described by the proposed framework. Then each one of the steps is applied to build the artificial cerebellum. Once this is done I explain how the artificial cerebellum was trained and how the control policy was extracted. I also offer a short discussion related to the advantages and disadvantages of using the real environment versus using a simulator to train the system. After this I show the results observed after training the artificial cerebellum and I offer the possibility to fine-tune the system even more, if so desired.

5.2 Motivation

Mining is one of the world's oldest industries. Minerals extracted from mines allow the making of metals, ceramics, fertilizers, pharmaceuticals, chemicals, electronics and a wide variety of other products [Rom06].

Mining techniques have come a long way since the times of the pick and shovel. Nowadays many mining operations make use of very specialized machinery and sophisticated computer systems.

Among this type of specialized machinery we have hauling trucks, which are able to move close to 400 tons of material per trip. Other specialized machinery called shovels and front-end loaders are used to load these trucks. These shovels and front-end loaders excavate using huge buckets with a capacity of several tens of tons of material. The picture below shows a hauling truck being loaded by a front-end loader.

Figure 5.1. An FEL (Front-End Loader) Loading a Hauling Truck.

Technology has allowed more efficient mining and a huge increase in production. However, in the last few years, the rapid industrialization of China, India and other developing countries has put a tremendous strain on the mining industry.

Many mines have expanded their operations and some old mines, which had been closed because of their low yield, have been reactivated. Unfortunately, the required rise in production has been delayed, mostly because of two reasons: first, the lack of qualified operators for this type of specialized machinery and, second, a worldwide tire shortage.

Learning to efficiently operate this type of machinery often requires several years of training and the current supply of experienced operators does not meet the current demand. Operating these pieces of equipment efficiently and safely can mean yearly savings of thousands and even millions of dollars in some mining operations. Nevertheless, in the last few years, existing inefficiencies have often been overlooked in order to satisfy the demand for materials

such as copper, gold and coal.

Furthermore, the worldwide production of tires for hauling trucks and front-end loaders has also not been able to catch up to the demand, in spite of hundreds of millions of investment in new plants. The demand is such that the price for these four meters tall tires has recently quadrupled to more than $40,000 each. But even at this price, many operations are unable to acquire tires in sufficient numbers. This shortage has caused some operations to stop plans of expansion. Furthermore, in some cases, this shortage has forced companies to scale back. For instance, Fording's coal, in British Columbia, was expected to reduce its yearly production from 28 millions tons to less than 25 millions [Rom06].

Perhaps a partial solution to these issues would be the development of autonomous vehicles. The task of these systems would be to drive this type of specialized machinery as efficiently and safely as very experienced operators. Furthermore, these autonomous systems would have the additional task of driving this type of machinery in a way that minimizes wear and tear of its tires.

A Japanese company called Komatsu already offers a hauling truck autonomous system called Front Runner [Mod10]. To avoid working on the same type of equipment as this system, I decided to focus on front-end loaders, which are used to load these hauling trucks.

5.3 Design Challenges

Designing and implementing this type of systems is neither easy nor quick. Komatsu's Front Runner system, the most advanced autonomous hauling system in the world, has been under constant development for more than 20 years.

One of the reasons why it is difficult and time consuming to develop this type of system

is because of its dependence on a large number of actuators and sensors. All these components need to work in unison and without failure, if the system is to work properly. The failure of one of these sensors or actuators may render the system unusable. To tackle this issue, Komatsu's system has added redundant components and policies. This has made the system more robust to component failure. However, this approach has also increased the complexity of the system. This has made the system not only harder to implement but also to test. There are just way too many factors that can change the system's environment and it has become very difficult if not impossible to test all scenarios.

Furthermore, there are currently no standard methods to design this type of systems. If ten engineers where given the task of designing such an autonomous system, we most likely would end up with ten different designs. This lack of formalization on the design of autonomous mining systems not only makes this a very ambiguous task but also makes it a very difficult one, which may only be tackled by very experienced engineers.

This has some serious implications. For instance, the fact that only a few people in the development team may have the skills and experience to develop and understand the overall architecture of the system increases the overall risk of the project. Because of the ambiguity and complexity associated with the development of such system, bringing in a new design leadership in the middle of the development cycle most likely means also enduring significant design changes and therefore delay the overall delivery of the system. Furthermore, the fewer the people who really understand the system, the lower the quality of the overall system will be. It is simple: many heads are better than one. If only very few of the members of the team are capable of really understanding the system, not only design and implementation will be poorer but the overall testing and deployment of the system will also be less successful.

5.4 Design Formalization

I propose to use the formalization described in Chapter 4 to design mining autonomous systems. I believe that the simplicity of this method will allow more people to be involved in the design process. Having more people involved with the design has several advantages:

For starters, the design will be peer reviewed by more people and, therefore, more design flaws will be detected and fixed early on.

Furthermore, the project of developing autonomous systems will be less risky as it will not depend on just a few members of the team.

The usage of a consistent method to develop autonomous systems will also bring other advantages:

First, it will allow the creation of a development tool that encapsulates this formalization and facilitates even more the design process by automatizing some of its steps.

In addition, the development of consistent autonomous systems will, most likely, lead to the creation of very robust core components which can be reused to facilitate and decrease the cost of developing new autonomous systems in the future, even in other fields different to mining.

5.5 Overview of the Designing Steps

As mentioned in chapter 4, the ***first step*** in the proposed methodology is to become familiar with the system we want to design. Gaining a lot of application specific knowledge is not absolutely necessary. However, it is typically a good idea as such knowledge may allow us to make significant simplifications to the system.

The ***second step*** of the design formalism is to enumerate all the factors we think that may

be significant to the system we have at hand. Examples of significant factors may be things like position, direction and speed of the system. On the other hand, there may be factors such as color, which may have such a little effect on the achievement of the goal of the system that we may as well disregard them. The main goal of this step is to come up with the minimum set of factors that are significant to the system we want to design.

The ***third step*** on the design formalism is to look at the list of factors and categorize them as *actuators*, *sensors* or *both*.

The ***fourth step*** is to decide what sanity points we want to have, if any. The use of sanity points is optional. However, they can make the system more discernible and less complex. Furthermore, they can shorten overall training time since it may be possible to train some components in parallel.

The ***fifth step*** is to find overloaded Learning Agents in the system and break them into simpler and more manageable LAs.

The ***sixth step*** is to use the formalizations described in chapter four to specify each LA in the system.

The ***seventh step*** is to apply the simplification rules described in chapter four to the description of the system. However, remember that a simplification rule should not be applied if it results in the deletion of a sanity point or if the resulting LA becomes way too complex.

The ***eighth step*** is to come up with a reward function for each LA. Keep in mind that a reward function does not only indicate what we want the LA to do but also what we want it to avoid doing.

Perhaps the best way to explain this methodology is by going through an example. The following sections do exactly that.

5.6 Design Step 1 (Familiarize with Task at Hand and its Environment)

As I mentioned before, I decided to use the task of automatizing a front-end Loader (FEL) to illustrate the proposed design formalism.

The main goal of a FEL is digging material from the ground or wall and loading it into a mining truck. Below I show a picture of a front-end loader loading a mining truck.

Figure 5.2. A Front-End Loader (FEL) Loading a Truck from a Different Point of View.

The diagram below describes some of the most important features of this vehicle:

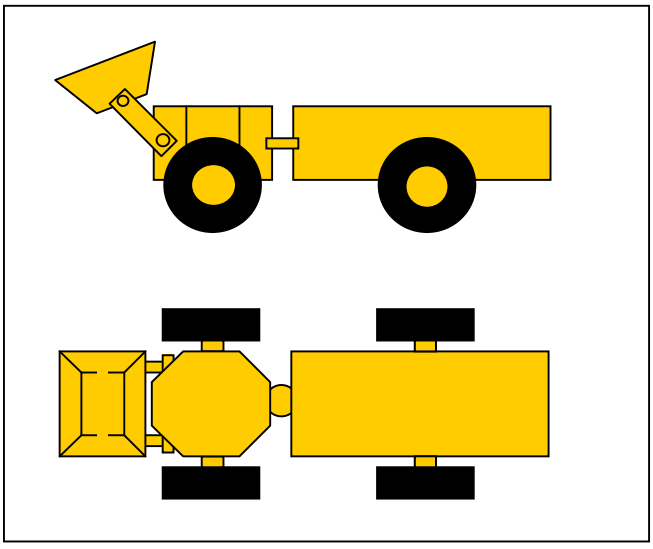

Figure 5.3. Side View and Top View Diagram of a FEL.

The main body of a Frond-End Loader is composed of two parts (shown in blue in the diagram below) and they are kept together by a Joint (in green). This Joint allows them to move independently with one degree of freedom:

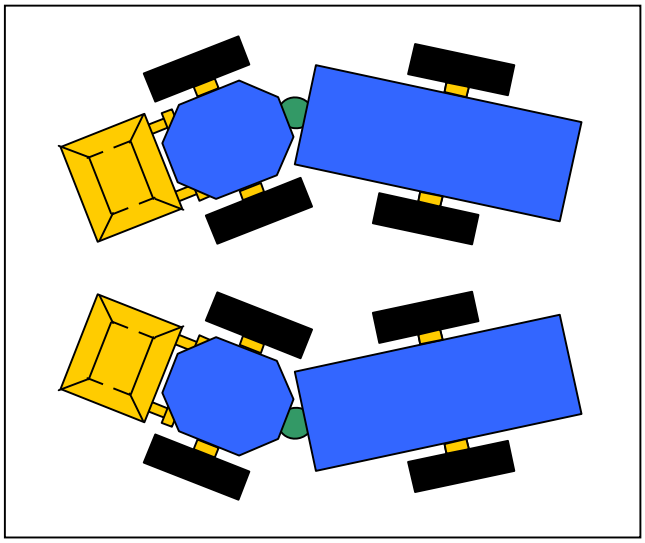

Figure 5.4. The Two Main Body Parts of a FEL and their Pivot Point.

The front wheels (shown in blue in the diagram below) in the front-end loader move freely and the back tires (in red) are the ones that push or pull the vehicle (i.e. this is a back traction vehicle):

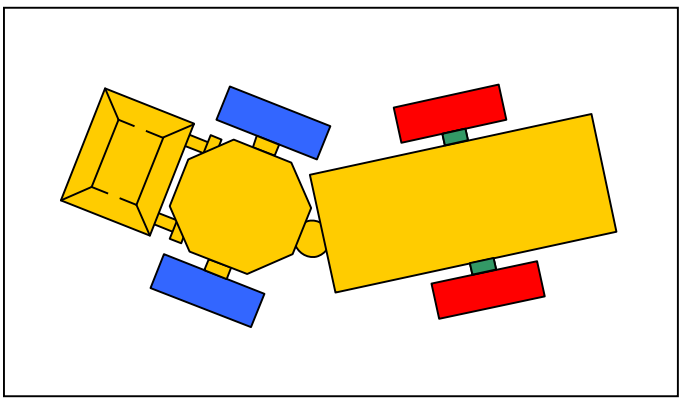

Figure 5.5. Back Traction Front-End Loader (Powered Joints are Shown in Green).

This vehicle has a huge bucket (shown in blue in the diagram below) at its front. This bucket is used to dig material from the ground or a wall, which is then dumped onto a hauling truck.

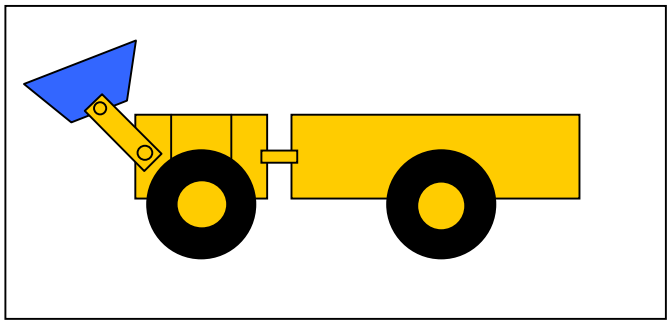

Figure 5.6. Bucket Position in a Front-End Loader.

The bucket is attached to the vehicle by an arm, which has two joints. The first one allows the rising and lowering of the bucket (in blue). The second allows the bucket to be tilted (in red):

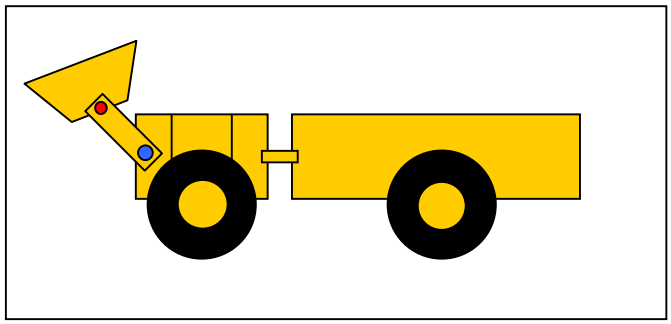

Figure 5.7. Two Actuators are Used to Rise and Tilt the Bucket.

When operated by humans, these equipments are typically driven in a path that resembles a Y as illustrated below:

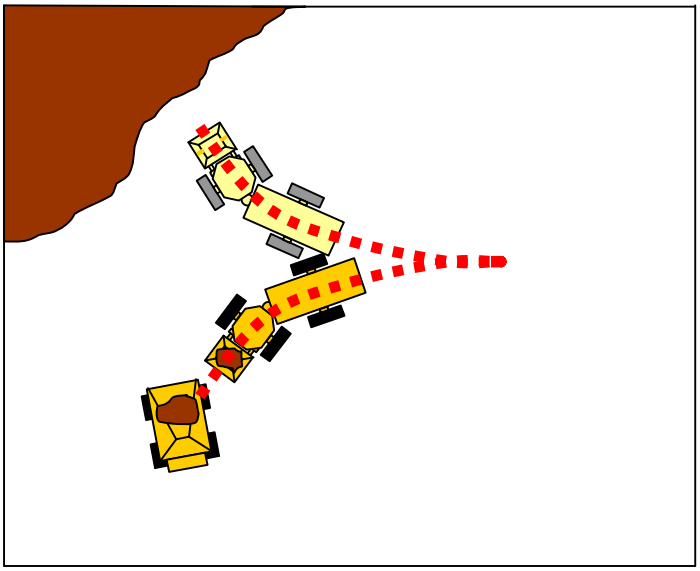

Figure 5.8. Common Driving Path Used by a FEL to Load a Hauling Truck.

Very often, the dirt extracted in a mine is not homogeneous and we have some areas in the mine that are composed of one type of material and some other areas of the mine that are of a different composition. The picture below shows areas with 3 different types of material:

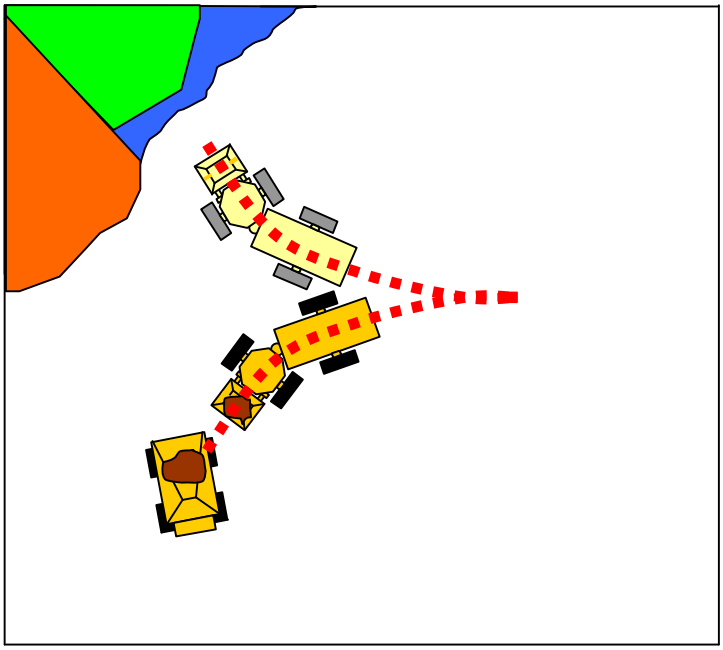

Figure 5.9. FEL is Working with Three Different Materials.

We would like to be able to tell the truck at which point we want to dig at the wall and at what point to dump this material into the truck.

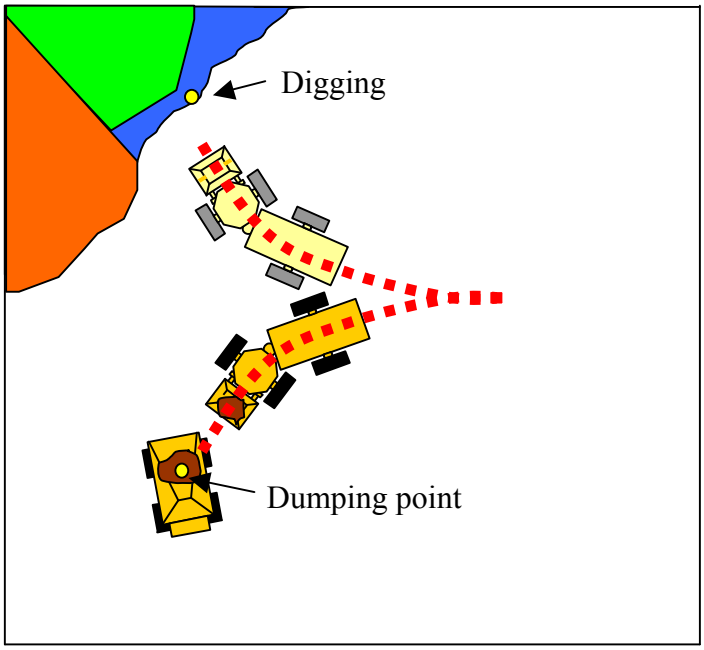

Figure 5.10. Digging and Dumping Points for a FEL.

There are already commercial systems, such as Dispatch [Mod10] and MineOps [Jig10], which can be used to generate the digging and the dumping points. Therefore, there is no need to pass this responsibility to the system we are designing.

However we would like to ask from our system to transport this material in a safe and efficient manner and in a way that minimizes maintenance (such as the wear of its expensive tires). Furthermore, we would like to be able to set priorities on these goals.

For example, most mines may want to have safety as the first priority (e.g. we would never want to ram into a truck with an operator inside). However, depending on the current price of things like gold, copper, fuel and tires, we may want the front-end loader to be operated in a

different way. For instance, if the price of the mineral we are mining is very high and the price of fuel and tires is low, we may want the FEL to be driven in a way that decreases the loading time, even if that means using fuel in a suboptimal way or wearing the tires sooner than necessary. On the other hand if the price of fuel or tires is very high and the price of materials such as gold or copper is very low we may want the system to drive the front-end loader in a way as to optimize the use of fuel and tires, even if that means loading trucks more slowly.

5.7 Design Step 2 (Generate list of significant factors)

Now that we have a better understanding of the system's goal and its environment, we need to come up with a set of related factors. As I mentioned before, you don't have to come up with the perfect set at the first attempt. This methodology is highly iterative and there are plenty of opportunities to refine this list without having to throw away all our work.

For instance, we know that things like the current location of the loader, the equipment's heading, the loader's speed are most likely very important factors to the success of this system.

Furthermore, things like equipment's weight, bucket elevation and tilt may also play very important roles.

Let us enumerate what we have so far:

1) **Loader's position**
2) **Loader's heading**
3) **Loader's speed**
4) **Loader's total weight**
5) **Bucket's elevation**
6) **Bucket's angle**

5.8 Design Step 3 (Generate List of Actuators and Sensors)

Once you have a preliminary set of *factors*, the next step is to try to categorize them as either *actuators* or *sensors* or both (yes, some factors may be associated with both actuators and sensors). Remember that sensors can be categorized as either *static* or *dynamic*.

For instance, in the case of the **loader's position** and **loader's heading** factors we may not only want to know where the loader is located and headed to, but we may also want to control where the loader will be located and headed to in the near future. Therefore, we will need *sensors* related to the equipment position and heading and also *actuators* that allow us to control its position and heading in the near future.

Notice that by taking the human driver out of the equation, the system is no longer able to tell the equipment's location and heading with respect to the mine. Therefore, we will need to add *sensors* not currently available in a standard FEL.

Using ***GPS receivers*** seems to be a practical option. By knowing the position of two points in the equipment (within a few centimeters of accuracy) we can easily calculate its location and its heading. The picture below shows where we could install the two GPS receivers in the front-end loader:

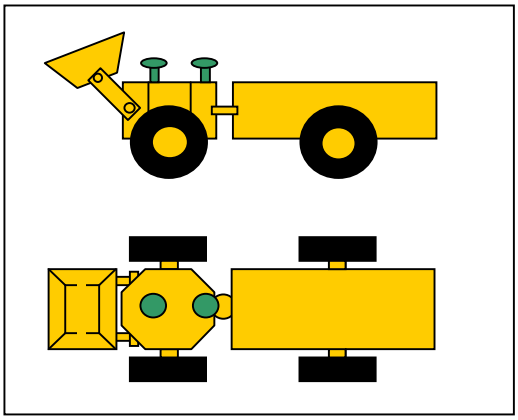

Figure 5.11. FEL with Two GPS Receivers that Allow Calculating Location and Heading.

The picture below shows two different headings of the equipment and how the position of the receivers can be used to calculate the current heading of the equipment:

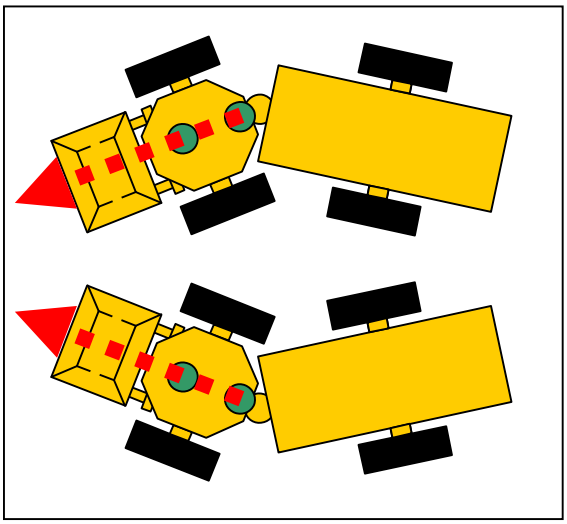

Figure 5.12. Using Two GPS Receivers to Calculate the Current Heading of a FEL.

As I mentioned before, the factors **loader's position** and **loader's heading** seem to also be associated with *actuators*. In order to control the equipment's location and heading in the near future, we probably will be ok with just using the equipment's ***gearbox*** and ***steering system***.

The next factor to analyze is the **loader's speed**. This one can be detected using the *GPS receivers* and can be controlled using the equipment's *gearbox*.

The factor **loader's total weight** would probably not be associated with an actuator since it doesn't seem likely that increasing the equipment's weight dynamically would have any advantage in this system. Perhaps for a submarine it would be beneficial but not for an FEL. However, it seems to be useful to have the total weight of the equipment handy since the vehicle may need to be driven differently depending on its current weight. At this moment we realize that the total weight of the equipment is actually composed of at least three parts: 1) the weight

of the equipment itself, 2) the weight of the fuel it has in its tank and 3) the weight of the material in its bucket, if any (this will require the installation of a sensor that does not come standard on front-end loaders).

Therefore the factor **loader's total weight** seems to be associated with three sensors: 1) a *static sensor* that specifies the weight of the FEL itself (which can be obtained from the vendor of the vehicle), 2) a *dynamic sensor* that detects the weight associated with the amount of fuel this vehicle currently has (the info from the odometer would probably suffice) and 3) a *dynamic sensor* that is able to tell the weight of the material in the FEL's bucket.

Since the system will need to be able to dig material, it is also very important to know and control the elevation and angle of its bucket.

Therefore, the factors **bucket's elevation** and **bucket's inclination** will be associated with *dynamic sensors*, which detect the elevation and inclination of the FEL's bucket. Furthermore, we will have actuators to control the bucket's elevation and inclination.

While the actuators are already part of the standard FEL, we will need to install specialized sensors (i.e. heavy duty inclinometers) to detect the bucket's elevation and inclination.

This means that we end up with the following list of sensors and actuators:

1) **first GPS receiver (dynamic sensor)**
2) **second GPS receiver (dynamic sensor)**
3) **equipment balance (static sensor)**
4) **odometer (dynamic sensor)**
5) **bucket balance (dynamic sensor)**
6) **arm inclinometer (dynamic sensor)**

7) **bucket inclinometer (dynamic sensor)**

8) **gear box (actuator)**

9) **steering system (actuator)**

10) **arm hydraulic system (actuator)**

11) **bucket hydraulic system (actuator)**

5.9 Design Step 4 (Select Sanity Points)

This step is optional but using *sanity points* has some very important advantages: They can make the system safer to use and it may accelerate learning by allowing parallel learning.

For this application I decided to use *sanity points*. One of the requirements for a sanity point is that the signals going through this point can be evaluated independently. At these points we can install alarms that will tell us if the system is no longer reliable. For instance we may want to have a sanity point that compares the current heading to the desired heading. If they vary too much, it probably means that either the *sensors* are broken or the environment has changed too much. In either case, the system is probably no longer fit to continue functioning and requires repairs or time to adapt to the new environment.

Using *sanity points* has more implications that may not be obvious at first. By creating a *sanity point* we are actually breaking our system into two systems. For example, if we assume that our sanity point compares desired heading to actual heading, one of the two systems would have the general task of minimizing the difference between desired heading and actual heading. Then, the other system would assume that the desired delta heading will be achieved, and would just need to indicate the optimal delta heading for the current situation. We would end up with two subsystems that are not only simpler than the original but we also have two systems that can now learn independently of each other.

Notice that we can also create three more sanity points:

a) One that compares desired speed to actual speed

b) One that compares the desired to the actual bucket elevation.

c) One that compares the desired to the actual bucket angle.

If we do this, we end up with a system whose subcomponents have the following responsibilities:

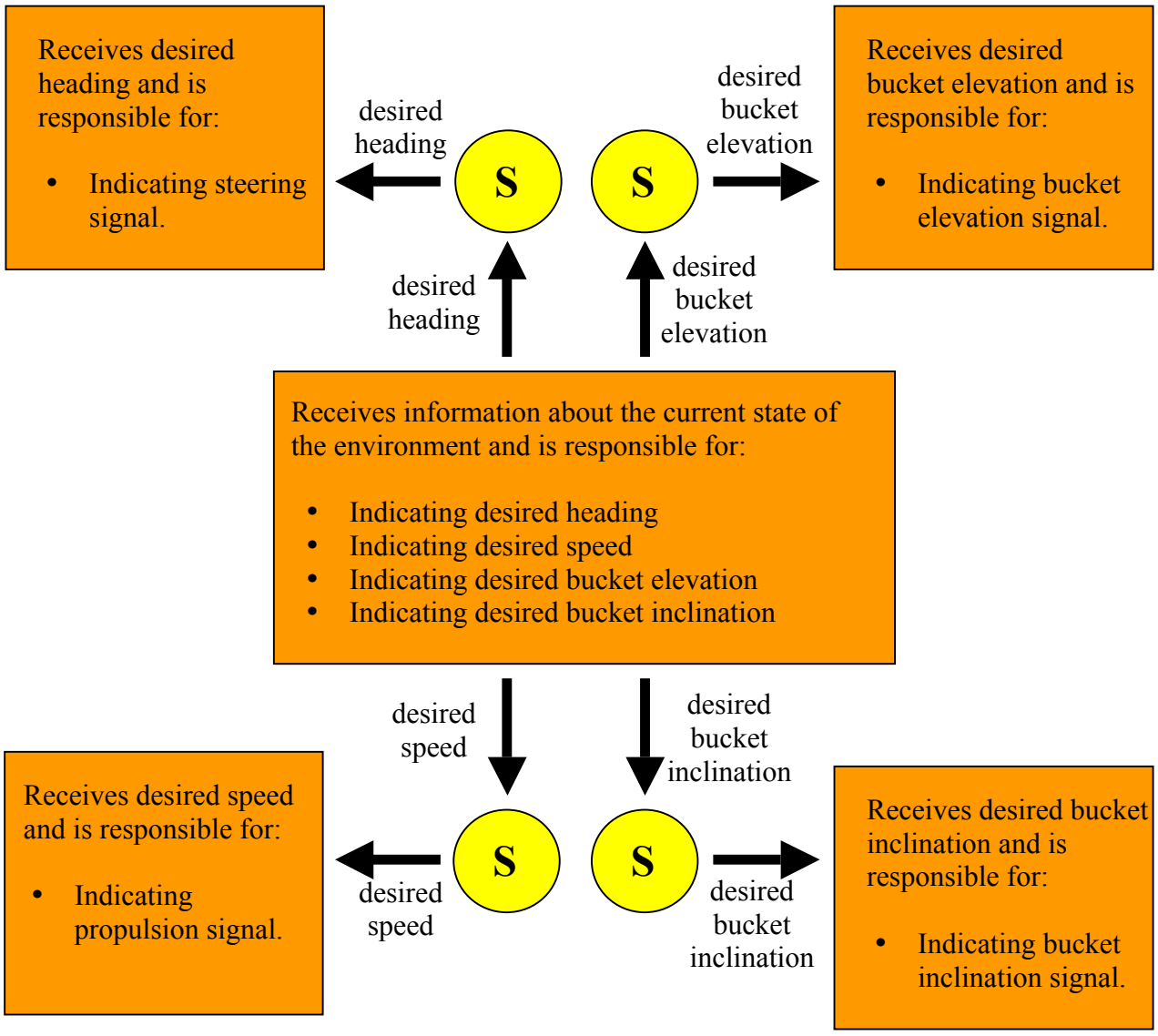

Figure 5.13. Subcomponents of our Autonomous System and their Sanity Points.

5.10 Design Step 5 (Break any overloaded LA into two or more LAs)

The next step is to review the responsibility given to each one of the learning agents. If you think one of them may be overloaded, break this LA into two or more LAs.

In our case, one of the Learning Agents, which was left off after introducing the four sanity points, still seems to be a little overloaded. I decided to break this learning agent into four more learning agents, which give us the following set of learning agents:

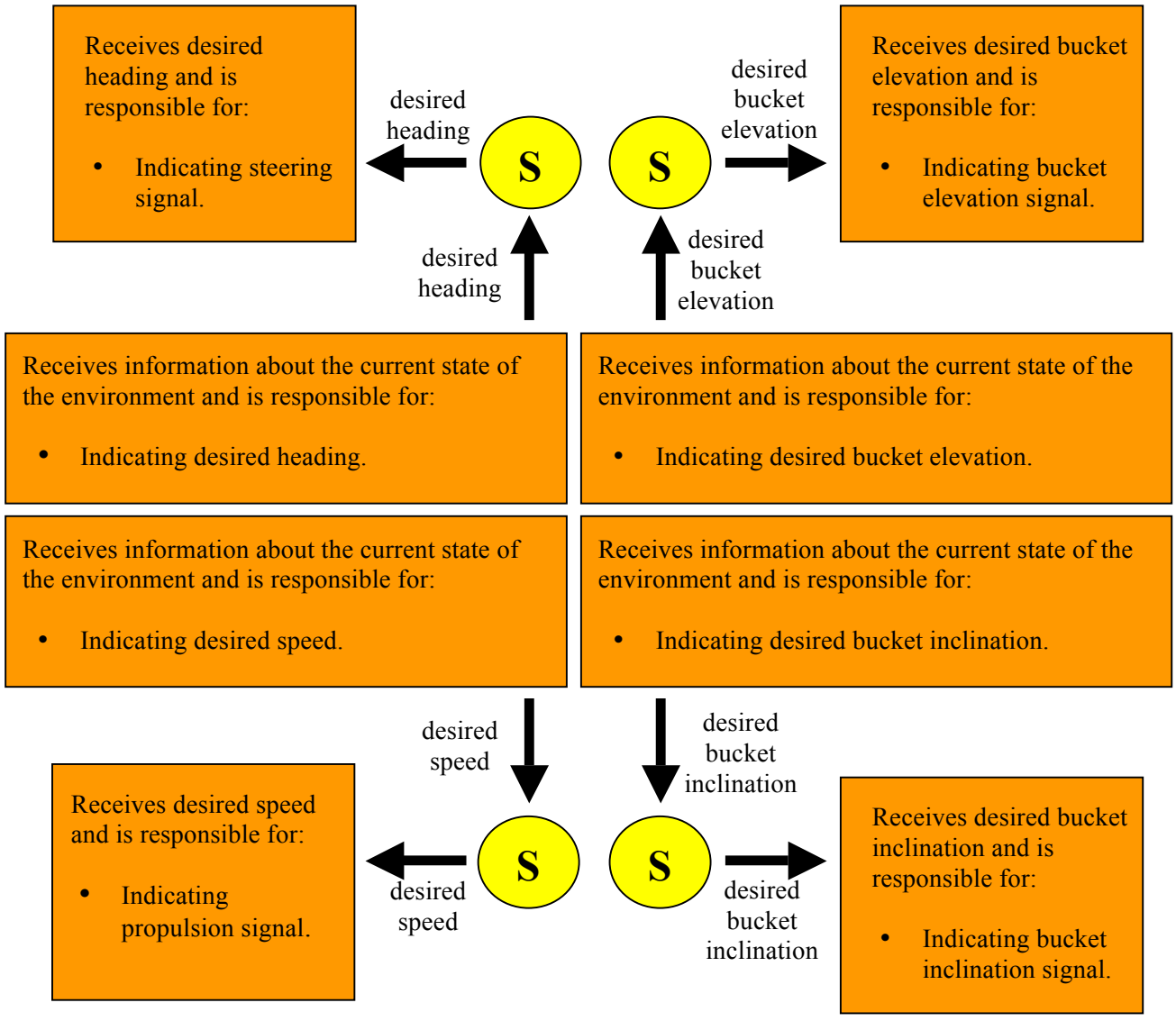

Figure 5.14. Autonomous System after Breaking Overloaded LA into Smaller LAs.

Now that we have defined the *sanity points*, have broken up the system accordingly and have broken overloaded LAs into smaller LAs, we end up with a set of simple learning agents that have very specific goals. Each of these learning agents can be considered to be an independent system and can be trained separately.

5.11 Design Step 6 (Use Formalization to Describe and Specify each LA)

At this point we use the proposed formalization to describe each one of the resulting LAs. The use of this formalization offers the following advantages:

a) Provides a short notation that still expresses important details about the LA such as:
    i) sensors
    ii) actuators
    iii) signal conversions
    iv) sub-learning agents
    v) connections between sub-learning agents
b) This notation provides a bird's view of the system and allows the manipulation of its components and the addition of new components
c) This formalization also provides rules that allow the transformation of the system into a simpler but equivalent system.

This is what each of the resulting sub-systems look like when represented using the proposed formalism:

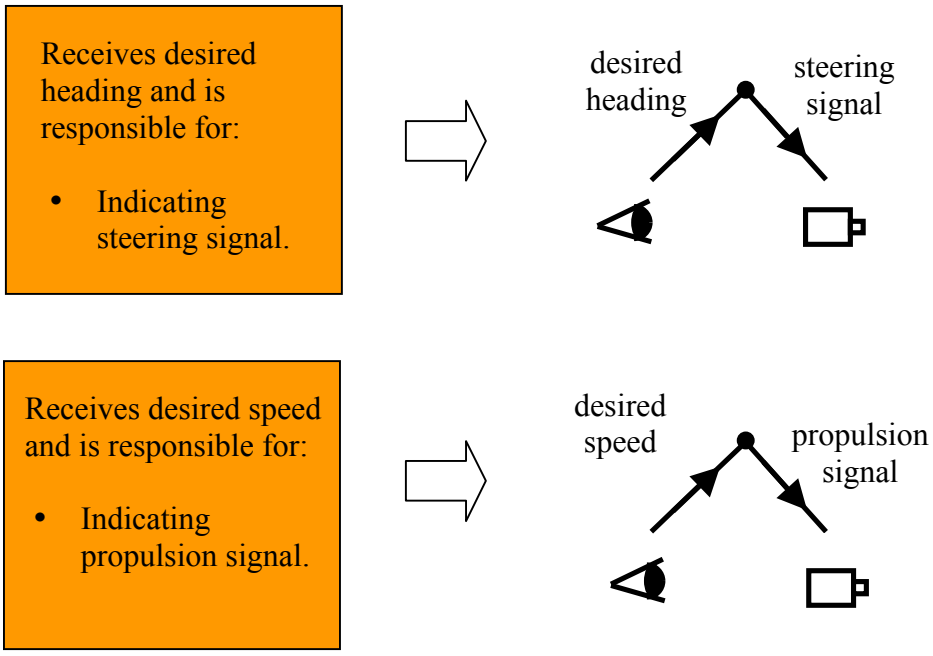

Figure 5.15. 1st and 2nd LA and their Description Using Proposed Formalism.

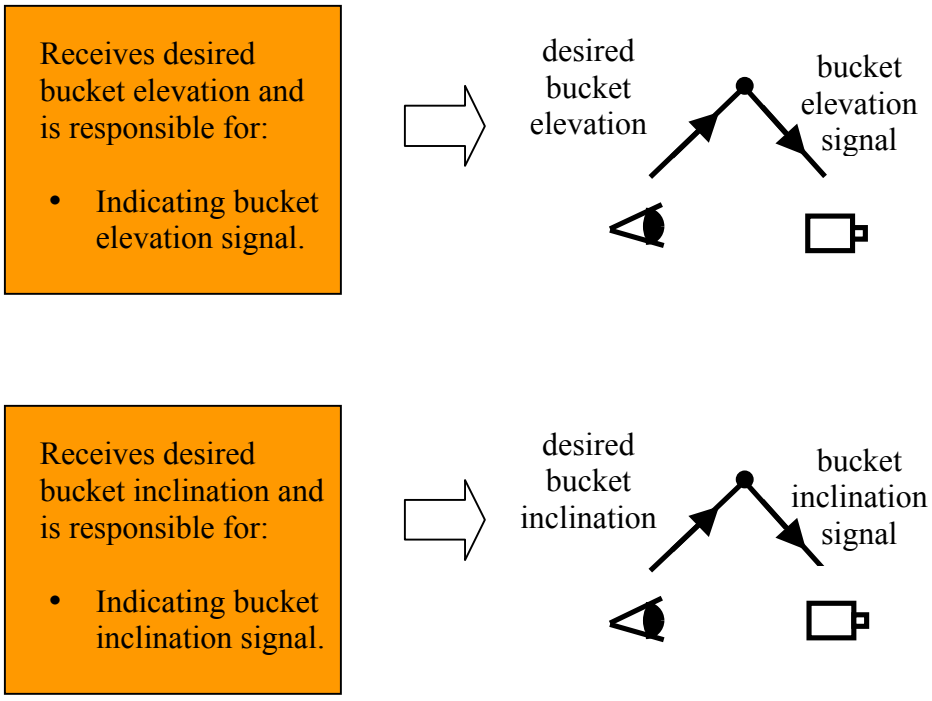

Figure 5.16. 3rd and 4th LA and their Description Using Proposed Formalism.

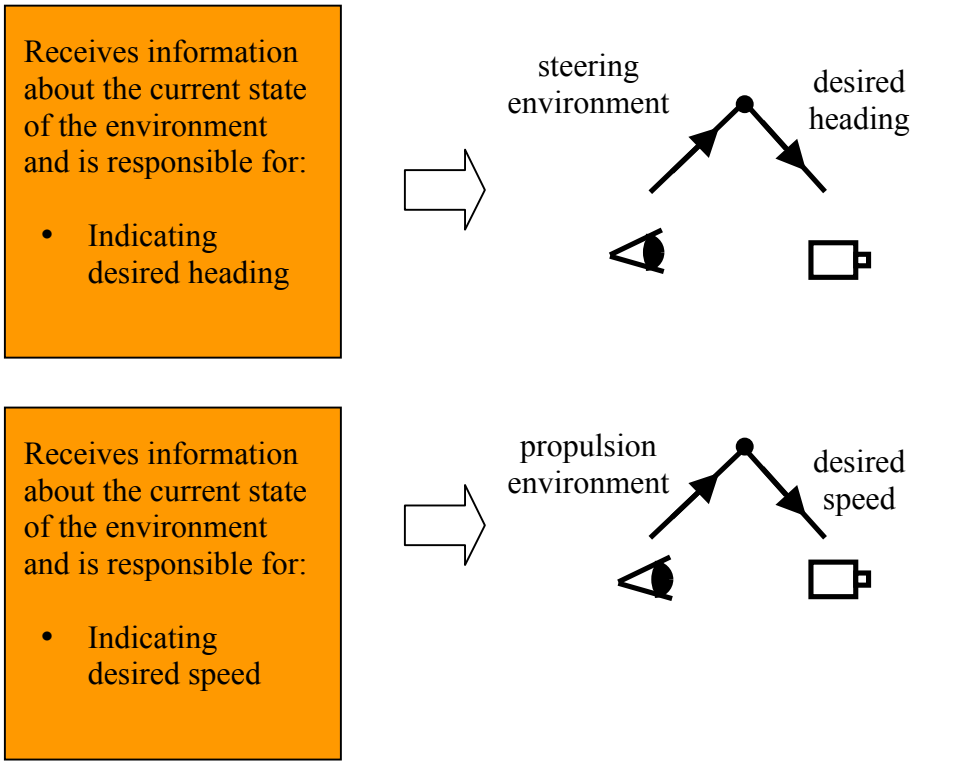

Figure 5.17. 5th and 6th LA and their Description Using Proposed Formalism.

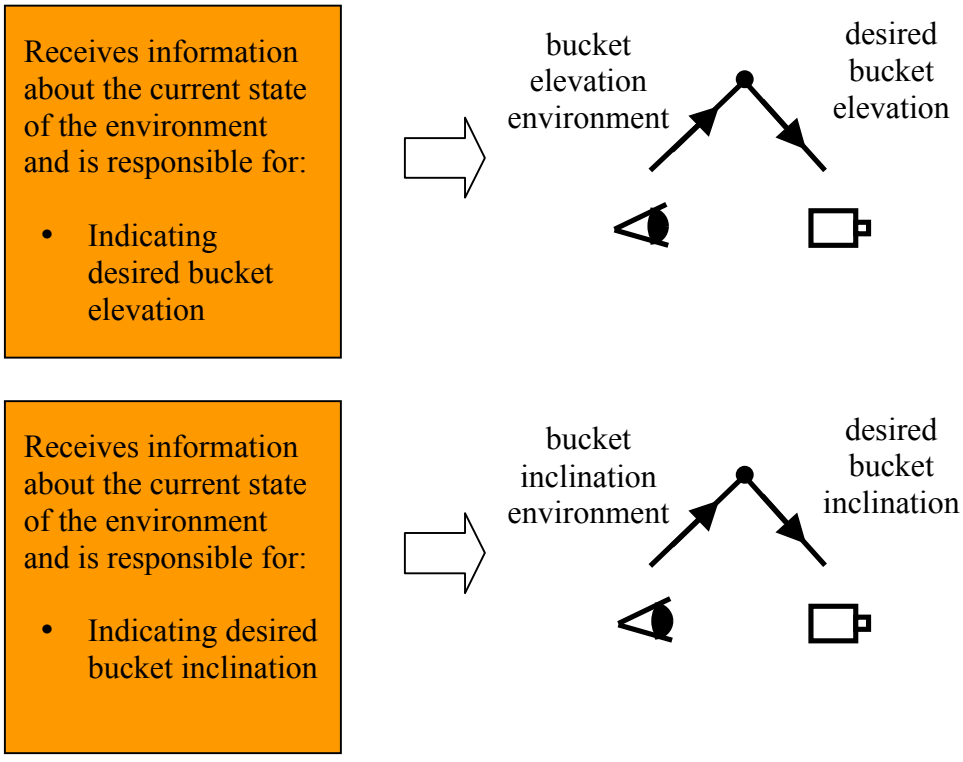

Figure 5.18. 7th and 8th LA and their Description Using Proposed Formalism.

Notice that this is still a very rough draft of our system. There are still plenty of details we need to work out. For instance, we still need to figure out what sensors and what signals we will use to define the state of the environment for several of these learning agents.

Notice that, in this specific application, the existing Learning Agents can be grouped in pairs that share a common signal. For instance, the LA below share the signal *desired heading*:

Figure 5.19. Two Learning Agents Sharing the Signal *Desired Heading*.

The pair below shares the signal *desired speed*:

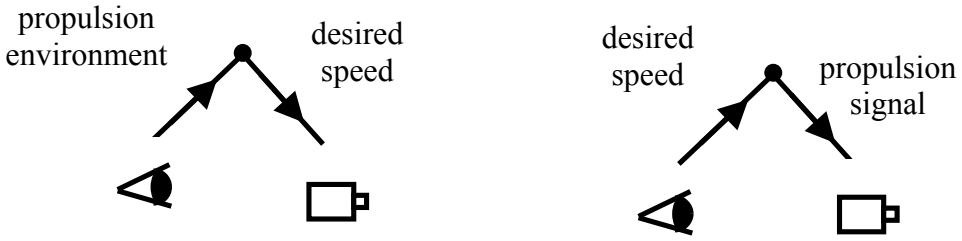

Figure 5.20. Two Learning Agents Sharing the Signal *Desired Speed*.

Another pair shares the signal *desired bucket elevation*:

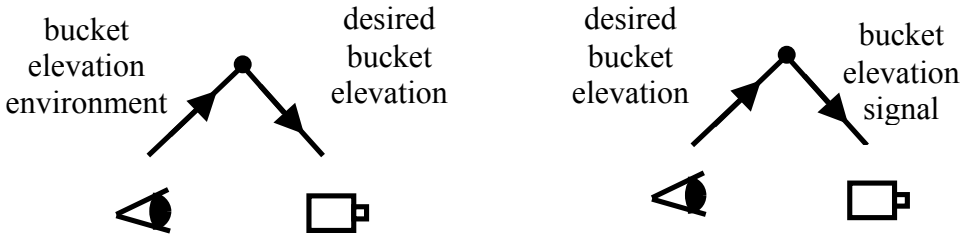

Figure 5.21. Two Learning Agents Sharing the Signal *Desired Bucket Elevation*.

A final pair shares the signal "desired bucket inclination":

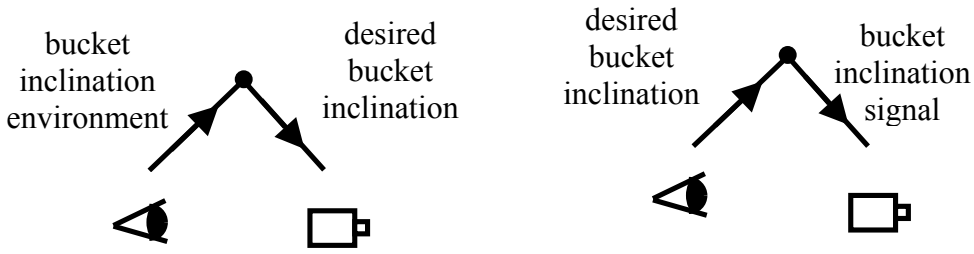

Figure 5.22. Two Learning Agents Sharing the Signal *Desired Bucket Inclination*.

Now we need to notice two things. First, that there is a sanity point between each pair of LAs. Second, that in each pair, there is one LA that is a master LA and another that is an actuator LA. Therefore our system can be represented as follows:

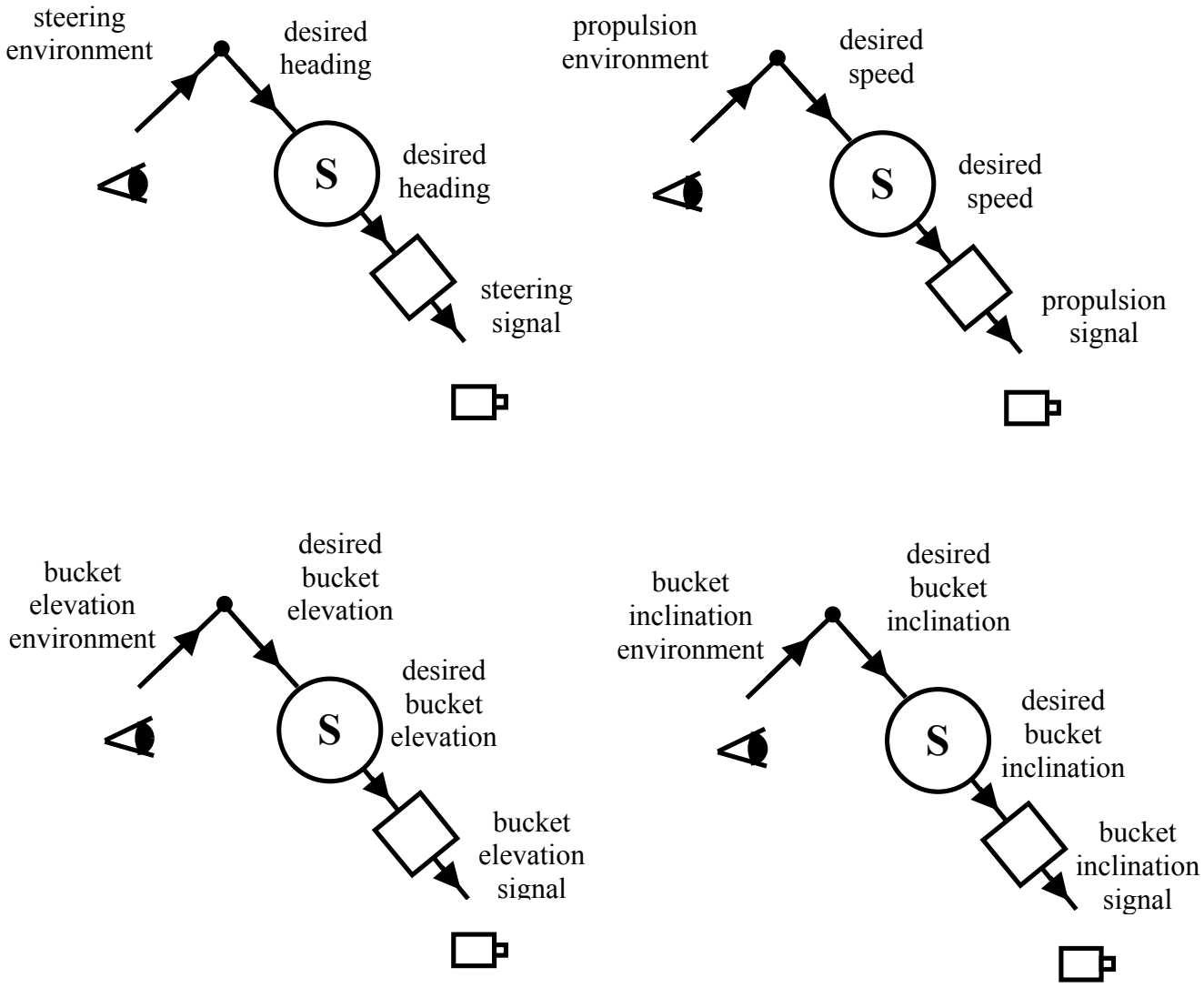

Figure 5.23. The Learning Agents in our Autonomous System.

Notice that, for this application, we ended up with more than one master LA. However, this is not always the case.

Furthermore, notice that we still have abstract sensors and abstract actuators. We now need to map things to actual sensors and actuators.

Let us start with the master LA in charge of controlling the steering of the vehicle:

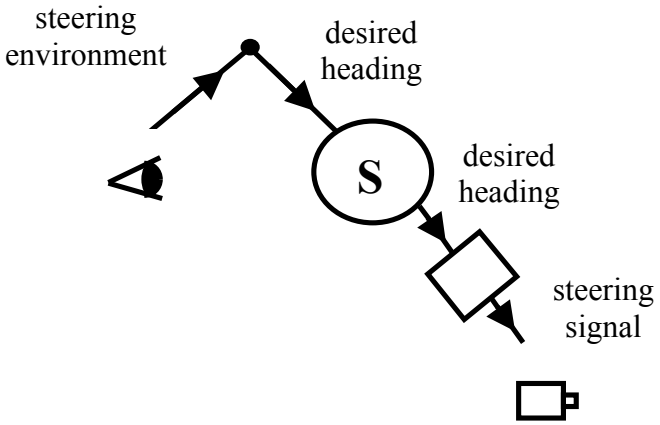

Figure 5.24. The Learning Agents in Charge of Steering the Vehicle.

In this case, we can install something like a stepper motor that controls the steering wheel. All we need to do is to pass this motor a value in the range [-10, 10]. If we pass a value of -10, it means that we want to steer to the left as much as possible. If we pass a value of 10, it means that we want to steer right as much as possible. If we pass a value of 0, it means that we want to keep the steering wheel straight. Now that we have specified this actuator, the right hand side section of our system is updated so that it looks like this:

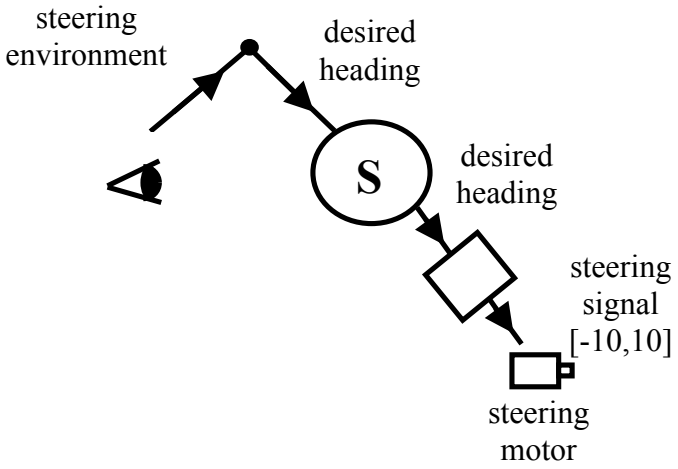

Figure 5.25. Learning Agents in Charge of Steering the Vehicle and Specific Actuator.

Now, let us work on the left hand side section of our system and replace the abstract detector *state of the environment* with specific sensors. The task of this master LA is to come up with an appropriate value for *desired heading* for any possible situation of the environment. Therefore, we now need to think of what factors may differentiate states that are significant to the task of finding the optimal *desired heading*. In this design iteration we believe that factors such as *distance to goal* and *angle to goal's perpendicular* may allow us to distinguish situations in which the *desired heading* value should be different from other situations. Both *distance to goal* and *angle to goal's perpendicular* can be generated from the GPS receivers and the optimization application. The GPS receivers provide the position of the first and second antennas. The optimization application provides the current *goal position*. Therefore, we now end up with the following representation of this section of our system:

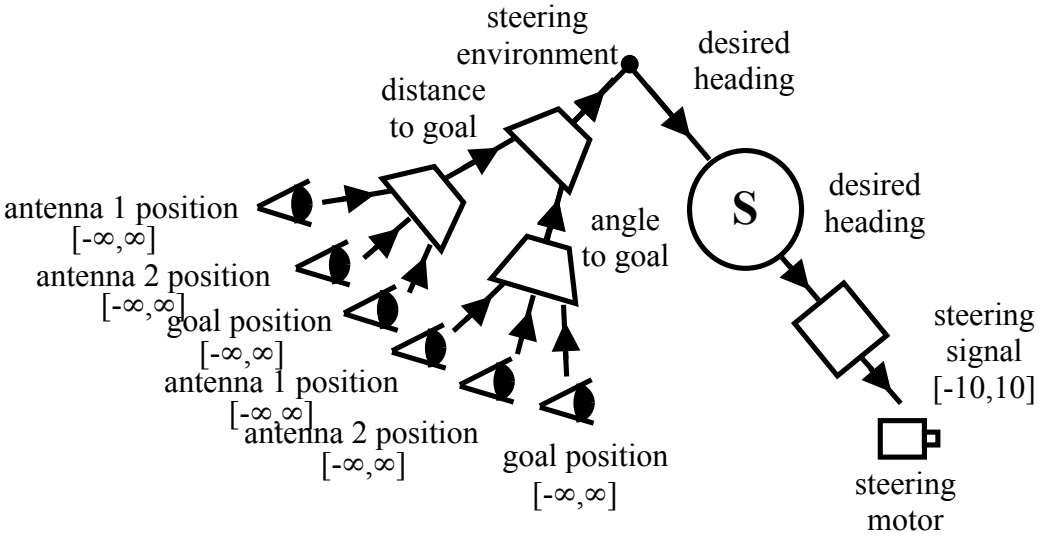

Figure 5.26. Learning Agents in Charge of Steering the Vehicle and Specific Sensors

Finally, we need to describe the functions used on the *distance to goal* and *angle to goal* encoders. Furthermore, we need to explain the details associated with the *desired heading* signal.

With respect to the implementation of the *distance to goal* encoder, we decided that it is the responsibility of this encoder to take the position of the two antennas and extrapolate the position of the front bucket of the equipment. Once this position is known the encoder should calculate the distance to the known goal position. Finally, this encoder should map this distance to one of the possible values for the *distance to goal* signal. In this specific case, we decided to limit the *distance to goal* signal to the range [0, 99]. The picture below describes this mapping:

If distance between bucket and goal is in this range (in meters)	*distance to goal* signal
[0,1)	0
[1,2)	1
[2,3)	2
[3,4)	3
.	.
[98,99)	98
[99, ∞)	99

Figure 5.27. Map between Actual Distance to Goal and *distance to goal* Signal.

The *angle to goal* encoder also calculates the position of the bucket. However, this encoder is a little more complex than the *distance to goal* encoder. The *angle to goal* encoder is responsible for finding two pieces of information: 1) the Heading to Goal angle and 2) the Heading to Desired Entrance angle. Once these two pieces of information are gathered, they are mapped to the signals *heading to goal angle* and *heading to desired entrance angle*. Finally these two signals are encoded into a single signal called *angle to goal*.

To calculate the Heading to Goal angle, the encoder uses the position of the two antennas to calculate the position of the bucket and heading of the equipment. Once this information is

known the encoder calculates the angle difference between the current heading of the equipment and the line connecting the bucket and the goal. The picture below shows examples of how this angle is calculated:

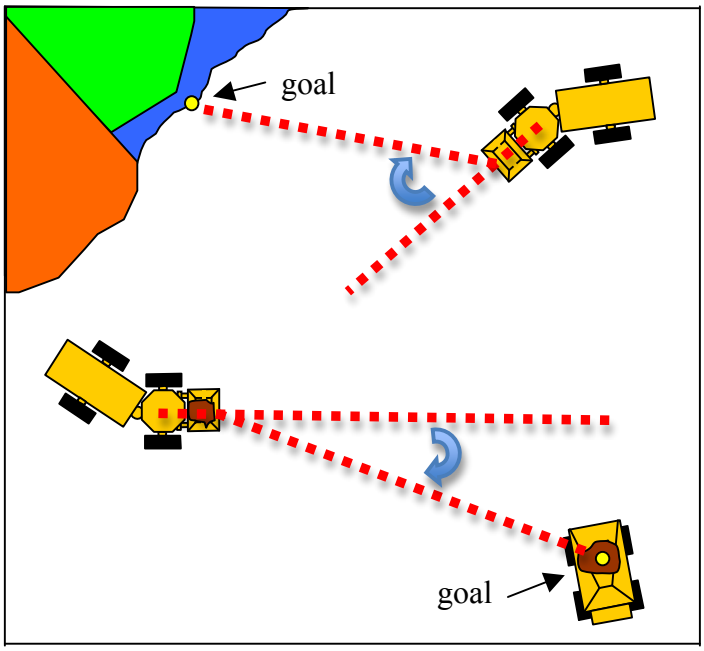

Figure 5.28. Examples of Heading to Goal Angle Calculations

To calculate the Heading to Desired Entrance angle, the encoder also uses the position of the two antennas to calculate the position of the bucket and heading of the equipment. Once this information is known the encoder calculates the angle difference between the current heading of the equipment and the desired entrance angle provided by the mining optimization program. The picture below shows examples of how this angle is calculated:

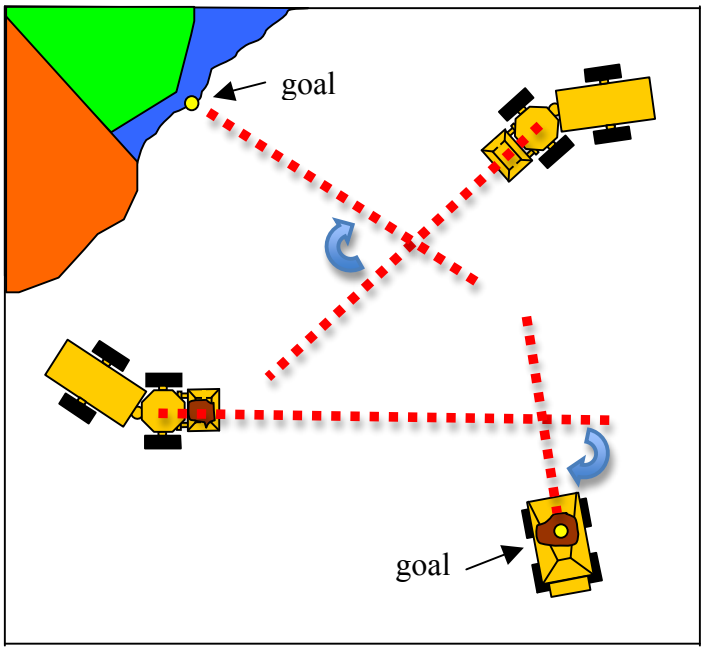

Figure 5.29. Examples of Heading to Desired Entrance Angle Calculations

I decided to limit the signals *heading to goal angle* and the *heading to desired entrance angle* to the range [0,14]. The two tables below describe the mapping between the Heading to Goal angle and Heading to Desired Entrance angle to the signals *heading to goal angle* and *heading to desired entrance angle* respectively:

If angle between heading line and bucket to goal line is in this range (in degrees)	*heading to goal angle* signal
[-180,-39)	0
[-39,-33)	1
.	.
[-3,3)	7
.	.
[33,39)	13
[39, 180)	14

Figure 5.30. Map between Heading to Goal Angle and *heading to goal angle* Signal.

If angle between heading line and desired entrance line is in this range (in degrees)	*heading to desired entrance angle* signal
[-180,-39)	0
[-39,-33)	1
.	.
[-3,3)	7
.	.
[33,39)	13
[39, 180)	14

Figure 5.31. Map between Heading to Desired Entrance Angle and its Related Signal.

Since we have 15 possible values for the *heading to goal angle* signal and 15 possible values for the *heading to desired entrance signal*, the *angle to goal* signal will have a total of 225 different values.

Finally, I decided to limit the desired heading signal to the range [0,9]. The table below describes the mapping between a desired heading angle and one of the possible values of the *desired heading* signal:

If the desired heading angle is in this range (in degrees)	*desired heading* signal
[-180,-16)	0
[-16,-12)	1
.	.
[-4,0)	4
.	.
[12,16)	8
[16, 180)	9

Figure 5.32. Map between Angle Equipment Should Achieve and *desired heading* Signal.

After this specification we end up with the following subsystem:

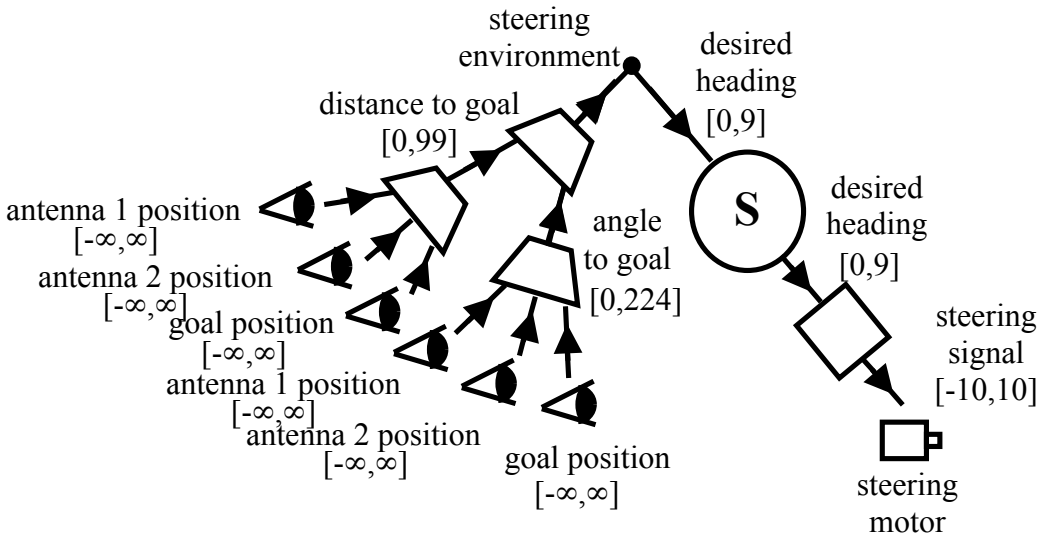

Figure 5.33. LAs in Charge of Steering the Vehicle and Description of Some Signals

Since the *distance to goal* signal has 100 possible values and the *angle to goal* signal has 225 possible values, the *steering environment* signal will automatically have 22500 possible values. After this realization, the subsystem looks like this:

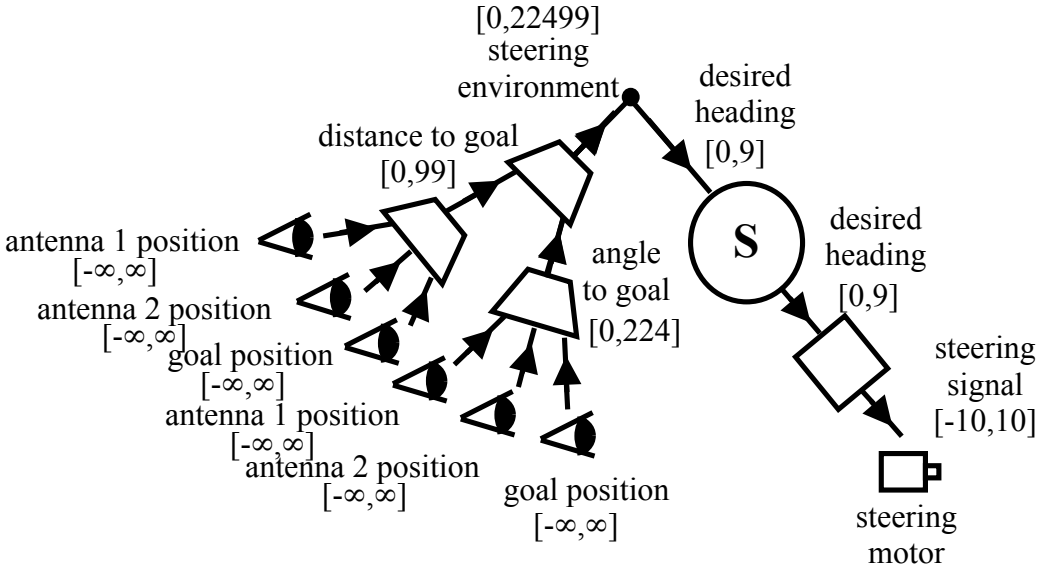

Figure 5.34. LAs in Charge of Steering the Vehicle and Description of All Signals.

Now let us consider the set of LAs that are in charge of propelling the vehicle:

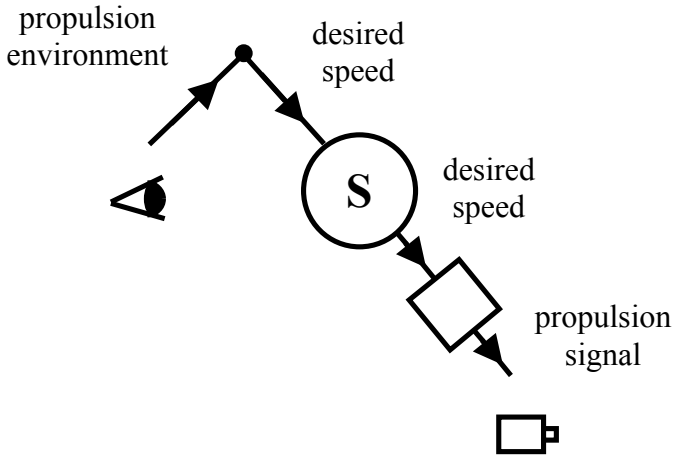

Figure 5.35. Learning Agents in Charge of Propelling the Vehicle.

At this point we could do something similar to what we did to the set of LAs in charge of steering the vehicle as shown below:

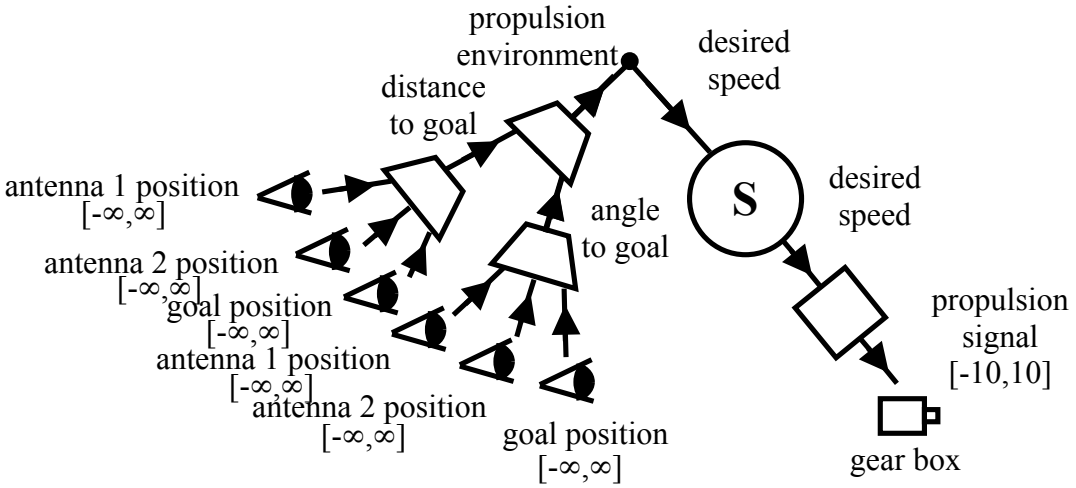

Figure 5.36. LAs in Charge of Propelling the Vehicle and its Sensors and Actuators

In this case, the actuator is the *gear box* which receives a signal with values in the range [-10, 10], where negative values represent deceleration and positive values represent acceleration. The more negative the value is, the harder the gear box will decelerate. The more positive, the more the gear box will accelerate the vehicle.

For this subsystem, the signals *distance* to *goal* and *angle to goal* have the same meaning and mapping as in the previous subsystem. So far, our subsystem looks like this:

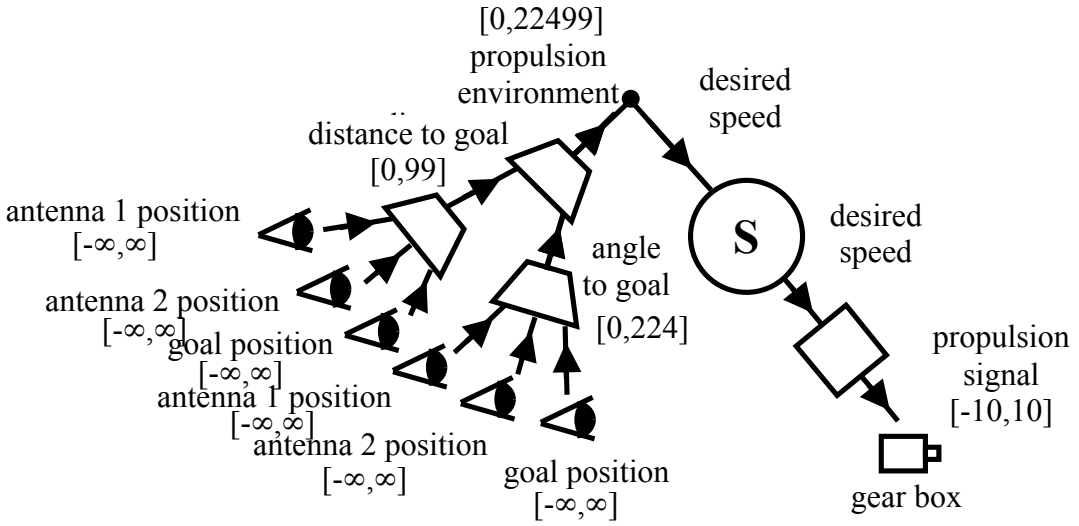

Figure 5.37. LAs in Charge of Propelling the Vehicle and Description of Some Signals.

In this case, the desired speed signal is limited to the range [0, 9]. The table below describes the mapping between a desired speed and the *desired speed* signal:

If the desired speed is in this range (in km/h)	*desired speed* signal
[0,5)	0
[5,10)	1
.	.
[20,25)	4
.	.
[40,45)	8
[45, ∞)	9

Figure 5.38. Map between Speed Equipment Should Achieve and *desired speed* Signal.

After describing the desired speed signal, our subsystem looks like this:

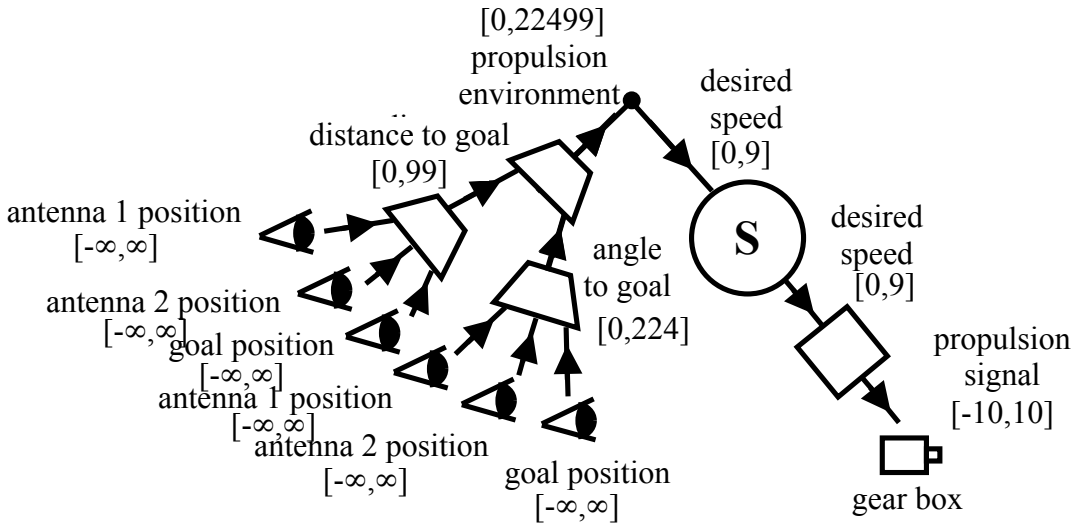

Figure 5.39. LAs in Charge of Propelling the Vehicle and Description of All Signals.

In the same way, we could take the abstract description of the following master LA:

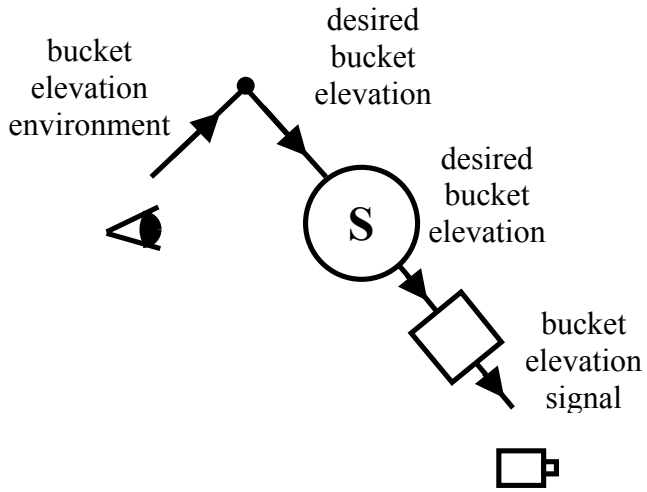

Figure 5.40. Learning Agents in Charge of Controlling the Bucket's Elevation.

The *bucket elevation hydraulic system* accepts values in the range [-10,10]. Passing -10 means we want the lower the bucket as fast as possible. Passing a 10 means we want to rise the bucket as fast as possible. Passing 0 means we want to keep the bucket where it is. Furthermore, we believe the desired bucket elevation should be based on distance and angle to goal. So far, our subsystem looks like this:

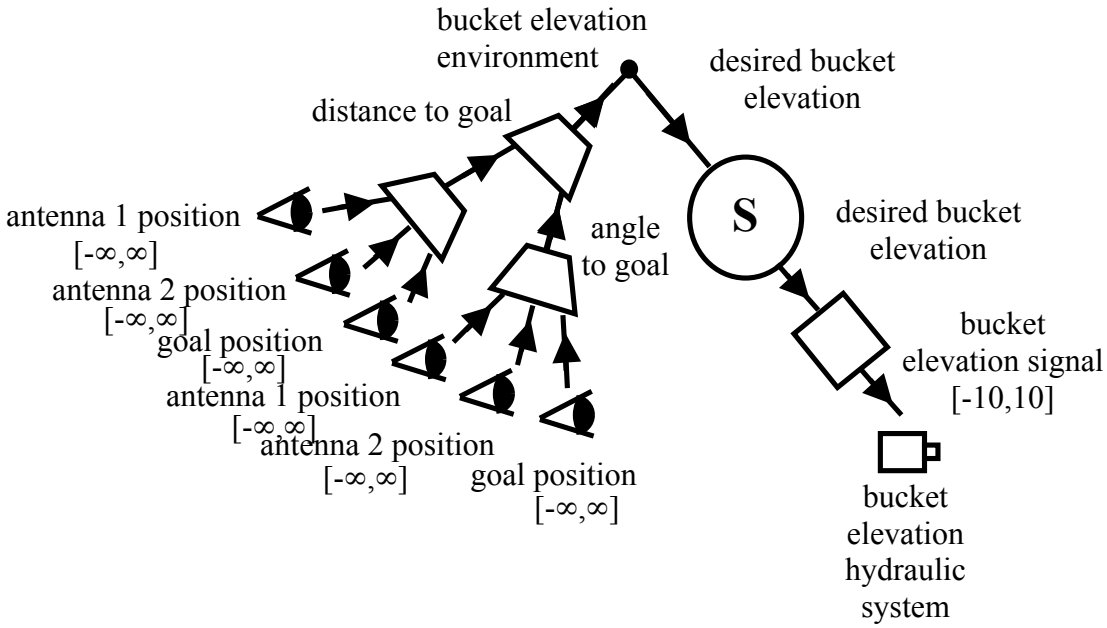

Figure 5.41. LAs Controlling the Bucket's Elevation and their Sensors and Actuators.

I decided to have the same definition for the signals *distance to goal* and *angle to goal*. The *desired bucket elevation* signal is given the range [0, 10]. The table below describes the mapping between the bucket elevation the equipment should reach and the *desired bucket elevation* signal:

If the desired bucket elevation from the ground is in this range (in meters)	*desired speed* signal
[0,1)	0
[1,2)	1
.	.
[4,5)	4
.	.
[8,9)	8
[9, ∞)	9

Figure 5.42. Map between Desired Bucket Elevation and *desired bucket elevation* Signal.

After describing these signals, our subsystem looks like this:

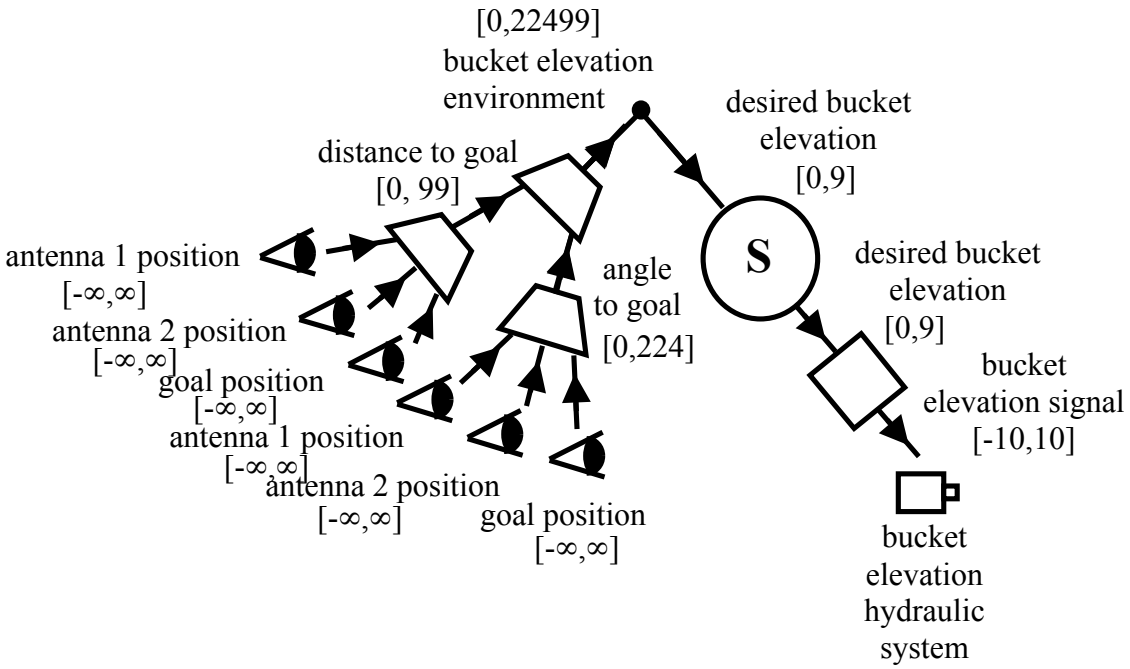

Figure 5.43. LAs Controlling the Bucket's Elevation and Description of All Signals.

Finally we take a look at the abstract definition:

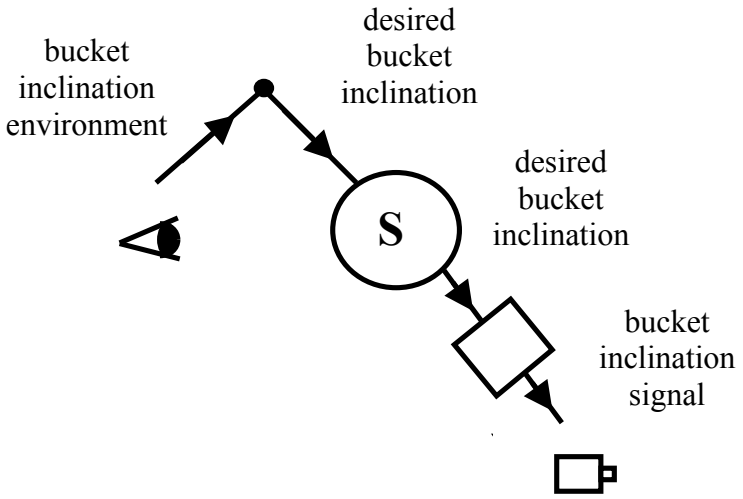

Figure 5.44. Learning Agents in Charge of Controlling the Bucket's Inclination.

The *bucket inclination hydraulic system* accepts values in the range [-10,10]. Passing -10 means we want the bucket to point downwards as much as possible. Passing a 10 means we want the bucket to point upwards as much as possible. Furthermore, we decide to use the *distance to goal* and *angle to goal* signals to specify the bucket inclination environment. After making these decisions the subsystem looks like this:

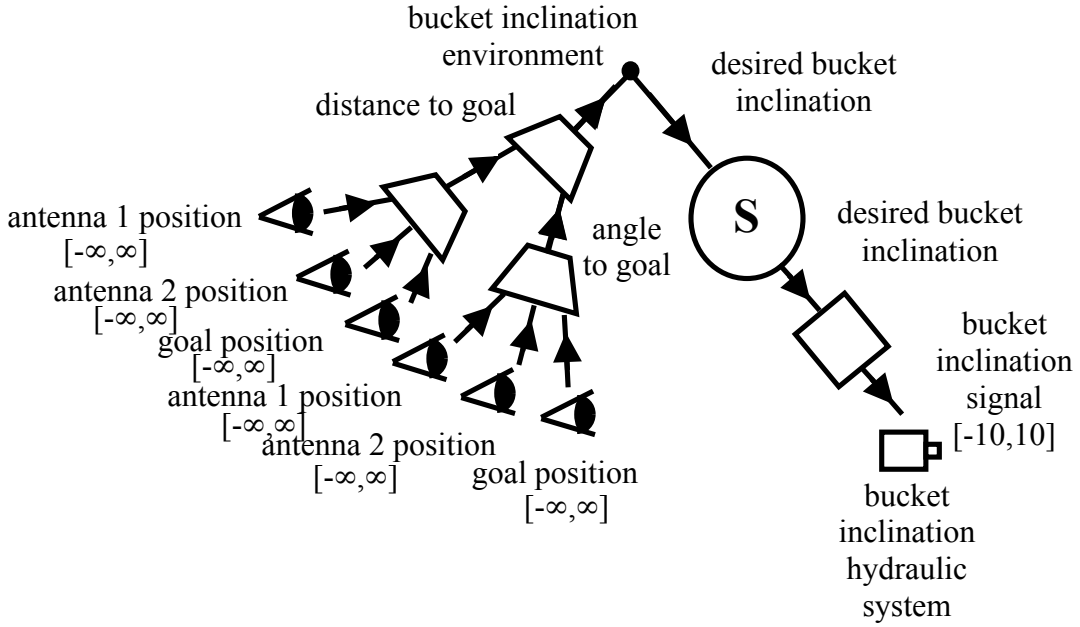

Figure 5.45. LAs Controlling the Bucket's Inclination and their Sensors and Actuators.

I now decide to map the signals *distance to goal* and *angle to goal* the same way we have done in the previous subsystems. After this we end up with a subsystem that looks like this:

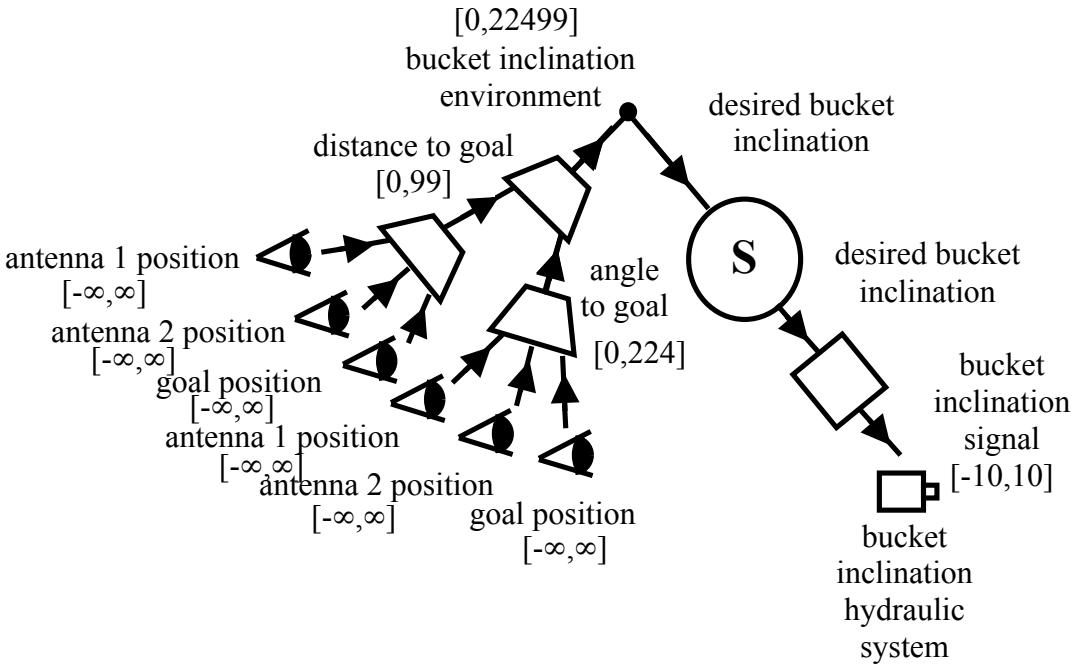

Figure 5.46. LAs Controlling the Bucket's Inclination and Description of Some Signals.

Finally, I decide to limit the *desired bucket inclination* signal to the range [0,9]. The table below describes the mapping associated with this signal:

If the desired bucket inclination angle is in this range (in degrees)	*desired bucket inclination* signal
[-90,-40)	0
[-40,-30)	1
.	.
[-10,0)	4
.	.
[30,40)	8
[40, 90]	9

Figure 5.47. Map between Angle Bucket Should Achieve and *desired inclination* Signal.

After the description of the desired bucket inclination signal we end up with a subsystem that looks like this:

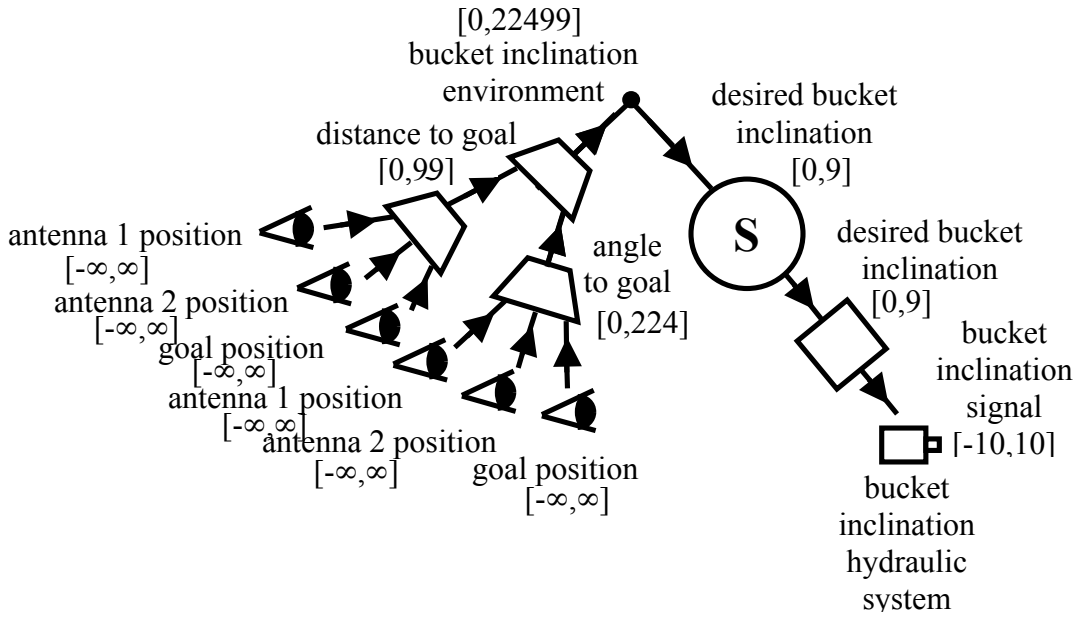

Figure 5.48. LAs Controlling the Bucket's Inclination and Description of All Signals.

After this process, we end up with an autonomous system that has four subsystems. One of the advantages of representing our system in this short notation is that we can easily browse it and pinpoint deficiencies. For instance, let us look at the following subsystem:

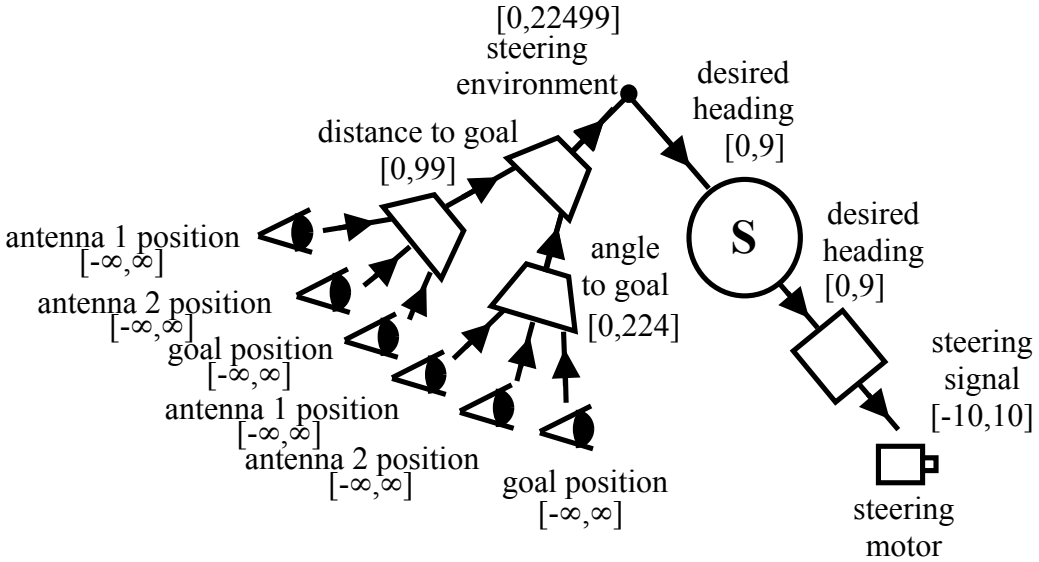

Figure 5.49. Subsystem in Charge of Controlling the Steering in our FEL.

One thing we can probably improve on is the way we calculate the steering signal. Right now we calculate the steering signal based only on the desired heading, but should we apply the same steering signal if the vehicle is driving 2 km/h or if it is driving 60 km/h? I believe we should not. For instance, it may be very dangerous to steer too hard if we are driving very fast but it would be ok if we were driving slowly. Also, the way we should steer may depend on how loaded we are. Perhaps if the vehicle is heavily loaded, we should not steer too hard. Finally, the actuator LA is asked to determine the appropriate steering signal given the desired heading but so far it is not provided with the current heading. We should provide this piece of information too. With these considerations, we should probably modify this section of the system so that it looks like this:

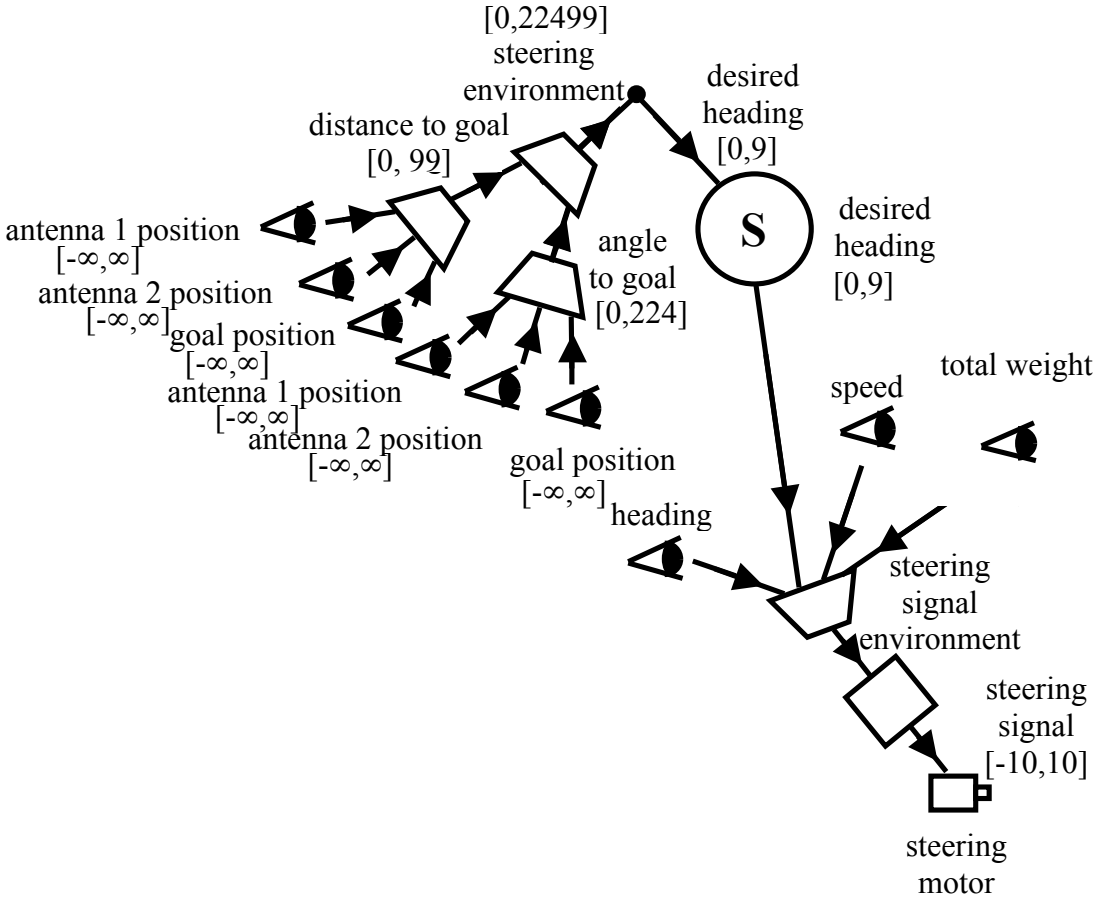

Figure 5.50. Improved Subsystem in Charge of Controlling the Steering in our FEL.

Notice that the description of total weight is still abstract. If we specify this abstract sensor we end up with:

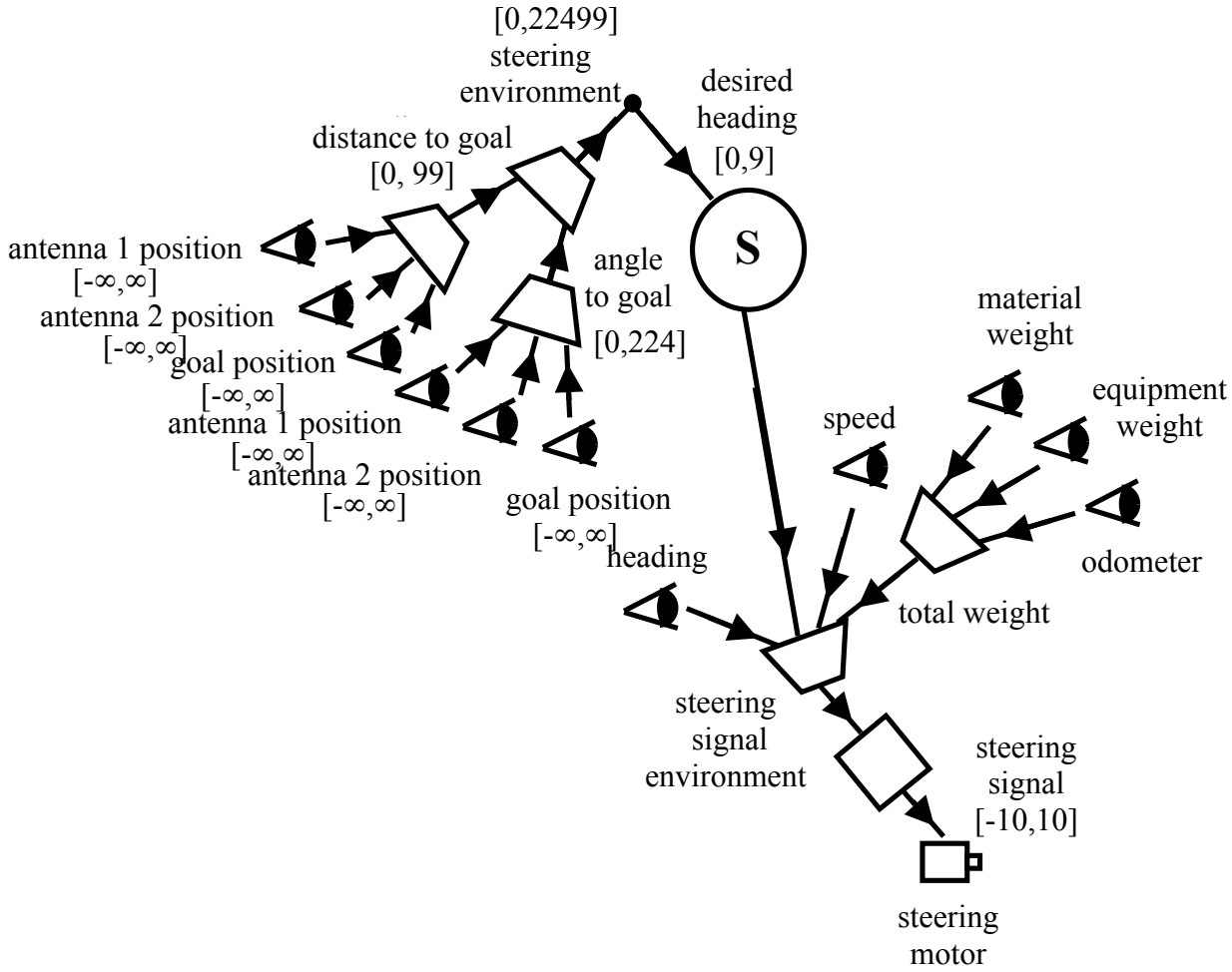

Figure 5.51. Improved Subsystem after Specifying the Total Weight's Sensors.

I decided to limit the *material weight* signal to the range [0,4]. The table below shows the mapping between the weight of the material in the bucket and the *material weight* signal:

If the weight of the material in the bucket is in this range (in tons)	*material weight* signal
[0,1)	0
[1,2)	1
.	.
[4,∞)	4

Figure 5.52. Map between Weight of Material in Bucket and *material weight* Signal.

Furthermore, I decided to limit the *odometer* signal to the range [0,4]. The table below shows the mapping between the weight of the diesel on the tank of the equipment and the *odometer* signal:

If the weight of the fuel is in this range (in tons)	*odometer* signal
[0.0,0.2)	0
[0.2,0.4)	1
.	.
[0.8,∞)	4

Figure 5.53. Map between Weight of Fuel in Tank and *odometer* Signal.

Finally, since the weight of the equipment does not change much during its working life, we assume the *equipment weight* signal stays the same. The table below shows the mapping between the weight of the equipment and the *equipment weight* signal:

The weight of the equipment is (in tons)	*equipment weight* signal
10	0

Figure 5.54. Map between Equipment Weight and *equipment weight* Signal.

After describing these three signals we have a subsystem that looks like this:

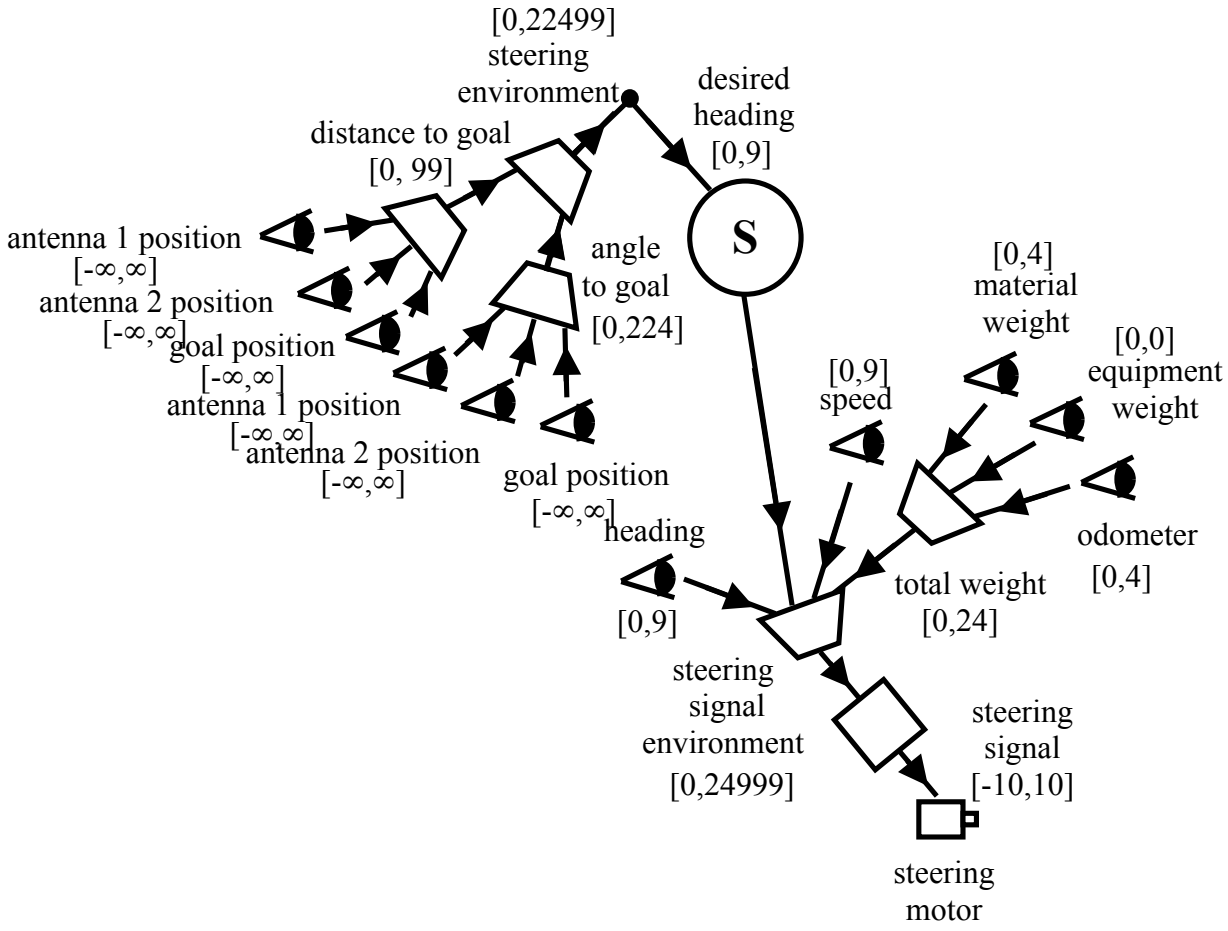

Figure 5.55. Improved Subsystem after Specifying All Signals.

The learning agent in charge of controlling the propulsion system should probably also take into account the current speed, steering angle and weight of the vehicle. For instance, if we are already going very fast we may not want to accelerate anymore. If the steering angle is too steep, we may not want to drive the vehicle too fast. If we are loaded we may need to accelerate more in order to accomplish the desired speed. If we are driving very fast we will need to break harder than if we are driving slowly. Furthermore, we may want to break harder if we are heavily loaded and not so hard if we have nothing in the bucket. After adding these sensors to the system we would end up with something like this:

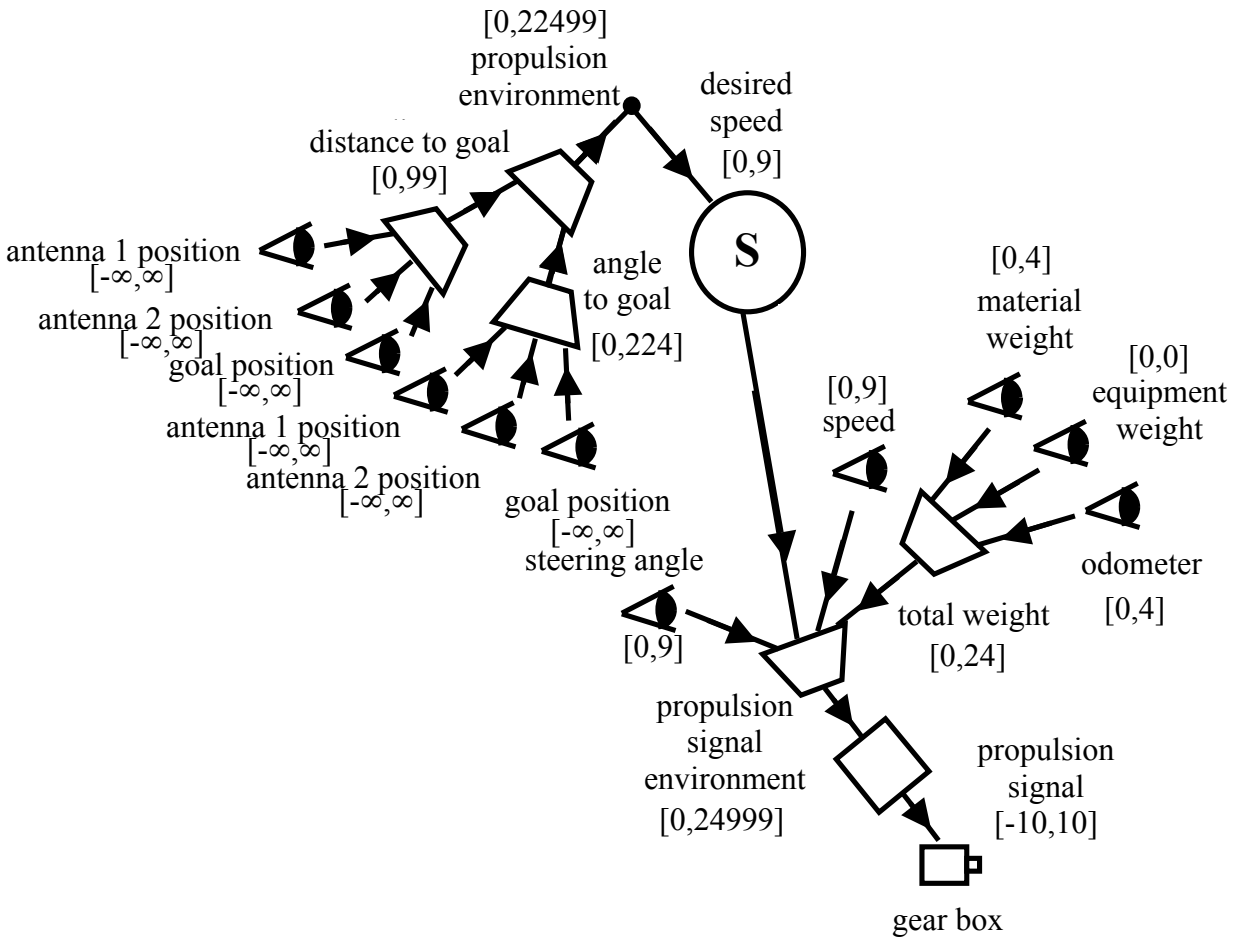

Figure 5.56. Improved Subsystem in Charge of Propelling the Vehicle.

Furthermore, the subsystem in charge of controlling the bucket elevation should also take into account the speed and weight of the vehicle. For example, we may not want to elevate the bucket if we are driving very fast since this may negatively influence the gravity point of the vehicle. Additionally, we may not want to rise the bucket very fast if it is carrying material that is almost as heavy as the equipment itself but it may be ok to raise it fast if the bucket is empty. If we consider these sensors, we end up with something like this:

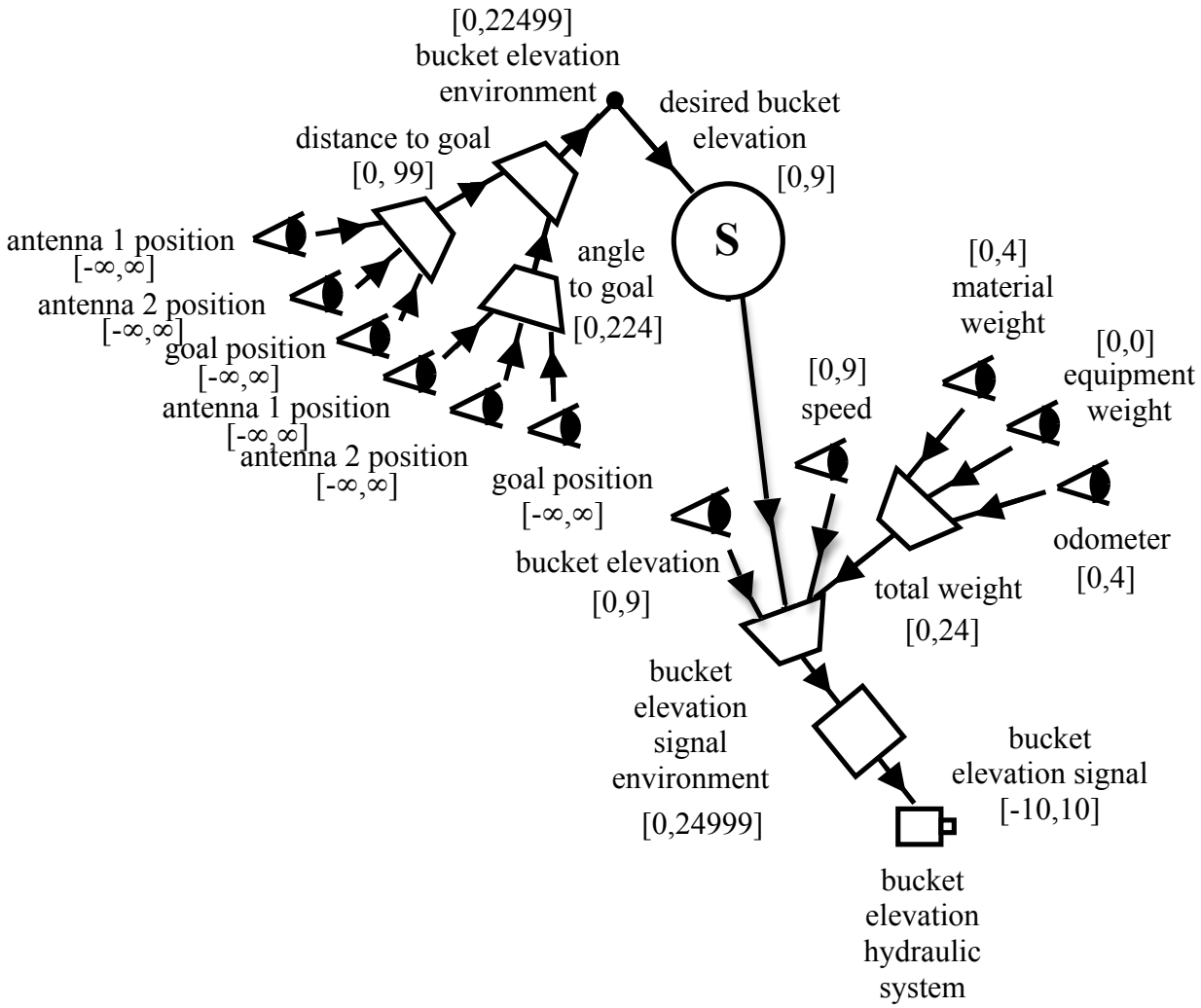

Figure 5.57. Improved Subsystem in Charge of Controlling the Bucket's Elevation.

Finally, the subsystem in charge of controlling the bucket's inclination should not only consider its position with respect to the goal, but also the weight of the material in its bucket. It is very possible that different signals will need to be used to achieve the same inclination depending on the current weight of the bucket.

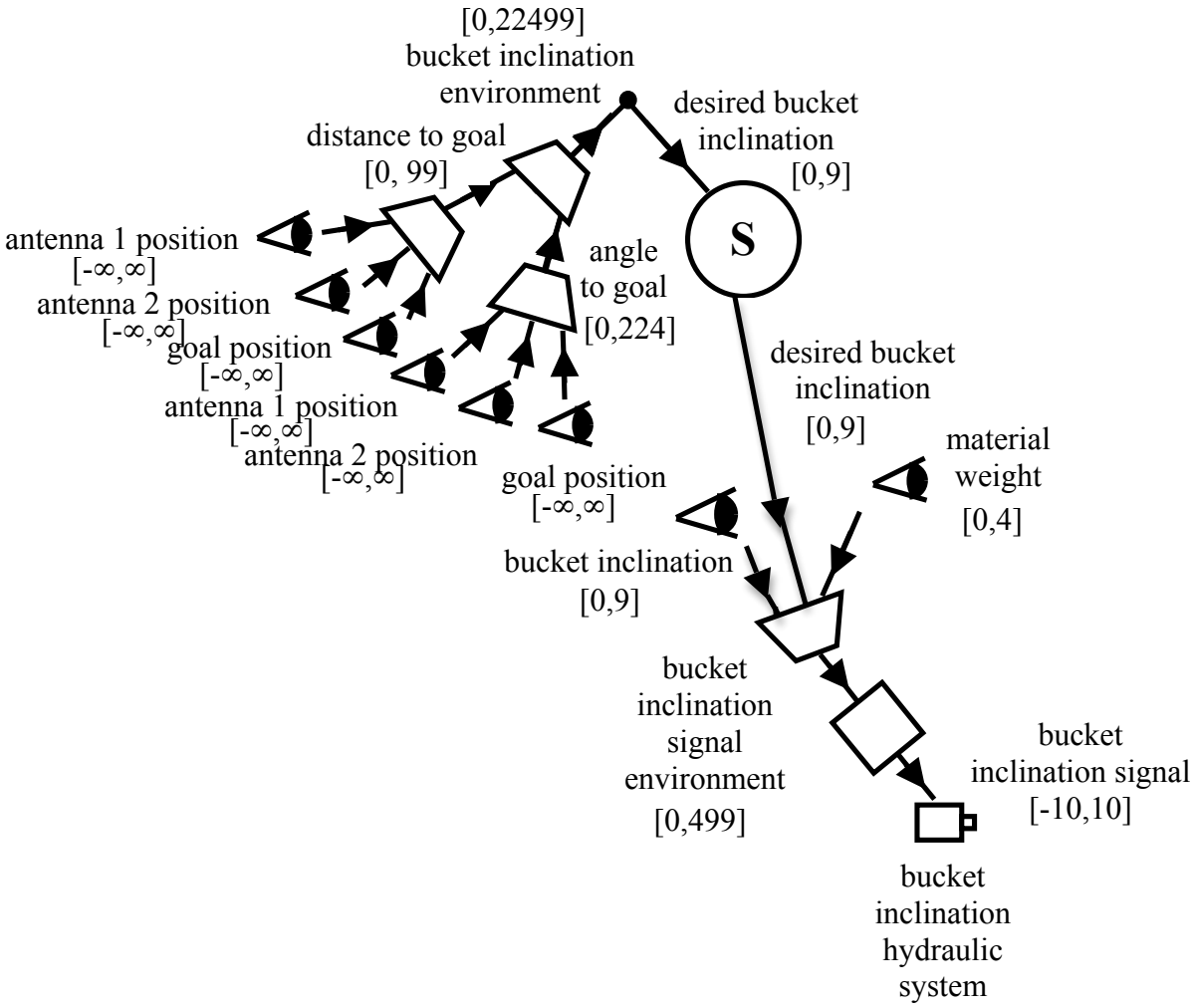

Figure 5.58. Improved Subsystem in Charge of Controlling the Bucket's Inclination.

Notice that we do not know, nor do we need to know, the exact implication of factors such as speed or weight on the control of the vehicle. This is the beauty of using learning agents. They will figure this out by themselves.

5.12 Design Step 7 (Apply Simplification Rules)

The seventh step is to use the simplification rules on the resulting system. Before we start, we need to remember that a simplification rule should not be applied if it results in the

deletion of a sanity point or if any of the resulting LAs is way too large to be handle by a single computer. In this specific application what we can simplify is some of our decoders.

After applying the simplification rules, our four subsystems would look like this:

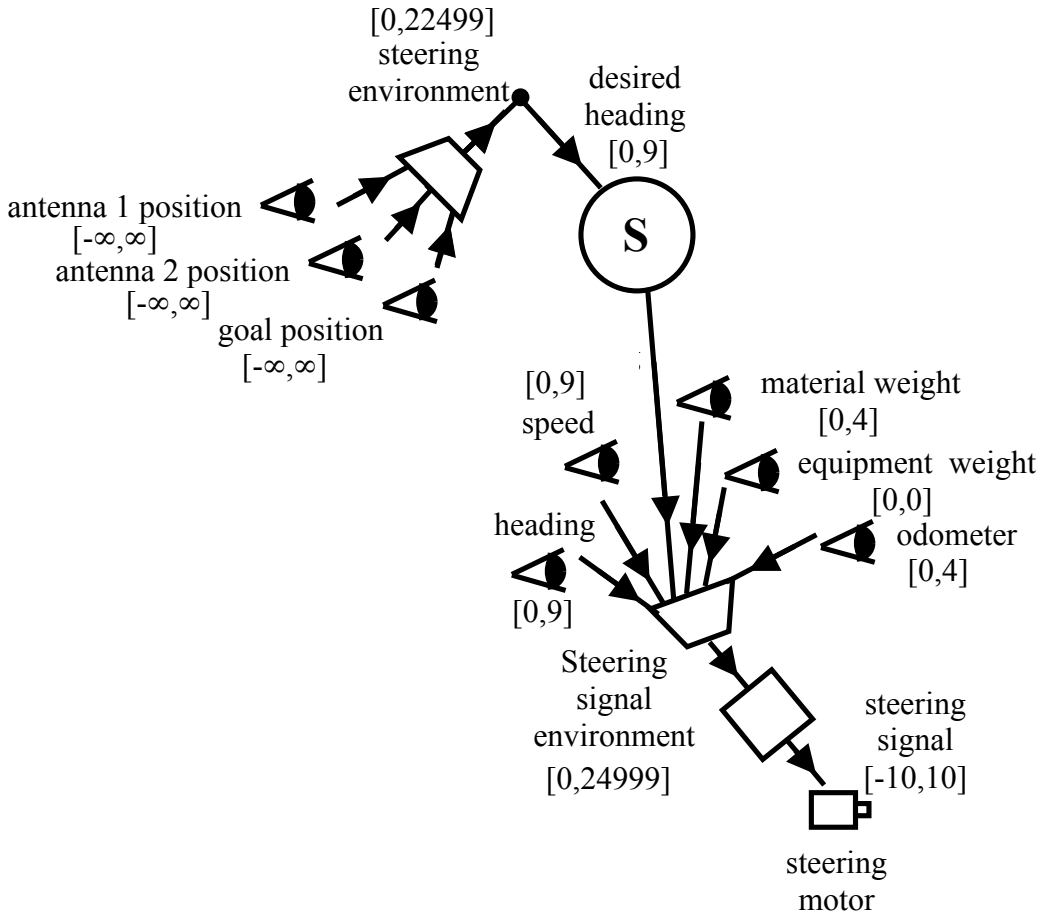

Figure 5.59. First Simplified Subsystem (Out of Four).

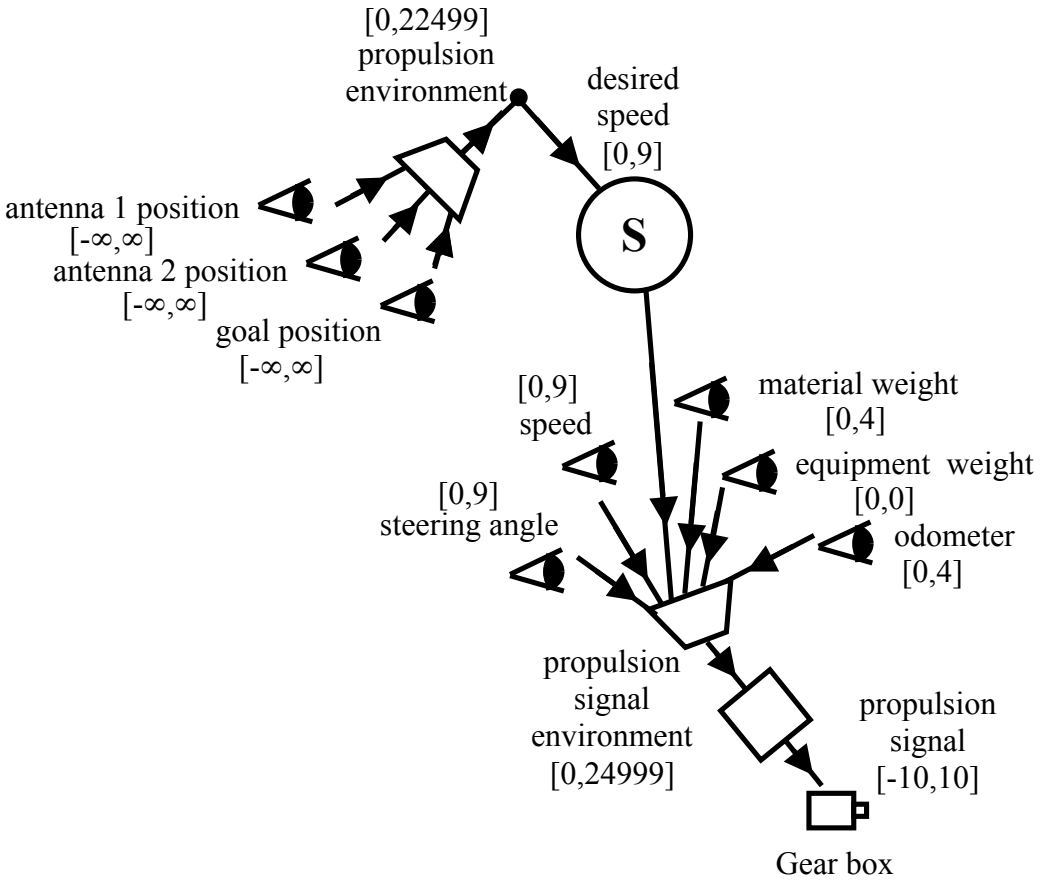

Figure 5.60. Second Simplified Subsystem (Out of Four).

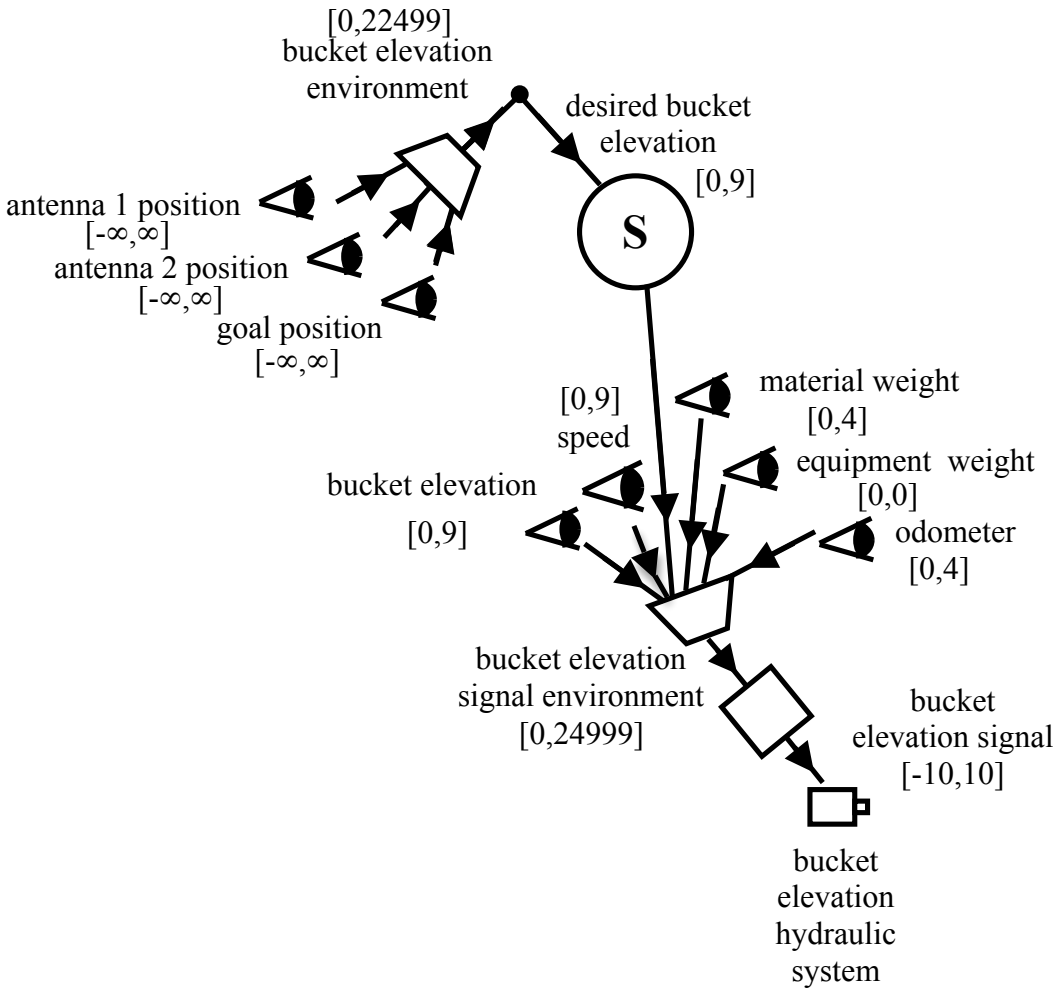

Figure 5.61. Third Simplified Subsystem (Out of Four).

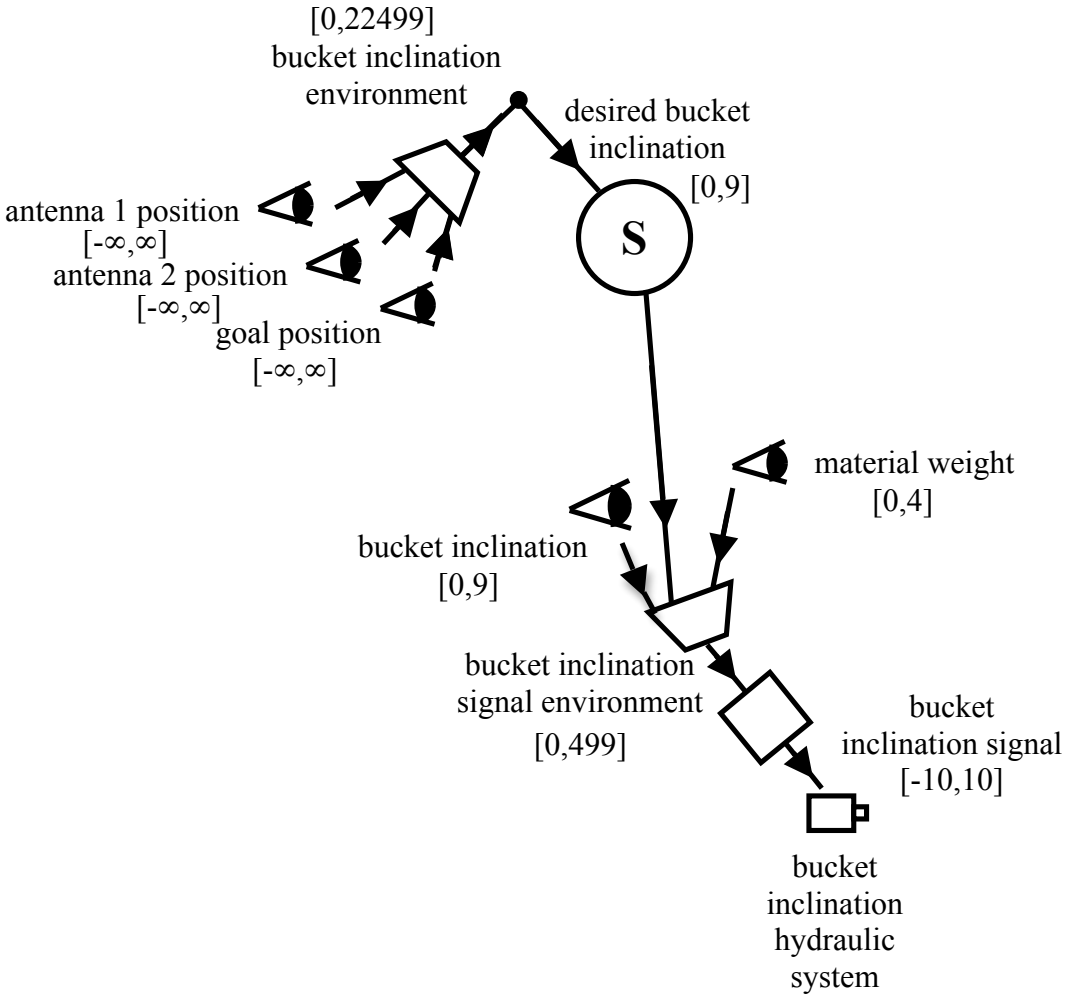

Figure 5.62. Fourth Simplified Subsystem (Out of Four).

Once we have applied the simplification rules to the description of our system, the next step is to specify the associated reward functions. This is done in the next section of this chapter.

5.13 Design Step 8 (Define a Reward Function for each LA)

Our current system has eight learning agents, four *master LAs* and four *actuator LAs*. Each one of these LAs has a very specific responsibility. These responsibilities are conveyed to

the LA by using reward functions.

For instance, one of the master LAs has the responsibility of coming up with the optimal heading that the equipment should try to accomplish at any given time. To come up with this reward function we need to think of indicators that would be able to tells us if we are doing a good or a bad job at coming up with the optimal heading. One indicator that we are doing a bad job would be that the equipment suffers some damage (e.g. by crashing or by being submitted to stress that goes beyond manufacturer's specifications). One indicator that we are doing a good job would be that we are getting closer to the desired position and that we are reaching the desired entry heading. Another negative indicator would be the amount of energy used. If we use these indicators, this Master LA and its reward function would look like this:

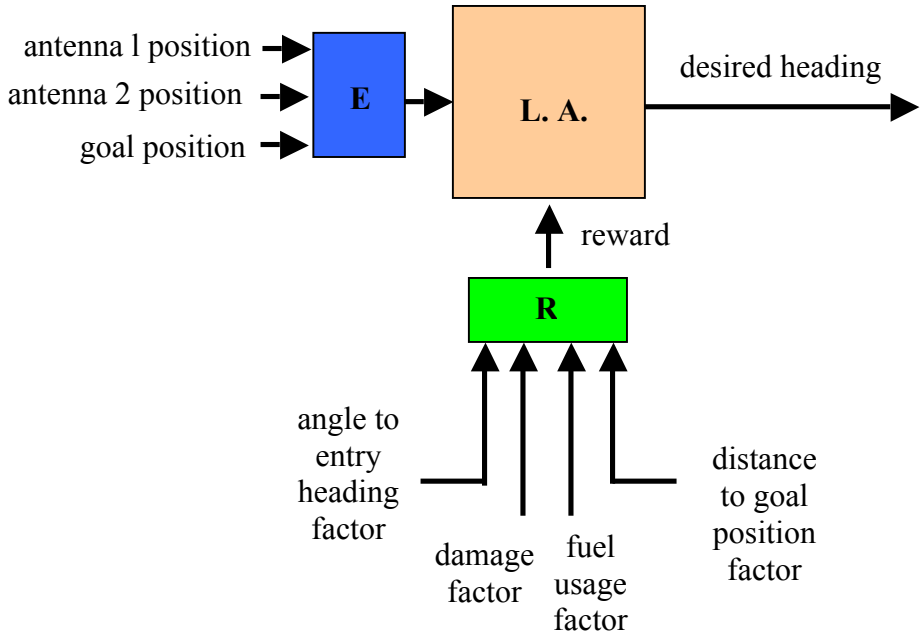

Figure 5.63. The *Desired Heading Master LA* and its Reward Function.

Notice that all the inputs to the reward function are *factors*. In this context, a factor can only have values from 0.0 to 1.0. The way we come up with factors for a given input is by

answering this question:

What are the maximum and minimum values for that input?

The maximum value will be mapped to 1.0 and the minimum value will be mapped to 0.0. All the other possible input values will be mapped to a value in the range (0.0, 1.0).

The maximum input value associated with *angle to entry heading factor* would be 180 and the minimum would be 0. Therefore, 0 would be mapped to 0.0 and 180 would be mapped to 1.0.

In the case of the *damage factor*, we could have three possible values: a) 0.0➔ no damage to the vehicle, b) 0.5 ➔ vehicle's body was strained beyond manufacturer's specifications and c) 1.0 ➔ vehicle crashed.

With respect to the *fuel usage factor*, we would need to find out what is the maximum amount of fuel the vehicle can burn in a single learning cycle. This value would be mapped to 1.0.

In the case of the *distance to goal position factor* we would also find out what is the maximum distance this vehicle could possibly travel during a learning cycle. That distance would be mapped to 1.0.

Now that we have the factors associated with the reward function, we need to think of whether these factors are things we want to achieve or things we want to avoid.

In this specific case, all *factors* represent things that need to be avoided. For instance, we want to avoid being at a high angle with respect to the entry heading's angle. Furthermore, we want to avoid being very far from the goal. Also, we want to avoid using gas, as much as possible. Finally we want to avoid damaging the vehicle.

Factors that represent things we want to achieve will be associated with a positive weight.

Factors that represent things we want to avoid will be associated with a negative weight. For instance:

Reward = (angle_to_entry_heading_factor)*(-1.0) +
(distance_to_goal_position_factor)*(-1.0) +
(fuel_usage_factor)*(-1.0) +
(damage factor)*(-1.0)

This value function is an easy way for a human to convey to the computer system what needs to be achieved and/or what things need to be avoided.

The equation above uses the same weights on all the factors. However, it does not need to be this way. For instance if, nowadays, the price for gas is very high, we may want to use this reward function instead:

Reward = (angle_to_entry_heading_factor)*(-1.0) +
(distance_to_goal_position_factor)*(-1.0) +
(fuel_usage_factor)*(-5.0) +
(damage factor)*(-1.0)

On the other hand, if gas is cheap and what we want to do is to decrease the time it takes to load a truck we may want to use this reward function:

Reward = (angle_to_entry_heading_factor)*(-1.0) +
(distance_to_goal_position_factor)*(-5.0) +
(fuel_usage_factor)*(-1.0) +

(damage factor)*(-1.0)

For our implementation I decided to use the first value function. However it would be straightforward to use any of the other functions in the future.

Now that we came up with a reward function for the Master LA in charge of determining the best heading angle for any given situation, let us look at its associated Actuator LA. The responsibility of this LA is to send the appropriate steering signal to the steering motor given a desired heading angle.

In this case, we would know that we are doing a good job if we are able to achieve the desired heading. Also we would know we are not doing a good job if we cause damage to the equipment while doing that. In the case of this LA, the damage we are talking about refers to lower level damage, more specifically to the wear of the tires. A good indication of tire damage may be acceleration (the more we accelerate the more we would wear off the tires).

After this discussion, we end up with two factors. The first one is the *achieved desired heading factor* and the second one is the *tire damage factor*. In the case of the *achieved desired heading factor*, it would get a value of 1.0 if the desired heading requested by the Master LA was perfectly accomplished. This factor would get a value of 0.0 if the difference between desired and accomplished heading were 180 degrees. In the case of the *tire damage factor*, we could give it a value of 1.0 if the acceleration of the vehicle reaches some outlandish value and we could give it a value of 0.0 if the acceleration is zero.

If we use these indicators, this Slave LA and its reward function would look like this:

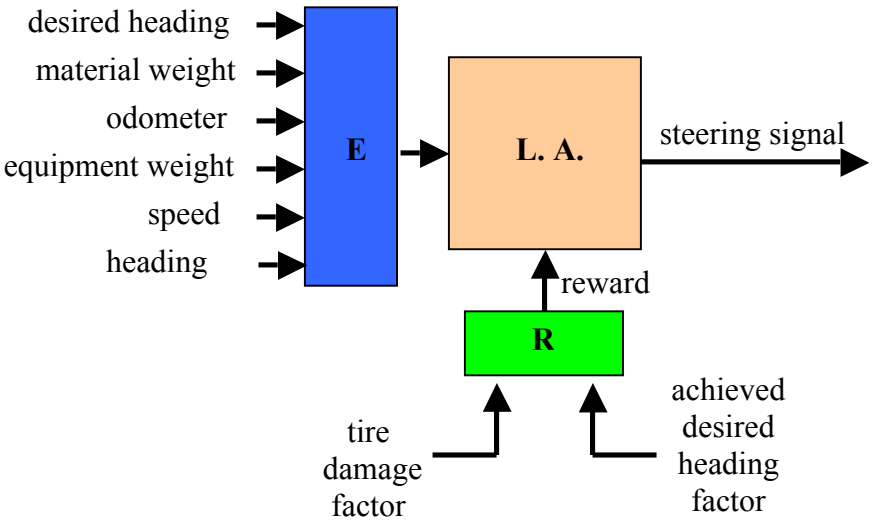

Figure 5.64. The *Steering Signal Slave LA* and its Reward Function.

The reward function could look like this:

Reward = (achieved_desired_heading_factor)*(1.0) +

(damage factor)*(-1.0)

Notice that in this case one of the factors has a positive weight while the other has a negative weight. This is because the first factor is related to something we want to achieve while the second factor is something we want to avoid.

The reward function above uses the same weight for both factors. However, if the price of the tires becomes very high, we may want to use this reward function instead:

Reward = (achieved_desired_heading_factor)*(1.0) +

(damage factor)*(-5.0)

If the autonomous system still does not perform as well as we want and it is still wearing off the tires too often, we could try using this reward function:

Reward = (achieved_desired_heading_factor)*(1.0) +
(damage factor)*(-10.0)

Reward functions are a way to tell the autonomous system what is important to achieve and to avoid and how important is one factor over the others.

Once again, for this experiment we selected the first one of the proposed reward functions.

Now, if we do the same with the other LAs, we could end up with something like this:

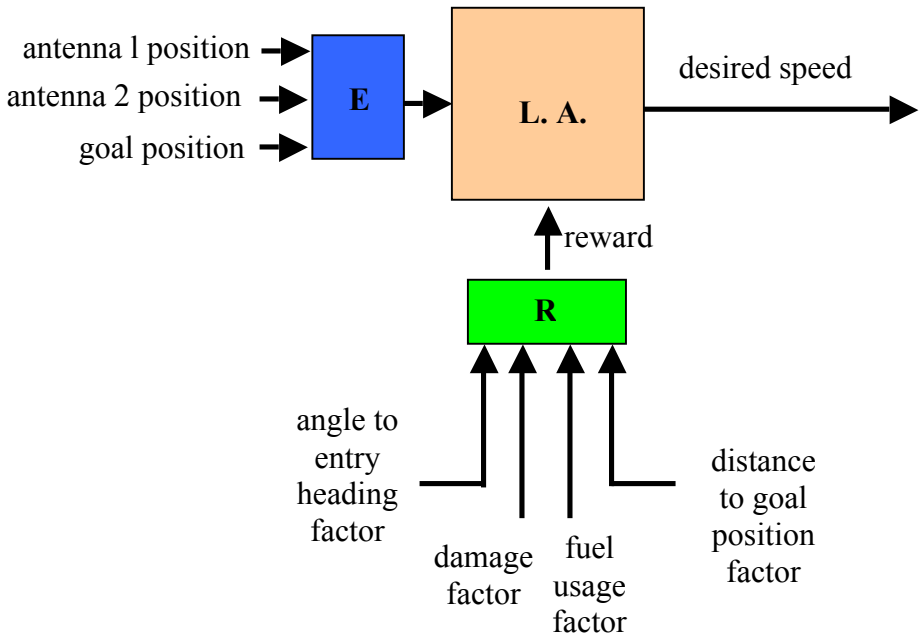

Figure 5.65. The *Propulsion Master LA* and its Reward Function.

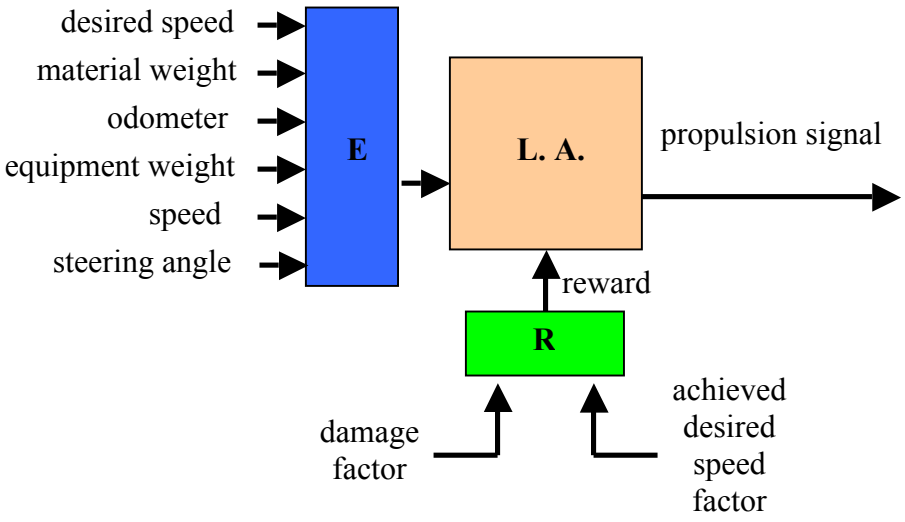

Figure 5.66. The *Propulsion Signal Slave LA* and its Reward Function.

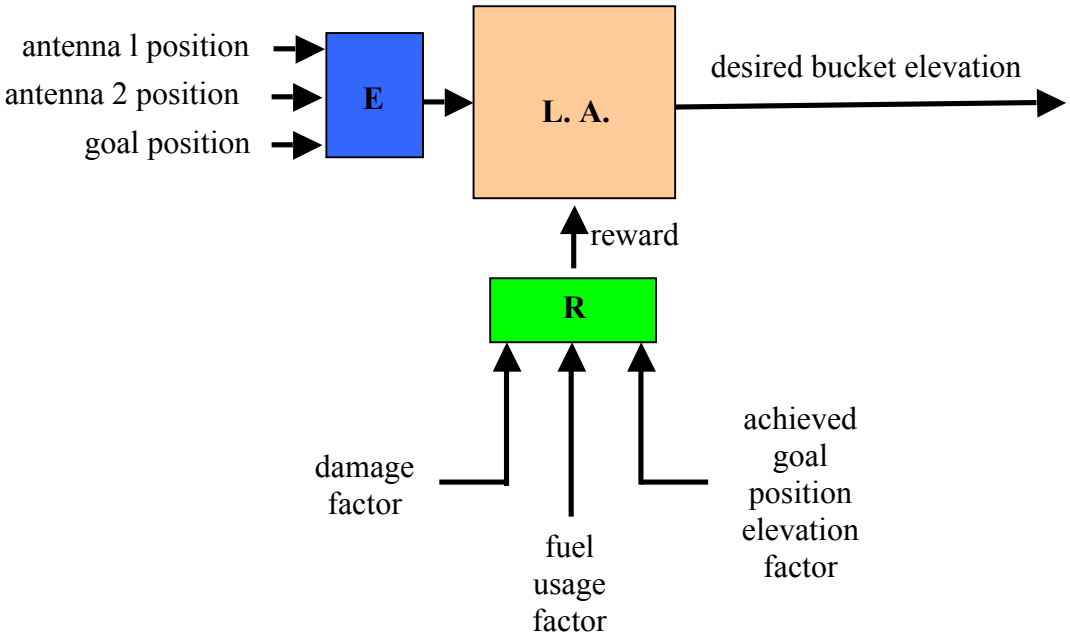

Figure 5.67. The *Bucket Elevation Master LA* and its Reward Function.

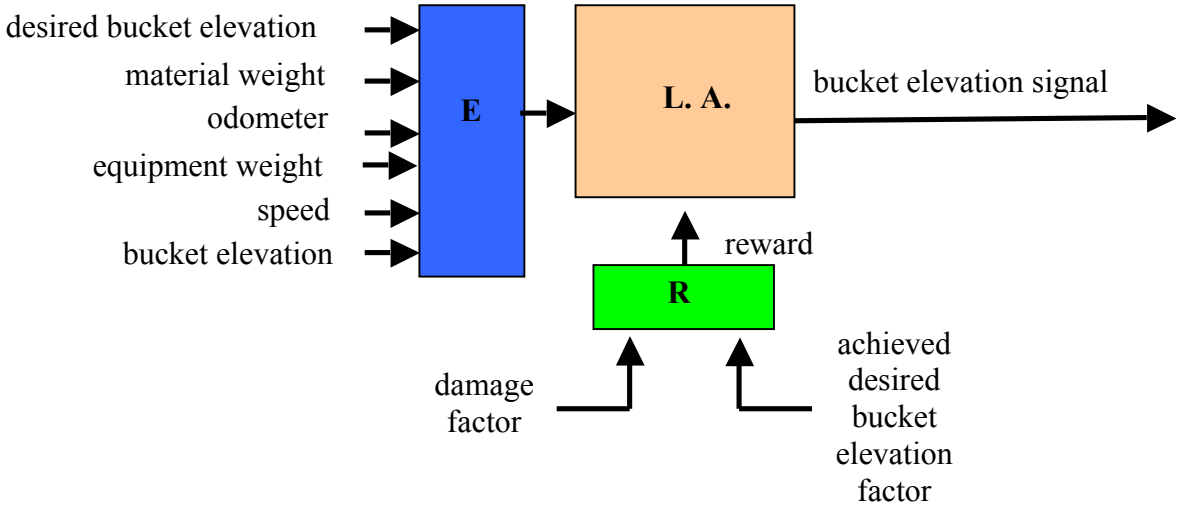

Figure 5.68. The *Bucket Elevation Signal Slave LA* and its Reward Function.

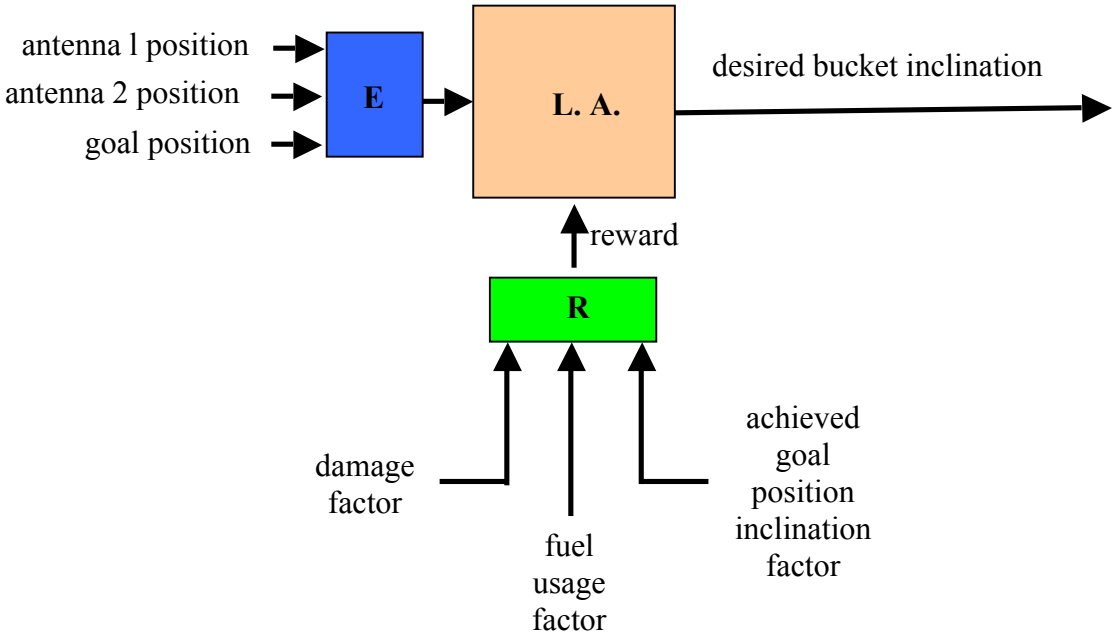

Figure 5.69. The *Bucket Inclination Master LA* and its Reward Function.

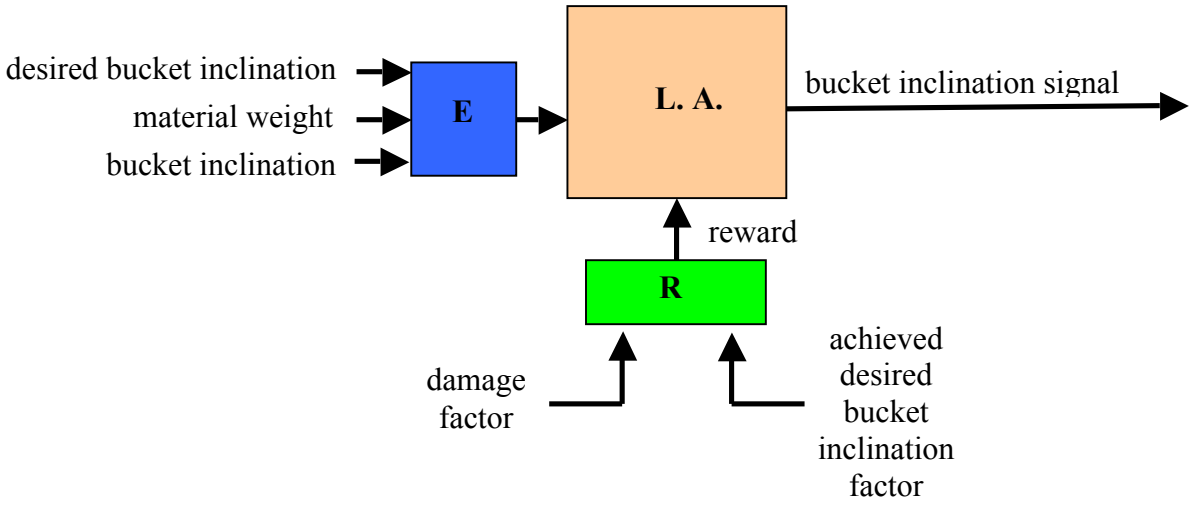

Figure 5.70. The *Bucket Inclination Signal Slave LA* and its Reward Function.

The weights associated with each one of these reward factors were kept to either 1.0 or -1.0 depending on whether the factor was something we wanted the learning agent to achieve or was something we wanted the agent to avoid.

5.14 Training the Learning Agents

You may have noticed that we have not really talked much about training in our formalism. The reason for this is that we believe this is a process the designer of artificial cerebellums does not have to be familiar with. We believe that training Q-learning agents is a very generic process, which requires very little input from the designer and is largely independent of the value function that needs to be learned. In the future, a training module should be implemented and added to the development tool used by the designer of artificial cerebellums. This module would allow the designers of artificial cerebellums to train their

systems without worrying about the details of the training algorithm.

This section describes in details how one of our Q-learning agents was trained. Furthermore, this section mentions the generic steps needed to train practically any Q-learning agent. It is my hope that this section is used, in the future, as an example of how such a generic trainer module could be implemented.

We chose the first one of our eight LAs to illustrate how training takes place. However, any of the other seven LAs would have also provided a good example. Below is a graphical description of the first LA using the proposed formalism:

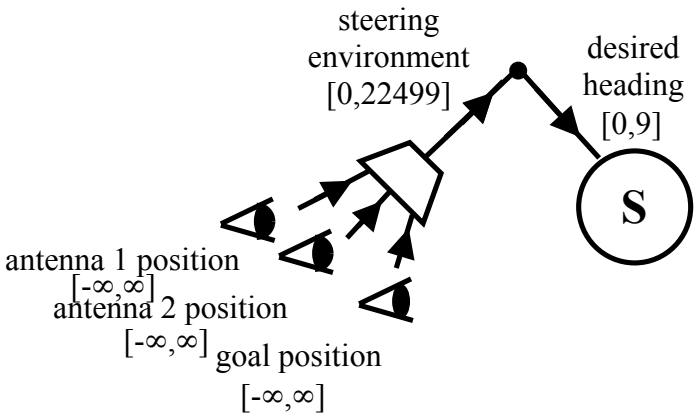

Figure 5.71. The LA Used to Illustrate the Generic Procedure of Training a LA.

From this description we can tell that the value function for this Q-learning agent has 22500 possible states and 10 possible actions. The value function for this Q-learning agent looks like this:

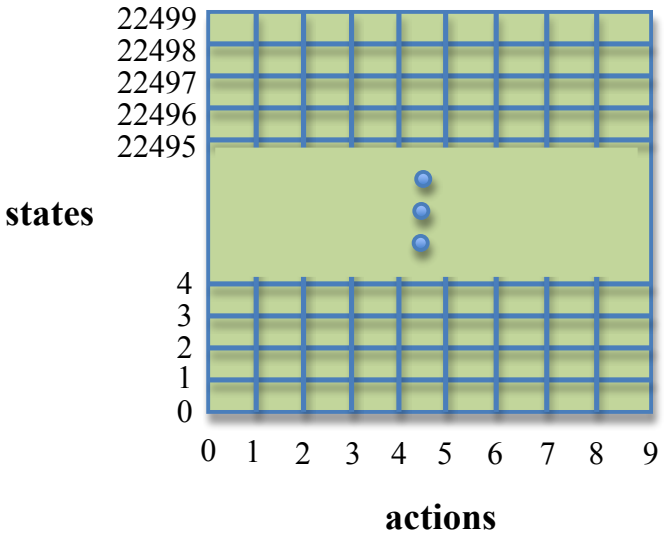

Figure 5.72. The Value Function Associated with the First LA.

As we discussed before, this value function is at the core of the Q-learning agent. The picture below represents the same LA but also describes the parameters used by the reward function:

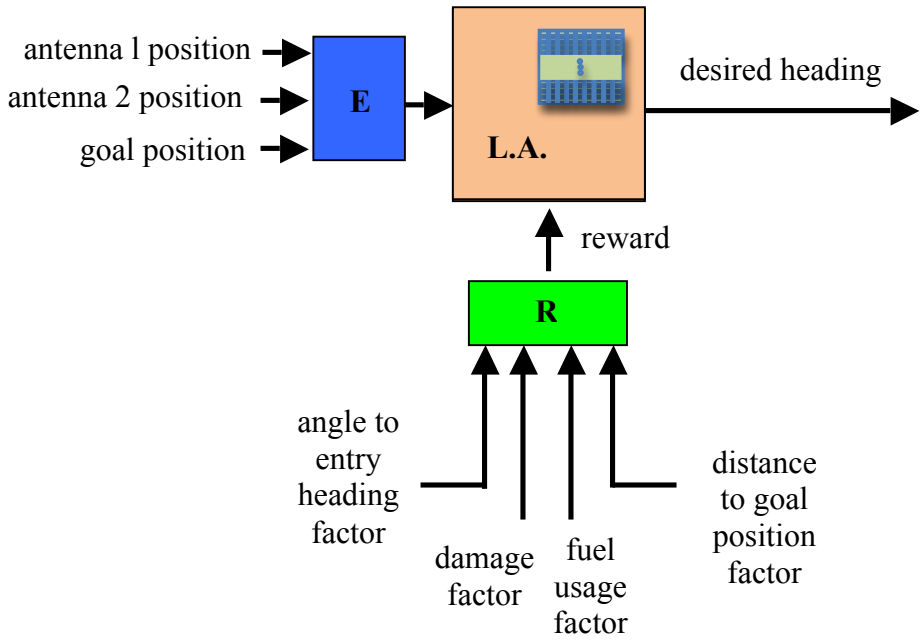

Figure 5.73. The First LA and Parameters Needed by Reward Function.

The way a Q-learning agent learns is by following these steps: 1) detecting the current state, 2) selecting an action, 3) evaluating the results of taking that action given that state using the reward function and, finally, 4) updating the value function using that reward. The formula used to update the value function is the following:

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

The training process can be described in procedural form as shown below:

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode)
   Initialize s
   Repeat (for each step of the episode)
      Choose a for s using an exploratory policy
      Take action a, observe r, s'
      Q(s, a) ← Q(s, a) + α [r + γ max Q (s', a') – Q(s, a)]
                                    a'
      s ← s'
```

Figure 5.74. The Training Process in Procedural Form.

You may have noticed that after each step in a given episode a new state s' is observed and that this new state s' becomes the current state s for the next step in that episode. The disadvantage of learning this way is that certain states may be visited very often while other states may not be visited at all. This may produce a value function with "immature areas". If we are using the real environment to find the resulting state we have no option but to learn this way

and increase the number of learning steps to improve our chances of visiting every state very often.

However, if we use a simulator instead of the real life environment, we can jump to any state any time we want. This allows us to visit all states in the value function with the same frequency. With this idea in mind, we decided to restart the simulator after each learning step so that each initial step is visited with the same frequency. In other words, during the first learning step the simulator is reset so that its initial state is state 0, during the second learning step the simulator is reset to the initial state 1, and so on until the 22500th step in which the initial state is set to the 22499 state. Then in the 22501st step the initial state 0 is selected again and in the 22502nd step the initial step 1 is selected and so on. However, we are forgetting about one more detail. We not only would like to visit all the states with the same frequency but we would also like to try using all the available actions for every state with the same frequency. For instance, if the initial state is 0 we would like to explore what happens when we use action 0,1,2,3,4,5,6,7,8 and 9. The same is true if the initial state is 1,2,3,4, … or 22499. Therefore we would end up exploring 225,000 different state-action combinations. For each one of these 225,000 state-action combinations we would follow the following steps: 1) initialize the simulator to the current state, 2) ask the simulator to simulate the current action, given this initial state for one unit of time (in our case 1 second), 3) request from the simulator all the information we need to evaluate the reward function associated with this learning agent, 4) update the value function using this reward value. For this specific LA, the reward function was defined by the equation below:

```
Reward =  (angle_to_entry_heading_factor)*(-1.0) +
          (distance_to_goal_position_factor)*(-1.0) +
          (fuel_usage_factor)*(-1.0) +
          (damage factor)*(-1.0)
```

We continue performing passes of these 225,000 state-action combinations until the value function before and after a pass differed by less that 1% at any position in its state-action space. Convergence for this LA took only a few seconds running on a computer using a Core Duo microprocessor.

Notice that the process of training a Q-learning agent is very generic. We can accomplish this by following these steps:

a) Find out how many possible states (#s) are associated with the LA. In my previous example #s is equal to 22500.
b) Find out how many possible actions (#a) are associated with the LA. In out previous example #a is equal to 10.
c) Define a *sweep* as the set of all possible state-action combinations. In my previous example a *sweep* was composed of had 225,000 possible state-action combinations.
d) For each possible state-action combination, ask the simulator to simulate what would happen given that we start in that state and apply that action.
e) Calculate the reward using the reward function associated with that equipment.
f) Update the value function using that reward value.
g) Repeat steps a) thorugh f) until the value function before and after a *sweep* differs by less than a certain percentage. In my previous example I selected 1% as my threshold.

I applied this method to the other seven LAs. For some LAs it took a few seconds to converge, for others it took a few minutes. After convergence I ended up with eight value functions, one for each one of the LAs. The following section explains what I did with these eight value functions.

5.15 Extracting the Policy Functions

Once a value function for a given LA has converged, the next step is to extract the associated policy function. This policy function indicates what action should be taken for each possible state.

The procedure to extract a policy function out of a mature value function is straightforward. Furthermore, this is a topic a designer of artificial cerebellums does not need to be aware of. I suggest that in the future this method is implemented within the design tool. That way, the LAs are not only automatically trained but their policy functions are also automatically generated.

The value functions associated with the eight LAs are quite large. Therefore, allow me to illustrate the process of extracting a policy function by using a much simpler example. Let us suppose we have a small value function, which looks like this:

States\Actions	0	1	2	3	4
0	123	453	-343	12	3
1	1	-5	542	445	11
2	645	43	5	3	34
3	2	-22	62	21	-32
4	2	12	4	65	667
5	434	43	21	1	-4

Figure 5.75. A small Value Function Used to Illustrate Policy Function Extraction.

The first step is to go through each one of the states and look for the action, which maximizes the expected reward. The picture below shows such a value in bold:

States\Actions	0	1	2	3	4

0	123	**453**	-343	12	3
1	1	-5	**542**	445	11
2	**645**	43	5	3	34
3	2	-22	**62**	21	-32
4	2	12	4	65	**667**
5	**434**	43	21	1	-4

Figure 5.76. A small Value Function with Largest Values for each State in Bold.

The final step is to create a table whose first column enumerates each one of the possible states and the second column indicates the action, which provides the largest expected reward given that initial state. The picture below shows the resulting policy function extracted from the above value function:

State	Best Action
0	1
1	2
2	0
3	2
4	4
5	0

Figure 5.77. The Policy Function Extracted from the Value Function.

Once this function is known, the LA can respond to any initial state by looking up that

state and selecting the associated action. For instance, if this LA is currently in state 5, the LA should perform action 0 if it wants to maximize its performance.

In the next section I provide a short discussion on the advantages of using a simulator versus using a real life environment to train LAs.

5.16 Simulation vs. Real Life Environment

After designing an Autonomous System, we could just deploy it in a real environment so that it can learn to optimize the task at hand. However, learning in this way may be very slow. For instance, a set of learning agents may require thousands of learning experiences in order to converge to their optimal policies. In a real life environment, thousands of learning experiences may be equivalent to several days or months of experimentation.

Furthermore, having an immature autonomous system in a mine may be very dangerous to the human workforce.

Finally, it may be very costly. Learning on a real environment, especially when the system has no experience at all, most likely will cause things like damage to the equipment and inefficient use of fuel. It may be much more cost effective to use a simulator, at least at the earliest stages of learning, and perhaps learn in a real life environment once the autonomous system has already formed reliable, if not optimal, policies.

5.17 Results

For this experiment, I implemented not only the eight learning agents described in 5.12 but also the reward functions described in section 5.13. In addition I implemented a mine's environment simulator. This simulator was designed to predict things like next position, speed and heading of the vehicle given things like its previous position, previous speed, previous

heading and weight of the vehicle and its payload. The internal model of this environment was kept secret from the implementation of the learning agents, as it should be. Under normal conditions, the learning agent would not have the internal model of the real environment either.

Once the implementation of the learning agents, reward functions and simulator were completed, I trained the agents in a way described in section 5.14. I forced a set of learning experiences on the system until the value functions stopped changing significantly. In this specific case the threshold was 1% or less of change after a *sweep*. After this the learning process was stopped and optimal policies extracted in a way described in section 5.15. As soon as the optimal policies were generated, I ran simulations of this autonomous system only using these policies to determine the best action given the current state. For example, in one of these simulations the front-end loader was asked to take material from a certain position at the wall and deliver this load to a nearby truck. The simulation started by positioning the loader with its bucket facing toward the wall and by making the position of the truck the "goal position" and the height of the bed of the truck the "goal bucket elevation". Once the simulator was initialized to this state, the artificial cerebellum was allowed to make a new decision every second based on its current state and its optimal policy. This process generated a file containing the position of the truck, its heading and the description of the wall plus a line for every second of simulation. Each one of these lines had the current position of the loader, its heading, and the elevation of its bucket. This file was then fed to the visualizer, which used this information to draw the wall and the truck. In addition, this visualizer drew the loader at the appropriate position and with the correct heading and bucket elevation. The loader drawing was updated every second until reaching the end of the file.

The visualizer allowed us to confirm that the loader behaved as expected. In other words,

the autonomous system was able to drive the FEL in a way that attained the desired loading and dumping locations by driving the vehicle in smooth curving trajectories (very similar to the one seen when very experienced drivers do it). Furthermore, the vehicle smoothly accelerated at the beginning of the trip and then slowed down before reaching its goal, as to avoid hitting the wall too hard or crashing against the truck. In addition, I observed that the bucket elevation was only achieved at the very end of the trip. After careful consideration I realized that this behavior made perfect sense since the vehicle was much more stable with the bucket lowered than with the bucket raised. I had introduced this feature on the simulator by making the gearbox less efficient when the bucket was elevated and I had forgotten about it. However, the autonomous system was able to discover it indirectly and adjust its policy as to deal with the issue as well as possible. In this specific case, the best way to deal with it was to delay the raising of the bucket until the FEL was very close to the truck or to the wall.

Below, I show some screen shots of this visualizer taken as I ran a simulation on my autonomous system. The first screenshot shows the FEL as it is filling up its bucket at the wall.

Figure 5.78. Screenshot of the FEL Filling Up its Bucket at the Wall.

The next screenshot shows the FEL backing up from the wall.

Figure 5.79. Screenshot of the FEL Backing up from the Wall After Loading.

The next screenshot shows the FEL moving forwards toward the truck:

Figure 5.80. Screenshot of the FEL Moving Toward the Hauling Truck.

The final shot shows the FEL dumping its load into the truck:

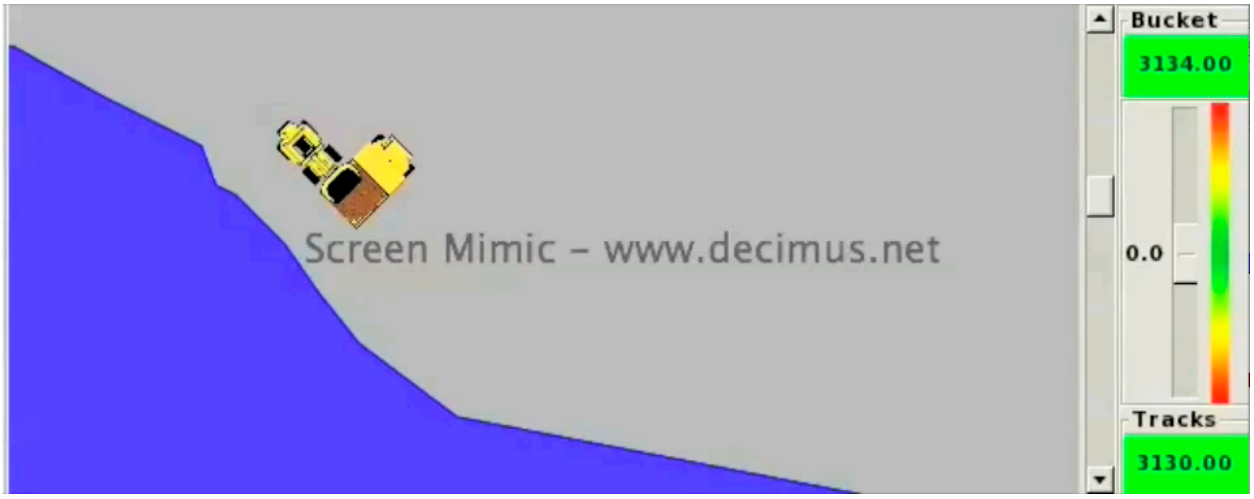

Figure 5.81. Screenshot of the FEL Dumping its Load onto the Hauling Truck.

5.18 Repeat as Needed

As I mentioned before, the proposed method to develop autonomous systems is highly iterative. We do not have to come up with a perfect system the first time around. Often we will find that the system still does not perform as well as we want to and we will have to re-analyze the task at hand and its environment. At this point we may discover new significant factors or discover that certain factors are not really significant and should be taken off the system. A simple way to tell if a factor is not significant would be to take it out of the system and see if the performance decreases.

If a new factor were to be added to the system, we would go through the same steps as we did before. That is, come up with a list of actuators and/or sensors associated with these new factors. Then, check if new *sanity points* may be useful. After this, we would check if there is any overloaded LA, in which case we would break it into two or more simpler learning agents. Furthermore, we would introduce these new learning agents to our system using the proposed formalism. Then, we would apply the simplification rules to this description of the system.

Finally, we would need to define reward functions for the new learning agents.

In this specific case, the performance associated with the current design of the autonomous system was considered to be sufficient as a proof of concept. However, if this system were to be deployed in a real mine in which even small improvements on performance can mean thousand if not millions of dollars in savings, we may want to continue exploring the use of other factors that may allow us to optimize the system even more.

5.19 Review

The main goal for this chapter was to illustrate the use of the framework described in chapter 4. I began this chapter by describing a real life problem in the Mining Industry: The shortage of experienced operators and rubber tires and the high price of fuel. I proposed tackling this problem by implementing an autonomous mining equipment. These systems would not only be able to work without the assistance of operators but also may be able to do their job while minimizing the wearing of their tires and consumption of fuel. Since there is already a system called Front Runner, which automatizes hauling trucks, I decided to select a different type of mining equipment for our illustration, namely a front-end loader. To implement this autonomous system I proposed to use the formalization described in chapter 4 to create an artificial cerebellum capable of controlling the movements of this type of equipment. Several of the sections in this chapter were dedicated to illustrate the use of each one of the steps in our formalism. After using this process I ended up with an artificial cerebellum consisting of eight different learning agents working in unison to accomplish the task of controlling the movement of this autonomous system. In addition, each one of these learning agents had associated a reward function and all its input and output information was clearly specified. The next step was

to train this system. Unfortunately, the astronomic cost of this type of equipment forced us to use a simulator instead of a real life front-end loader to train our system. However, I am optimistic, that someday this research will instill enough confidence in a big mining corporation as to support the development of this autonomous system in a real life front-end loader.

Even though I did not have the opportunity to implement this autonomous system in an actual FEL and in a real mining environment, but instead using simulators, I believe the autonomous system developed is a good proof of concept. The reason why I believe this is because the learning agents did not have the intrinsic model of the simulator and still were able to optimize their task without this information. I believe that the learning agents would also be able to optimize their performance if the environment model changed, including the case in which the model was the real life environment.

CHAPTER 6: DISTRIBUTED LEARNING AGENT

6.1 Overview

In the previous chapter we saw an example of how we can use several Q-learning agents to solve a real life problem. It would be possible to place each one of these agents in separate computers and with the help of simulators train them in parallel. However, in some cases, one of these LAs may be too large to be implemented by a single computer. A reason for this is that we may have assigned a really large and complex state-action space to this agent. Another reason is that we may have a very limited amount of time available to train this agent. In either of these cases we may want to use a distributed system to implement a single learning agent.

One implementation of this distributed system could be to break up the value function into sections and place each one of them into in a separate computer. In a sense, we would be breaking a large learning agent into a set of smaller co-depending learning agents. Each one of these smaller learning agents would run on its own computer and would be responsible for updating its own value function section. By distributing the task of generating the value function, a distributed learning system would be able to tackle problems that a centralized learning system would be unable to do because of lack of memory and/or CPU resources.

Nevertheless, in order to make good use of this type of distributed Q-learning system, we need to first try to understand where its performance bottlenecks are and what we can do to ameliorate these pitfalls.

In other words, we need to be able to formally analyze this type of distributed Q-learning system and find its performance bottlenecks. Furthermore, we need to come up with a set of

techniques that may address these performance bottlenecks. Finally, we need to formally verify the virtues of each one of these new techniques.

6.2 Analysis and Verification Tool

DEVS Java is a practical and easy to use simulation tool that allows the user to model and perform discrete simulations. This tool is based on Discrete EVent System specification (DEVS), which is a well-defined mathematical formalism.

I decided to use DEVS Java to simulate distributed Q-learning scenarios. This allowed us to analyze distributed Q-learning systems and find their performance bottlenecks. Furthermore, I used this simulation tool to validate or refute several techniques I came up with. This led to the improvement on the efficiency of the proposed distributed Q-learning system.

6.3 Details on DEVS

As I mentioned before, I believe that DEVS Java is a good choice when presented with the problem of selecting a modeling and simulation tool. DEVS Java allowed us to carry out the necessary simulations required to support my research on distributed learning.

The following subsections describe in more detail some DEVS Java properties and how these properties match the requirements for a modeling and simulation tool.

6.3.1 Continuous Time Support:

DEVS Java is a tool that is based on the Discrete EVent System specification. This does not mean that this mathematical formalism is only applicable to scenarios in which time is discrete. As a matter of fact, DEVS assumes that the time base is continuous. The type of

scenarios we would like to simulate requires that the used time base be continuous. The way DEVS accomplishes this is by assuming that state variables remain constant for variable periods of time. These state changes are called *events*. This assumption is valid for our scenarios since modules can only be in a small number of states (e.g. “waiting for largest value function entry”, “waiting for simulator response”, “calculating new reward”, “forwarding a message”, etc). Furthermore, the modules in these scenarios stay in these states for variable periods of time.

6.3.2 Appropriate Modeling Paradigm

Under the DEVS formalism, the models specify how events are scheduled and what state transitions they cause. This perfectly satisfies our requirements since we are looking for a tool that allows us to model modules such as a “learning agent”, a “hub”, a “simulator” etc, for which we have a very clear idea of what states they go through, and what events trigger changes on these states. This formalism makes it very easy for us to model the components associated with the scenarios we want to simulate. Furthermore, DEVS Java supports object-oriented modeling which facilitates the modeling of very complex modules.

6.3.3 Do not Worry About the Simulator:

DEVS Java has an associated simulator that handles the processing of events as dictated by the models. This is very convenient, since we do not have to worry about implementing a simulator. We only have to concern ourselves with modeling the components and plugging them together. The provided simulator will take care of processing the events as dictated by our models.

6.3.4 Concurrency Support:

DEVS and therefore DEVS Java support concurrency among the interacting simulation modules. This is a very important feature since concurrent events will often occur in the scenarios we would like to simulate. For instance, we need to be able to properly handle the case in which we are ready to start a new learning cycle but at the same time we receive a request for information from another learning agent. DEVS allows us to easily specify what to do in such cases.

6.4 Details on Proposed Distributed Learning System

I assumed that the distributed learning system was composed of a set of learning agents. Each one of these learning agents is in charge of generating a section of the whole value function associated with the learning problem at hand. To be more specific, the particular problem I selected to test the proposed techniques was the one of learning to "balance a pole on a 2-dimensional setup". Each one of the learning agents was assumed to live in its own computer but be able to communicate with each other by making use of a common communication backbone. To be specific, this communication backbone was assumed to be comprised of a set of Ethernet cables and some Ethernet hubs. In the most basic setup, a simulator of the problem at hand would run on its own computer and would be connected to the learning agents through the common communications backbone. This simulator would be used by the learning agents to learn their next state, given an initial state and a given action. For instance, if the current angle of the pole is 82 degrees and the action to be taken is to move the pole forward 2 centimeters in a second, the simulator would be able to tell us what would be the resulting angle. Providing an answer to this kind of question would be the responsibility of this simulator.

6.5 Details on the Distributed Learning System Implementation

As I mentioned before, I decided to use DEVS Java to model and simulate any scenario required by my research. This required modeling a distributed learning system. This system initially consisted of nine ***Learning Agents*** (each one identical to the others):

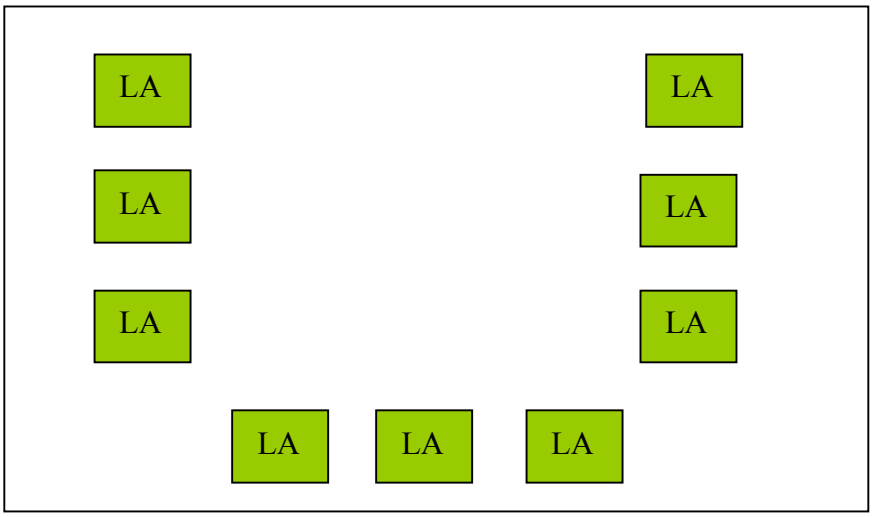

Figure 6.1. The Learning Agents in the Implemented Distributed Q-Learning System.

Each one of these modules was able to maintain a section of the value function.

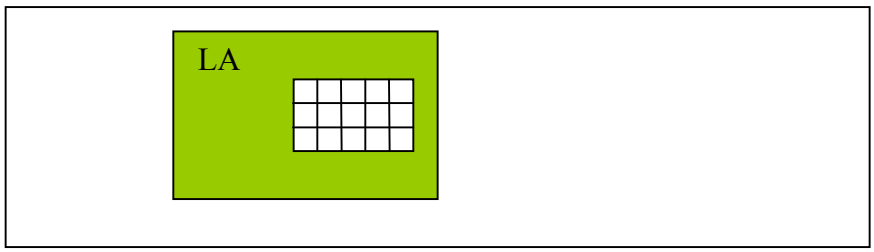

Figure 6.2. A Q-Learning Agent and its Value Function Section.

Furthermore, this module had to be able to repeatedly go through learning cycles in order to update this table. In order to do this, the module has to be able to go through different states, each one indicating the current step that is being executed in the learning cycle.

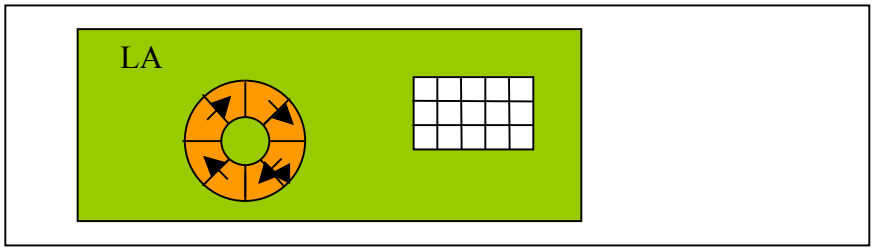

Figure 6.3. A LA, its Value Function Section and its State Machine.

If the required information needed to execute a given step was not available locally, it was the responsibility of this module to send a message out to other learning agents asking for this piece of information. Furthermore, it was the responsibility of this module to accept information requests from other learning agents and service these requests if the information was available locally.

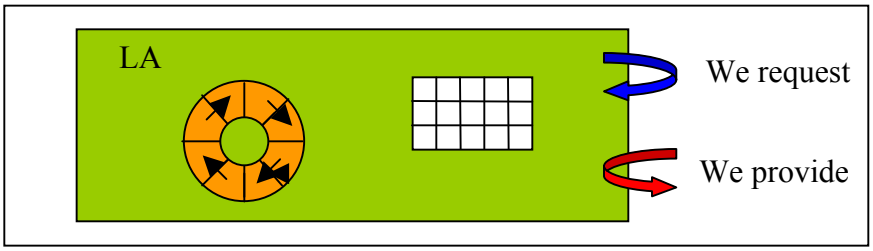

Figure 6.4. A LA Requesting and Providing Info.

As expected, in order to simulate a distributed learning system I needed to model and simulate the communication backbone. I selected to model an Ethernet network backbone. This required the implementation of a ***Hub*** module whose input is connected to the output port of all the other modules and whose output is connected to the input of all the other modules:

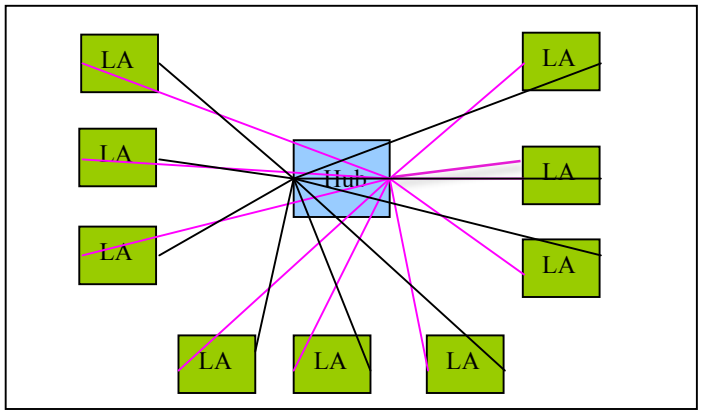

Figure 6.5. The Nine LAs, the Hub and their Connections.

Furthermore, I had to model the environment in which this distributed learning agent learns. This was implemented by a module I called ***Simulator***. This module's responsibility was to receive messages indicating current state and current action taken by a learning agent and respond by sending a message back indicating the resulting new state. This module was also connected to the *Hub* module like any of the learning agents.

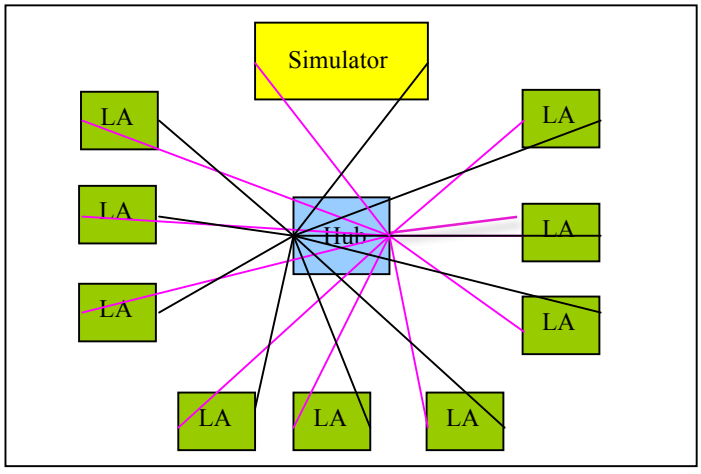

Figure 6.6. The LAs, the Hub, the Simulator and their Connections.

Finally I modeled a ***Message*** (which inherits from the DEVS Java class "entity") to be used to send requests and responses between the modules in the distributed learning system.

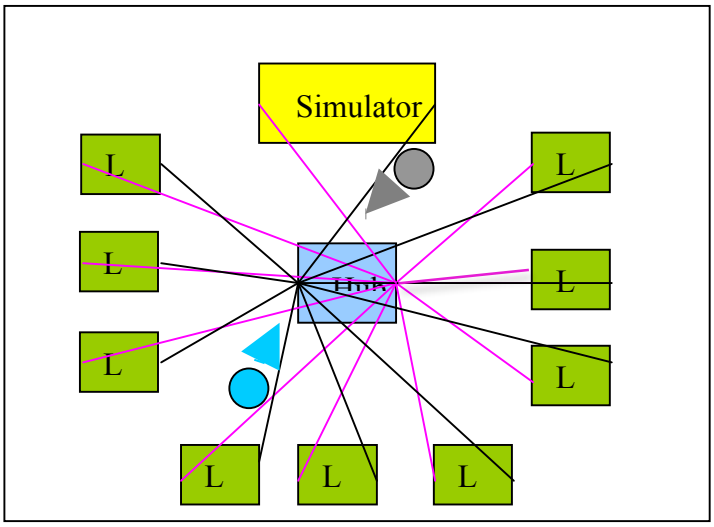

Figure 6.7. The LAs, the Hub, the Simulator, a Request and a Response Message.

6.6 The Initial Hypothesis

Since the beginning, I suspected that learning in a distributed Q-learning system would occur sooner if I distribute the load in a way as to minimize traffic among the learning agents. Unfortunately, I realized that traffic between the learning agents and the simulator could not be changed since a call to the simulator would be required for every update of the value function. Therefore, no matter how the load was to be distributed, the total number of calls to the simulator would end up being the same. However, I suspected that there might be a way to diminish the number of messages sent between the learning agents.

Before I discuss how we were able to do this we need to mention that the reason for having learning agents sending messages to each other is because, in order to update an entry in the value function, the learning agent needs to know what is the largest expected reward associated with the next state. This value can be obtained from the value function itself.

However, since the value function is now distributed, it is possible that the section or sections of the value function that is/are needed to answer this question may not be in the same learning agent that is trying to update this value function entry. Therefore, this learning agent will need to ask the other learning agents to help answer this question. This question may be directed to one or more learning agents, depending on whether the value function entries needed to answer this question are located in a single node or in several nodes.

Since I knew that breaking the value function along the "action axis" implied that any such questions would be answered by exactly one learning agent (as opposed to many learning agents), I came up with the following theory:

Theorem 1: *Learning will occur faster if we split the value function in slices along the "action" axis.*

To validate or refute this theory I proposed the two scenarios described in the following section.

6.6.1 The Initial Hypothesis: Two Scenarios

In order to validate or refute *Theorem 1*, I proposed to simulate the following two scenarios:

Scenario 1: *The value function is divided into 9 sections along the "state" axis and each learning agent is given one of these 9 sections.*

Scenario 2: *The value function is divided into 9 sections along the "action" axis and each learning agent is given one of these 9 sections.*

In order to experimentally verify this theory I modeled and simulated the two scenarios using DEVS Java. The results of these two simulations are described in the following section.

6.6.2 The Initial Hypothesis: Results

After simulating *Scenario 1* using DEVS Java, I found out that it took the distributed learning agent 3500 time units to learn to balance a pole.

On the other hand, DEVS Java showed that, under *Scenario 2*, it took 1700 time units to learn to balance a pole.

6.6.3 The Initial Hypothesis: Analysis of Results

These results seemed to confirm the hypothesis that cutting the value function along the "action axis" resulted in less traffic and therefore faster learning. However, I also noticed that not all the learning agents learned at the same pace. I noticed that some learning agents generated their final value function much sooner than other learning agents.

6.7 The Second Hypothesis

I suspected that some learning agents became busier than other learning agents. I proposed some possible explanations:

a) *Some learning agents became very busy answering requests from other learning agents and therefore their performance became poor when compared to the other learning agents that did not need to answer so many requests.*
b) *Also, some learning agents ended up having many of their next states covered by the value function in the same learning agent. This accelerated learning since many "Max Value Function" requests were resolved locally.*
c) *Finally, it was also possible that some learning agents learned very slowly because they spent most of their time waiting for other learning agents to answer their question.*

However, no matter the reason, it was very clear that learning did not occur at the same pace in all the learning agents.

This took us to the following theorem:

Theorem 2: *No matter which problem we choose to solve, it is very likely that learning will not occur at the same pace among all the learning agents. We suspect that we may be able to ameliorate this problem by making "sluggish" learning agents pass some of their load to "speedy" learning agents. When doing this, we recommend that a "sluggish" learning agent pass a small section of its value function (more specifically, a slice along the "action" axis) to a "speedy" learning agent. We suspect that this method will result in faster overall learning.*

To validate or refute this theorem I proposed the two scenarios described in the following section.

6.7.1 The Second Hypothesis: Two Scenarios

In order to validate or refute *Theorem 2*, I proposed to simulate the following two scenarios:

Scenario 3: *The value function was divided into 9 equally sized sections along the "action" axis and each learning agent was given one of these 9 sections. These sections were not modified during the learning simulation.*

Scenario 4: *The Value function was initially divided into 9 equally sized sections along the "action" axis and each learning agent was given one of these 9 sections. These sections were modified during the learning simulation as to try to make all the learning agents learn at the same pace (i.e. by moving value function slices along the "action" axis from "sluggish" learning agents to "speedy" learning agents). We used the number of processed learning experiences as a measure of "sluggishness".*

I suspected that the distributed learning system would perform better under the second scenario but I was not sure.

In order to "experimentally" verify this theory I modeled and simulated the two scenarios using DEVS Java. However, this time I made the state space 25 times larger. The results of these two simulations are described in the following section.

6.7.2 The Second Hypothesis: Results

After simulating *Scenario 3* using DEVS Java, I found that it took the distributed learning agent 52000 time units to learn to balance a pole.

On the other hand, DEVS Java showed that, under *Scenario 4*, it took 42000 time units to learn to balance a pole.

6.7.3 The Second Hypothesis: Analysis of Results

These results seem to confirm the hypothesis that learning at the same pace seems to improve on the overall learning performance (I saw about a 20% improvement over the already improved technique). At first glance, it may appear that this improvement is solely explained by the realization that learning will no longer have to continue until the "slow" learning agents finish learning (using the new technique will make all the learning agents finish learning at about the same time). However, there is one more explanation: "fast" learners may also require information from the "slow" learners in order to update their own value functions. Therefore if this information comes from a "slow" learner, the values supplied will not be frequently updated, which in turn will make the value function of the "fast" learner not as quickly updated as it could be. In other words, "slow" learners slow down "fast" learners. Therefore, getting rid of "slow" learners will have the added advantage that "fast" learners will not be slowed down.

6.8 The Third Hypothesis

As I mentioned during discussions of the first hypothesis, I suspected that lowering traffic would probably improve learning performance. Therefore, I suspected that giving each LA its own local simulator (as opposed to sharing the same global simulator) will probably improve learning performance. This takes us to the following theorem:

Theorem 3: *Giving each distributed LA its own local simulator should improve learning performance significantly.*

To validate or refute this theory I propose the two scenarios described in the following section.

6.8.1 The Third Hypothesis: Two Scenarios

In order to validate or refute *Theorem 3*, I propose to simulate the following two scenarios:

Scenario 5: *A 90x90 value function is divided into 9 equally sized sections along the "action" axis and each learning agent is given one of these 9 sections. There is only one simulator for all the learning agents.*

Scenario 6: *A 90x90 value function is divided into 9 equally sized sections along the "action" axis and each learning agent is given one of these 9 sections. Each learning agent is given its own local simulator.*

I suspect that the distributed learning system will perform better under the second scenario.

In order to "experimentally" verify this theory I modeled and simulated the two scenarios using DEVS Java. The results of these two simulations are described in the following section.

6.8.2 The Third Hypothesis: Results

After simulating *Scenario 5*, using DEVS Java, I found that it took the distributed learning agent 29,000 time units to learn to balance a pole.

On the other hand, DEVS Java shows that, under *Scenario 6*, it took 14,000 time units to learn to balance a pole.

6.8.3 The Third Hypothesis: Analysis of Results

These results seem to confirm the hypothesis that using local simulators improves the performance significantly versus using a single global simulator (I saw more than a 50% improvement over the global simulator technique).

6.9 The Fourth Hypothesis

Since assuming that lowering network traffic improves the learning performance has resulted so fruitful I make use of this premise one more time as a base for the next hypothesis.

This time I changed the way we propagate max value information among the learning agents. Instead of requesting max values, I let each agent have a local list of max values (one for each one of their covered states) and report to everybody if one of these max values increases.

In other words, I would like to replace the poll policy with a push policy. This takes us to the following theorem:

Theorem 4: *Using a push policy instead of a poll policy to propagate max-values among the learning agents should improve learning performance significantly.*

To validate or refute this theory I proposed the two scenarios described in the following section.

6.9.1 The Fourth Hypothesis: Two Scenarios

In order to validate or refute *Theorem 4*, I proposed to simulate the following two scenarios:

Scenario 7: *A 90x90 value function is divided into 9 equally sized sections along the "action" axis and each learning agent is given one of these 9 sections. Each learning agent is given its own local simulator. We use a poll policy to propagate max-values among the learning agents.*

Scenario 8: *A 90x90 value function is divided into 9 equally sized sections along the "action" axis and each learning agent is given one of these 9 sections. Each learning agent is given its own local simulator. We use a push policy to propagate max-values among the learning agents.*

I suspected that the distributed learning system would perform better under the second scenario.

In order to "experimentally" verify this theory I modeled and simulated the two scenarios using DEVS Java. The results of these two simulations are described in the following section.

6.9.2 The Fourth Hypothesis: Results

After simulating *Scenario 7*, using DEVS Java, I found that it took the distributed learning agent 14,000 time units to learn to balance a pole.

On the other hand, DEVS Java shows that, under *Scenario 8*, it took 6,000 time units to learn to balance a pole.

6.9.3 The Fourth Hypothesis: Analysis of Results

These results seem to confirm the hypothesis that using a push policy to propagate max-value info among the learning agents improves performance significantly versus using a poll policy (I saw more than a 50% improvement over the poll policy technique).

6.10 The Fifth Hypothesis

After seeing such positive results, I wondered whether the push policy puts relatively insignificant overhead on the distributed learning system. In other words, if I were to use the push policy and twice as many learning agents, would I get the work done in about half the time?

This takes us to the following theorem:

Theorem 5: *Using again the push policy to propagate max-values among the learning agents and twice the number of learning agents may allow us to cut learning time in almost half.*

To validate or refute this theory I proposed the two scenarios described in the following section.

6.10.1 The Fifth Hypothesis: Two Scenarios

In order to validate or refute *Theorem 5*, I propose to simulate the following two scenarios:

Scenario 9: *A 180x180 value function is divided into 9 equally sized sections along the "action axis" and each learning agent is given one of these 9 sections. Each learning agent is given its own local simulator. We use a push policy to propagate max-values among the learning agents.*

Scenario 10: *A 180x180 value function is divided into 18 equally sized sections along the "action axis" and each learning agent is given one of these 18 sections. Each learning agent is given its own local simulator. We use a push policy to propagate max-values among the learning agents.*

I suspected that the distributed learning system might be able to learn in about half the time under the second scenario.

In order to "experimentally" verify this theory I modeled and simulated the two scenarios using DEVS Java. The results of these two simulations are described in the following section.

6.10.2 The Fifth Hypothesis: Results

After simulating *Scenario 9*, using DEVS Java, I found that it took the Distributed Learning Agent about 24,000 time units to learn to balance a pole.

On the other hand, DEVS Java shows that, under *Scenario 10*, it took about 12,000 time units to learn to balance a pole.

6.10.3 The Fifth Hypothesis: Analysis of Results

These results seem to confirm my hypothesis that using a push policy to propagate max-value info among the learning agents and twice as many learning agents I cut the learning time by about half, which implies that the push policy overhead is not very significant.

6.11 Review

In this study I identified the inherent performance bottlenecks of one type of distributed Q-learning agent. Furthermore, I developed policies that attempted to overcome these bottlenecks. Finally, I simulated a distributed Q-learning system using these different policies and evaluated their performance. For instance, I saw a 50% reduction on learning time by dividing the value function into sections along the "action axis", as opposed to sections along the "state axis". Then I saw a further 20% reduction of learning time by using a policy in which "slow" learners passed slices of the value function sections to "fast" learners. Furthermore, I discovered that by giving each learner its own local simulator (as opposed to a single global simulator), learning time got reduced by about 50%. Then I came up with a "push policy" to replace the normal "poll policy" used by this type of distributed Q-learning system. I found out that using this "push policy" reduced the learning time by almost 60%. Finally I discovered that the "push" policy did not require a lot of overhead, which meant that I could almost decrease the learning time by half by using twice the number of learning agents in the distributed Q-learning

system. This last finding looks very promising since this may allow us to solve really large learning problems by just adding more learners to the system.

CHAPTER 7: SUMMARY

7.1 Dissertation Overview

I started the dissertation by reviewing the development of the cerebellum in animals such as sharks. This coprocessor allowed sharks to improve their movement coordination and become one of the most successful animals around. I also highlighted that the dexterity observed in animals thanks to their cerebellum has become an inspiration for engineers and scientists looking to provide our machines with better motion coordination.

This dissertation introduced the topic of biological cerebellums, also called natural cerebellums. I believe this was necessary in order to better understand the current literature on artificial cerebellums as previous work on this subject makes frequent references to natural cerebellums. This discussion did not go deep into the subject of natural cerebellums but at least touched on some of the most commonly found topics in the literature associated with artificial cerebellums.

After this, I reviewed the most important work on the subject of artificial cerebellums. This was followed by a review, which focused on gathering advantages and disadvantages associated with these methods. With the help of this analysis I determined in what cases previously proposed methods appeared to be a good choice and where there was room for improvement. In cases where there was room for improvement I proposed general techniques, which may help fill up the gap.

The identified shortcomings associated with artificial cerebellum methodology were grouped together into two broad issues: *framework usability* and *building blocks incompatibility*. The *framework usability* issue was tackled by proposing the use of a simple yet powerful

framework to create artificial cerebellums. The *building blocks incompatibility* issue was addressed by using as the engine for this new framework a new method called Moving Prototypes. I argued and illustrated why I believe that this method is more compatible with the currently typically available computers than other methods used in previously proposed frameworks.

I recognized that in order to really grasp the concepts introduced by a new framework I had to put it to work. To accomplish this I decided to implement an artificial cerebellum, which was able to solve a real life problem. The selected problem was the shortage of experienced operators and rubber tires in the mining industry in addition to the high price of fuel. To tackle this problem I developed an artificial cerebellum capable of driving a piece of mining equipment in a way that minimizes the wearing of its rubber tires and its fuel consumption. The selected piece of equipment was a rubber tire front-end loader in charge of loading hauling trucks. To create this artificial cerebellum I made use of the newly proposed framework. This exercise illustrated the use of each one of the steps associated with this framework and also helped clarify some of its concepts. The resulting artificial cerebellum was trained and evaluated using a simple simulator and a visualizer. After this training session the artificial cerebellum appeared to be able to drive the vehicle in a manner very similar to the way very experienced operators do.

In addition, this dissertation recognized that in some situations a single computer might not feasibly implement one of the components associated with the resulting artificial cerebellums, namely one of its learning agents. This may happen when one of these agents requires more memory resources than are available in a single computer. This may also occur when the time allocated to train this agent is not sufficient. To tackle this issue I proposed to break this complex agent into a distributed agent system and train each one its parts in parallel on

separate computers. However, I recognized that in order to make good use of this type of distributed agent system, I needed to first try to understand where its performance bottlenecks were and what I could do to ameliorate these pitfalls. In other words, I need to be able to formally analyze this type of distributed agent system and find its performance bottlenecks. Furthermore, I need to come up with a set of techniques that may address these performance bottlenecks. Finally, I need to formally verify the virtues of each one of these new techniques. One of the first findings by this analysis was that we could achieve a 50% reduction on learning time by dividing the value function into sections along the "action axis", as opposed to sections along the "state axis". Then I saw a further 20% reduction of learning time by using a policy in which "slow" learners passed slices of the value function sections to "fast" learners. Furthermore, I discovered that by giving each learner its own local simulator (as opposed to a single global simulator), learning time got reduced by another 50%. Then I came up with a "push policy" to replace the normal "poll policy" used by this type of distributed Q-learning system. I found out that using this "push policy" reduced the learning time by almost 60%. Finally I discovered that the "push" policy did not require a lot of overhead, which meant that we could almost decrease the learning time by half by using twice the number of learning agents in the distributed Q-learning system. This last finding looks very promising since this may allow us to solve really large learning problems by just adding more learners to the system.

7.2 Future Work

I believe that in the near future, it will be possible to build computers composed of billions of parallel computing units that will rival the computing power of natural cerebellums. Examples of this type of highly parallelized hardware can be found in modern graphics

processing Units (GPUs) [Gpu10]. This type of hardware is a specialized microprocessor that is typically used to accelerate 3D or 2D graphics rendering [Wik10]. However, modern GPUs are programmable and therefore could potentially be used to implement artificial neural networks or other similar methodologies based on natural learning systems. GPUs have been used to model agents [Pau09, Bra10]. Nevertheless, to my knowledge, GPUs have not yet utilized to implement full-blown artificial cerebellums. However, as this type of hardware improves its power and usability, I believe many researchers will create artificial cerebellums based on this hardware platform.

The eventual development of parallelized computer units more powerful than the human cerebellum will make the method Moving Prototypes less attractive. However, I believe that the proposed framework mentioned in this dissertation will still offer a simple and practical way to build artificial cerebellums. The only difference will be that the Q-function will be implemented using a method that is more compatible with the currently available building blocks, in this case a highly parallel computing unit.

Furthermore, we argue that the framework proposed in this dissertation could be used to build a development tool that would provide people with the ability to quickly design and implement customized cerebellums for a wide range of applications. This development tool would not require its users to be experts in the area of Artificial Intelligence, Machine Learning or Neuroscience as many of the currently available frameworks do. In addition, some of the most complex steps associated with the proposed formalism, such as training and optimal policy extraction, could be automatized by the implementation of this development tool. This would allow humans to place our efforts on what we are really good at (i.e. looking at a problem and extracting from it factors which may be significant to its solution and disregarding others which

are probably not helpful).

I propose to put extra effort on making this development tool very easy to use and intuitive. For instance, the user should be able to drag and drop components such as sensors, actuators, master learning agents, etc to a canvas and then join these components any way the user wants as long as the connections are allowed. For instance, the development tool would not allow a signal to start at a learning agent and end at a sensor but it would allow a signal to start at a sensor and end at a learning agent. In other words the development tool itself would make sure the designed artificial cerebellum is well formed. In addition, it may be a good idea for this tool to allow the user to double click on a sensor or an actuator and open a new window in which the details associated with this component are specified. For instance, it would be possible to double click on a sensor and specify what type of hardware will be used to implement it and how its signals will be defined.

In addition, I suggest including an easy to use graphical interface in which the user can quickly specify the mechanical characteristics of the machine we want to control with the artificial cerebellum and perhaps some characteristics of its environment. For instance, the tool may allow the user to specify that the vehicle to be controlled has 6 wheels, weighs 50 tons and has a center of mass position at 30 centimeters above the ground and 2 meters behind its first set of tires. The user should also be able to specify environment characteristics such as friction coefficients between the tires and the floor, etc. This information would be used by any of a set of commercially available simulators already incorporated into the development tool. These simulators would be able to emulate complex mechanical systems.

Finally, I believe this tool should provide the possibility of adding libraries of well-tested and possibly already trained components, which the user may select from a menu to construct

his/her new artificial cerebellum. This may not only make the development cycle shorter by providing components that have already been trained but also more robust as some of the components would have been already thoroughly tested.

References

Alb75 Albus, J.S., *A New Approach to Manipulator Control: The Cerebella Model Articulation Controller (CMAC)*, Transactions of ASME, Journal of Dynamic Systems, measurement and Control, September 1975, pp.220-227.

Albu75 Albus, J.S., *Data Storage in the Cerebella Model Articulation Controller (CMAC)*, Transactions of ASME, Journal of Dynamic Systems, measurement and Control, September 1975, pp.228-233.

Alm03 Paulo E. M. Almeida*, Student Member, IEEE,* and Marcelo Godoy Simões. *Parametric CMAC Networks: Fundamentals and Applications of a Fast Convergence Neural Structure*. IEEE Transactions of Industry Applications. Vol. 39, No 5, September/October 2003.

Bal00 Balac, N., Gaines D. and Fisher D. (2000). *Using Regression Trees to Learn Action Models*. Proceedings of the IEEE Systems, Man, and Cybernetics Conference, Nashville.

Bar02 Barlow, J. S., *The Cerebellum and Adaptive Control*, 1st Edition, 2002, Cambridge University Press.

Bar96 Barto, A. and Crites, R. (1996). *Improving Elevator Performance Using Reinforcement Learning*. In Advances in Neural Information Processing Systems 8 (NIPS8). D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (Eds.), Cambridge, MA: MIT Press.

Bar98 Barto, A. G. and Sutton, R. S. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT press.

Ben08 Bengtsson F. and Jörntell H. *Sensory transmission in cerebellar granule cells relies on similarly coded mossy fiber inputs*. Edited by Masao Ito, RIKEN, Wako, Japan, and approved December 10, 2008 (received for review September 1, 2008).

Bez04 Bezzi M, Nieus T, Coenen O.J-M.D., D'Angelo E. *An integrate-and-fire model of a cerebellar granule cell*. Neurocomputing, 2004; 58-60:593-598.

Bou05 Christian Boucheny, Richard Carrillo, Eduardo Ros and Olivier J.-M.D. Coenen, *Real-Time Spiking Neural Network: An Adaptive Cerebellar Model*, Springer-Verlag Berlin Heidelberg 2005.

Boy04 Boyden ES, Katoh A, Raymond JL (2004). *Cerebellum-dependent Learning: The Role of Multiple Plasticity Mechanisms*. 2004, Annu Rev Neurosci 27: 581–609

Bra10 Brandon G. Aaby, Kalyan S. Perumalla, Sudip K. Seal. *Efficient simulation of agent-based models on multi-GPU and multi-core clusters*. SIMUTools '10 Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques.

Bru04 Brunel N, Hakim V, Isope P, Nadal JP, Barbour B. *Optimal information storage and the distribution of synaptic weights: perceptron versus Purkinje cell*. Neuron. 2004 Sep 2; 43(5):745-57.

Cai06 Caihong Li, Jingyuan Zhang and Yibin Li. *Appplication of Artificial Neural Network Based on Q-learning for Mobile Robot Path Planning*. 2006, Proceedings of the 2006 IEEE International Conference on Information Acquisition August 20 - 23, 2006, Weihai, Shandong, China.

Car06 Ros, E., Carrillo R., Ortigosa, E.M., Barbour B. (2006). *Event-Driven Simulation Scheme for Spiking Neural Networks Using Lookup Tables to Characterize Neuronal Dynamics*. Neural Computation 18, 2959–2993 *Motor Systems and Control Theory.*

Car07 R. Carrillo, E. Ros, S. Tolu, T. Nieus, E. D'Angelo. (2007) *Event-driven simulation of cerebellar granule cells*. Information Processing in Cells and Tissue, 66-75, 2007. (An extended version submitted to the Biosystems).

Dan04 D'Angelo E, Rossi P, Gall D, Prestori F, Nieus T, Maffei A, Sola E. *Long-term potentiation of synaptic transmission at the mossy fiber-granule cell relay of cerebellum*. Prog Brain Res. 2004; 148:69-80.

Dan05 D'Angelo E. *Synaptic Plasticity at the Cerebellum Input Stage: Mechanisms and Functional Implications*. To appear in: Arch Ital Biol.

Dew97 Dewdney, A. K. *Yes, We Have No Neutrons*. John Wiley & Sons, New York, 1997, page 82.

Die96 Dietterich, T. G. and Zhang, W. (1996). *High-performance job-shop scheduling with a time-delay TD(λ) network*. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo (Eds.), Advances in neural information processing systems.

Die99 Dietterich, T. G. and Wang, X. (1999). *Efficient Value Function Approximation Using Regression Trees*. Proceedings of: IJCAI-99 Workshop on Statistical Machine Learning for Large-Scale Optimization, Stockholm, Sweden.

Fin02 Fine EJ, Ionita CC, Lohr L., *The History of the Development of the Cerebellar Examination*. 2002, Semin Neurol 22 (4): 375–84.

Gal05 Gall D, Prestori F, Sola E, Roussel C, D'Errico A, Forti L, Rossi P, D'Angelo E. *Intracellular calcium regulation by burst discharge determines bidirectional long-term synaptic plasticity at the cerebellum input stage*. To appear in: J Neurosci.

Gie86 Gielen CC, van Zuylen EJ. *Coordination of Arm Muscles During Flexion and Supination: Application of the Tensor Analysis Approach*. 1986. Neuroscience 17 (3): 527–39.

Gpu10 http://www.nvidia.com/object/GPU_Computing.html

Her24 Herrick, C.J., *Neurological Foundation of Animal Behavior*. 1st Edition, 1924, Henry Holt and Co.

Hil66 R. Llinkr, D. E. Hillman and W. Precht. *Functional aspects of cerebellar evolution*. *The Cerebellum in Health and Disease. Green* 4. 269-291, St. Louis. 1966

Hin06 Hines, E. L. (2006). *Intelligent Systems Engineering – Major characteristics of Expert Systems and Artificial Neural Networks*. Intelligent Systems Engineering (ES3770) Lecture Notes.

How98 Howe, A. E. and Pyeatt, L. D. (1998). *Decision Tree Function Approximation in Reinforcement Learning*. Technical Report CS-98-112, Colorado State University.

Igo00 Igor N. Aizenberg, Naum N. Aizenberg and Joos Vandewalle, *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications*. 2000, Kluwer Academic Publishers.

Ing08 Ingram JN, Koerding KP, Howard IS & Wolpert DM (2008). *The statistics of natural hand movements*. Experimental brain research vol. 188 (2) pp. 223-36.

Iso04 Isope P, Franconville R, Barbour B, Ascher P. *Repetitive firing of rat cerebellar parallel fibres after a single stimulation*. J Physiol. 2004 Feb 1; 554(Pt 3): 829-39.

Ivr88 Ivry RB, Keele SW, Diener HC. *Dissociation of the Lateral and Medial Cerebellum in Movement Timing and Movement Execution*. 1988, Exp Brain Res 73 (1): 167–80

Jig10 http://www.jigsawtech.com/mineops.html

Jor08 Jörntell H., and Ekerot C. F. *Synaptic Integration in Cerebellar Granule Cells*. The Cerebellum, Springer New York, Vol 7, Number 4, December 2008.

Jun04 Lu Jun, Xu Li, and Zhou Xia0-ping. *Research on reinforcement learning and its application to mobile robot*. Journal of Harbin Engineering University, Vol. 25, No 2, pp. 176-179, April 2004.

Kae96 Kaelbling, L.P., Littman, L.M. and Moore (1996), A.W., *Reinforcement learning: a survey*, Journal of Artificial Intelligence Research, volume 4.

Lag99 Lagoudakis, M. G. (1999). *Balancing and Control of a Freely-Swinging Pendulum Using a Model-Free Reinforcement Learning Algorithm*. Duke University.

Lar67 Larsell, O., *The comparative Anatomy and Histology of the Cerebellum*,

Lee02 Lee, J. and Ni, Jun. *An introduction of Embedded Infotronics Agent for Tether-Free Prognostics*. AAAI Technical Report SS-02-03.

Lee92 Lee, J. and Kramer, B.M., *Analysis of Machine Degradation using a Neural Networks Based Pattern Discrimination Model*, J. Manufacturing Systems, Vol. 12, No. 3, pp. 379-387, 1992.

Lee99 Lee, J. (1999) *Machine Performance Assessment Methodology and Advanced Service Technologies*, Report of Fourth Annual Symposium on Frontiers of Engineering, National Academy Press, pp.75-83, Washington, DC.

Lli91 Llinas R. (1991). *The Noncontinuous Nature of Movement Execution*. 1991. Motor Control: Concepts and Issues, eds. D.R. Humphrey and H.J. Freund, John Wiley & Sons Ltd., Dahlem Konferenzen pp 223-242.

Lon08 Lyle N. Long and Ankur Gupta, *Scalable Parallel Artificial Neural Network*. Journal of Aerospace Computing, Information and Communication, Vol. 5, No 1, January 2008.

Mit97 Mitchell, T. M. (1997). *Machine Learning*. Boston, MA: WCB/McGraw-Hill.

Mod10 http://www.mmsi.com/Frontrunner.html

Pad08 Padois, V. (2008). *Dealing with challenging robots, environments and missions: operational space control in the context of motor learning*. First workshop for young researchers on Human-friendly Robotics, Naples, Italy.

Pan08 Panzer, Heiko; Eiberger, Oliver; Grebenstein, Markus; Schaefer, Peter; van der Smagt, Patrick (2008): *Human motion range data optimizes anthropomorphic robotic hand-arm system design*. MOVIC 2008 (in print).

Pau09 Paul Richmond, Simon Coakley, Daniela M. Romano. *A high performance agent based modeling framework on graphics card hardware with CUDA*. AAMAS '09 Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2.

Pel80 Pellionisz, A., Llinás, R., *Tensorial Approach to the Geometry of Brain Function: Cerebellar Coordination Via a Metric Tensor*. 1980, Neuroscience 5: 1125-1136

Pel82 Pellionisz A, Llinas R., Space-time Representation in the Brain. *The cerebellum as a Predictive Space-time Metric Tensor*. 1982. Neuroscience 7 (12): 2949–70.

Pel85 Pellionisz, A., Llinás, R., *Tensor Network Theory of the Metaorganization of Functional Geometries in The Central Nervous System*. 1985, Neuroscience 16 (2): 245–273

Pel86 Pellionisz, A. *David Marr's Theory of the Cerebellar Cortex: A Model in Brain Theory for the 'Galilean Combination of Simplification, Unification and Mathematization'*. 1986. Palm G., & Aertsen, A.: Brain Theory. Springer Verlag, 253–257.

Phi04 Philipona D, Coenen O.J.-M.D. *Model of granular layer encoding of the cerebellum*. Neurocomputing, 2004; 58-60:575-580.

Pre07 http://prensa.ugr.es/prensa/research/verNota/prensa.php?nota=456

Rap01 Rapp, Brenda. *The Handbook of Cognitive Neuropsychology: What Deficits Reveal about the Human Mind*. 2001. Psychology Press, 48

Ric08 Richard R. Carrillo, Eduardo Ros, Christian Boucheny and Olivier J.-M.D. Coenen, *A Real-time Spiking Cerebellum Model for Learning Robot Control*, 2008,.

Rom06 Simon Romero, *Wanted: Big wheels for mining industry*, International Herald Tribune, April 20 2006.

Ros06 Ros, E., Ortigosa E.M., Agis R., Carrillo R., Arnold M., (2006). *Real-Time Computing Platform for Spiking Neurons (RT-Spike)*. IEEE Transactions On Neural Networks, Vol. 17, No. 4.

Sen09 http://www.sensopac.org/

Sma08 Smagt van der P. and Stillfried G. (2008). *Using MRT data to compute a hand kinematic model*. In Proc. 9th International Conference on Motion and Vibration Control (MOVIC).

Smi98 Russell L. Smith. *Intelligent Motion Control With an Artificial Cerebellum*. Thesis submitted to the Department of Electrical and Electronic Engineering. University of Auckland, New Zealand.

Sot04 Soto, M. (2004). *On-line Q-learner Using Moving Prototypes*. Master of Science Thesis submitted to the Department of Electrical and Computer Engineering. University of Arizona.

Spi06 SpikeFORCE: Real-time Spiking Networks for Robot Control. 2006.

Sri00 Sridharan, M. and Tesauro, G. (2000). *Multi-agent Q-learning and Regression Trees for Automated Pricing Decisions*. ICML 2000: 927-934. Stanford University.

Sut05 Sutter H. (2005), *A Fundamental Turn Toward Concurrency In Software: Your Free Lunch Will Soon Be Over. What Can You Do About It?*. Dr.Dobb's Journal, 30(3), pp. 16-22.

Tes00 Sridharan, M. and Tesauro, G. (2000). *Multi-agent Q-learning and Regression Trees for Automated Pricing Decisions*. ICML 2000: 927-934. Stanford University.

Tes95 Tesauro, G. (1995). *Temporal Difference Learning and TD_GAMMON*, Communications of the ACM, Volume 38, Number 3.

Thi09 Thibault J.C. and I. Senocak (2009), *CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows*. 47th AIAA Aerospace Sciences Meeting, January 5-8, 2009, Orlando, pp. AIAA 2009-758: 1-15

Thr96 Thrun, S. (1996). *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Boston, MA: Kluwer Academic Publishers.

Vij00 Vijayakumar, S. and Schaal, S. (2000), *Locally Weighted Projection Regression: An O(n) algorithm for incremental real time learning in high dimensional space*, In Proceedings of the International Conference In Machine Learning (ICML 2000), pp. 1079-1086, Stanford (USA).

Wat89 Watkins, C. (1989). *Learning from delayed rewards*. Ph.D. Thesis, Cambridge University, Cambridge, UK.

Wik10 http://en.wikipedia.org/wiki/Graphics_processing_unit