INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Towards Real Learning Robots

Getachew Hailu

Bericht Nr. 9906

Oktober 1999

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Towards Real Learning Robots

Getachew Hailu

Bericht Nr. 9906
Oktober 1999

e-mail: gha@informatik.uni-kiel.de

Dieser Bericht gibt den Inhalt der Dissertation wieder, die der Verfasser im Mai 1999 bei der Technischen Fakultät der Christian-Albrechts-Universität zu Kiel eingereicht hat.
Datum der Disputation: 7. Juli 1999

1. Gutachter	Prof. Dr. Gerald Sommer
2. Gutachter	Prof. Dr. Werner Kluge
3. Gutachter	Prof. Dr. Gerd Pfister
Datum der mündlichen Prüfung:	7. Juli 1999

Abstract

The subject of this thesis is learning in a large and continuous space with a physical robot. In so doing, it focuses on: processing of the perceptual apparatus, questing of the right type of bias, and learning of the task with the real robot.

Task based multi-sensor processing method that combines two computing components: a **hashing technique** and a **fusion method** is proposed. The components operate in tandem. The hashing technique collapses the perceptual space by re-grouping the sensors into clusters to abstract away important control details and to match to the complexity of the task. Sensors that measure the same object form a cluster. The fusion method integrates measurements gathered over an extended time period through the Kalman filter algorithm. The proposed processing method is tested in two experiments that involve an off-line and an online robot navigation problems and the results are compared with other existing processing techniques.

The influence of the amount and quality of bias on the speed of reinforcement learning is studied for the first time. For a chosen class of learning problem different forms of biases are initially identified. Some of the bias are extracted from the knowledge of the environment, others from the task, and yet a few from both. Belief matrices, which reset Q-tables before learning commence, encode the biases. The average number of interactions between the agent and the environment is used to quantify the biases. Based on this performance measure the biases are graded and new results are reported.

The result obtained from the earlier bias analysis is utilized to design two bias forms that make learning possible on a physical robot. The first bias pre-labels the world into appropriate structures to diminish the learning space. The second bias is a collection of simple reflex-like rules that focuses exploration. The robot faces a delayed reinforcement learning task where a global minimum cost path is sought in a large continuous space. Learning is structured around the value function—TD method is used to estimate the values of taking actions in states and actions are extracted from state values. The result asserts that **by introducing sufficient bias it is possible to realize reinforcement learning on a real robot.**

Acknowledgments

Life has its peak moments that seem to stand separate from and above all others. Seeing this dissertation cross the finish line is such an occasion for me.

I am by no means the only one to take credit for the work leading to the results presented here. There are so many individuals whom I owe a special debt for contributing to the four great years I have had at the Christian Albrechts University (CAU) of Kiel, and without whom this thesis would never have been brought about.

Unlimited thanks go to Prof. Dr. Gerald Sommer for being my advisor and an excellent motivator. He has been a major source of inspiration throughout the work. Most of all, I am thankful that he spent so much time with me, sharing his ideas about learning philosophy. I am very grateful for his helpful suggestions, encouragement, and unreserved support.

Much gratitude is due to Dr. Jörg Bruske for being a supportive officemate, a wonderful friend, and a source of great discussion. I am indebted to him for proof reading my thesis, initiating me into the culture of Kiel, such as surfing. My indebtedness also goes to the entire cognitive system group for providing me with a very inspirational and helpful environment throughout the entire thesis conception, implementation and documentation process.

Wholehearted and deepest thanks to Dr. Jose Millán at the Joint Research Center laboratory of the European Commision (JRC) for showing interest in my work and providing me with numerous pieces of invaluable advice in the field of reinforcement learning. Particularly, Dr. Millán has been very instrumental in my thinking of TESEO's architecture. Furthermore, he has directly contributed to some of the ideas presented in chapter 5.

Deepest thanks to Prof. Dr. habil. Klaus Kißig without whom I could not have had the opportunity to come to Kiel and began this work. My warmest thanks to Mr. Rex Slate and Mrs. Leslie Slate for patiently reading the early drafts of my thesis and fixing the bunches of flaws in my usage of English. My sincere affection goes to them for their parental character and true fellowship, too. I wish to thank dozens of international students—from Finland to Thailand—with whom I have had splendid interactions and have shared exciting indoor and outdoor activities. Undue appreciation and respect goes to the outskirts of Kiel for offering me an enjoyable landscape and a refreshing Baltic sea air so that I could wind-up my long and hectic days in the laboratory with a pleasant jog—behind me, I leave indelible memories of Kiel and its environs.

Infinite and incomparable gratitude is extended to the Gullasch family, the most wonderful family I have ever met. Many many thanks for the accommodation I received way back in the days of my language course in Bremen. I am equally thankful for the afterward unabated invitations and untold support. But all these add up to nothing compared to what the family has imparted to me to make a personal life-deciding choice.

Finally, enduring thanks and praise go to my Heavenly Father who saw me through when the bottom seemed to drop out at the times of hardware frustrations and battling temptations. Surely, He will lead me to a large field where I may harvest the fruits of my labors . . . the cupcakes are waiting me.

The research presented in this thesis was sponsored by the German Academic Exchange Service under grant code R-413-A-94-01537, which is duly acknowledged.

Contents

List of Figures

Chapter 1

Introduction

I can not say what intelligence is, but I know it when I see it.

Stewart

1.1 Thesis Overview

Intelligent systems naturally have to contain multiple sensors to perceive their environments. Multi-sensor based intelligence has been studied extensively in the context of autonomous mobile robots and there are numerous papers in the literature suggesting various type of controllers; some of which are: potential field [61], behavior based [17, 116], and fuzzy controller [77]. But for most part, the control of mobile robots has escaped the attention of sensor processing. It has become clear that intelligence and flexibility of future robots will come in large part from effective processing of sensory information and the integration of this information into useful motor actions [2]. Therefore, one of the goals of this thesis is to address the problem of task based multi-sensor processing. In particular we ask: are there mechanisms we can successfully apply to extract workable, if not accurate, descriptive information from multiple and uncertain sensors?

The majority of learning research in artificial intelligence, computer science, and psychology has studied the case in which the agent begins with no knowledge at all about what it is trying to learn. In some cases it has access to examples presented by its experience; nevertheless, although this is an important case it is by no means the general case. Most human learning takes place in the context of a good deal of background knowledge of the world. Some studies even claim that newborn babies exhibit knowledge of the world [111]. Whatever the truth of this claim, there is no doubt that prior knowledge helps learning enormously. The approach taken in recent years has been to design agents that already know something and are trying to learn some more. The second goal of the thesis goes beyond this approach. Instead of simply biasing and learning a task, we ask the more fundamental questions—how do we even bias the learning system or what types of information should be learned and what should be built-in? In other words, we are questing for biases that make learning not only possible but also efficient.

Reinforcement learning is especially attractive for problems in robotics, as it is often told that the creator of the robot does not have enough knowledge to program or teach the robot everything it needs to know. Reinforcement learning allows the programmer: to set a goal for the robot, to tell it whatever can be easily taught, and to let the robot discover the details on its own accord by adapting to the changing world. This is all speculative stuff; as of yet there has been scant demonstrations in the way of success in combining reinforcement learning and robots. The third goal of the thesis is aimed at applying reinforcement learning to a real robot. Is it possible to implement a self-learning robot based on a reinforcement signal? What are the appropriate means of investigating the architecture? These are the questions we will try to answer in the last part of this thesis.

1.2 Thesis Contributions

Having stated the challenges this thesis tries to address, we will now briefly describe the experiments and summarize the responses to the various challenges.

First and foremost, a mobile robot must have an accurate perception of its world. Without this prerequisite no sensor-based robot motion control is possible, no learning or adaptation can take place, and no behavior can be observed. In response to this challenge, we propose a sonar processing method and validate the method through a set of experiments. The method consists of a hashing technique and a fusion process working in tandem. The hashing technique collapses the huge raw sensory data so that it can represent only the conditions needed to be dealt with by the robot; whereas the fusion process, implemented as a Kalman filter, fuses the range returns gathered from the sonar sensors over an extended period of time into a consistent unified perception of the state of world. The method is able to:

- show a good success, even when some of the assumptions made on the Kalman filter are not satisfied,
- deliver a usable value that is far better than the existing processing methods, and
- work online with the fuzzy controller to control the robot for a long period of time with obstacles appearing and disappearing rapidly.

Appropriate biases, supplied by a human programmer or a teacher, are able to lever complex reinforcement learning problems, but the problem is in finding these biases. For a particular class of problems, we search for an appropriate bias from a set of biases. Some of the biases are derived from the environment, others directly from the task, and yet a few from the insight of the problem and the environment. An index that measures the appropriateness of each of the bias is computed from the knowledge of the average number of interactions between the agent and the environment. Based on this index it is found out that,

- some biases speed up learning faster than others,
- biases that simply cut the search space may not always be the best choice, sometimes insight biases perform better even with a large search space, and
- there is a danger when more biases are introduced into the system, the learning system may miss the optimal and settle to a sub-optimal solution.

Reinforcement learning is a weak learning method that presents unreasonable difficulties, especially when applied to real robot tasks. The final experiment aims at making this method, through proper and sufficient biasing, possible on a real robot. Two kinds of biases are introduced to the learning system. The unstructured input utilization of the traditional reinforcement learning approach is eliminated by giving the robot domain knowledge that determines how much the robot knows about different portions of its state space. Moreover, in order to make the robot to stay operational right from the beginning a reflex bias is introduced [88, 97]. Reflex bias covers the state space of the robot with vector fields, some of which are feasible with respect to the goal [47]. The task of the robot has many interesting characteristics which are unknown and must be learned during the trials. After the robot has been given internally and externally motivated needs and placed in the world, it is able to:

- explore safely on its own possible alternative trajectories,
- learn the necessary skills *only* with a handful of trials, and
- retain the learned skill in subsequent trials.

Having introduced the subjects of this thesis, we will present in the rest of the chapter the foundational theories on which the thesis is based upon. We begin by reviewing the application of the Kalman filter in multi-sensor integration. The emphasis is particularly on signal level integration and when the integration is over time. We then go on to explain why learning is useful and elaborate the issues associated with learning. We define neural networks as computational entities that capture human heuristics, and present the important network structures and training algorithms. Based on the available feedback, learning paradigms are classified into three classes—one of which is reinforcement learning, our subject. Though the state of the art of reinforcement learning will be given in chapter 3, some aspects of it will be briefly touched upon. Finally, the thesis is outlined.

1.3 Sensor Fusion

Without sensors, robots are merely open-loop machines incapable of acting to changes in their environment. The use of sensors in a robot is an acknowledgment of the fact that it may not be possible or feasible for a robot to know a priori the state of the outside world to a degree sufficient for its autonomous operation. The diversity of information needed by a robot to learn about its environment requires that it be equipped with multiple sensors. Other motivation for using multiple sensors in a robot system can be considered as a response to the question: if a single sensor can increase the capability of the robot, would the use of more sensors increase it even further?

Some of the most complex operations in robotics are those involving navigation and manipulation. In addition to the need for non-contact sensors, such as vision, range, and proximity, in order to recognize objects to be manipulated, there is an evident need for contact sensors, such as force and tactile sensors. With this diversity of sensing modalities, comes the need for a system that is capable of acquiring and fusing data from these different sensors to yield an intelligent interpretation of the scene. The sheer amount and diversity of data warrants the fusion of multi-sensory data [1, 70].

There are a number of different means of integrating the information provided by multiple sensors into the robot operation. The information to be used may come from multiple sensory devices during a single period of time or from a single sensory device over an extended period of time. The most straightforward approach is to let the information from each sensor serve as a separate input to the robot controller. This approach may be the most appropriate if each sensor is providing information concerning completely different aspects of the robot environment. The major benefit gained through this approach is the increase in the scope of the environment that can be sensed.

However, if there is some degree of overlap between the sensors, it may be possible for a sensor to directly influence the operation of another sensor so that the value of the combined information that the sensors provide is greater than the sum of the information provided separately by each sensor. The information from the sensors can be fused at a variety of levels of representation. One of the methods used to integrate sensor values at the signal level is the *Kalman filter*.

Fusion through Kalman Filtering

The Kalman filter [84] can be used in a number of multi-sensor systems whenever it is necessary to fuse dynamic low-level redundant data in real-time. The filter uses the statistical characteristics of a measurement model to recursively determine the estimate of the fused data. If a system can be described with a linear model and both the system and the measurement error can be modeled as white Gaussian noise, the Kalman filter will provide the unique statistical optimal estimates for the fused data [85]. Examples of the use of the filter for multi-sensor fusion includes object motion recognition, robot navigation, and target tracking.

The measurement from a group of n sensors can be fused using a Kalman filter to provide an estimate of the current state of a system and a prediction of the future state of the system. The state being estimated may, for example, correspond to the current location of a mobile robot, the position and velocity of an object in an environment, or the actual measurements themselves. Given a system represented as a linear discrete Markov process, the state space model

$$\boldsymbol{x}(t+1) = \boldsymbol{\phi}(t)\, \boldsymbol{x}(t) \;+\; \boldsymbol{B}(t)\, \boldsymbol{u}(t) \;+\; \boldsymbol{G}(t)\, \boldsymbol{w}(t) \tag{1.1}$$

and the measurement model

$$\boldsymbol{z}(t) = \boldsymbol{H}(t)\, \boldsymbol{x}(t) \;+\; \boldsymbol{v}(t) \tag{1.2}$$

can be used to describe the system, see figure 1.1. Here $\boldsymbol{x}$ is an m state vector, $\boldsymbol{\phi}$ is an $m \times m$ state transition matrix, $\boldsymbol{B}$ is an $m \times p$ input transition matrix, $\boldsymbol{u}$ is a p input vector, $\boldsymbol{G}$ is an $m \times q$ process noise transition matrix, $\boldsymbol{w}$ is a q process noise vector, $\boldsymbol{z}$ is an n measurement vector, $\boldsymbol{H}$ is an $n \times m$ measurement matrix, and $\boldsymbol{v}$ is an n measurement noise vector.

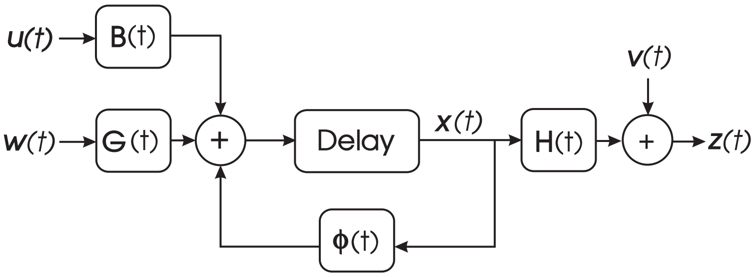


Figure 1.1: *The Kalman filter block diagram.*

The vectors $\boldsymbol{w}$ and $\boldsymbol{v}$ are uncorrelated discrete-time, zero-mean, and white Gaussian noise sequences with covariance kernels,

$$E\{\boldsymbol{w}(t_i)\ \boldsymbol{w}^T(t_j)\} = \boldsymbol{Q}(t_i)\delta_{ij} \tag{1.3}$$

$$E\{\boldsymbol{v}(t_i)\ \boldsymbol{v}^T(t_j)\} = \boldsymbol{R}(t_i)\delta_{ij} \tag{1.4}$$

where $E\{\dots\}$ denotes the expectation operator and δ_{ij} is the Kronecker delta function. If all the parameters of the model are known the optimal filtering equations are:

$$\hat{\boldsymbol{x}}(t|t) = \hat{\boldsymbol{x}}(t|t-1) + \boldsymbol{K}(t)\ [\boldsymbol{z}(t) - \boldsymbol{H}(t)\ \hat{\boldsymbol{x}}(t|t-1)] \tag{1.5}$$

$$\hat{\boldsymbol{x}}(t+1|t) = \boldsymbol{\phi}(t)\ \hat{\boldsymbol{x}}(t|t) + \boldsymbol{B}(t)\ \boldsymbol{u}(t) \tag{1.6}$$

where $\hat{\boldsymbol{x}}(t|t)$ is the estimate of $\boldsymbol{x}(t)$ based on the measurements $\{\boldsymbol{z}(0), \dots, \boldsymbol{z}(t)\}$ and $\hat{\boldsymbol{x}}(t+1|t)$ is the prediction of $\boldsymbol{x}(t+1)$ based on the measurements $\{\boldsymbol{z}(0), \dots, \boldsymbol{z}(t)\}$. The $m \times n$ matrix $\boldsymbol{K}$ is the Kalman filter gain and is defined as,

$$\boldsymbol{K}(t) = \boldsymbol{P}(t|t-1)\ \boldsymbol{H}^T(t)\ [\boldsymbol{H}(t)\ \boldsymbol{P}(t|t-1)\ \boldsymbol{H}^T(t) + \boldsymbol{R}(t)]^{-1} \tag{1.7}$$

where $\boldsymbol{P}(t|t-1) = E\{[\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t|t-1)]\ [\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t|t-1)]^T\}$ is an $m \times m$ conditional covariance matrix of the error in predicting $\boldsymbol{x}(t)$ and is determined using

$$\boldsymbol{P}(t+1|t) = \boldsymbol{\phi}(t)\ \boldsymbol{P}(t|t)\ \boldsymbol{\phi}^T(t) + \boldsymbol{G}(t)\ \boldsymbol{Q}(t)\ \boldsymbol{G}^T(t) \tag{1.8}$$

where $\boldsymbol{P}(t|t) = \boldsymbol{P}(t|t-1) - \boldsymbol{K}(t)\ \boldsymbol{H}(t)\ \boldsymbol{P}(t|t-1)$. The initial condition for the recursion are given by $\hat{\boldsymbol{x}}(0|0) = \hat{\boldsymbol{x}}_0$ and $\boldsymbol{P}(0|0) = \boldsymbol{P}_0$.

An Example

The application of Kalman filtering for multi sensor fusion can be illustrated using a one dimensional position estimation problem [85]. Let us assume that we are lost at sea during a night and have no idea at all of our location. Suppose we observe two sensors, a star and a light beacon to estimate our position. The position measurement z_1 from the star and z_2 from the light beacon can be modeled as, $z_1 = x + v_1$ and $z_2 = x + v_2$ respectively, where v_1 and v_2 are independent zero-mean Gaussian random variables with standard deviation σ_1 and σ_2. We assume that $\sigma_2 < \sigma_1$, since the measurement of the light beacon would be more accurate than that of the star. If the two measurements are available simultaneously, batch processing can be used for fusion, where

$$\boldsymbol{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} x + \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \boldsymbol{H}x + \boldsymbol{v} \tag{1.9}$$

If the measurements are available sequentially, recursive processing can be applied to update the estimate of x as new measurements become available. Assuming that the measurement from the star is available initially, $\hat{\boldsymbol{x}}_0 = \hat{x}_0 = z_1$ and $\boldsymbol{P}_0 = P_0 = \sigma_1^2$ can be considered the a priori information available about x before the receipt of the measurement from the light beacon. When z_2 becomes available the optimal estimate, equation (1.5), of x is given by,

$$\begin{aligned} \hat{x}_1 &= \hat{x}_0 + K[z_2 - H\hat{x}_0] = \hat{x}_0 + P_0 H^T (H P_0 H^T + R)^{-1} [z_2 - H\hat{x}_0] \\ &= z_1 + \sigma_1^2 (\sigma_1^2 + \sigma_2^2)^{-1} [z_2 - z_1] = \frac{1}{\sigma_1^2 + \sigma_2^2} (\sigma_2^2 z_1 + \sigma_1^2 z_2) \end{aligned} \tag{1.10}$$

where $R = \sigma_2^2$. It is interesting to note that σ_1 and σ_2 are used as a means of weighting each measurements so that the measurement with the least variance, i.e., z_2 of the light beacon, is given the greatest weight in the fused estimate. The variance of the estimate, equation (1.8), is given by,

$$\begin{aligned} \sigma &= \sigma_1^2 - P_0 H^T (H P_0 H^T + R)^{-1} P_0 \\ &= \sigma_1^2 - \sigma_1^2 (\sigma_1^2 + \sigma_2^2)^{-1} \sigma_1^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2} \end{aligned} \tag{1.11}$$

From equation (1.11) it is easy to see that the variance of the estimate is less than either σ_1 or σ_2. In other words, the uncertainty in our estimate of position has been decreased by integrating the two pieces of information. Thus, even poor quality data provides some information and increases the precision of the filter output.

1.4 Learning vs. Wiring

Although many advances have been made in conventional control and artificial intelligence, neither approach seems capable of realizing autonomous operation in robots. That is, neither can produce robots capable of interacting with the world with an ease comparable to that of humans or at least higher animals. Conventional robots rely upon humans to pre-wire and explicitly supply appropriate commands for all possible and foreseeable situations. Robot systems based upon these methodologies are restricted to operation in well known, easily defined, highly structured, and static environments. Any deviation from this initial development condition results in an unrecoverable failure.

As robots are expected to operate in unstructured and dynamic environments, control problems arise whose solutions are not intuitively obvious, difficult to foresee, or impossible to pre-wire. For a robot to operate in such challenging environments, one could attempt to determine and explicitly supply all possible skills or behaviors of which the machine is capable, regardless of what skills are relevant. This method, in addition to being cumbersome, assumes that the robot will perform identically in both the developmental and operational environments and that it will be possible to predict and account for all the environmental changes that the robot may undergo during its operation life. This represents an extremely difficult and non-realistic approach which simply does not work.

Why Learning?

A different approach would have been a robot learns only the skills or behaviors it finds useful, and then adapts to changes, if and when they occur. The robot would adjust its operating characteristics based upon the current environment in which it finds itself and the specific task(s) required of it. Such self-learning robots would be more flexible and capable of functioning in a broader range of conditions than those using the conventional approach. To realize such intelligent robots, one would require developing and implementing the principles of self-exploration and self-learning robot control systems. In short, learning has two main benefits; it is used to [58, 82]:

- adapt to external and internal changes, and
- simplify genetic materials or built-in knowledge.

What to Learn?

Having decided that the agent must learn about its environment, we must decide exactly what the agent is to learn. According to what is being learned, existing approaches can be classified into two groups [58]:

- learning declarative knowledge, and
- learning functional knowledge.

The only type of declarative knowledge that the agent can learn is the map of the environment [79, 132]. Maps or world models are closely tied to action in the world. That is why it is a primary type of declarative knowledge. If the agent could learn the map or the model of the environment, it would have enough information to satisfy any task within its physical abilities. In order to take action, however, it needs the task description and then uses that description, together with the learned world model, to decide what to do. This process will, in many cases be reduced to the general planning problem, which has been shown intractable in the majority of cases [23]. Another option is to compile the world model and the task description into an action mapping that can be executed efficiently. If the task changes, the map needs only be recompiled [58].

The other approach and the one pursued here, is to learn the functional mapping from the perceptual states to effector actions. Since everything a robot learns must eventually be connected to the way it acts, a functional representation about the world is more desirable. Instead of converting the information from the world into an abstract map and then re-converting it back into actions, the structure to be learned can be represented as control law (*policy*). Unfortunately, this functional mapping is tailored for a particular task, so that if the task changes, an entirely new mapping must be learned [82].

What to Learn from?

Whether the agent is learning the action map directly or is learning the map of the environment, it must observe and store information from the experience it has in the world. In addition to learning how the world works, the agent must learn what its task is. There are two methods, that will cause the agent to learn to carry out a particular task at a particular time. One method is to provide a teacher, so that the agent after learning would behave in the same way as the teacher, even when the teacher is no longer present. Another method is to provide a reinforcement value. This is basically a mapping from each state of the environment into a scalar value, encoding how desirable that state is for the agent. The agent's goal, in this case, would be to take action that would maximize the received reinforcement value [58].

1.5 Learning in Neural Networks

Heuristics

Most complex problems require the evaluation of an immense number of possibilities to determine an exact solution. The time required to find an exact solution is often more than a life time. Heuristics play an effective role in such problems by indicating a way to reduce the number of evaluations and to obtain solutions within a reasonable time constraint. Heuristics are criteria, methods, or principles for deciding which among several alternative courses of actions promises to be the most effective in order to achieve some goal. They compromise between two requirements: the need to make such criteria simple and at the same time, the desire to see them discriminate correctly between good and bad choices.

A heuristic may be a rule of thumb that is used to guide one's action. For example, a popular method of choosing ripe cantaloupe involves pressing the spot on the selected cantaloupe where it was attached to the plant, and then smelling the spot. If the spot smells like the inside of a cantaloupe, it is most probably ripe. This rule of thumb does not guarantee choosing only ripe cantaloupe, nor does it guarantee recognizing each ripe cantaloupes judged, but it is effective most of the time [101].

It is often said that heuristic methods are unpredictable; they work wonders most of the time, but may fail miserably some of the time. Some heuristics greatly reduce search effort but occasionally fail to find an optimal or even near optimal solution. Others work efficiently on most problems in a given domain; yet, a rare combination of conditions and data cause the search to continue forever. Much of the excitement about artificial neural networks revolves around the promise to avoid this tedious, difficult, and generally expensive process of articulating heuristics and rules for machines that are to perform difficult tasks [67].

Neural Networks

A neural network is composed of a number of nodes or units, connected by links. Each link has a numeric modifiable weight. Some of the units are connected to the external environment, and can be designated as input or output units. Each unit has a set of input links from other units, a set of output links to other units, a current activation level, and a means of computing the activation level at the next step in time. The main idea of computing in a neural network is that every unit does a local computation based on inputs from its neighbors, without the need for any global control over the set of units as whole. A learning algorithm modifies the weights to bring the network's input and output characteristic more in line with that of the environment that provides the inputs.

Weights are the primary means of long-term storage in neural networks. Storing information in network's connection weights via a learning process is a more general version of storing information in a look up table. Accessing information from a network is a relatively shallow computation that can be accomplished in roughly the same amount of time for all input data; it can replace a deep sequential computation whose duration varies with the input data. Consequently, artificial neural networks, should have great utility in application such as control, where real-time operation is essential [52].

Network Structures

There are a variety of network structures that result in different computational properties. But, the main structural distinction to be made is between *recurrent* and *feed-forward* networks. Recurrent networks have arbitrary topologies with a rampant direct or indirect back-connections. Because activation is fed back to the units that caused it, recurrent networks have internal states stored in the activation levels of the units. This also means that computation can be much less orderly than in feed-forward networks. Furthermore, recurrent network can become unstable or exhibit chaotic behavior. Given some input values, it can take a long time to compute a stable output, and learning is made more difficult [111]. Hopefield networks and Boltzmann machines are the two best-understood classes of recurrent networks.

Instead of recurrent networks, we are usually dealing with networks that are arranged in layers. In this type of networks, each unit is linked only to units in the next layer, there are no links between units in the same layer, no links backward to a previous layer, and no links that skip a layer. This simple connections of units enable computation to proceed uniformly from input to output units. By now, the mechanics of feed-forward neural networks and their gradient descent algorithm are quite well known, but a brief description introducing terms and notations can not hurt.

Perceptrons

Layered feed-forward networks or perceptrons were studied in detail by Rosenblatt some 30 years ago [108]. Although networks of all size and topologies were examined, the only effective learning network at that time was the single-layer network, so that is where most of the effort was spent. Today the name perceptron is used as a synonym for a single layer feed-forward network.

Suppose we want to train a perceptron network to produce a desired state in the output units for each member of a set of input vectors. A measure of how poorly the network is performing with its current set of weight is:

$$E = \frac{1}{2} \sum_{\mu} \sum_{j} (y_j^{\mu} - d_j^{\mu})^2 \tag{1.12}$$

where y_j^{μ} is the actual state of output unit j for the input output case μ, and d_j^{μ} is the desired state. We can minimize the error measure given in equation (1.12) by starting with any set of network weights and repeatedly changing each weight by an amount proportional to the negative gradient of the error with respect to the network weight, i.e.,

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \sum_{\mu} \frac{\partial E}{\partial y_j} \frac{dy_j}{dx_j} \frac{dx_j}{dw_{ji}} = \eta \sum_{\mu} (d_j - y_j) \frac{dy_j}{dx_j} \, y_i \tag{1.13}$$

Note that the index μ has been suppressed for clarity. Here w_{ji} is the weight of the connection from the i-th input unit to the j-th output unit, and x_j is the combined effects of the rest of the network on the j-th unit, i.e., $x_j = \sum_i w_{ji} \, y_i$. The relation between x_j and y_j is defined by the unit's activation function. When the activation function is linear, the term dy_j/dx_j is a constant. However, the most useful activation is the sigmod function $y_j = g(x_j) = (1 + e^{-x_j})^{-1}$. Equation (1.13) is guaranteed to find the set of weights that give the least mean square (LMS) error.

Multi-layer Networks

Multi-layer networks are feed-forward nets with one or more layers of nodes between the input and output nodes. These additional layers contain hidden units that are not directly connected to the outside world, neither input nor output. Though multi-layer networks were invented at the same time as perceptrons, due to the lack of effective training algorithms they had not been in use until the arrival of a back-propagation training algorithm. Back-propagation algorithm was invented by [110] and remained central to much current work on learning in neural networks; but a similar idea has been developed earlier by [99, 138]. It is a generalization of the least square algorithm, equation (1.13), that can compute more complicated functions than networks that lack hidden units.

In multi-layer network it is possible to compute $\partial E/\partial w_{ji}$, using equation (1.12), for all the weights in the network provided we can compute $\partial E/\partial y_j$ for all the units that have modifiable incoming weights. In a system that has no hidden units, this is easy because the only relevant units are the output units, and for them $\partial E/\partial y_j$ is found by differentiating the error function. But for hidden units it is harder to compute the error derivative.

The idea in back-propagation is that these derivatives can be computed efficiently by starting with the output layer and working *backwards* through the layers. For each input output case, we first use the forward pass, starting at the input units, to compute the activity levels of all the units in the network. Then we use a backward pass, starting at the output units to compute $\partial E/\partial y_j$ for all the hidden units. For the hidden unit i at a certain layer, the only way it can affect the error is via its effects on the unit j in the immediate upper layer. So we have,

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{dy_j}{dx_j} \frac{dx_j}{dy_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{dy_j}{dx_j} w_{ji} \tag{1.14}$$

where again the index μ is ignored. If $\partial E/\partial y_j$ is already known for all units in the upper layer, it is easy to compute the same quantity for all units immediately beneath this layer.

However, unlike the simple LMS algorithm, the convergence of the back-propagation can not be proven hence not guaranteed to find the minimum error. Nevertheless, it has been shown successful in many areas of interest: sonar target recognition [40], inverse kinematics [66], pattern recognition [28], and automatic control and navigation of mobile robots. Two good examples in mobile robots are Pomerleau's work [103] wherein he used a multi-layer perceptron on NAVLAB vehicle and a video camera (mounted on the roof of the vehicle) to control the vehicle's steering direction on a winding road, and Hailu's work [43] wherein he used the same type of network on a TRC robot equipped with ultrasonic sensor to determine both the steering angle and velocity in an indoor environment.

Radial-basis Function Networks

Yet another example of a feed-forward network structure is a radial basis function (RBF) network [93]. Radial-basis function networks have Gaussian activation functions whose output is determined by the *distance* between the input vector $\boldsymbol{x}$ and a prototype vector $\boldsymbol{\mu}_i$, i.e.,

$$g_i(\boldsymbol{x}) = \phi_i(\boldsymbol{x}) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu}_i||^2}{2\sigma_i^2}\right) \tag{1.15}$$

Each input unit evaluates the kernel function on the incoming input and the output of the j-th unit in the output layer is simply a weighted linear combination of the kernel functions, i.e., $y_j = \sum_i w_{ji}\ \phi_i(\boldsymbol{x})$. Associated with every input unit are the prototype vector $\boldsymbol{\mu}_i$ and the variance σ_i, whereas with the input units to output units are conventional signal multipliers w_{ji}. The network is trained in two training procedures. In the first stage, the optimal coverage of the input space is sought by training the input unit parameters from the input data set alone. In the second stage, these parameters are kept fixed and the multiplier weights are trained. This idea of a layer by layer training without having the input signal going through multiple layers makes training of an RBF network appealing.

In the first stage, the parameters governing the Gaussian functions are determined using a relatively fast competitive learning algorithm. The second stage that involves the training of the multiplier weights requires the solution of a linear problem. In general, competitive learning divides a set of input vectors into a number of disjoint clusters such that the input vectors in each clusters are similar to one another. It is called competitive because there are a set of input units which compete with one another to become active. There are variations of this same basic idea, one of which is the *self organizing feature map* [62].

The self organizing feature map trains the input units of RBF network by mapping a higher dimensional vector space of the input signal onto a usually two dimensional, discrete lattice of formal neurons [31, 62]. The map contains a close correspondence between the input signal and the neurons in the lattice, such that the neighborhood relationships among the input are reflected as faithfully as possible in the topological arrangements of the neurons in the lattice. To create the map, the neuron responding most to an input signal is identified in the lattice. That neuron and its neighbors change their weights of connections, such that the neighbors now respond more like the particular neuron than they did before. The result is a map with neighborhoods possessing similar response properties and showing a smooth and gradual change from one to another [52].

Model Estimation

A major goal of connectionist learning is to produce networks that correctly generalize new cases after training on a sufficiently large set of typical cases of some domain. Up to now we have focused on the minimization of an error function as the basic technique for determining values of the network parameters. Unfortunately, network parameters that give the smallest error with respect to the training data do not have good prediction or generalization capability for new data. To a large extent, the heart of the generalization problem lies in the number of adjustable parameters or equivalently, in the optimum size of the network. Thus in order to achieve a good generalization, we need to optimize the network model. Model estimation is perhaps one of the major challenges in neural networks, and one that usually limits the practical application of neural networks.

A fixed model or network architecture, say a fixed radial basis function units, either may be too small to represent the problem or may involve excessive units that result in poor generalization. One approach for model estimation is to search through a restricted class of network architectures and select the model from that class that has the best generalization performance. This approach requires all the network models in that class to be trained and their generalization performance to be evaluated before the best network is chosen. Thus, exhaustive model search, though most widely adopted, involves significant computational effort [14].

An alternative approach is to consider a network which initially is small, but allows new units and connections to be added during the course of learning. Generally, techniques of this form came under the name *growing algorithms* and various network topologies and training algorithms have been proposed [4, 21, 78]. These algorithms address the issue of completeness, efficiency and generalization and have been in use in a variety of applications, such as saccade control of a binocular head [20] and adaptive state space quantizations [47, 63, 88].

1.6 Learning Taxonomy

There are many methods in which learning can be classified, see [18, 135] for the full list. We are interested here in the classification of different types of learning based on the *feedback* signal available.

For some systems, such as systems for predicting the outcome of an action, the available feedback generally tells the agent what the correct outcome is. That is, the agent predicts that a certain action will have a certain outcome, and the environment immediately provides a percept that describes the actual outcome. Any situation in which both the input and output of a system can be perceived is called *supervised learning*. On the other hand, in learning the condition-action mapping, the agent receives some evaluation of its action but is not told the correct action. This is called *reinforcement learning*. Learning when there is no hint at all about the correct output is called *unsupervised learning*. In this case, the teacher or the learning rule is fixed and built-in into the system at the start. Thus, unsupervised learning can always learn relationships among its percepts; this can be useful for example in clustering or pattern recognition tasks. But, due to the absence of a utility function or an external teacher, it can not learn what to do.

Our main subject in this thesis is reinforcement learning. Even though a full definition of this learning problem in terms of optimal control of Markov decision process must wait until chapter 3, we would like to motivate readers by emphasizing its biological relevance and by comparing it with other learning paradigms.

Reinforcement Learning

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays or waves its arms it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequence of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence [124].

Reinforcement learning is learning what to do—how to map situation to actions so as to maximize a numerical reward function. It is defined not by characterizing a learning method, but by characterizing a learning problem [59]. Any method that is well suited to solving that problem, is considered to be a reinforcement learning method. Clearly, such an agent must be able to sense the state of the environment to some extent and must be able to take actions that affect the state of the environment. Thus in the simplest possible form, reinforcement learning includes three aspects—sensation, action, and goal.

Dimensions of Complexity

Reinforcement learning has many different dimensions of complexity that are caused by the environment, the robot sensors, and the task. A few of these complexities are listed in table 1.1. For example, if the mapping between the sensors (world states) and the agent states (perceived states) is not one-to-one, which is the case in general, then a single agent state could arise from many world states. When some of these world states respond differently to different actions, the world will appear *inconsistent* to the agent. We will revisit some of these dimensions in chapter 3, however, we would like to point out that no method, including the work in this dissertation, addresses all the dimensions perfectly.

Dimensions	Complexities
underlying model	Markov-n, $n = 0 \dots \infty$
sensor and action	discrete vs. continuous
sensor $\rightarrow$ state	one-to-one vs. many-to-one
reward	immediate vs. delayed
exploration	directed vs. undirected
next state model	deterministic vs. stochastic
planning step	$0 \dots \infty$

Table 1.1: *Complexity of reinforcement learning. The left column of the table is the dimension, while the right column gives the extreme values for that dimension.*

Comparison with Supervised Learning

Fundamentally, the difference between supervised learning and reinforcement learning lies in what information they are fed with about their performance. The measure of learning can be thought of as a function defined over the set of possible system models. Each model in this space induces a system behavior. Hence the function defines a hyper-surface, on which each point corresponds to the performance of a particular system behavior. To improve its behavior, the system has to move towards higher and higher points on this performance surface [67].

In supervised learning, the system has information about the gradient of the performance function with respect to the model parameters. That means, it has directed information about how to change its behavior in order to improve its performance. The directed information is not often the true gradient itself, but rather an error vector computed as the difference between the desired and actual system responses. In supervised learning the system is told the correct responses to the different stimuli. That means the learning task has to be solved from the beginning, at least for some representative cases, from which the system can generalize by interpolation.

In reinforcement learning, on the other hand, instead of the teacher there is only a reward that tells the system how good or bad its performance is. Eliminating the teacher removes any bias that might be present in the training set. The reward indicates the value, with no directed information, of the performance function at a given model parameter. Therefore, reinforcement learning relies entirely on the environment to encode and manifest an observable and learnable mapping between the states the agent can perceive and the action it can perform. The dependence of reinforcement learning on the environment rather than on the training set can be recast as the reliance on the designer to properly structure the perceptual apparatus and the reinforcement function [82].

Rewards vs. Values

The reward function defines the goal in a reinforcement learning problem. It maps each perceived state of the environment to a single number. It is appropriate to associate rewards with pleasure or pain existing in biological system. They are the immediate and defining features of the problem faced by the agent. Whereas a reward function indicates what is good in an immediate sense, a *value* function specifies what is good in the long run. For example, a state might always yield a low reward but still have a high value, because it is regularly followed by other states that yield high rewards. Without rewards, there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is the value function with which we are more concerned when making decision [124].

1.7 Thesis Outline

The preceding sections summarized the contributions of the thesis and briefly introduced some background information. This section outlines the structure of the thesis and gives a short summary for each of the chapters.

Chapter 2 describes in detail the proposed sonar processing technique. Chapter 3 through 5 are devoted to reinforcement learning and the last chapter summarizes the thesis. Readers familiar with reinforcement learning and want to go directly to the learning part of the thesis should skip chapter 3.

Chapter 2 starts off with the description of the experimental testbed used to verify our sonar processing method. To address the inconsistency between the input dimension and the task dimension, a hashing method is presented. Hashing abstracts away the input space by partitioning, in a natural way, the sensors mounted on the robot into few and relevant regions. Following this, a linear recursive Kalman filter is introduced. The filter estimates a region depth by propagating an assumed conditional probability density function. The remaining sections present open and closed loop experiments where this processing technique is tested and compared with other existing methods. The chapter also includes material on fuzzy controller that is implemented in the closed loop experiment.

Chapter 3 begins with a simple hypothetical experiment that is used to define the vocabulary and elements of a reinforcement learning paradigm. Then, it discuss the different optimality criteria by which learning algorithms can be judged. Known problems associated with the measure of convergence and the tradeoff between exploitation and exploration are also raised. Afterwards, delayed reinforcement where actions are chosen to optimize future reinforcements is dealt with. In subsequent sections, we review existing techniques, model based and model free approaches of handling delayed reinforcement learning problems. The chapter closes by presenting a hybrid architecture called Dyna that integrates planning, learning and action.

Chapter 4 first presents the essence of bias–variance dilemma by decomposing the estimation error into two components. Following, bias in the context of learning is formally defined and some useful forms of biases that have been dominant in the past are discussed. Since it is difficult how to bias a general problem towards important aspects, a particular problem domain that eases the study of amount and quality of bias is first chosen. Further, as a way of introducing prior knowledge into the problem, the notion of belief matrix is introduced. Then, a variety of biases each altering the belief matrix differently are derived. While some of the biases are derived by considering only the environment, others are derived directly from the task. A performance index that measures the effectiveness of a bias is defined. Based on this index, the biases are assesed, and new results are reported. Before closing the chapter, the advantage of continual learning over learning from scratch is also presented.

Chapter 5 reports an implemented biased reinforcement learning experiment. The most pressing issue in this chapter is the integration of learned knowledge with existing knowledge. The learner is suppled with two kinds of knowledge: domain knowledge that provides symbolic knowledge to the learner on key world features and basic reflex rules that enables the robot to be operational right from the beginning. The learning architecture is a two layered neural network. The first layer is a growing radial basis function network (section 1.5) that is used to construct state space adaptively, and the second layer is a stochastic layer that performs the reinforcement learning algorithm. The reinforcement learning method is structured around estimating the long term value function from which actions are chosen. The kind of application presented is to learn to reach a goal position following a minimum cost path or trajectory.

Chapter 6 summarizes the results of the thesis and ends with the author's final closing thoughts.

Chapter 2

Mobile Robot Sensor Processing

The hallmark of an intelligent robot is the sense–think–act cycle,
and the sense part is the most difficult.

Thorpe

2.1 Motivations

Often in a mobile robot control, sensor uncertainty is overlooked by working on a simulated robot that assumes that robot sensors deliver accurate values [56, 106, 112, 118, 125]. Furthermore, the number of sensors considered in simulations are far fewer than the number of sensors that one finds on real robots. In practice, real world sensors deliver very uncertain values, even in a stable world! Moreover, real robots are equipped with multiple sensors that enable them to capture as much information as possible about the environment and their internal conditions. Therefore, even if we assume that the sensors are accurate, there is a huge amount of raw data that is difficult to apprehend.

It has been estimated that far more neurons in the human brain are devoted to *processing* and *analyzing* sensory inputs than organizing and executing complicated motor outputs [34]. There is no reason to believe that this will not be true of effective robots. In as much as the navigation property of mobile robots are influenced and affected by the sensors mounted on their body, sensor processing deserves equal attention as the robot controller does. Therefore, to successfully operate an autonomous mobile robot in a real world, some form of processing scheme that minimizes the noise and collapses the huge amount of raw sensory data into few and relevant information is necessary.

This chapter presents a novel noise tolerant mobile robot sensor processing technique that is capable of dealing with multiple and noisy sensors. The technique combines a *hashing technique* that uses some form of domain knowledge to collapse a huge sensory data and a *fusion method* that uses a Kalman filter to estimate the proximity of a region by integrating sensory values taken at different time instances. The proposed technique is compared with the existing method by carrying out a simple as well as a realistic experiment on a real mobile robot. In the former experiment the perception-action cycle is not complete, it involves only the perception cycle and processes the perceived data off-line without using it for control. The latter experiment, however, includes an on-line controller, as well. The controller is a fuzzy logic controller that has a handful of fuzzy rules. The results of both experiments not only showed that our processing technique is superior to the existing technique, but also brings to light the flaw of the existing technique when applied to real mobile robots.

2.2 Experimental Platform

The primary testbed for demonstrating the applicability of the idea described in this chapter is the TRC robot shown in figure 2.1. The robot is a two wheeled autonomous vehicle with the dimensions of 28 cm H $\times$ 75 cm L $\times$ 70 cm W. Basically, it has two types of motions: translational and rotational. For each motion type the corresponding motion quantities: position, velocity, and acceleration are defined as the basic control commands of the robot. It also has other commands that are derivatives of the basic commands, however, they are not used in this thesis. The robot motor gets its power from a series of two re-chargeable batteries, each having a capacity of 60 Amp-Hours at 12 V.

Sensor Subsystems

Four types of sensors, which enable the robot to perceive the environment as well as detect the internal and external events, are mounted on the robot. These sensors are: a current sensor at the base of the motor to detect high current whenever the robot tries to push heavy loads, two bumpers around the base that stop the robot when it is in contact with objects, 16 Polaroid proximity sensors, and a compass both used for navigation. Each of the sonars has a 30 degree beam opening angle and measures the distance to the nearest obstacle within the range of six inches and a user programmable value. The user programmable value determines the highest reading obtainable from the sonars as well as the maximum time the scanner waits for an echo before it disables the particular sonar [134]. If the maximum time is elapsed before receiving an echo, the sonars return the programmed value as a measured distance. The sonars are continuously scanned by an on-board micro-controller in a round robin manner. The bias voltage for the ultrasonic diaphragms and the proximity sensor electronics is derived from a separate battery that provides 12 Amp-Hours at 12V.

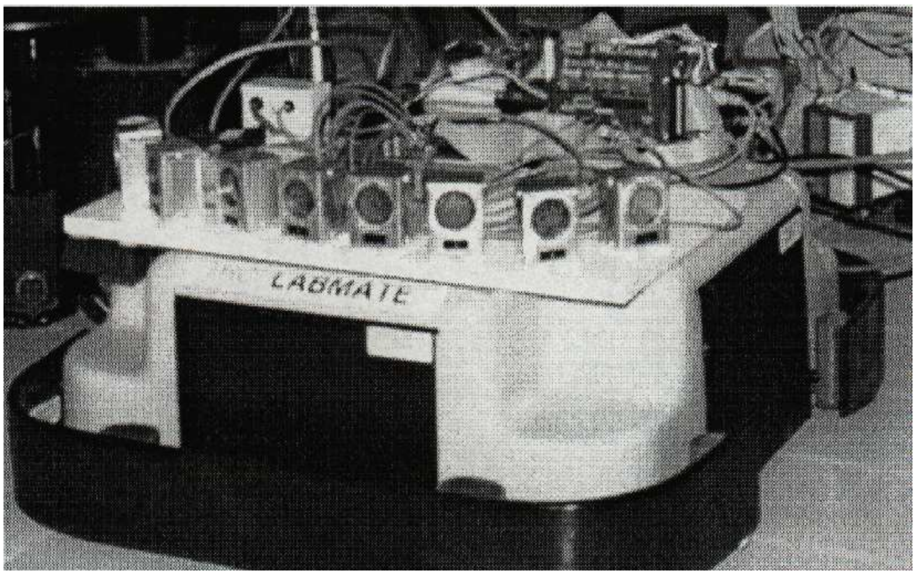


Figure 2.1: *The experimental TRC mobile robot.*

Sonars Placement

Among the available 16 sonar sensors, only ten sonars are chosen to cover the front side of the robot. Furthermore, each of the sonars is programmed to measure a maximum distance of 2 *m* and arranged to span a frontal beam angle of 120 degrees.

Unlike circular robots whose geometry encourages uniform sonar distributions, rectangular type robots demand non uniform sensor distributions. That is to say, it is not enough to simply place sonars only along the uniform sides. Such arrangements often result in *specular reflection* on the part of the robot where there is an abrupt change in geometry. Specular reflection is a phenomena that occurs when the incident angle of a sonar normal to the surface is greater than a certain critical angle. In this case, instead of reflecting, the beam totally deflects and no echo is received by the sonar receptor. In order to combat, at least to a certain extent, this problem, a sparse sonar distribution that places more sonars on the non-uniform (corner) sides of the robot than on the uniform sides was followed. This is shown in figure 2.2, where the sonar density increases outwardly from the center.

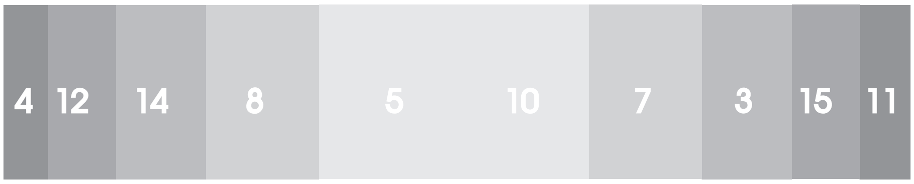


Figure 2.2: *The sonars are distributed sparsely and placed in a staggered way. The sparse distribution is shown by the non-uniform sonar density, which is represented as a grey value—the darker the grey value the denser the sonars; whereas the staggered placement is shown by the non-sequential labeling of the sonars.*

When there exists a systematic interaction between any two or more sensors in a multi-sensor environment, we say that the sensory system suffers from a *crosstalk* problem. Since almost any environment promotes multiple reflections of ultrasonic waves, crosstalk noise is a common problem, specially in sonar-based robots. But unlike other types of noises that appear and disappear rapidly, the noise due to crosstalk is particularly damaging—once it occurs it may continuously cause erroneous readings until some random time.

Borenstein et al. [15] have used a special sensor firing scheme called Error Eliminating Rapid Ultrasonic Firing (EERUF) to minimize the noise due to sensor crosstalk. However, their method is not easily transferable to a system, such as ours, where both firing sequence and firing intervals are hardware fixed. We combat sensor crosstalk by staggering the sonars so that their spatial sequence do not agree with their firing sequence. Since the sonars are fired in a fixed sequence by the on board controller, there will be a high crosstalk between any two sonars when their spatial ordering coincide with their firing sequence. Therefore, sonars that are fired consecutively are placed, as much as possible, far apart.

2.3 Sensor Abstraction

Human beings exhibit an impressive performance in a complex and rapidly changing environment by continually monitoring a large number of sensory inputs and detecting statistically significant patterns in them. For example, the human mind can glimpse at a rapidly changing scene and immediately discard 98 percent that is irrelevant and instantaneously focus on a woodchuck at the side of a winding forest road or a single suspicious face in a tumultuous crowd [120]. Furthermore, by monitoring their own actions, they are able to associate patterns of sensations and actions and determine the effect that actions are likely to have in the future. This mixing of actions with sensations enable them to seek desired actions based on past experiences. This will be the topic of discussion in subsequent chapters.

Unfortunately, present day machines are not yet able to prune and process their sensory information effectively. Instead, they try to detect significant patterns that affect the output by considering the entire state space. But monitoring a high dimensional space and detecting significant patterns is almost an impossible task. Even if it is assumed possible, it could not be achieved without ever surpassing the available computational resources. For instance, in the scenario of figure 2.2, assuming hypothetical sensors each having only three outputs, the number of states that need to be considered is in the order of 10^5. To detect a handful of significant patterns from this huge dimension is indeed a very difficult and complex undertaking.

This problem arises due to our attempt to conclude the task complexity from the dimension of the input space. Instead of focusing on the input state space, a more useful way is to focus on the complexity of the *target* function itself, as separate and distinct from the size and dimensionality of the input state space. A large state space is not so much a sign of a difficult problem—the size of the input state space may give an upper bound on the complexity, but short of that high bound, task complexity and input dimension are unrelated. For example, one might have a large dimensional task where only one of the dimensions happens to matter [124].

From this point of view, the real source of the problem is the complexity of the target function, not the dimensionality of the state space. Thus, adding dimensions to the task such as new sensors or new features, should be almost without consequence if the complexity of the needed approximations remain the same. The new dimensions may even make things easier if the target function can be simply expressed in terms of them. Therefore, we need methods which are unaffected by the dimensionality per se, but limited only by, and scale well with, the complexity of what they approximate. A coding scheme called *hashing* that first extracts the complexity of the target before it approximates the target function is hereby proposed.

Hashing

The earlier hypothetical example on the scenario of figure 2.2 suggests that, the number of sensors mounted on the robot does *not* necessarily correspond with the required dimensions of the target function. In fact, the dimension of the target function *is* much less than the number of sensors available and is often determined from some prior domain knowledge. This does not mean, however, that most of the robot sensors are redundant and can be thrown away. Rather on the contrary, a wealth of sensory information is a key to intelligent control, so any further direction in control must be helped, rather than hurt, by an increased amount of sensors [82].

On the one hand, it is claimed that a wealth of sensors is important for a faithful perception of the environment, but on the other hand, it is argued that not all sensors are important for the generation of control signal. This conflict between perception and control calls essentially for some form of processing mechanism that will collapse the sensor space to the level corresponding to the task without loss of significant sensory information. Hashing is an appropriate method of collapsing the sensor state space. It is a consistent way of reducing the large state space into a much smaller set of spaces. In general, hashing produces state space consisting of noncontiguous and disjoint regions randomly spread through out the state space, but that still form an exhaustive partition. Through hashing, memory requirements are often reduced by a large factor with little loss of performance. It frees us from dimensionality problem in the sense that memory requirements need not be exponential in the input dimension, but need merely match the demand of the task.

Regions Labeling

Sensory information can be used to segment the environment into regions with properties that are useful for spatial reasoning. The known characteristics of different types of sensory information can be used to label some useful property of each region so that symbolic reasoning, like fuzzy rules, can be performed at higher levels in the control structure.

Let us consider an autonomous robot navigation task. Obviously, depending on the particular geometry of the robot, this task calls for monitoring the various sides of the robot. It is therefore clear that the task dimension is linear with the number of the sides needed to be monitored. Since only the front side of our robot is monitored, the simplest approximation of the task dimension is one. But this approximation is very coarse and the resulting target function is oscillatory hence, not interesting. Let us now begin hashing the robot's front side into three regions: `left`, `center`, and `right`. Notice that these regions correspond to the physical geometry of the robot. In this case, the task dimension is three, because there are three separate regions that need to be controlled. Still another possibility is to hash the front side into five regions and get a corresponding high dimension. We can keep on increasing the task dimension by hashing the front side into finer regions and get by with a proportionate increase in computational resources. In the limit, the task dimension is stretched to the dimension of the input space.

What we normally do is to fix the level of complexity and try to approximate accurately the target function of that complexity or less. But the problem is how to fix the level of complexity? Intuitively, a low level of complexity results in a trivial target function or robot behavior, whereas a high level of complexity makes it difficult even to arrive at any solution. Based on the physical dimension of the robot and a few trial and error experiments, we have chosen a task dimension equal to five. Notice that while the lower bound of the task dimension is one (no hashing), the upper bound is ten (equal to the available number of sensors). Therefore, the chosen task dimension corresponds to a complexity that is neither trivial nor difficult.

Once we have chosen the task complexity, the sonar sensors (covering the front side of the robot) are carefully hashed into five overlapping regions. These are: `far left`, `left`, `center`, `right`, and `far right` (figure 2.3). Each region covers only a limited part of the robot's view *but* their union still exhausts the front view. Also shown in figure 2.3 is an overlapped regions, i.e., sonars adjacent to two hashed regions are considered in both. The main purpose of overlapping the regions is to account for the sonar beam angle, but it increases also the estimation accuracy.

The presented hashing scheme can be considered as a way of elevating the state description of the input to an abstract level. In other words, instead of the individual sensors snapshot, it is now abstract states called region depths that constitute the input state space.

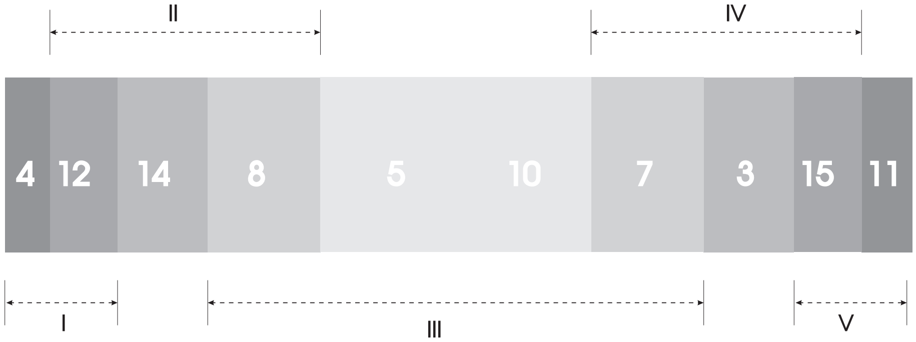


Figure 2.3: *Hashing method. The sensors are placed on the front view of the robot. Each hashed region is associated with a part of the robot's view:* `I - far left`, `II - left`, `III - center`, `IV - right`, *and* `V - far right`.

2.4 Sensor Fusion

Remember that instead of the individual sensor snapshots, we are now interested in a *region depth* that provides an object level description of the robot environment. In this section, we will see how to compute a region depth from a set of sonar snapshots. In order to simplify our discussion, let us consider only one hashed region that covers a fixed number of sensors. Nevertheless, whatever is being said about this region also applies, perhaps with minor changes, for the other regions.

Basically, a region depth conveys the distance of a particular hash region (figure 2.3) of the robot from a nearby object. We seek to compute this distance from the aggregate of the sonars covering the region. Sonar sensors due to effects such as material properties, multiple reflections, specularities, etc. deliver inaccurate and sometimes unexpected readings in complex environment. Hence, the need for a filter that estimates the region depth from the available noisy measurements is apparent. Despite the typical connotation of a filter as a black box containing electrical networks, the filter we are interested in is just a computer program that processes discrete-time measurement samples.

One of the simplest filtering processes proposed by Reignier [106] is to keep only the minimum sensor reading in the hash region and to discard all the others,

$$\acute{d} = \min\left(s_1(t), s_2(t), \ldots, s_N(t)\right) \tag{2.1}$$

where N is the number of sensors contained in the hash region, $s_i(t)$ is the reading of sensor i at time t, and $\acute{d}$ is the depth estimate. Unfortunately, this method assumes that sonar sensors deliver precise depth readings at all times. But most easy-to-build sensors are incredibly noisy and difficult to interpret; this is no less true for the sonar sensors of our robot. Later in the chapter, conclusive results that refute Reignier's method of estimating a region depth will be presented.

Instead of over simplifying the task, a more rigorous sensor fusion technique of processing the sonar readings is proposed. Generally, the advantage of fusing multiple sensory readings is to obtain a more accurate value of the desired information (section 1.3). However, in order for the data from each sonar to be used for integration, it must first be modeled. The model represents the uncertainty in the reading of the sonars and provides a measure of the measurement quality that can be used by the subsequent integration functions. The fusion involves two stage filtering processes.

Filtering

The first filter, figure 2.4, is a median filter that computes the *measured depth* $z(t)$, from the present set of sensor readings, i.e.,

$$z(t) = \text{median}\left(s_1(t), s_2(t), \ldots, s_N(t)\right) \tag{2.2}$$

It is worth noting that equations (2.1) and (2.2) deliver different values but their noise characteristics are the same, because both choose the reading of a single sensor as their estimate. Therefore, there is no noise reduction by going from the minimum filter to the median filter. Generally, it is difficult to faithfully estimate the region depth from a single sensor snapshot.

Instead of a single sensor snapshot, Cameron et al. [104] and Hailu [43] have enlarged the sensor space by taking multiple sensor snapshots at every robot location. Though this method has worked quite well, it was not without the undue influence of multiple sonar perception on the speed of operation of the robot. Multiple perception also depletes the battery power and causes the robot to die quickly.

The other method of enlarging the sensor space is to augment the present measured depth with the past measured depth profiles so that estimation will now be based on the present as well as the past sequence of snapshots. Clearly, this method requires past measured profiles to be stacked in a sliding window or memory. The addition of past sensor information to the present perception is not new, it has been applied to distinguish perceptually aliased or hidden states in agents that learn from memory [12, 126, 142]. Unfortunately, the approach has its own drawbacks, it leaves to intuition to guide the choice of the window length. In most cases the designer must understand the task well enough and estimate its maximum memory requirement. For the navigation problem we have at hand, very past measured depth values, taken at different spatial locations, have little or no correlation at all with the current depth value. Therefore, it makes sense to estimate the present region depth only from the current and few past measured depth values. Based on this intuition and some trial and error experiments, we have chosen a window of length three.

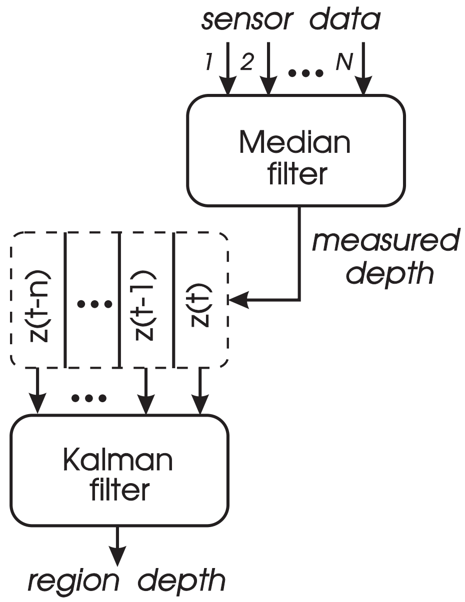


Figure 2.4: *A cascade of two filters used to estimate the region depth. The estimation is based on the present and past measured depth profiles.*

The second filter is a Kalman filter that estimates the desired quantity, i.e., the region depth, from the measured depth data stored in memory. We define optimality in terms of a Bayesian view point where a given likelihood function is to be maximized [14]. The chosen likelihood function is the conditional probability density function of the region depth conditioned on the knowledge of the present $\{z(t-n+i)\}$, and the past $\{z(t-n), \ldots, z(t-n+i-1)\}$, measured depth values. It is denoted by,

$$f_{\acute{d}|\{z(t-n),\ldots,z(t-n+i)\}}\left(x|\{z(t-n), z(t-n+1), \ldots, z(t-n+i)\}\right) \tag{2.3}$$

where n is the length of the window. Equation (2.3) indicates that for a given sequence of measured depth values, what the probability would be of, $\acute{d}$, assuming any particular value or range of values.

Since the system is dynamic, the robot position and hence sensor values change with time. The dynamic Kalman filter is appropriate to the scenario; unfortunately, this filter requires a model, equations (1.1) and (1.2), for the rate of change of the sonar returns. For a situated agent, this change depends among other things on: the speed and rotation of the robot, the direction of motion, the environment and its acoustic property, the dynamic property of each sensor, the position of the sensors on the robot, and the frequency of sensor crosstalk. Looking at the parameters involved, it is extremely difficult to determine the system coefficients that the model requires. Due to the lack of these coefficients, a linear recursive Kalman filter is employed.

Sensor noise in robots is caused by different uncorrelated noise sources. For example, the noise due to sensor crosstalk is uncorrelated with the noise due to specular reflection. Furthermore, since each noise source acts independently, the distribution of their summed effect could be approximated by a normal distribution function, provided that the individual noises are white (see section 2.9). Under this assumption, the conditional probability density function of a region depth is described by a Gaussian distribution below,

$$f_{\acute{d}|\{z(t-n),\ldots,z(t-n+i)\}}\left(x|\{z(t-n), \ldots, z(t-n+i)\}\right) = \exp\left(-\frac{||x-\mu||^2}{\sigma^2}\right) \tag{2.4}$$

where μ is the first and σ is the second order statistics of the distribution.

The region depth estimation is based on propagating the conditional probability density function through all the stored measured depth values, i.e., $\{z(t-n), z(t-n+1), \ldots, z(t)\}$. But before we begin the process, the conditional probability density function has to be initialized. The first measured depth value in the series, i.e., $z(t-n)$ is used to initialize the density function. In other words, we look for the mean and variance of the density function,

$$f_{\acute{d}|z(t-n)}(x|z(t-n)) \tag{2.5}$$

conditioned only with the knowledge of $z(t-n)$. When only one measured depth exists, the best estimate or the highest likelihood value of the region depth is the measured depth itself. But because of the assumed Gaussian distribution, the highest likelihood value is the mean. Hence, the mean of the initial distribution is the measured depth, i.e.,

$$\mu = z(t-n) \tag{2.6}$$

Since the best initial estimate of the region depth is the measured depth, its uncertainty or variance is the same as the variance of the measured depth. From equation (2.2), the variance of the measured depth is the same as the measurement variance of the sonars. So, the initial variance of the distribution is equal to the variance of the sonars, i.e.,

$$\sigma = \varsigma \tag{2.7}$$

where ς^2 is the sonar variance.

Computation of ς

To compute the variance of the sonars, an arbitrarily random sensor is selected (the assumption here is that the variance obtained for a single sensor will be representative for all the others. This makes sense because all the sonars are of the same type.) and the following experiment is carried out on it. The selected sonar is placed in a different environment, orientations and proximity that the robot could possibly face when it is in operation (e.g., corners, corridors, doors edges, extended walls, free ways, etc). In each of these conditions, the true and measured distance pairs (d, r) have been recorded. After a sufficient number of range pairs have been taken, the variance is computed by,

$$\varsigma^2 = \frac{k}{N} ||\boldsymbol{d} - \boldsymbol{r}||^2 \tag{2.8}$$

where $\boldsymbol{d}$ and $\boldsymbol{r}$ are the true and measured range vectors respectively, and N is the number of elements in each vector. Note that this experiment was carried out in a static condition, only a single sensor was fired and the robot was not moving. Therefore, the obtained value does not represent the actual variance when all the sensors are fired sequentially and the robot is moving continuously. In order to account for these dynamics, a multiplying factor $k > 1$, has been added to equation (2.8).

Estimation Based on Likelihood

Once the initial conditional probability density function is initialized with equations (2.6) and (2.7), it is propagated through the remaining stored measured depth values, $\{z(t-n+1), z(t-n+2), \ldots, z(t)\}$, using the Kalman filter algorithm (algorithm 1). Notice that at each update the variance is decreasing, indicating that every new data provides some information thus, increases the estimation accuracy. At the last update, the conditional distribution of the region depth given the present and all the past stored measured depth values is summarized in the parameters μ and σ.

Earlier we mentioned that the likelihood function $\mathcal{L}(x)$, is the same as the conditional probability density function. So, in order to estimate the true region depth, we apply the Maximum Likelihood (ML) analysis on the conditional probability density function, i.e.,

$$\begin{aligned}\acute{d} &= \mathcal{L}(x) \\ &= \arg\max_x f\left(x | \{z(t-n), z(t-n+1), \ldots, z(t)\}\right) \\ &= \arg\max_x \exp\left(-\frac{||x-\mu||^2}{\sigma^2}\right) \\ &= \mu \end{aligned} \tag{2.9}$$

The algorithm estimates the present region depth by propagating the conditional probability density function from $t-n$, up to the present time t. At the end of the propagation, the mean of the density function is the optimal estimate of the region depth, while the variance indicates the error associated with the estimate.

initialize the density function
  mean: $\mu = z(t-n)$
  variance: $\sigma = \varsigma$
for $i = 1$ to n do
  compute Kalman gain: $G = \sigma^2\left(\sigma^2 + \varsigma^2\right)^{-1}$
  update mean: $\Delta\mu = G\,(z(t-n+i) - \mu)$
  update variance: $\Delta\sigma^2 = -G\,\sigma^2$

Algorithm 1: *The linear recursive Kalman filter algorithm.*

2.5 A Simple Experiment

To compare the proposed sensor processing technique with the method suggested in [106], a simple experiment, described below, was carried out. The robot was first placed at a distance of 2 m in front of an extended wall. Then it was set to approach the wall at a constant velocity. While the robot was moving, the sonar readings of `region III` (figure 2.3) were recorded until the robot has come close to the wall. Later, Reignier's method, equation (2.1), and the sensor processing scheme presented here were applied to the data gathered to compute the distance of the robot from the wall.

Figure 2.5 shows the variations of the distance of the robot from the wall as obtained by the two methods. From the plots it is clear that the method suggested by [106] was susceptible to noise and practically failed to deliver a distance variation that corresponds to the motion of the robot. Whereas the proposed sensor processing scheme successfully removed the noise and delivered a clean depth estimate that could be used to generate control signals. Specifically, the Kalman filter held (sustained) the depth estimate at a relatively high value without much swing while the robot was far from the wall. Undeniably, there was some swing in our preprocessor, too. However, the fuzzy controller (section 2.7) was not sensitive to such little noise, because such noise affected the membership function only slightly and changed the final control command in a minor way.

2.6 Fuzzy Systems

In the previous section, the utility of the proposed sensor processing method was demonstrated in a simplified experiment, where the robot was moving against a wall without being controlled. In practice, however, the robots' environments are complex and the robots are controlled by data coming either directly from the sensors or some processing block. Therefore, in order to substantiate the usefulness of the proposed method, it was necessary to carry out a sensor based navigation experiment. Thus a realistic experiment was conducted by tying the sensors to a fuzzy logic controller, whose purpose was to drive the robot based on the data coming from the sensor processing. Before describing the implemented fuzzy logic controller and presenting the experimental results, a review of the notion of fuzzy logic control will be made.

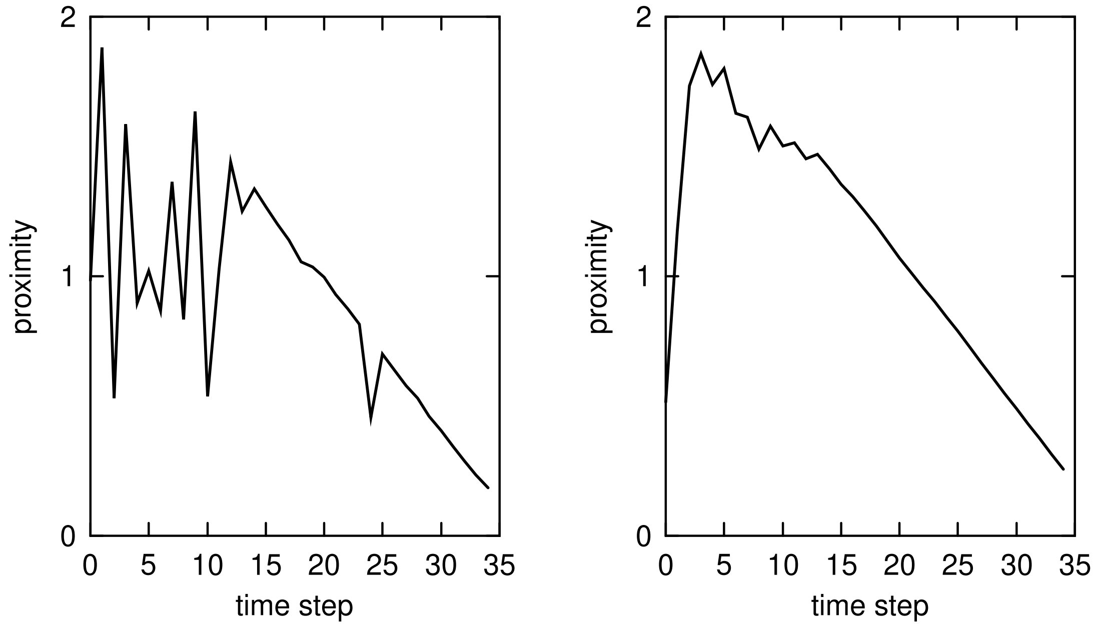


Figure 2.5: *Left: Variations of the robot distance from the wall as computed with the method proposed in [106]. Right: Variations of the robot distance from the wall as computed with the method proposed in [44].*

Fuzzy Sets

A classical set, in the universe of discourse $\mathcal{U}$, is normally defined as a collection of elements or objects $u \in \mathcal{U}$, that can be finite or infinite. Each element can either belong or not belong to a set $\mathcal{A}$, $\mathcal{A} \subset \mathcal{U}$. A *binary* valued characteristic function, $\chi_{\mathcal{A}}(u)$, represents whether the object u belongs to the set $\mathcal{A}$ or not.

$$\chi_{\mathcal{A}}(u) = \begin{cases} 1 & : \quad u \in \mathcal{A} \\ 0 & : \quad u \notin \mathcal{A} \end{cases} \tag{2.10}$$

The binary valued sets to represent relative concepts such as weight or height is restrictive, whereas linguistic values such as `heavy` or `short` are appropriate and context dependent. The fuzzy set theory [148] provides a mathematical framework to capture such imprecise linguistic values. It represents a linguistic value by a membership function that assigns a value to every object $u \in \mathcal{U}$ from the unit interval [0,1]. The value assigned is the degree to which the object belongs to the linguistic value. The membership function is an extension of the characteristic function in the sense that neither $u \in \mathcal{A}$ nor $u \notin \mathcal{A}$ holds. The set defined on the basis of such extended membership function is called a *fuzzy set*.

Formally, the membership function $\mu_{\mathcal{A}}$, of a fuzzy set $\mathcal{A}$ is a function,

$$\mu_{\mathcal{A}} : \mathcal{U} \to [0,1] \tag{2.11}$$

This means every element $u \in \mathcal{U}$ has a membership value $\mu_{\mathcal{A}}(u)$, and the fuzzy set $\mathcal{A}$ is completely described by the set of tuples,

$$\mathcal{A} = \{(u, \mu_{\mathcal{A}}(u)) \mid u \in \mathcal{U}\} \tag{2.12}$$

In classical set theory the union ($\cup$), intersection ($\cap$), and complement ($'$) of sets are simple logical operations that have a well defined meaning. These logical operations are also defined in fuzzy set theory, however, due to the use of fuzzy values, their interpretation is not so clear as in the classical set theory. Among the variety of definition of fuzzy set operations, the definition proposed by Zadeh [148] is most frequently used. Let $\mathcal{A}$ and $\mathcal{B}$ be two fuzzy sets in $\mathcal{U}$ with the membership functions $\mu_{\mathcal{A}}$ and $\mu_{\mathcal{B}}$ respectively. Then,

$$\forall u \in \mathcal{U}: \quad \mu_{\mathcal{A}\cap\mathcal{B}}(u) = \min(\mu_{\mathcal{A}}(u), \mu_{\mathcal{A}}(u)) \tag{2.13}$$

$$\forall u \in \mathcal{U}: \quad \mu_{\mathcal{A}\cup\mathcal{B}}(u) = \max(\mu_{\mathcal{A}}(u), \mu_{\mathcal{B}}(u)) \tag{2.14}$$

$$\forall u \in \mathcal{U}: \quad \mu_{\mathcal{A}'}(u) = (1 - \mu_{\mathcal{A}}(u)) \tag{2.15}$$

The above fuzzy operations are simple extensions of the classical set operations, other definitions are possible as well [33]. For example, equations (2.16) - (2.18) are the fuzzy set operations that have been used in our robot control application (to be explained shortly).

$$\forall u \in \mathcal{U}: \quad \mu_{\mathcal{A}\cap\mathcal{B}}(u) = \mu_{\mathcal{A}}(u) \times \mu_{\mathcal{A}}(u) \tag{2.16}$$

$$\forall u \in \mathcal{U}: \quad \mu_{\mathcal{A}\cup\mathcal{B}}(u) = \min(1, \mu_{\mathcal{A}}(u) + \mu_{\mathcal{B}}(u)) \tag{2.17}$$

$$\forall u \in \mathcal{U}: \quad \mu_{\mathcal{A}'}(u) = (1 - \mu_{\mathcal{A}}(u)) \tag{2.18}$$

Fuzzy Logic Control

The assumption that the behavior of a process can be modeled by an exact set of differential equations has been challenged by the introduction of *fuzzy logic control* [77, 119, 148], which recognizes that precise modeling of processes is difficult and not necessary for effective control. Processes, such as car driving, are readily controlled by humans without recourse to rigorous mathematical models, algorithms or deep understanding of the physical processes involved.

In a similar way, fuzzy logic control is aimed at modeling the *operation*, how a process behaves, rather than the underlying physical dynamics. It models the operation by a set of linguistic structural knowledge that has a canonical form,

$$\texttt{if } A_i \texttt{ then } B_i\ ,\ i = 1, \ldots, n \tag{2.19}$$

The expression describes the causal relationship between the process states and the control output variables and its form is called a fuzzy associative memory (FAM) rule, with A_i forming the antecedent part and B_i the consequent part of the rule. Each FAM rule captures some structural knowledge of the process operation and a fuzzy logic controller is constructed by covering the input and output spaces of the process with as many FAM rules as needed.

Both the antecedent and consequent parts of the rule are either simple or compound propositions. A proposition is an expression that relates a fuzzy variable to a linguistic value. By a linguistic variable we mean a variable whose values are words or sentences from a natural language [149]. For example, `weight` is a linguistic variable whose linguistic values may be {`heavy, light`}, likewise the linguistic variable `height` may have {`long, short`} as values.

The expression A_i : `the weight is heavy` is a simple proposition, which has a formal symbolic translation, $A_i:\ u \in \mathcal{A}_i$ (where u is the linguistic variable `weight` and $\mathcal{A}_i$ is the linguistic value (set) `heavy`). Since truth in fuzzy is a matter of degree, the truth value of the above proposition (statement) is in the unit interval [0,1] and given by $\mu_{\mathcal{A}_i}(u)$. Simple propositions, like the above one, are not useful in most fuzzy control applications, where multiple input and multiple output are common. Therefore, instead of simple propositions compound or multi modal [67] propositions are common. A compound proposition is formed by a logical combination of two or more simple propositions and its value is determined from the values of the constituent propositions, equations (2.16) to (2.18).

Fuzzy Logic Control Structure

The basic configuration of a fuzzy logic control, as illustrated in figure 2.6, consists of the components: knowledge base, fuzzification process, inference engine, and defuzzification process [50]. Since in the next section we will entirely devote our attention to the implementation detail of this control structure on the TRC robot, we will describe the function of each component here and introduce the design parameters involved.

Knowledge Base

The knowledge base is divided into two parts: data and rule bases. The data base provides the necessary information for the proper functioning of the fuzzifier, the inference engine and the defuzzifier modules. Particularly, it contains information on: the quantization and scaling of the physical domains and their normalized counter parts, the fuzzy partition of the input and output spaces, and the definition of the membership functions. For the prevailing cases of continuous, normalized domains the design parameter of the data base includes mainly the choice of membership functions and the scaling factor.

Whereas the rule base contains the operation rules that are derived from an experienced skill operator who can express qualitatively (in words) the operation of the system. The design parameters involved in the construction of the rule base include: the choice of process state and control output variables, the determination of the content of the antecedent and the consequent parts of the rule, and finally the derivation and the writing of the rules.

Fuzzification

The fuzzification process initially scales the physical value of the process state variable, so that it lies within the universe of discourse of the control rules. After scaling, the resulting process state u, is fuzzified into linguistic values so that, it will be compatible with the fuzzy set representations of the process state variable in the antecedent part of the rules. The fuzzifier operator is defined as follows,

$$\boldsymbol{u} = \texttt{fuzzifier}\,(u) = \begin{pmatrix} \mu_{\mathcal{A}_1}(u) \\ \vdots \\ \mu_{\mathcal{A}_n}(u) \end{pmatrix} \tag{2.20}$$

where $\boldsymbol{u}$ is a vector holding the fuzzy values of the process state u, $\mu_{\mathcal{A}_i}$ is the membership function of the input fuzzy set $\mathcal{A}_i$ contained in the input proposition of the i-th rule, and $\mu_{\mathcal{A}_i}(u)$ is the corresponding truth value of the proposition to the process state, u.

Inference Engine

One of the nice features of fuzzy logic is that it provides a mathematical framework that allows us to reason in a manner that resembles human reasoning.

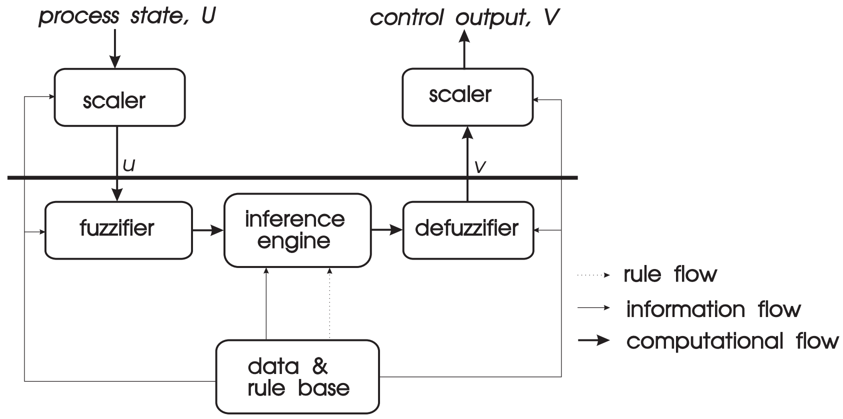


Figure 2.6: *The fuzzy logic control structure. The components above the thick horizontal line are optional. In domains where the range of the physical domain is the same as the universe of discourse of the fuzzy rules these components may be ignored.*

Fuzzy inference is a way of inferring the overall value of the control output variable. The inference process is based on an individual rule firing scheme and involves two steps. First, the output of the fuzzifier, which represents the fuzzy value of the process state variables, is matched to each antecedent part of the rules and the degree of match is determined. Second, based on this degree of match, the consequent part of the rule is modified, i.e., the clipped fuzzy set representing the fuzzy value of the control output variable is determined. The overall output fuzzy set of the final control output is the union of the clipped fuzzy sets of the individual rules.

Defuzzification

The defuzzification process produces a crisp output v, that optimally represents the overall output fuzzy set B. It also re-scales the crisp control output to match to the physical domain. The design parameter is the choice of the defuzzifier operator. The two defuzzification methods that commonly used are the mean of maximum (MOM) and the center of area (COA) [33].

The mean of maximum relies on selecting the control value corresponding to the maximum value in the output membership function,

$$v = \texttt{defuzzifier}\,(B) = \arg\max_{x \in B}\, \mu_B(x) \tag{2.21}$$

The popular probabilistic methods of maximum-likelihood and maximum posteriori parameter estimation motivate this defuzzification scheme. However, the scheme fails when there is more than one maximum membership values, since the control value can not be determined uniquely. One way of generating a single defuzzified output in multiple maxima cases is to take the mean or average of all local maxima values,

$$v = \texttt{defuzzifier}\,(B) = \frac{1}{N}\sum_{i=1}^{N} v_i \tag{2.22}$$

where $v_i = \arg\ \max_{x \in B}\ \mu_B(x)$ and $N = |\{v_i\}|$ is the number of times the membership function reaches the maximum value. The MOM is a simple defuzzification algorithm that does not consider the shape of the output membership function, it depends only on the points where the output distribution attains its maximum. But it is obvious that there are infinitely many output distributions that share the same maximum points.

The center of area method divides the first moment of area under the output membership function into half and the value marking the dividing line, called the centroid, is the defuzzified value. Mathematically this is expressed as,

$$v = \texttt{defuzzifier}\,(B) = \left(\int_{-\infty}^{\infty} \mu_B(x)\,dx\right)^{-1} \int_{-\infty}^{\infty} x\ \mu_B(x)\,dx \tag{2.23}$$

For the discrete case with a set of n control values $\{v_1, \ldots, v_n\}$, the centroid is,

$$v = \left(\sum_{i=1}^{n} \mu_B(v_i)\right)^{-1} \sum_{i}^{n} v_i \cdot \mu_B(v_i) \tag{2.24}$$

As already stated the MOM method ignores the information in much of the waveform. Therefore, it is analogous to a multi-level relay switch where a small change in the input of the controller results in a large change in the output. On the other hand, the COA method always delivers a unique value and considers the shape and area of the output fuzzy set as a whole. Hence it has a smooth transition between output values for variable inputs. This property of COA makes it the preferred defuzzification technique in most control applications.

2.7 Implementation Details

Earlier we have outlined the basics of a fuzzy logic controller without going into specific design details. We are now going to address the implementation or planning issues. Specifically, we determine the values of the design parameters for each of the components of figure 2.6.

Rule Base

Choice of Variables

The parameters that have to be specified in the rule base are the input and output variables. Any of the traditional motion quantities is a candidate for the output variable. In the thesis the linear velocity v, and the rotational velocity w, of the robot are chosen as output variables. Further, the range and unit of these variables are $[-10, 15]$ cm/sec and $[-10, 10]$ $degrees/sec$ respectively.

Since it is a sensor based navigation, the input variables are the external and internal sensors of the robot, $u = (u_1, u_2, u_3, u_4, u_5, u_6)$. The first five variables are the region depths each having a range of $[0, 2]$ m and derived from the external sensor readings. The last variable is the linear velocity of the robot, derived from the robot's internal sensor. The inclusion of the velocity as a feedback renders a smooth motion to the robot.

Content of Rules

Unlike the more generalized fuzzy rules, Sugeno rules [119] that have fuzzy sets in the premise part and simple coefficients in the conclusion part are used here. Therefore, the antecedent of our rule, equation (2.25), consists of a compound proposition whereas, the consequent is an assignment expression that assigns crisp values to the two output variables.

$$\texttt{if}\ (P_{i1} \cap P_{i2} \cap P_{i3} \cap P_{i4} \cap P_{i5} \cap P_{i6})\ \texttt{then}\ v = v_i\ \texttt{and}\ w = w_i\ ,\ i = 1, \ldots, n \quad (2.25)$$

The resulting rule is a multiple input multiple output (MIMO) FAM rules bank, where the P_{ij}'s, $j = 1, \ldots, 6$ are simple propositions each associating one of the earlier input variables to a linguistic value, v_i and w_i are the output of the controller when *only* the i-th rule is fully active, and n is the number of rules. The approach used in formulating the initial fuzzy rules is pure human intuition. But once a rough controller is build, we will be writing and rewriting the rules until an acceptable control response is obtained.

Data Base

The primary concern in the design of the data base is the choice of the membership functions for the input as well as the output variables. In the implementation the rules are Sugeno type; consequently, the choice of the membership functions is limited only to the input variables.

As mentioned earlier, the selected input variables are the five region depths and the robot base velocity. Akin to the work of [81], the universe of discourse of the regions depth value has been segmented into three relevant linguistic values. These are `dangerous`, `steer`, and `safe` with the qualitative meanings of,

- `dangerous:` in this linguistic value, a region is in a very tight situation and the only way to escape the situation is by backing up.
- `steer:` this is a linguistic value a region assumes before it enters into the tight situation, the robot should be steered to prevent its occurrence.
- `safe:` in this linguistic value, a region is totally free and the robot can move toward the region's direction without causing any danger.

Note that it is possible to choose some more intermediate linguistic values, but it trades with the complexity of the controller. The three fuzzy sets, whose quantitative interpretation is defined by the overlapping trapezoidal membership functions of figure 2.7, are a good compromise between the simplicity and smoothness of the controller.

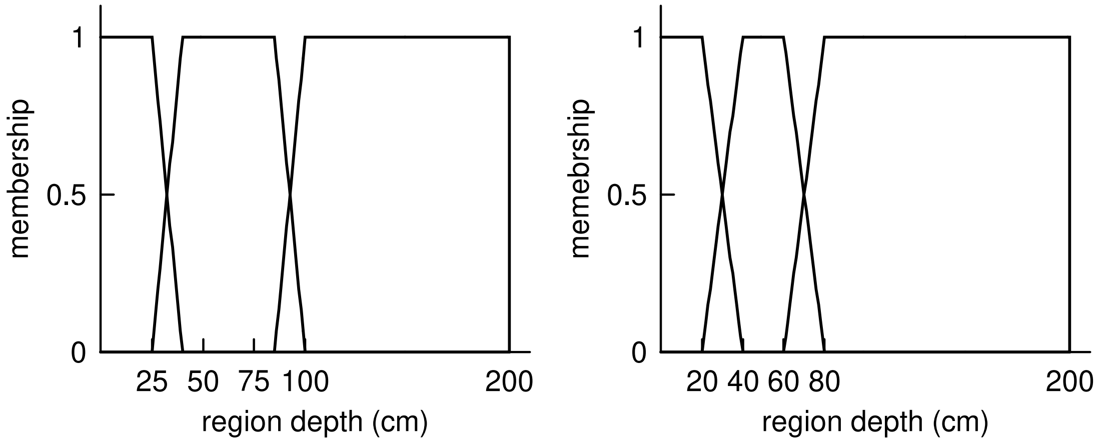


Figure 2.7: *Three overlapping trapezoidal membership functions defining the fuzzy sets:* `dangerous`, `steer`, *and* `safe`*. Left - membership functions of regions* `II, III, IV`*. Right - membership functions of regions* `I, V`*.*

Two things are worth noting in the plots of membership functions of figure 2.7. First, it is obvious that the three front regions of the robot demand a higher degree of safety than the two far side regions, see figure 2.2 for the regions. Therefore, in order to account for this variation, two different sets of membership functions are used. Second, since the domain of both sets of membership functions is the same as the universe of discourse of the physical measurements, the scaling component shown in figure 2.6 is bypassed.

Similar to the region depth values, the velocity of the robot is quantized into four linguistic values: `reverse`, `slow`, `normal`, and `fast` whose qualitative meanings are already clear; their quantitative meanings, however, are defined by the fuzzy membership functions of figure 2.8. Also scaling is skipped here, since the domain of the membership function and the universe of discourse of the base velocity are identical.

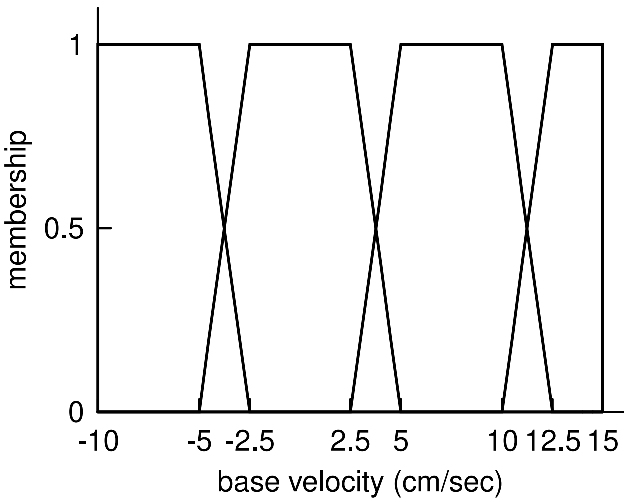


Figure 2.8: *The continuous robot velocity is patched (discretized) into four overlapping trapezoidal fuzzy sets:* `reverse`, `slow`, `normal`, *and* `fast`.

As shown in equation (2.25) the rule-antecedent is a conjunction of six propositions. The five depth variables take any of the three linguistic values (figure 2.7) and the velocity variable takes any of the four linguistic values, (figure 2.8). So, it is evident that a total of $n = 3^5\text{x}4^1 \approx 1000$ variations of propositions are constructed and each variation is a FAM rule.

Inference Engine

Our choice of inference is the individual-rule based inference, where one first fires each rule with the crisp input to obtain the individual rule strengths. The strength of a rule is the same as the value of the proposition of the rule-antecedent. Since the rule-antecedent is a compound of six simple propositions, see equation (2.25), its value according to the fuzzy set operation of equation (2.16) is,

$$\mu_{i1}(u_1) \times \mu_{i2}(u_2) \times \mu_{i3}(u_3) \times \mu_{i4}(u_4) \times \mu_{i5}(u_5) \times \mu_{i6}(u_6) \tag{2.26}$$

where the u_j's are the six crisp input values and $\mu_{ij}(u_j)$ is the membership value of input unit j in rule #i. Once the individual strengths of the rules are computed, their outputs are weighted by the rule strengths and combined into an overall output of the controller, see algorithm 2.

In the algorithm n is the number of FAM rules, u_j is the value of input j, μ_{ij} is the membership function of input j in rule #i, r_i is the strength of rule #i, b_{il} is the l-th scalar output suggestion of rule number #i, and b_l is the l-th total output.

compute rule strength:

$r_i = \prod_j \mu_{ij}(u_j) \quad j = 1, \ldots, 6\ ,\ i = 1, \ldots, n$

inferred scalar output:

$b_l = (\sum_i r_i)^{-1} \sum_i r_i b_{il}\ ,\ i = 1, \ldots, n\ ,\ l = 1, 2$

Algorithm 2: *Inference algorithm of MIMO Sugeno type rules.*

2.8 Computational Environment

The computing system is build around a parallel virtual machine (PVM) [36] and is designed in a modular form. PVM is a computational platform that supports a distributed computing environment on a set of loosely coupled heterogeneous machines. A module is a process that performs a specific task and resides in a specified machine. In PVM only one module, called a parent, is responsible to spawn all the other modules called slaves. However, as soon as a slave is spawned, it runs asynchronously on the machine the parent assigned during spawning. Modules communicate by sending and receiving messages.

The architecture consists of three processors that appear as one virtual machine to the PVM (figure 2.9). The 68HC11 micro-controller mounted on the board

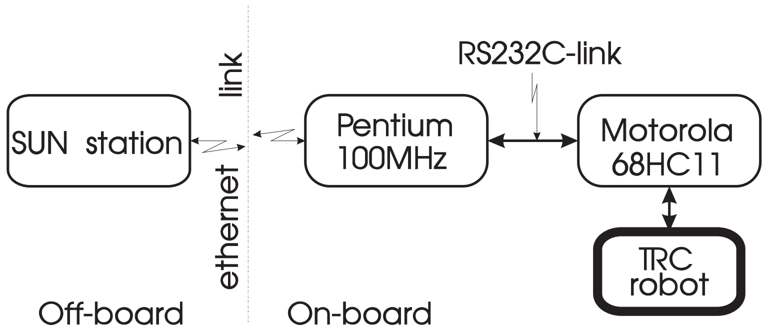


Figure 2.9: *Distributed computing environment.*

runs a module that fires and reads individual sonars, sends motor commands to the servo controller and handles time related synchronization details, whereas the other on board processor performs the perception action cycle at a higher level and routes sensor values to and motor commands from the host. The off-board processor is a SUN workstation that handles the sensor processing (algorithm 1) and the fuzzy controller (algorithm 2).

2.9 Experiments

Once again the aim is to test the sensor processing stage by carrying out a realistic experiment. The fuzzy controller described earlier gets its input from the sensor processor to control the TRC robot in a laboratory environment. Before the fuzzy controller was transferred to the robot, its behavior had been tested in a simulation for a variety of situations.

Simulations

In order to test the fuzzy controller, a TRC robot simulator was constructed. The simulated robot was equipped with as many sensors as the real robot, took into account the dimensions of the robot by reducing its size proportionally and placed the sonars in the same orientation as in the real robot. Furthermore, neither the rules nor the shape of the membership functions were tampered when they were carried over to the simulation. As the purpose of the simulator was to test the controller alone, the simulator assumed that the robot sensors were ideal. So region depth values were computed using equation (2.1). The simulator had also a simplified dynamics to account for the inertia of the robot.

The size of the simulation world is 47.5 unit W $\times$ 40 unit L while the simulated robot size is 4 unit L $\times$ 4 unit W and its perceptual range is 10 unit. The controller was tested in a variety of situations and figure 2.10 shows two such sample situations with fairly dense block obstacles. The arrows inside the two world situations indicate the trajectories the simulated robot followed in each situation.

From the path of the robot one can see that our fuzzy controller drove the simulated robot safely and smoothly, slowing down rarely in areas where obstacles were tight (figure 2.10 above). Saffiotti et al. [112] advocated a fuzzy behavior architecture for atomic tasks, like `avoid obstacle`. However, we argue that there is little benefit in breaking down an atomic task, especially when one employs fuzzy systems. Since, in fuzzy systems we can incorporate all the available information into one clean *monolithic* rule bank. To ground this claim, a monolithic rule bank for an atomic task, we subjected the TRC simulator robot to the same situation (figure 2.10 below) as that used in [112]. As seen from the robot path, the monolithic rule exhibited the same trajectory as reported in [112] without breaking down the task into two modules and later blending them together.

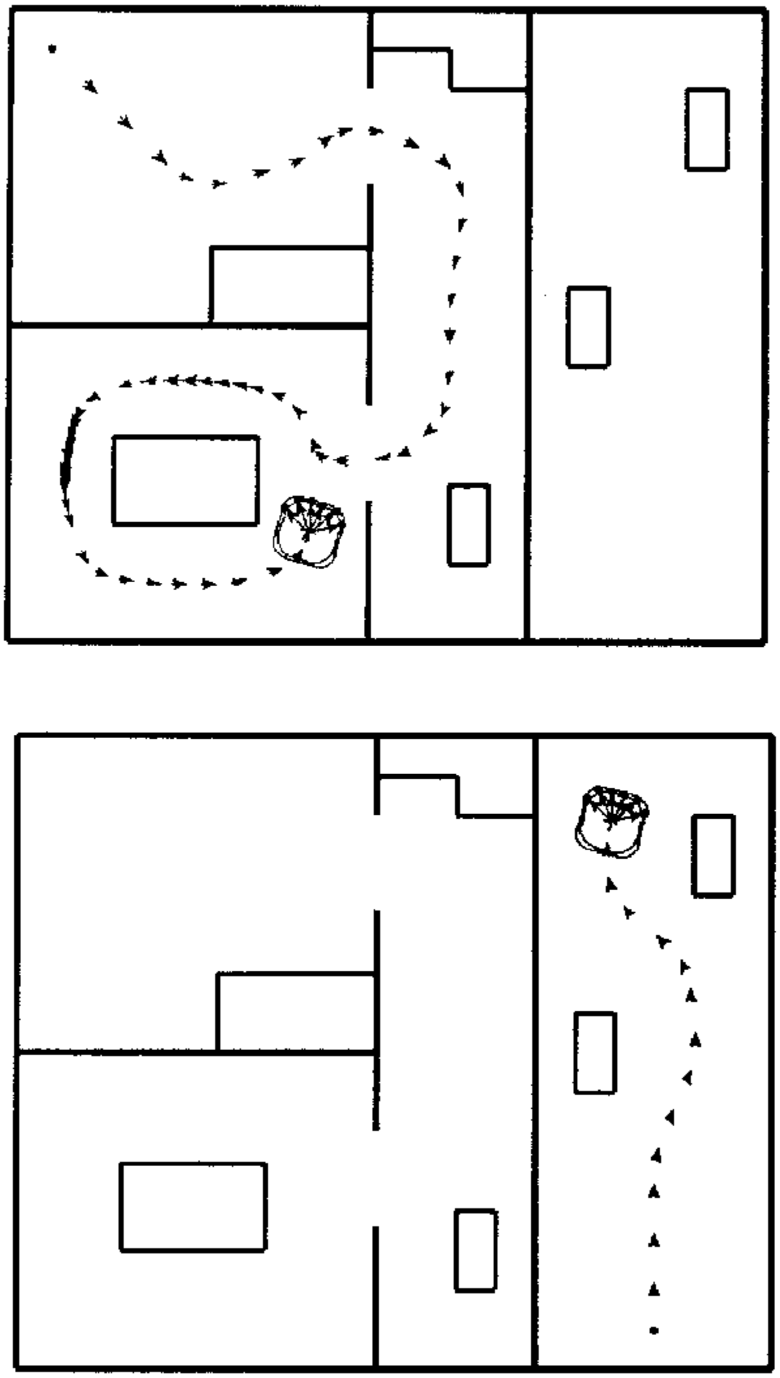


Figure 2.10: *Two sample block world situations of the robot. As shown from the ghost paths, the monolithic fuzzy controller drove the robot safely and smoothly in both situations. The robot slows down only in an environment where obstacles are crowded.*

Real Robot

After the fuzzy controller had been tested and debugged in the simulation, it was transfered to the robot. The world was the robotics laboratory that consisted of narrow doors, a long corridor way and moving objects (humans). Two kinds of experiments were carried out, each using the same controller but different sensor processing. While the first experiment used the approach proposed by Reignier [106], the second experiment used the approach presented here and reported in [44]. Apart from the way they process their sensors, other settings of both experiments were kept identical.

Due to drift error it was difficult to accurately record, as done in the simulation, the robot trajectory on the blue print of the floor map. Fortunately, it was not necessary to have the robot trajectory; it was enough to observe the behavior of the robot in the two experiment to compare the processing methods. Therefore, we will report here only qualitative results.

From the simulation result, we knew that the fuzzy controller had exhibited a collision free navigation behavior; so one would expect a similar behavior on the real robot, too. However, despite the use of the same controller in both experiments, the observed behaviors were quite different, see page 49. In the first experiment, where Reignier's method was used, the robot hardly moved through the long corridor. It turned now and then left or right without a clear forward motion in situations where the robot path was free. In the second experiment, however, the robot moved successfully through the long corridor way passing even through narrow doors that were comparable to the width of the robot (actual results are available on a video). The most striking thing is that the Kalman filter has showed a good success even when all the assumptions were not satisfied, e.g., the sensor model used in equation (2.4). From the result, we concluded that the type of sensor processing employed has made a significant influence on the final controller performance—the behavior of the controller is observed in the second experiment but remain hidden in the first.

The Experiments at a Glance

Experiment title: Collision free navigation of a mobile robot in unknown environment

Robot used: TRC

Sensors used: an array of ultrasound sensors for navigation, bumper for detecting collision and internal sensor for measuring the robot velocity

Motor actions used: continuous translational and rotational velocities

Controller used: a fuzzy logic controller with about 1000 rules

Robot world: a laboratory environment with 100 m long corridor way, three doors each with 1.8 m width and human beings moving around

Experiment I

Sensor processing: the method proposed by Reignier [106]

Observed behavior: the robot has moved haphazardly without a clear forward motion, despite the free corridor way. Also, it couldn't pass through the doors - it was seen often colliding with door edges

Experiment II

Sensor processing: Hashing technique and a Kalman filter proposed by Hailu and Sommer [44]

Observed behavior: The robot was seen moving straight down the corridor, turning sideways when it detected obstacle or human. It also passed through the narrow doors without touching the edges

2.10 Summary

Ultrasonic sensors are a cheap means of acquiring the proximity of objects from the time of flight (TOF) signals. The TOF signals do not deliver high or object level information, rather they measure indirect aspects of the world. In addition, they tend to be highly susceptible to random noise, even carefully calibrated sonars produce unexpected readings. Analytical models of the TOF for some surfaces are derived in the literature [64, 65]. But these closed form solutions assume conditions that are hardly met in practice. In general, the TOF is a complex signal and requires specific interpretation or processing before it can be used directly. To deal with the problem, this chapter has presented a processing technique that involves a hashing and a fusion process.

The most important factor determining the performance of a particular control architecture on a task is its input representation. It is argued that a large state space is not so much a sign of a difficult problem. As Anderson and Rosenfeld put it—a good representation does most of the work [5]. This is universal that applies to any type of controller, hand wired or learned. The best input representation is one that has been extensively preprocessed to make important features prominent. Hashing or sensor aggregation is one such feature extraction techniques. It *matches* sensors to the task by lifting the sensors to object level description. Nevertheless, there is no straightforward method or technique of hashing. Sensors can be hashed in different levels or complexity and it is up to the designer to choose the level that he deems appropriate to the task.

The aim of the fusion process is to integrate sonar data over time so that the robot maintains a consistent picture of its local world. It consists of two filters: the median and the Kalman filters. The median filter chooses the most likely sensor reading of the desired quantity from the aggregate of the sensor snapshots. But it is difficult to get a faithful estimate of the region depth from sensor snapshots alone. Therefore, we have extended the space on which the estimation is based by augmenting to the present reading past sensor readings that are stored in a FIFO stack of fixed memory size. The linear recursive Kalman filter estimates the region depth by processing the data in the stack. The filter works by propagating an assumed conditional probability density distribution through all the stored sensory profiles. At the end, the maximum likelihood (ML) criteria is used to extract the best estimate of the region depth from the conditional probability density function.

The proposed method was compared with Reignier's technique in a number of experiments. His method does use a hashing technique to match sensors with tasks, but throws away all the sensors, except the one that records the minimum. The flaw of this method was first showed in a simple off line experiment where the variation of the distance of the robot is linear. Reignier's method has totally failed to produce the known linear variation of the distance of the robot from a fixed obstacle. On the other hand, the method presented here has successfully generated the expected linear variation. The effectiveness of this approach was then demonstrated by conducting a more realistic on-line experiment that uses the fuzzy logic controller. The controller was driven by the output of the sensor processor. Two identical experiments differing only in the way they process their sensory data were conducted. But, despite the use of identical controllers, the observed final behavior of the robot was quite different. The difference in the observed behavior is attributed to the sensor processing used in the two experiments, thus signifying the role of effective sensor processing that was claimed at the beginning of the chapter.

A fuzzy controller controls a process by modeling the operation of a process by linguistic FAM rules, which form the skeleton of a fuzzy control. Each rule defines a patch and the fuzzy control describes the process operation by covering the input-output spaces with rule patches. A rule is an association between input and output fuzzy sets. Designing a fuzzy logic controller means determining the premise and consequent part of the rules, deciding the form of the rules and writing them down concretely. Before writing the rules, however, three things have to be known: identification of process input and output variables, discretization of their universe of discourse into set of linguistic values, and choice of parameters that ascribe a membership function to each linguistic value.

The major difficulty in the design of a fuzzy controller is that there are no systematic procedures of deriving control strategies. The utility of fuzzy control is explored in a highly experimental fashion; one tries several rules to see which produces the best performance. It is easy to see that this approach is undisciplined from the mathematical and engineering viewpoint. One might even argue that it is unorthodox to use the fuzzy controller, since there is no guarantee of achieving an optimal performance with rules deduced from a human experience. But until a systematic approach is developed to extract knowledge from a human operator, the nature of the current fuzzy control design will remain ad hoc.

Chapter 3

Reinforcement Learning

Who learns by finding out has sevenfold the skill of him
who learns by being told.

Spenser

3.1 Motivations

Neural networks based learning and *control* go beyond monitoring or classifying their input signals to actually influencing them. Unlike most neural networks, these systems are explicitly designed to learn from a closed-loop interaction with their environment. Closed-loop control involves a rather different set of requirements of learning methods than those often considered in neural network research. For example, in control it is much more important to learn on-line, incrementally and without an explicit supervisor specifying desired behavior.

This class of control problems are properly addressed under *reinforcement learning*. Reinforcement learning based neural networks learn and control systems online and incrementally. Work in reinforcement learning dates back to the earliest days of Samuel's checker program [113]. More recently, there have been some advances both in theory and practice of reinforcement learning, notably [7, 27, 58, 121, 127, 136, 143, 150]. In this chapter, we survey previous theoretical and practical works of reinforcement learning that are closely related to the work of this thesis.

In the first part of the chapter, an hypothetical example that allows us to get a clear statement of a reinforcement learning task is provided. Based on the hypothetical example, reinforcement learning is modeled as a sequential decision making process. Next, various task models that ultimately affect the final learned behavior (policy) of reinforcement learning systems are examined. Open issues such as the choice of exploration strategy and the practical difficulty of evaluating reinforcement learning systems are also addressed. In the latter half of the chapter, three learning categories that are widely employed in the general delayed reinforcement learning tasks are described one by one.

The first category, model based learning, is a direct on-line application of dynamic programming techniques that have traditionally been used to solve problems of optimization and control [7, 9]. The second category, model free learning, is a method envisaged when reinforcement learning is first conceived. Under this method Sutton's temporal credit assignment technique, TD(λ) [121, 122], and the two learning methods that make use of TD(λ): Adaptive Heuristic Critic AHC [8, 140] and Q-learning [136, 137] are sufficiently described. The last category, learning by learning world model, is a kind of learning that aims at utilizing the advantages of model based and model free learning methods. In this category Dyna [123] and prioritized sweeping [96, 102] learning methods are surveyed.

3.2 Learning Models

Imagine a man or a computer, hereafter a real or a computational agent, sitting in front of a row of levers, $U = \{u_1, u_2, \ldots, u_n\}$ (figure 3.1). Beside him is a big meter measuring discrete values $R = \{r_1, r_2, \ldots, r_p\}$ that indicates how well the agent is doing at each instance. The blinking lights, $X = \{x_1, x_2, \ldots, x_m\}$, are simply a source of information which the agent can use in deciding which lever to pull and when to pull it. In general, the agent could start out with no knowledge of the lights or the meter, allowing for the possibility of nonlinear fluctuations and noise in the external environment.

The goal of the agent is to learn which lever to pull at each instance of action so that it can optimize a given performance function, J (a similar index has been used in control and optimization theory [22]). For example, if the agent task is to choose actions that tend to maximize the instantaneous meter reading, then the performance of the agent at time t is $J(t) = r_t$, and the goal of the agent is to look for action that maximize $J(t)$, i.e., $\max_{u \in U} J(t)$.

The hypothetical example highlights the basic constituents of a reinforcement learning model. Formally, a reinforcement learning model consists of a discrete set of environment states $x_i \in X$, a discrete set of agent actions $u_i \in U$, and a set of scalar reinforcement values, $r_i \in R$. In reinforcement learning the agent is simply told the task to achieve, which is usually coded into some performance function to be optimized. The agent then learns how to achieve the task by *sequentially* interacting with its environment.

The sequential interaction takes the form of the agent sensing the environment $x_i \in X$, and based on this sensory input choosing an action $u_i \in U$, to perform in the environment. The action changes the environment in some manner and this change is communicated to the agent through a scalar reward $r_i \in R$. The agent then discovers which action yields the highest reward by trial and error process. In the most interesting and challenging cases, action may affect not only the immediate reward but also the next situation, and through that all successive rewards. These two characteristics, trial and error search and delayed reward, are the two most important distinguishing features of reinforcement learning.

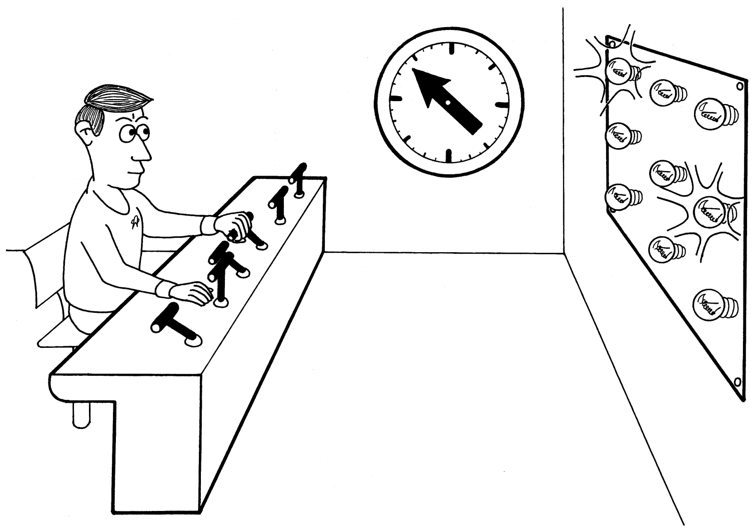


Figure 3.1: *Hypothetical reinforcement learning, taken (with modification) from [140].*

Algorithm 3 is the generic form of reinforcement learning algorithm; the agent improves its behavior through out its life time as long as it receives a reinforcement value from the environment.

The optimality model of the hypothetical example is one of the simplest model, in which the reinforcement is instantaneous and the goal is to optimize local or immediate reinforcement. In this simple model, the reinforcement values at a latter time do not matter and the action maps to be learned are pure functions. Next, we will discuss other interesting models that can handle complex tasks.

initialize the learner
  do forever :
    observe the current world state $x_i \in X$
    choose an action $u_i \in U$
    execute action u_i
    observe the reward $r_i \in R$
    learn from experience tuple $< x_i, u_i, r_i >$

Algorithm 3: *Generic reinforcement learning algorithm [58].*

3.3 Models of Optimality

The choice of the optimality model is crucial in reinforcement learning, because it affects, as we will see shortly, the final learned action. There are interesting and challenging models that capture how the agent should take the future into account in the decision it makes about how to behave now.

Finite-horizon model

The finite horizon model is the easiest model to think about. At a given moment in time, the agent attempts to optimize its expected reward for the next h time steps, i.e., the model looks only h steps ahead and needs not to worry what will happen thereafter. It can be formulated in a non-stationary or stationary policy. In the former case, the agent has a fixed time h and its policy changes as the length of the remaining life time of the agent decreases. At time n, $0 \leq n \leq h$ the agents policy is,

$$J(n) = E\left(\sum_{t=n}^{h} r_t\right) \tag{3.1}$$

In the latter case, however, the agent performs a *receding-horizon control* [60]. That is, the agent always acts according to the same policy, but the horizon h, limits how far ahead it looks in choosing its action, i.e.,

$$J(n) = E\left(\sum_{t=n}^{n+h} r_t\right) \tag{3.2}$$

Infinite-horizon model

This is the most widely used model in the machine learning community. The model takes the long run reward of the agent into account. However, rewards that are received in the future are progressively discounted by a factor γ, i.e.,

$$J = E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \tag{3.3}$$

where $0 \leq \gamma \leq 1$. The discount factor γ determines the contribution of future rewards in selecting the present action. It can also be thought of as a mathematical trick to bound the infinite sum: the ultimate contribution of future rewards to the present action is zero.

Average-reward model

This model takes actions that optimize its long-run average reward, i.e.,

$$J = \lim_{h \to \infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right) \tag{3.4}$$

The policy acquired using this model is referred to as *gain optimal* policy. The average reward optimality model is unable to distinguish between two policies; one of which gains a large amount of reward in initial phases and the other of which does not. In [60] a *biased gain optimal policy* is discussed where rewards obtained during initial phases are used to break ties between two policies that have the same average reward.

An Example

Figure 3.2 is an environment taken from [60] to study the influence of the models' parameters on the final learned policy. Figure 3.3 is plot of the performance of the finite horizon model and the infinite discounted sum model versus their respectively model parameters, i.e., h and γ respectively.

In both plots the optimum policy, the policy that yields a higher optimum reinforcement value, varies with the model parameters. In the finite policy model, the first policy a_0 is optimum for small values of h. However, as the horizon increases the optimum policy changes to a_1, for still higher values of h, a_2 becomes the optimum policy through out. Similarly, for the infinite discounted model case

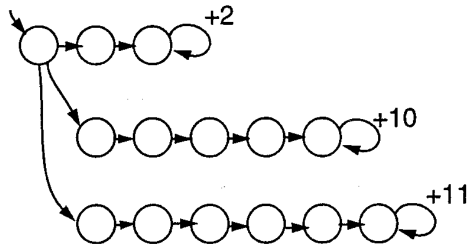


Figure 3.2: *Circles represent the state of the environment and arrows are state transitions. There is only a single policy in every state except the start state, which is the upper left circle. In the start state three possible policies: a_0, a_1, and a_2 can be chosen. When a_0 is chosen it leads to the upper chain. Similarly, when a_1 and a_2 are chosen, they lead to the middle and bottom chains respectively. All unlabeled state transition arrows produce a reward of zero, taken from [60].*

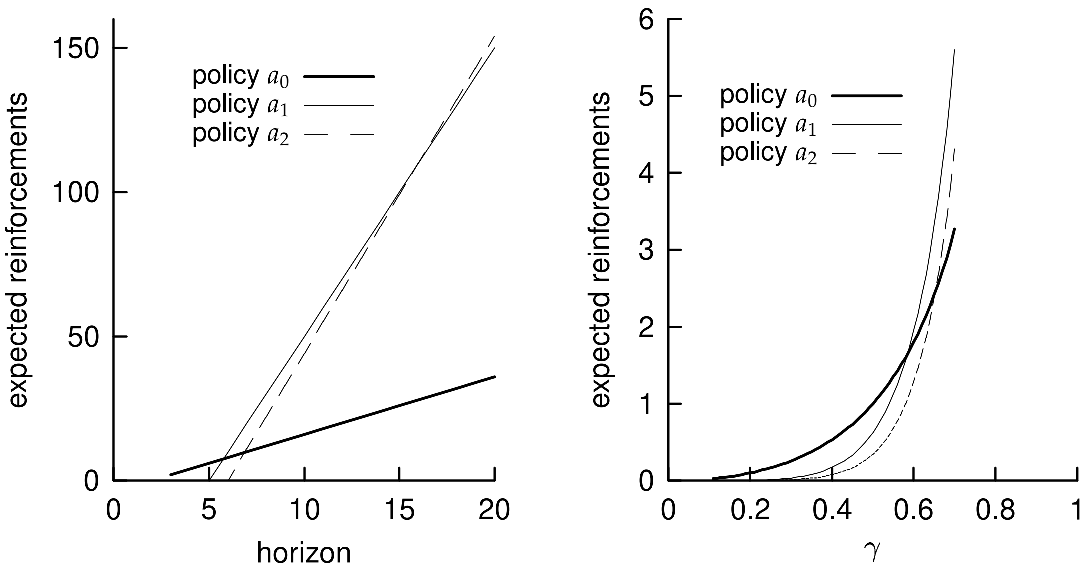


Figure 3.3: *Finite horizon model (left): The total expected reinforcement value of the three policies versus the horizon h. Thick solid line - policy a_0 that leads to upper chain, thin solid line - policy a_1 that leads to middle chain and broken line - policy a_2 that leads to lower chain. Infinite discounted sum model (right): The total expected reinforcement value of the three policies versus the discount factor γ. Thick solid line - policy a_0 that leads to the upper chain, Thin sold line - policy a_1 that leads to the middle chain and broken line - policy a_2 that leads to the lower chain.*

small value of γ yields a_0 as an optimum policy, when γ increases, a_1 takes lead of a_0. Since the average reward model selects a policy that has a long term reward, its optimum policy is a_2 with an average long term reward value of 11.

3.4 Evaluating Learning Performance

While the previous section assesses the final learned policies of different optimality models, this section is about learning performance that tries to evaluate the learning itself. This evaluation is important in order to compare different learning algorithms.

In supervised learning, where parameter estimation is the main concern, it is common to measure the learning performance by the sum of the square error between the output of the learner and the desired target value, section 1.5. But in reinforcement learning, where the target policy is not known in advance, it is not possible to use this kind of measure. One possible approach is to first determine the correct (optimal) policy of an agent and to measure the speed at which the learned policy converges to the correct policy. This performance measure is known as the *eventual time to converge* [82].

Evaluating a reinforcement learning agent based on eventual time to converge is, however, extremely difficult and does not directly apply. The amount of time required for an agent to discover the correct policy depends among other things on the external events that trigger different states in its learning spaces. In addition, noise and errors in the system can make certain parts of the learned policy fluctuate; thus waiting for specific point of eventual convergence is not feasible.

Another weakness of eventual time to converge is that it explicitly divides the agent life time into two parts: a learning phase, during which a performance index is available, and an acting phase, during which there is no performance index. But this arbitrary division is inappropriate because it assumes that learning ceases after the agent converges. In reality, an agent learns life long [131]; it may seem to converge for a certain aspect of the environment, but as it discovers new parts or aspects it goes back to a learning phase again.

As Kaelbling et al. [60] pointed out, eventual time to converge has also additional weakness. It may not be enough to evaluate a learning algorithm only in terms of its eventual time to convergence. An algorithm that tries to achieve optimality as fast as possible may incur large penalties during the learning period.

In this type of situation it is preferable to have a strategy that takes a longer time to converge, but incurring less penalties during its learning.

Another performance measure is *regret* or unhappiness [10]. The regret of a policy is defined as the expected loss of reward due to executing that particular policy rather than the optimal one. Regret is an appropriate performance measure, since it measures exactly what the agent tries to minimize. Actually, the agent tries to gain a high reward but, gaining a high reward and minimizing regret are the two sides of the same coin. This measure has its own weakness, it assumes that the expected reinforcement value of the optimal policy is known.

Due to these difficulties in evaluating the performance of a reinforcement learning agent, what is done in practice is to adopt an intuitive evaluation. In most cases learning proceeds until a *good enough* performance is obtained. There is also a growing interest of looking for good enough solutions in other area of artificial intelligence where it is difficult to find optimal solutions.

3.5 Exploration

Another major difference between reinforcement learning and supervised learning is that a reinforcement learner must explore its environment. To better understand the need for exploration in reinforcement learning, we need to consider the previous hypothetical scenario (figure 3.1). The problem can be reformulated to the *k-armed bandit problem* that was studied in [10]. Each lever i, is called a one armed bandit and when pulled delivers a payoff value of r_i, according to an underlying probability distribution p_i, which is unknown to the agent. The agent can pull any arm but it is only allowed to pull a fixed number of arms, say h. The only cost the agent incurs is when he pulls an arm with a lesser payoff. The task of the agent is to maximize his total payoff within his life time.

Suppose that the agent pulls an arm initially and receives a high payoff (according to what he taught) and decides to play only this arm throughout. Will this strategy maximize his total pay off? Shouldn't the agent explore other arms that can potentially deliver a higher payoff? The heart of the problem is that in order to say a given strategy is optimal, it is necessary to prove that every other strategy will lead to worse payoff values than the value obtained by the given strategy. Therefore, it is essential to do *exploration* in order to determine whether other arms are worse or better than the one obtained. The strategy of always pursuing an action for which its payoff is already known is called *exploitation* strategy [130]. Pure exploitation strategy is not always the optimal strategy.

Hence, a reinforcement learning agent must do enough exploration to discover a new and potentially optimum policy. Exploration ceases only† if the agent finds a strategy that works optimally or as good as necessary. Exploration, however, is not the only issue; an on-line reinforcement learning agent must also optimize its performance by exploiting its best strategy. For simple reinforcement learning problems algorithms, like dynamic programming [10] and Gittins allocation table [38], and ad hoc techniques, like greedy and randomized strategies, exist which work well by balancing exploitation with exploration. Kaelbling et al. [60] have surveyed these techniques. But to date there has been no technique that resolves the tradeoff between exploration and exploitation for more complex and delayed reinforcement tasks. In order to ease the problem of exploration in complex learning task, the agent has to be endowed with a priori knowledge so that it has some expectation of how well it needs to perform in the environment‡.

3.6 Learning in Delayed Reinforcements

In the simplest reinforcement learning method, the immediate reinforcement signal is generated at each time step that gives all of the information the agent needs to know about the success or failure of the action it has just taken. This is a simple instance of the more general case, in which action taken at a particular time may not be rewarded or punished until sometime in the future.

In this section, we will look at existing methods that enable agents to learn which of their actions are desirable based on the reward that can take place arbitrarily far in the future—*delayed reinforcement*.

Markovian Processes

Most computational models of reinforcement learning are based on the assumption that the agent-environment interaction can be modeled as a Markovian Decision Process (MDP). In MDP environment, an agent observes discrete states of the world x ($x \in X$, a finite set) and can execute discrete actions u ($u \in U$, a finite set).

†In a non-stationary environment, where the agents' previous strategy after a while is not valid, the environment has to be explored continually.

‡This is the subject of the next chapter.

At each discrete time step, the agent observes state x, takes action u, observes new state y, and receives immediate reward r. Transitions are probabilistic—y and r are drawn from stationary probability distributions $P_{xu}(y)$ and $P_{xu}(r)$, respectively. Here $P_{xu}(y)$ is the probability that taking action u in state x will lead to state y and $P_{xu}(r)$ is the probability that taking action u in state x will generate reward r. Both probability distributions satisfy,

$$\sum_y P_{xu}(y) = 1 \quad \text{and} \quad \sum_r P_{xu}(r) = 1 \tag{3.5}$$

Viewing the transition probability as a conditional probability, $P_{xu}(y) = P(y|(x,u))$, with x_t and u_t values of x and u at time t, the *Markov* [9, 54, 105] and the *stationarity* conditions are expressed respectively by,

$$P(y|(x_i, u_i\,;\, i = 0, \dots, t)) = P(y|x_t, u_t) \tag{3.6}$$

$$P(y|x_t, u_t) = P(y|x, u) = P_{xu}(y)\ ,\ \forall t \tag{3.7}$$

Deterministic Worlds

A deterministic world is a special case in which all transition probabilities are equal to 1 or 0. For any state action pair (x, u) there will be a unique state $\acute{y}$ and a unique reward $\acute{r}$ such that,

$$P_{xu}(y) = \begin{cases} 1 & : \text{ if } y = \acute{y} \\ 0 & : \text{ otherwise} \end{cases} \qquad P_{xu}(r) = \begin{cases} 1 & : \text{ if } r = \acute{r} \\ 0 & : \text{ otherwise} \end{cases} \tag{3.8}$$

Different Action Sets

The set of actions available may differ from state to state. Those actions that do nothing in a particular state x can be represented within the model as,

$$P_{xu}(x) = 1\ ,\ u \notin U(x) \tag{3.9}$$

Absorbing States

An absorbing state x is one for which,

$$P_{xu}(y) = \begin{cases} 1 & : \text{ if } y = x \\ 0 & : \text{ if } y \neq x \end{cases} \qquad \forall u \in U \tag{3.10}$$

Unless some sort of artificial resetting mechanisms exist, learning stops for all other states once an absorbing state x is reached. In a real environment, where x contains information from the sensors, it is hard to imagine how there could be such a thing as an absorbing state, since the external world will be constantly altering irrespective of what the agent does.

Notes on Expected Reward

When we take action u in state x, the reward we expect to receive is,

$$E(r) = \sum_r r\; P_{xu}(r) \tag{3.11}$$

Often, the reward function is not written in terms of what action we took, but rather what new state we arrived at. That is, r is a function of the transition x to y. Writing $r = R(x, y)$, the probability of a particular reward r is,

$$P_{xu}(r) = \sum_{\{y|R(x,y)=r\}} P_{xu}(y) \tag{3.12}$$

and the expected reward becomes,

$$\begin{aligned} E(r) = \sum_r r\; P_{xu}(r) = \sum_r R(x,y)\; P_{xu}(r) &= \sum_r \sum_{\{y|R(x,y)=r\}} R(x,y) P_{xu}(y) \\ &= \sum_y R(x,y)\; P_{xu}(y) \end{aligned} \tag{3.13}$$

Due to the above equality, the expected reward is often specified in terms of the transition probability $P_{xu}(y)$ and the associated reward $R(x, y)$ [7]. Kaelbling [58] defines a globally consistent world as one in which, for a given x and u, $E(r)$ is constant—this is equivalent to saying that $P_{xu}(r)$ is stationary.

The Learning Task

A policy π specifies the action of the agent in terms of the observed state. A policy that specifies a unique action at each state is called a deterministic policy. On the opposite, a stochastic policy specifies an action u probabilistically from a distribution P_x^π with probability $P_x^\pi(u)$. A policy whose action does not change over time is called a stationary[¶] or memory less policy, i.e.,

$$x_t = x_{t+n} \Rightarrow \pi(x_t) = \pi(x_{t+n})\ ,\ \forall n \tag{3.14}$$

[¶]Policies which are time dependent are called non-stationary policies. For instance, a policy that produce actions u_1, and u_2, alternatively is a non-stationary policy.

Following a stationary policy, the value function $V^{\pi}(x)$, of a state $x(\in X)$, under policy π, is defined as the expected infinite discounted sum of reward that the agent will gain if it starts at that state and executes policy π throughout, i.e.,

$$V^{\pi}(x) = E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \ , \ \forall x \in X \tag{3.15}$$

The task facing the agent is that of determining an optimum policy π^*, that maximizes the value function of equation (3.15), i.e.,

$$V^*(x) = V^{\pi^*}(x) = \max_{\pi} \ E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \ , \ \forall x \in X \tag{3.16}$$

This is the general delayed reinforcement learning problem for which we look for appropriate learning algorithms. There exists a variety of algorithms that are able to learn the optimal policy iteratively.

Model Based Learning

If there is a complete and accurate model of the decision problem, i.e., if transition probabilities $P_{xu}(y)$ and $P_{xu}(r)$ are explicitly known, the optimum value function, equation (3.16), is written as,

$$V^*(x) = V^{\pi^*}(x) = \max_{u \in U(x)} \left(\sum_r r P_{xu}(r) + \gamma \sum_y V^{\pi^*}(y) P_{xu}(y)\right) \tag{3.17}$$

because for the action u, that the agent takes at state x, it receives a reward r, with $P_{xu}(r)$ immediately, and then moves to a state y that is worth of $V^{\pi^*}(y)$, with probability $P_{xu}(y)$.

Equation (3.17) is one form of the Bellman optimality equation [7] which asserts that the optimum value of a state $x \in X$ is the expected immediate reward plus the discounted optimum value of the next state, using the best available current action. Although the optimal value function, equation (3.17), seems circular, it is a well defined system of n (number of states) non linear simultaneous equations which can be solved by dynamic programming algorithm [9]. The solution of these non linear simultaneous equations is *unique*—for any, x, there is a unique value $V^*(x)$, which is the best the agent can do from x. However, there may be more than one optimal policy π_i^*, that satisfies the equality,

$$V^*(x) = V^{\pi_i^*}(x) \ , \ \forall i \text{ and } \forall x \tag{3.18}$$

To find the value of a state from any given arbitrary initial value, a simple iterative algorithm called *value iteration* has been developed [9, 11]. The basic operation in value iteration is backing up estimates of the optimal state value (algorithm 4).

$P_{xu}(y)$ and $P_{xu}(r)$ are the probabilities of taking action u in state x leading state y and generating reward r respectively. $V(x)$ is the value function of state x and γ is the discounting factor.

initialize $V(x)$ arbitrarily $\forall x \in X$
do forever
  loop for $x \in X$
    loop for $u \in U$
      $Q(x, u) = \sum_r r P_{xu}(r) + \gamma \sum_y V(y) P_{xu}(y)$
    end loop
    $V(x) = max_{u \in U(x)} Q(x, u)$
  end loop

Algorithm 4: *Value iteration algorithm.*

This iterative dynamic programming process is obviously an off-line process—the agent ceases interacting with the world while it runs through this loop. Moreover, the algorithm does not require state values to be backed up in any systematic order. There are in fact several variations of the value iteration depending on how the computations are ordered [7, 13].

It is not clear, however, when the algorithm terminates. One method is to test the greedy policy (the policy obtained by choosing in every state the action that maximizes the estimated discounted reward, using the current estimate of the value function) at every update until the resulting solution is considered to be good enough. There are also other terminating criteria discussed in the literature [11, 105, 144]. Among them, the criterion discussed in [144] puts a bound on the performance of the greedy policy. It states: if the maximum difference between two successive value functions is less than ϵ, the value of the greedy policy differs from the value function of the optimal by no more than $2\epsilon\gamma/(1-\gamma)$ at any state.

Once the optimal value function has converged, the optimal policy is easily computed by,

$$\pi^*(x) = \arg \max_{u \in U(x)} Q(x, u) \ , \ \forall x \tag{3.19}$$

There is also another type of DP algorithm called *policy iteration* [8, 13] that manipulates the policy directly, rather than finding it indirectly through the optimal value function. Policy iteration alternates between two phases. The first phase involves policy evaluation in which the value function of the current policy is determined, and the second phase is a policy improvement stage in which the current policy is altered to be greedy with respect to the new value function [7].

Model Free Learning

Unlike the previous section where we discussed model based (indirect) learning of the optimal policy for an MDP, model free (direct) learning is primarily concerned with how to obtain the optimal policy when such a model is not available or difficult to obtain. In this method the agent interacts directly with the environment and compiles the information it gathers into a reactive like structure without learning the model. The method learns the optimal policy through what Watkins [136] described as "incremental dynamic programming by the Monte Carlo method: the *agent's experience*—the state transition and the reward that the agent observes—are used in place of transition and reward models".

Temporal Difference Method

The biggest challenge facing on-line delayed reinforcement learning is how to account for the link between the present action and future consequences - *temporal credit assignment*. There are two basic approaches to meet this challenge. The first approach is to wait until the "end" and reward or penalize every action taken in the past on the eventual outcome. An example of this method is discussed in [140] where the controller is a layered neural network that uses back-propagation through time (BTT) [139] learning algorithm. In BTT, temporal credit assignment is performed by unfolding the network and explicitly computing the effect of each action on the final outcome. However, as Kaelbling et al. [60] pointed out it is difficult to know for ongoing tasks what the "end" is. Besides, even if it is known it might require a great deal of memory.

The other approach is the *temporal difference method* proposed by Sutton [122]. In the temporal difference method, a special network is adapted to learn to associate local reinforcement values with states that are intermediate in time between the action and the external reinforcement. One important idea is to make the local reinforcement value of an intermediate state regress toward the reinforcement value of its *successor*. In the limit, this causes the local reinforcement value of each state to be equal to the expected reinforcement of its successor, and hence equal to the expected final reinforcement. Classes of learning algorithms which work with local reinforcement values efficiently propagate global reinforcement value back along the chain of actions without waiting for the "end".

Two learning algorithms that work with temporary difference have been proposed: AHC-learning due to Sutton [122, 123] and Q-learning due to Watkins [137]. Both algorithms involve learning a local reinforcement value called *evaluation function* and *Q-function* respectively. Below we discuss each learning algorithm for a discounted infinite horizon model.

Adaptive Heuristic Critic Learning

Figure 3.4 shows the block diagram of an adaptive critic network. Two networks are adapted over time: an action network (labeled RL) which outputs the actual control signal, and a critic network (labeled AHC) which guides how the action network is adapted. The critic network uses the real external reinforcement signal r, to learn a heuristic evaluation value of each state $v(x)$, created by the action network. Viewing in another way, the AHC module is learning to transduce the delayed reinforcement signal into local reinforcement signal. The action network in turn learns (modifies its action map) to maximize the heuristic value computed by the critic network.

If we assume that the two networks operates alternatively, there is a close resemblance between AHC learning and the policy iteration that has been briefly described before. If we fix the action network to the current policy π, the critic network learns the evaluation function $V^{\pi}(x)$, $x \in X$ - this is policy evaluation. Next, we fix the critic network, the action network learns a new greedy policy that maximizes the current evaluation function - this is policy improvement. In most applications, however, both networks learn concurrently.

Algorithm 5 formally describes how the critic element learns the value of a policy. The vector V holds the current best estimate of the discounted value of each state, with discount rate γ. The vector e represents the *eligibility trace* of the states. The eligibility trace of a state x is a measure of the degree to which it has been visited recently and is defined to be,

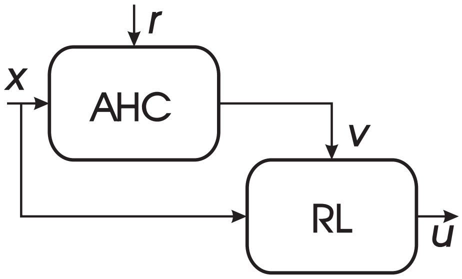


Figure 3.4: *AHC learning architecture - a two component design.*

$< x, u, y, r >$ is an experience tuple that summarizes a single transition. In the tuple, x is the agent's state before transition, u is its choice of action, r is the instantaneous reward after transition, and y is the resulting state. $0 \leq \lambda \leq 1$, $0 \leq \gamma < 1$, and $0 \leq \alpha < 1$.

loop for $z \in X$
  $e[z] = \gamma\lambda e[z]$
$e[x] = e[x] + 1$
loop for $z \in X$
  $V[z] = V[z] + \alpha(r + \gamma V[y] - V[x])e[z]$

Finally, based on the current learning instances, adjust the critic and the action networks using local learning algorithms.

Algorithm 5: *The AHC algorithm.*

$$e(x) = \sum_{i=1}^{t} (\lambda\gamma)^{t-i} \delta_{x,x_i} \tag{3.20}$$

where δ_{x,x_i} is a Dirac function with $\delta_{x,x_i} = 0$; $\forall\, x$, except $x = x_i$ and λ is a parameter that controls the extent to which the eligibility trace is spread backward from the currently active state (figure 3.5). The evaluation values of states are adjusted in proportion to their eligibility, so for $\lambda = 0$ only the current active state value is updated. This is an instance of the more generalized class of algorithm called TD(λ), with $\lambda = 0$. The general TD(λ) rule updates any number of state values by controlling the parameter λ.

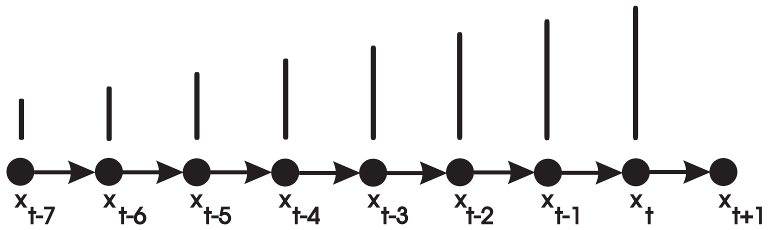


Figure 3.5: *A sequence of states and their eligibility traces shown as vertical bars. States that have been visited recently have high eligibility traces and those that have not been visited recently have low traces.*

For on-line implementation equation (3.20) is written as,

$$\begin{aligned} e(x) &= \sum_{i=1}^{t-1} \lambda\gamma(\lambda\gamma)^{(t-1)-i}\delta_{x,x_i} + \delta_{x,x_t} = \lambda\gamma\sum_{i=1}^{t-1}(\lambda\gamma)^{(t-1)-i}\delta_{x,x_i} + \delta_{x,x_t} \\ &= \begin{cases} \gamma\lambda e(x) + 1 & : \text{ if } x \text{ is current state} \\ \lambda\gamma e(x) & : \text{ otherwise} \end{cases} \end{aligned} \tag{3.21}$$

Before learning starts, the vectors V and e are initialized to zero. At each learning instant the algorithm updates the eligibility trace of all states using equation (3.21). Next the evaluation values of the states in vector V are updated by,

$$V(z) = V(z) + \alpha(r + \gamma V(y) - V(x))e(z)\ ,\ \forall\ z \in X \tag{3.22}$$

Each state value is incremented by the product of its eligibility $e(z)$, the learning rate α, and the one step prediction difference $r + \gamma V(y) - V(x)$. The quantity $r + \gamma V(y)$ is a one-step lookahead value of state x. This update rule is analogous to the sample backup rule of the value iteration (algorithm 4), the only difference is that the samples are now provided by the real world. Any more functional dependence of the value function on the adaptable parameters in the AHC and RL network require the application of local learning algorithms.

Note that the algorithm assigns a value $V(x)$, to each state. This is feasible as long as states are discrete and manageable. When states are continuous, instead of storing the separate state values $V(x)$, we represent the value function by a neural network or some other differentiable function approximator[††] of the form $V(x, w)$, where w is a vector of adjustable weights. With this representation, we still can't directly assign a value to a state, but we *can* adjust the weights so that $V(x, w)$ can accurately approximate the true value function $V(x)$.

[††]See section 5.7 on the specific function approximator used in this thesis.

Q-learning

Q-learning [136] is a simple method for agents to learn how to act optimally in Markovian domains by experiencing the consequence of actions, without requiring them to build a map of the domains. Learning proceeds similarly to Sutton's AHC learning method [121, 122], but the functions of the two networks (AHC and RL) are now unified into a single Q-function. The Q-function defined as $Q(x, u), x \in X, u \in U$, represents the expected discounted reward of taking action u, in state x, and following a greedy policy thereafter.

Let $Q^*(x, u)$ be the expected discounted reinforcement of taking action u in state x, then continuing by choosing actions optimally. Note that $V^*(x)$ is the value of x assuming the best action is initially taken. So,

$$V^*(x) = \max_{u \in U(x)} Q^*(x, u) \tag{3.23}$$

Q-learning work by successively improving its evaluation of the "quality" of a particular action at a particular state. Computationally, it can be viewed as asynchronous dynamic programming [7]; only the experienced (sampled) state action pair is backed up and the control policy may visit them in any order. Formally, Q-learning is described in algorithm 6.

$< x, u, y, r >$ is an experience tuple as described in the AHC algorithm. Q is a matrix indexed by the state $x \in X$ and action $u \in U$ and its elements are initialized to some constant values, normally zero. $0 \leq \gamma < 1$, and $0 \leq \beta < 1$.

$$Q[x][u] = Q[x][u] + \beta(r + \gamma \max_u Q[y][u] - Q[x][u])$$

Algorithm 6: *Q-learning algorithm.*

Like the AHC case, the quantity $r + \gamma \max_u Q[y][u]$ is a one step lookahead value of $Q(x, u)$. Since the one-step lookahead value is a better estimate than the stored value, the update rule adjusts the stored Q value of the previous state and action in the direction of $r + \gamma \max_u Q[y][u]$. The rule also illustrates the temporal difference learning, which allows the agent to learn a chain of actions without waiting for a final reinforcement value. It seems as if the agent can see the future, though in reality it lives only in the present (it can not even predict what the next state will be).

Watkins et al. [137] have proven that Q-learning will converge to an optimal policy, if: 1) the learning rate β with each update takes decreasing successive values $\beta_1, \beta_2, \ldots$, such that $\sum_i \beta_i = \infty$ and $\sum_i \beta_i^2 < \infty$, and 2) each pair of (x, u) is stored in a lookup table and visited by the agent an infinite number of times. In a real system, however, it is impossible to visit each state action pair infinitely. What is done in practice is to approximate Q-learning by carrying out a random policy for a limited span of time and then exploit the knowledge gathered [130]. Sutton [123] suggested a stochastic action selection that uses a Boltzmann distribution, see equation (4.11).

AHC vs. Q-learning

In the inner loop of the learning algorithm, both AHC and Q learning use temporal difference technique to update the function they are trying to learn. However, while the convergence of Q-learning has been proven by Watkins and Dayan [137], it is still unclear whether AHC-learning will always converge and find the optimal control policy. It is only recently that some convergence proofs for a class of adaptive heuristic critics have started to appear [67]. In AHC-learning, there are two concurrent learning processes: learning the value function and learning the policy. Both processes interact closely, a change in the policy will re-define the target value function and an update to the value function will cause the policy to alter. It is possible that these two processes may interact to either prevent or favor convergence. Hence, AHC learning, though biologically inspired, is generally more difficult than Q-learning.

3.7 Learning by Learning Model

In the previous section we have seen two types of approaches for learning an optimal policy for a class of MDP problems. The model based approach learns by *planning* in advance and compiling the result into a set of rapid reactions or situation-reaction rules, which are then used for real time decision making. This approach, however, is limited by its dependence on a complete and accurate probabilistic model of the world. The model free approach arrives at the same optimal policy without knowing the model of the world and without even learning it. It interacts directly with the environment and learns the correct situation-reaction rules. This approach, though guaranteed to find the optimal policy eventually, has its own weakness. It doesn't use the information it gathers from the world optimally. When either the world or the problem is altered a little, the approach requires everything to be re-learned.

Yet a third approach is to integrate the advantages of the two approaches that leads to a type of learning where the agent *learns to plan*. That means the agent learns the world model first in order to plan the optimal reaction. This method has been used in the dynamic programming community [109], but it has a serious practical difficulty when used in on-line learning. In order to learn the world, it must first gather data about the world. But how can the agent gathers data in the absent of planning[‡‡]? Pure random exploration is neither safe nor effective in sampling the world. In addition to the above practical limitation, it decomposes the problem into a learning phase, a compilation phase, and a performance phase [58]. This decomposition doesn't allow the agent to make use of partial information it gathers during the course of learning. Sutton [123] has broken the circular problem between learning and planning in a simple Dyna system.

Dyna

Sutton's Dyna architecture [123] consists of four primary components, interacting as shown in figure 3.6. The architecture is closely related to AHC architecture (figure 3.4), in which the critic and actor networks are concurrently updated. However, Dyna includes also an explicit world model. The world model is intended to mimic the one step input-output behavior of the real world.

Dyna learns from an experience tuple $< x, u, y, r >$ through a trial-and-error learning process. The experience is generated by sampling either the real world—which results in learning or the world model—which results in planning (algorithm 7). Learning and planning results are accumulated in the policy and critic networks.

Sutton [123] has demonstrated the benefit of integrating learning and planning in a deterministic world with a state space small enough to use a lookup table. For each experience with the real world, n hypothetical experiences (planning) were generated with the model. His result shows that intensive planning between real experiences accelerates the learning process.

[‡‡] This is exactly the egg and the hen problem.

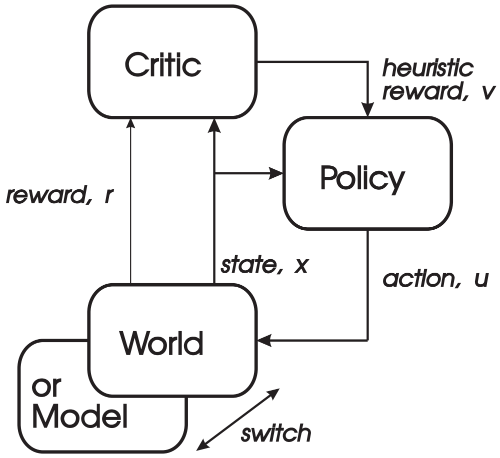


Figure 3.6: *Sutton's Dyna system, Dyna uses real-world experience to learn a model, derives a policy from the model, uses simulated experience and RL to plan [123].*

In the algorithm, $< x, u, y, r >$ is either a real or an hypothetical experience. $0 \leq \gamma \leq 1$.

do
  pick a state x either from the world or world model
  compute *a prior* state value : $\acute{v} \leftarrow \mathrm{AHC}(x)$
  choose action: $u \leftarrow \mathrm{RL}(x)$; execute action u
  obtain next state y and reward r from the world or world model
  compute *posteriori* state value : $v \leftarrow r + \gamma\ \mathrm{AHC}(y)$
learning :
  if it is real experience adapt world models $p_{xu}(y)$ and $p_{xu}(r)$
planning :
  adapt the critic and the policy based on $\Delta = v - \acute{v}$

Algorithm 7: *Dyna algorithm.*

Prioritized Sweeping

Prioritized sweeping is a technique proposed by [96] to accelerate further Dyna's learning algorithms. After each real experience, Dyna picks n random states from the world model to perform planning. This random choice of states does not allow planning to be done in a directed way. It is advantageous to direct planning to regions where learning has taken place. Therefore, instead of sampling states randomly, prioritized sweeping concentrates its planning on the part of states that are more interesting. To identify interesting states, the model of the world is enlarged to hold additional information—each state stores its priority value and all its *predecessors*, states from which it can be reached by one transition.

The algorithm works as follows. After every real experience $< x, u, y, r >$, the priority of all predecessors of state x are promoted to,

$$P_{\acute{x}} = \begin{cases} \max_u p_{\acute{x}u}(x) \times \Delta & : \text{ if } \max_u \; p_{\acute{x}u}(x) \times \Delta \leq \Delta_{max} \\ \Delta_{max} & : \text{ otherwise} \end{cases} \quad (3.24)$$

where Δ is the change in the value of state x and $P_{\acute{x}}$ is the priority of predecessor $\acute{x}$ and Δ_{max} is the highest allowable priority.

The net effect of equation (3.24) is that if the real world makes an interesting transition, say arrival at a goal or collision, Δ becomes high and all states that lead to this event (predecessors of x and their ancestors) are promoted to the top of the priority queue so that their state values and policies get the chance to be updated. If however the real world experience is not new, Δ remains low and the priorities of the predecessors are belittled and the planner continues to concentrate on the previously interesting regions. The performance of prioritized sweeping has been compared with the Dyna and Q-learning on a number of examples. In all cases, prioritized sweeping has reached the optimal policy faster and has required less computation (planning) than Dyna, see [60].

3.8 Summary

Dynamic programming is a field of mathematics that has traditionally been used to solve problems of optimizations and control. Unfortunately, traditional dynamic programming is limited in size and complexity of the problems it can address. Supervised learning is a general method for training a parametrized function approximator, such as neural network, to represent a function or a plant controller. However, supervised neural network can not learn to control the plant unless there is a set of known input-output pairs; so if we do not know how to build a controller in the first place, simple supervised learning will not help.

Reinforcement learning is an approach to machine intelligence that combines the fields of dynamic programming and supervised learning to successfully solve problems that neither discipline can address individually. It is an extension of classical dynamic programming in that it greatly enlarges the set of problems that can practically be solved. Unlike supervised learning, it does not require explicit input-output pairs of training. By combining dynamic programming with neural networks, the machine learning community is optimistic that classes of problems previously unsolvable will finally be solved.

This chapter began by defining reinforcement learning as a non-deterministic Markov decision process (MDP). A non-deterministic MDP consists of a mapping from a given state and action into a probability distributions over successor states. Similarly, the reinforcement model must map states and actions into probability distribution over reinforcement values. If such a mapping is already available, then it is shown that the optimal policy is a solution to the set of equations defined by Bellman optimality equation. The process of learning was subsequently described as the process of improving an approximation of the optimal value function by iteratively finding a solution to this set of equations.

Next, two on-line reinforcement learning methods, namely AHC-learning and Q-learning were presented. Both methods are based on temporal difference to learn an evaluation function of states or state-action pairs. The main idea is to write down a recursive definition of the target evaluation function and then incrementally construct a function to satisfy this definition.

The on-line algorithms work effectively in small domains. As the size of the domain increases, the effectiveness of the algorithms tends to degrade. The degradation is not primarily in the space requirements of the algorithms or the time per learning instance to execute. Rather, it is in the number of learning instances, or interactions with the environment, that the agent must have in order to learn an effective policy. One way to learn with fewer interactions with the world is to allow the agent to do some of its experimentation "in its head" rather than directly with the world, by using an internal model of the world.

Some ancillary issues such as measure of learning performance and the choice of optimality model have been raised. Measures related to speed of learning (e.g., eventual time to converge) have inherent weakness. Many learning algorithms come with a provable guarantee of eventual convergence [67, 137]. This is reassuring, but useless in practice. An agent that quickly reaches a plateau at 99% of optimality is more practical than an agent that has a guarantee of eventual optimality, but a sluggish early learning [58]. Likewise, it has been shown that the choice of the model optimality eventually dictates the final learned policy.

It is noteworthy that throughout the discussion no attention has been paid to the question of how to combine reinforcement learning with prior knowledge, and how to deal with continuous state space domains. These are important questions in the development of learning system capable of solving truly complex tasks.

Chapter 4

Prior Knowledge in Learning

... Therefore, progress toward understanding learning mechanisms depends upon understanding the sources of, and justification for, various biases.

Mitchell

4.1 Motivations

Reinforcement learning is widely regarded as elegant in theory but hopelessly slow in practice. This is because it is often studied under the assumption that there is little or no prior information about the task at hand. But this assumption is not the defining characteristic of learning. Learning, whether it is reinforcement or any other type, by no means entails a tabula rasa view. Rather on the contrary, it involves the incorporation of a priori knowledge or bias that can greatly accelerate or otherwise improve the learning process.

Biasing, once regarded as "cheating" in the machine learning community, is now understood and accepted as a necessary part of designing a useful learning system. In the past, reinforcement learning has been applied in many areas, ranging from robotics [76, 82, 87, 88], to industrial manufacturing [16, 27], to combinatorial search problems such as computer games [127, 128]. The most striking part of all these applications, except the last one, is that it has proved necessary to supplement the fundamental algorithm with pre-programmed or biased knowledge.

Though biasing a learning system is agreed as a necessary and crucial step, it is not yet clear by how much and with what quality the system should be biased. Intuitively, the more human effort and insight is, the less time is required to learn.

In the limit, however, the system becomes less autonomous and non-interesting. The challenge is, therefore, the amount, quality, and way of expressing this bias in a systematic way that will give enough inductive leap to the learning system [91].

This chapter addresses the challenge for a particular reinforcement learning task. The task is a typical navigation task where an agent in a given environment learns the shortest path to the target position. First, a variety of biases that can be of help in tackling the problem will be identified. We define a performance index that measures the effectiveness of each bias in levering the learning process. The index is the relative gain a particular bias brings in the reduction of the average number of actions taken (which is directly linked with the agent's learning time) before the path is learned. Based on this index the influence of the various biases on the learning process is assessed. The results presented in this chapter, to the best of the author's knowledge, are the first attempt to shed light on the influence of different forms of biases in reinforcement learning.

In addition to the proper choice biases, learning can also be accelerated by allowing agents to use previously learned policy to adapt to a new policy. That is, if agents follow a continual learning scheme rather than learning every time from scratch. A section is also devoted to continual learning and we will presents conclusive results that supports the above claim. We will begin the chapter by discussing the bias variance dilemma.

4.2 Bias-variance Dilemma

Reinforcement learning is a stochastic learning process where the stochasticity is in the choice of action and as a consequence in the resulting perception. We adopt the view that adapting the synaptic weights in a parametric function is one method in which empirical knowledge about the environment may be encoded in a model [67]. By empirical we mean that our knowledge of the environment is due to a set of measurements that characterize a phenomenon we wish to describe or predict.

Let us now consider predicting the value function V, with a model $V = f(\boldsymbol{x})$, where the vector $\boldsymbol{x}$ is the state of the environment for which we want to predict its value. In accordance with the reasoning below, the *regression* is the best predictor of the value in the mean square sense [37]. For any function $f(\boldsymbol{x})$ and any state $\boldsymbol{x}$,

$$\begin{aligned} E\left\{[V - f(\boldsymbol{x})]^2 \mid \boldsymbol{x}\right\} &= E\left\{[V - E\{V \mid \boldsymbol{x}\} + E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x})]^2 \mid \boldsymbol{x}\right\} \\ &= E\left\{[V - E\{V \mid \boldsymbol{x}\}]^2 \mid \boldsymbol{x}\right\} + [E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x})]^2 + 2 \times \\ &\quad\; E\left\{[V - E\{V \mid \boldsymbol{x}\}] \mid \boldsymbol{x}\right\} \times [E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x})] \\ &= E\left\{[V - E\{V \mid \boldsymbol{x}\}]^2 \mid \boldsymbol{x}\right\} + [E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x})]^2 \\ &\geq E\left\{[V - E\{V \mid \boldsymbol{x}\}]^2 \mid \boldsymbol{x}\right\} \end{aligned} \tag{4.1}$$

The regression,

$$E\{V \mid \boldsymbol{x}\} \tag{4.2}$$

is that function of $\boldsymbol{x}$ that gives the mean value of V conditioned on $\boldsymbol{x}$. Due to the appearance of the expectation operator, the evaluation of the regression requires an infinite number of samples. Therefore, the regression can not be represented in a closed form, it is simply a mathematical entity which at best is represented by an infinite parametrized function,

$$E\{V \mid \boldsymbol{x}\} = g(\boldsymbol{x}; \boldsymbol{w}) \quad \text{where} \quad \boldsymbol{w} = \begin{pmatrix} w_0 \\ \vdots \\ w_\infty \end{pmatrix} \tag{4.3}$$

Error Decomposition

What we can aim for is an approximation of the value function by a *fixed* parameterized model such as a feed forward neural network whose output is $f(\boldsymbol{x})$. To be explicit about the dependence of the approximating function on the parameter, we write $f(\boldsymbol{x}; \boldsymbol{w})$ instead of simply $f(\boldsymbol{x})$, where $\boldsymbol{w}$ is the weight vector that depends on the size of the network employed. Again, the natural measure of the effectiveness of the approximating function as a predictor of the value V, is the mean square error,

$$E\left\{[V - f(\boldsymbol{x}; \boldsymbol{w})]^2 \mid \boldsymbol{x}, \boldsymbol{w}\right\} \tag{4.4}$$

Using equation (4.1), the above mean square error can be decomposed into two terms,

$$\begin{aligned} &E\left\{[V - f(\boldsymbol{x}; \boldsymbol{w})]^2 \mid \boldsymbol{x}, \boldsymbol{w}\right\} = \\ &\qquad E\left\{[V - E\{V \mid \boldsymbol{x}\}]^2 \mid \boldsymbol{x}\right\} + [E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x}; \boldsymbol{w})]^2 \end{aligned} \tag{4.5}$$

The first term does not depend on the weight vector of the estimator. It is simply the variance of V given $\boldsymbol{x}$ and so can not be minimized. It is only the second term, the squared distance to the regression that can be minimized and measures the effectiveness of $f(.;.)$ as a predicator of the value V. Similar to equation (4.1), the mean-squared error of the approximating function as an estimator of the regression is,

$$E_N\left\{[E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x};\boldsymbol{w})]^2\right\} \tag{4.6}$$

where E_N represents the expectation with respect to the fixed network weights,

$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} \tag{4.7}$$

A useful way of assessing the sources of estimation error is via the bias variance decomposition, which can be derived in a similar way to equation (4.1),

$$\begin{aligned}
& E_N\left\{[E\{V \mid \boldsymbol{x}\} - f(\boldsymbol{x};\boldsymbol{w})]^2\right\} \\
= \;& E_N\left\{[E\{V \mid \boldsymbol{x}\} - E_N\{f(\boldsymbol{x};\boldsymbol{w})\} + E_N\{f(\boldsymbol{x};\boldsymbol{w})\} - f(\boldsymbol{x};\boldsymbol{w})]^2\right\} \\
= \;& E_N\left\{[E\{V \mid \boldsymbol{x}\} - E_N\{f(\boldsymbol{x};\boldsymbol{w})\}]^2\right\} + E_N\left\{[E_N\{f(\boldsymbol{x};\boldsymbol{w})\} - f(\boldsymbol{x};\boldsymbol{w})]^2\right\} + 2 \times [\\
& E\{V \mid \boldsymbol{x}\} - E_N\{f(\boldsymbol{x};\boldsymbol{w})\}] \times E_N\{E_N\{f(\boldsymbol{x};\boldsymbol{w})\} - f(\boldsymbol{x};\boldsymbol{w})\} \\
= \;& E_N\left\{[E_N\{f(\boldsymbol{x};\boldsymbol{w})\} - f(\boldsymbol{x};\boldsymbol{w})]^2\right\} + [E\{V \mid \boldsymbol{x}\} - E_N\{f(\boldsymbol{x};\boldsymbol{w})\}]^2
\end{aligned} \tag{4.8}$$

The essence of the bias variance dilemma lies in equation (4.8), which decomposes the estimation error into two components, known as bias (second term) and variance (first term). Either the bias or the variance can contribute to poor performance. Incorrect models lead to high bias whereas model-free inference suffers from high variance. Thus, a learning system that starts with a tabula rasa is slow to converge, as it requires large training samples to achieve acceptable performance. This is the effect of high variance, and is the consequence of the large number of parameters, indeed an infinite number in truly model-free learning, that need to be estimated. Prohibitively large training sets are then required to reduce the variance contribution to the estimation error. The only way to control the high variance is to introduce the right bias that properly constrains the problem and thereby mitigates the issue of large training samples. However, this is the other side of the dilemma; for a complex system, it is difficult to determine how to bias the system toward the most important aspects of the problem and any bias is likely to be incorrect.

4.3 Classes of Biases

Bias, in the context of learning, is the term used to describe a learning system's predisposition for learning something at the expense of others. By having varying degrees of built-in structure, learning networks can fall almost anywhere in the continuum from unbiased to highly biased memory systems. Lookup tables are near the unbiased end of this continuum because they do not impose constraints other than a certain grain of quantization on the data they store. Whereas memory systems are near the highly biased end that assume specific functional relationships between their inputs and outputs. Highly biased memory systems generally have few degrees of freedom, but the form of their bias enables them to generalize beyond the data with which they have direct experience.

Examples of memory systems are connectionist networks. Connectionist networks are *data-driven* learning system that are biased by the structure of the network and the training set. They have been shown to be sensitive to initial conditions, very specific and of limited ability to generalize. On the other end of the data-driven learning lie *knowledge-based* learning schemes. They employ some form of domain knowledge in order to minimize the amount of deduction left to the agent, as well as the the amount of new information needed from the world. Explanation based learning (EBL) [30] and explanation based generalization (EBG) [92] belong to this category. These approaches are constrained by the structure and amount of information provided by the domain and rely on its completeness and accuracy. These properties have earned them the label *strong* methods as compared to *weak* connectionist approaches [52].

Reinforcement learning is situated between the two extremes. In order to avoid preprocessing the raw data, reinforcement learning approaches manipulate the raw input vector. To establish a correlation between each state and the desired action, the algorithm searches through the entire space of state-action combinations—requiring a large amount of trials to find the optimum policy. In contrast, knowledge-driven learning approaches rely on very few carefully constructed examples, since they encode most of the domain knowledge in the system.

In this section we will examine some useful classes of bias that have been dominant in the past.

Modularization

The problem of learning the optimal policy, in general, can be cast as searching for a path in action space which connects the current state with the goal state. The longer the distance between a state and the goal, the longer it takes to learn the policy or the path [142]. This is why policies for large state spaces take a longer time to be learned. Breaking the task into modules or sub-tasks effectively shortens the distance between the reinforcement signal and the individual actions. Consequently, the length of the action sequence to be learned is decreased.

We will clearly be interested in a kind of bias that *decomposes* the task into modules (sub-tasks) which can work with smaller state spaces and simpler reward functions and have some method of prioritizing scheme to coordinate or resolve conflicts among the various modules. A *gated behavior* architecture (figure 4.1) allows this kind of bias to be incorporated into the learning system [55]. It consists of a collection of behaviors [17] each handling a specific module and a gated network that decides which of the behavior(s) should be used at each instant. Since breaking up the problem into an appropriate set of behaviors requires domain information, this part is entirely done by hand. Nevertheless, the behaviors themselves could either be hand wired or learned.

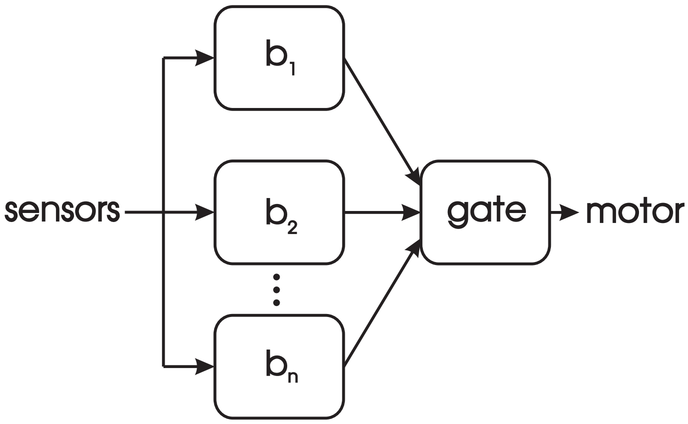


Figure 4.1: *The global task is decomposed into several sub-tasks and later combined by the gate. Since learning takes place in the individual sub-tasks that have manageable states, the algorithm converges quicker on the sub-tasks than on the global task.*

Moore [94] and Maes & Brook [74] have used pre-wired behaviors and learned only how to coordinate the behaviors, while Mahadevan and Connell [76] have used the dual approach where they manually fixed the gated network and learned the individual behaviors by reinforcing them separately. Singh [114] and Tham & Prager [129] have dealt with problems where the sub-tasks have termination conditions and the gate have combined the sub-tasks sequentially to solve the main problem. In general, however, the gate is concerned with sub-tasks acting in parallel and interrupting each other rather than running to completion. Typically, each sub-task can only be satisfied partially [73].

Even though the modularization of a huge learning problem is a very powerful technique of biasing, it is unlikely that any universal strategy for dividing a task into a collection of small tasks exists. To date, task decomposition is done entirely on a problem basis. But it would be useful to derive a few principles of task decomposition at least for a particular class of learning problems. Another interesting question is whether the modularization of a task is dependent on the learning algorithm, i.e., whether there exists some optimal set of modules which is independent of the way modules are learned, but is tied instead to the semantics of the problem [82].

Advice

Another way of biasing a reinforcement learning agent is by allowing the agent to accept advice given, at any time in a natural manner, by an external observer. Figure 4.2 shows the structure of a reinforcement learning agent with an observer that provides advice. The actual advice taking mechanism can be implemented in many different approaches. One approach often used in real robots is that the observer occasionally imparts appropriate motor commands to the robot through external devices such as a joystick or a steering wheel [89]. Another approach, mostly used with computational agents is that, the observer expresses his advice using a simple language construct and a list of task-specific terms [72].

The two main constructs used in advice-taking language are: `if-then` rules and loops, both `while` and `repeat`. In each of these constructs, the observer lists logical combinations of conditions (basic sensors and derived features) in the preconditions and specifies either a single action or a plan in the post-conditions. One such advice-taking language is shown in table 4.1, where an observer provides advice to the agent learning to escape an enemy. The `if-then` construct actually serves two purposes. It can be used to specify an action to be taken in a particular situation or to create a new intermediate term; in this case, the conclusion of the rule is some descriptive term based on the sensed feature.

Most advice is too imprecise for the learning agent to understand. Therefore, the observer uses fuzzy logic (section 2.6) to articulate the imprecise advice by creating an explicit mathematical expression that determines the fuzzy truth values of the preconditions as a function of the sensor values. Furthermore, due to the imprecise nature of the advice, the advice expressed in these language constructs is translated directly into a neural network for further tuning [68, 72, 107].

The general advice taking framework involves three major sequences:

- request/receive advice,
- integrate the advice into the agent's knowledge, and
- evaluate the value of the advice.

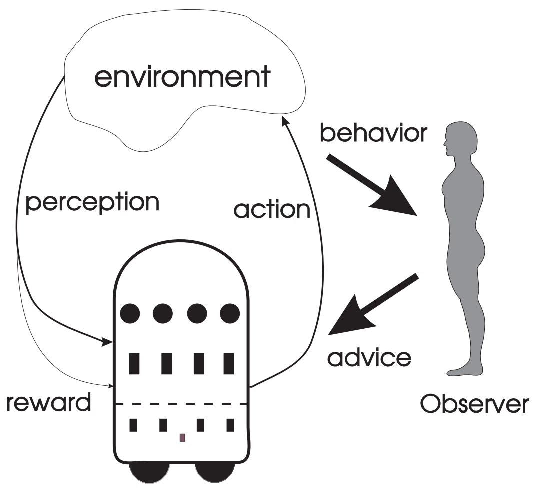


Figure 4.2: *The basic reinforcement learning agent is augmented with a process that allows an observer to watch the agent and to suggest advice based on the agent's behavior.*

Request/Receive Advice

To begin the process of advice taking, a decision must be made that advice is needed. Often, approaches to advice focus on having the agent ask for advice when it needs help [25]. This approach, however, places more burden on the observer, as the agent may require advice more frequently than the human advisor is willing to provide. The other approach, in which the external observer provides advice whenever the observer feels it is appropriate [72], does not require a lot of interaction between the agent and the observer. Besides, it is an open question how to create the mechanism for the agent to recognize or express its need for advice.

Verbal	Advice

If an enemy is near and west and an obstacle is near and north, hide behind the obstacle.	`if (Enemy (Near && West) &&`
	`    Obstacle (Near && North))`
	`then`
	`  Move East`
	`  Move North`
	`end`

Table 4.1: *An* `if-then` *advice-taking construct [133]. Before using the advice the learner converts it into a collection of directly interpretable statements.*

Integration

Integration is concerned with incorporating the acquired advice into the controller of the agent. Incorporating the observer advice calls for updating the structure of the agent's neuro-controller. In most cases, the idea of knowledge based global neural networks [68, 107, 133] is utilized to directly install the new advice into the agent. Since in a knowledge based neural network a set of propositional rules is represented as a neural network, it converts a rule set into a network by mapping the target concept of the rule set to the output unit and creating hidden units that represent the intermediate conclusions. Figure 4.3 illustrates the general approach used for adding advice into a knowledge based network.

Evaluation

The final step of an advice taking process is to evaluate the advice. The evaluation can be from the agent's point of view, who must decide if the advice is useful or from the observer's point of view, who must decide if the advice has the desired effect on the behavior of the agent. The agent evaluates the advice by continuing the operation in the environment; the feedback provided by the environment offers a crude measure of the advice's quality. Similarly, the advisor judges the value of his advice by watching the agent's post-advice behavior.

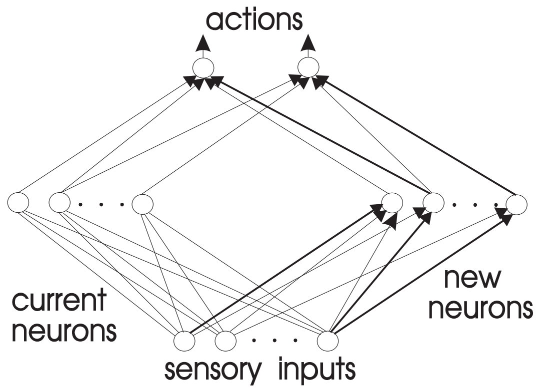


Figure 4.3: *Incorporating advice involves adding to the existing units new hidden units. The thick links capture the semantics of the advice and the thin links initially have zero weights; during training their weights change so as to refine the original advice [71].*

Reflex

It is often said that "one can not learn anything unless one almost knows it already" [145]. One of the weaknesses of reinforcement learning agents is that whenever they perceive a new situation they do not know where to search for interesting parts of the action space [88, 115]. Often, they either wander around never getting to the goal or will be killed immediately after a pre-set trial time is elapsed. A way of overcoming this problem is to program a set of reflex rules that enable agents to behave in some reasonable way.

Reflex use domain knowledge to restrict the set of actions to be explored, namely the optimal action map is within a restricted class. A learning agent that uses this form of bias works by continually interacting with the reflex component (figure 4.4). Each time the learner fails to generalize its previous experience to the current situation, it invokes the reflex component. Upon request, the reflex component provides an initial safe and near optimum action. Subsequently, the learner overrides the acquired reflex action by a more accurate learned action.

Reflex based learning system involves three phases. Initially, it operates in *reflex mode* by relying more often on the reflex to determine a rough-cut task performance. As learning proceeds, however, the learner starts to suggest its own actions for situations that it can generalize but continues to rely on the reflex for other situations, *hybrid mode*. Finally, after acquiring sufficient situation-action pairs, the reflex no longer or seldom intervenes with the learner, which is now operating completely in *reinforcement mode* [49].

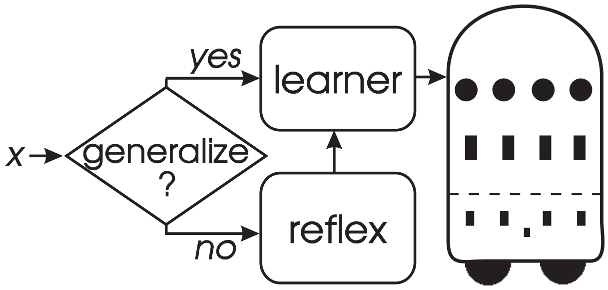


Figure 4.4: *Reflex guiding the learner to concentrate its exploration in regions where it is most needed, adapted from [88].*

Domain-rich Reward

The amount and quality of the reinforcement signal determines how quickly an agent will learn. In non-deterministic and uncertain world, learning in bounded time requires the shaping of the reinforcement signal in order to take advantage of as much information as is available to the agent.

Rather than encoding knowledge explicitly, a reinforcement learning method hides it in a single monolithic reward function. The reward function can be made to encode domain knowledge, thus biasing what the agent can learn. Simplifying or lumping the reinforcement signal diminishes this bias, but it also handicaps, and in real domains, completely debilitates the learner. Domain knowledge can be embedded through a reward rich and complex reinforcement function. Matarić [83] proposed two methods of shaping the reward function to accommodate domain knowledge: *heterogenous rewards* and *progress estimators*.

Instead of an impulse reward, a heterogenous reward provides large amount of intermediate reinforcements to aid the agent in learning. The central argument in generating intermediate reinforcements is that agents maintain and pursue multiple concurrent goals. These goals are maintained and achieved by using behaviors as basic building blocks of control and learning [17]. Thus, a task can be represented with a collection of such concurrent goal achieving behaviors. Reaching each of the goals generates an event that provides primary reinforcement to the learner. Intuitively, the more subgoals are used, the more frequently reinforcement can be applied, and the faster the learner will converge.

While heterogenous rewards, which are generated when an agent accomplishes sub goals, are intermittent, progress estimators are immediate feedback to the agent and are generated during a behavior. Progress estimators use domain knowledge to measure the progress of a behavior, and if necessary, terminate a behavior. An agent has no impetus for terminating a behavior and attempting an alternative, since any behavior may eventually produce a reward. The learning algorithm must use some principled strategy for terminating a behavior. Progress estimators provide such a method: if a behavior fails to make progress relative to the current goal, it is terminated and another behavior is tried. Hence, instead of an arbitrary or random behavior selection, progress estimators induce exploration by terminating behavior according to common sense [82].

4.4 The Learning Problem

To study the influence of the amount and quality of bias in reinforcement learning, a deterministic world with denumerable states is considered. Moreover, the agent is assumed to be a point robot with simplified motor actions (such as move to the next square and turn 90 degrees). Such a robot world configuration is often called a *labyrinth* world. The labyrinth world is a highly simplified scenario of a real robot world. It is unrealistic to think of a dimensionless robot or denumerable world states. Similarly, it is impossible to throw away the details of low level control and deal with only simplified motor actions.

Nevertheless, despite these unrealistic assumptions, we have based the experiment on the labyrinth world for three justifiable reasons.

Cost

Since the influence of different types of biases on the learning speed is to be investigated, a reinforcement learning experiment has to be set up as many times as the number of available biases. Usually, however, each reinforcement learning experiment requires a large number of expensive learning trials, expensive in many ways: wall clock time, power consumption, danger to the robot and to the surrounding, etc. Techniques such as Dyna [123], experience replay [69], transition proximity Q-learning [6], and asynchronous dynamic programming [7] are all examples of efforts to cut this expensive learning trial by substituting world experience with storage and computation. Therefore, it is clearly expensive to carry out a reinforcement learning experiment on a real robot for each and every bias introduced into the learning system.

Representation

Once again, the main goal is to identify those bias types that enhance learning. Clearly a problem domain that enables us to vary the strength of the bias and also qualify this variation is an important consideration. As we shall see shortly, external inductive biases that are hard to manipulate in real domains can be easily manipulated and represented in labyrinth domains.

Noise

Even if we decided to undertake the experiment on a real robot, there is a danger of coming up with an incorrect conclusion. Varying the bias and studying the learning performance of a physical agent is notoriously difficult. As we have stated in section 3.4, noise and error can make certain parts of the agent policy to fluctuate. So, despite the fact that the learning system is appropriately biased, due to noise and error it may still exhibit a bad performance, unless it is smoothed out by averaging over a large set of experiments.

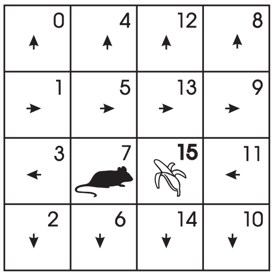
Therefore, a more efficient and inexpensive method is to perform the experiment in an artificial world, that requires much less experimental effort than running on the real domain and yet to come up with a domain free proposal that suggests the best way of biasing a reinforcement learning system. The proposed bias can then be built into the real robot to provide faster learning—this is the approach we followed.

The Labyrinth World

The robot world, figure 4.5(a), consists of 16 states or cells, one of which, marked bold, is identified as a target state and any of the states can be chosen as a start state[†]. It is assumed that all the states are distinct and completely distinguishable. Furthermore, there are three possible actions: `left`, `forward`, and `right` which the agent can choose from. All actions can be tried in all states. Figure 4.5(b) defines the state transition as a function of the present state and action taken. The navigation task is to learn to reach the goal state that holds food through the shortest path. Reward is zero for all transitions except for those into the goal state, in which case it is +1. Upon entering the goal state, the system is instantly

[†]Previously, Matarić [80] used this same world to analyze and compare the performance of Q-learning and the Bucket Brigade algorithm [53].

transported back to the start state to begin the next trial. Attempting an action against the world boundary does not change the state. None of this structure and dynamics is known to the learning system a priori.



(a)

state	action: left	action: forward	action: right
↑ 0 4 12 8	← 3 7 **15** 11	↑ 0 4 12 8	→ 1 5 13 9
→ 1 5 13 9	↑ 0 4 12 8	→ 5 13 9 9	↓ 2 6 14 10
← 3 7 **15** 11	↓ 2 6 14 10	← 3 3 7 **15**	↑ 0 4 12 8
↓ 4 6 14 10	→ 1 5 13 9	↓ 2 6 14 10	← 3 7 **15** 11

(b)

Figure 4.5: *Two-dimensional maze problem: (a) The point robot must find the shortest path from any start state to the goal state. (b) Its state transition table.*

4.5 Belief Matrices

As discussed in section 4.3, bias comes in different forms and shapes different parts of the reinforcement learning components. For example, domain rich heterogeneous reinforcement is fundamentally used to ease the problem of temporary credit assignment. Likewise, reflex is primarily used to focus exploration.

In this work, I have used *belief matrices*, a version of reflex for discrete state-action space that restricts the set of possible hypothesis by putting a belief value on each hypothesis, so that a strong negative belief is needed to eliminate an hypothesis from consideration. An hypothesis is a pairing of any state with any action and is associated with a value that represents the belief about the appropriateness of that action in that state. In short, belief values either eliminate or put preference on the set of possible actions that can be tried at each state by encoding domain knowledge in a matrix, equation (4.9). In general, for any state $i \in \{1, \ldots, p\}$, and any two actions j and k, $i, j \in \{1, \ldots, q\}$, $j \neq k \Rightarrow b_{ij} \neq b_{ik}$.

$$\boldsymbol{b} = \begin{pmatrix} b_{11} & b_{12} & b_{13} & \ldots & b_{1q} \\ b_{21} & b_{22} & b_{23} & \ldots & b_{2q} \\ & & & \vdots & \\ b_{p1} & b_{p2} & b_{p3} & \ldots & b_{pq} \end{pmatrix} \tag{4.9}$$

4.6 Bias Design

For the navigation problem of figure 4.5, four different types of biases are considered, namely

- `unbiased`,
- `environment`,
- `insight`, and
- `goal`.

The choice of these biases is primary guided by the particular task at hand. Other tasks may require different forms of biases. It is worth noting that in the case of tasks not requiring reaching a given destination, biases such as `goal` are neither useful nor available. But other biases such as `environment` are more generic, and could be applied across tasks. While each of the above biases defines the quality, the amount in each type can be varied by altering the elements of the corresponding belief matrix.

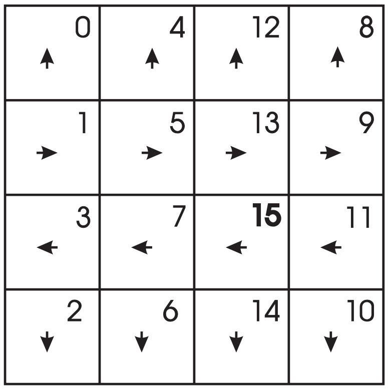
Unbiased, $\mathbf{B}_0$

In this case, the agent does not know beforehand the nature of the problem; therefore, its action selection strategy is *optimism in the face of uncertainty*. That is, at each state all actions are equally preferred or the belief matrix is like an unused blackboard and learning proceeds from scratch, figure 4.9(b).

Environment Bias, $\mathbf{B}_1$

In this type of bias, a part of the environment knowledge that informs the agent to stay away from likely collisions is encoded in the belief matrix. Under this category, two forms of biases are identified.

The first form $\mathbf{B}_{10}$, excludes those actions that have *immediate* consequences. Referring to figure 4.7(a) among the three actions at cell #0, the action `forward` has an immediate consequence of collision with the world boundary. Hence, in the belief matrix, figure 4.7(b), this action is discriminated by putting a high negative value so that it will not be chosen by the action selection mechanism.



(a)

state \ action	left	forward	right
↑ 0	1	1	1
→ 1	1	1	1
↓ 2	1	1	1
← 3	1	1	1
↑ 4			

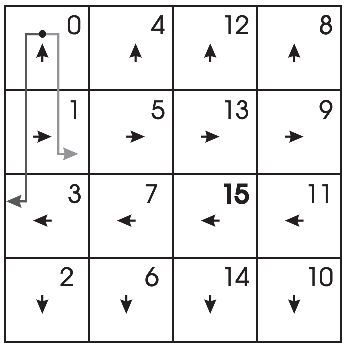
(b)

Figure 4.6: *Unbiased: a) No prior knowledge is encoded in the state space. b) The elements of the belief matrix have identical values.*

The second form $\mathbf{B}_{11}$, which is the more general case, looks one step ahead and puts preference to actions in the belief matrix. That means, actions that have *potential* consequences after a transition has occurred are less preferred to those actions that do not have consequences. Again referring to figure 4.7(a), like bias $\mathbf{B}_{10}$, the `forward` action at cell #0 is excluded in this case, too. In addition, the action `left` potentially results in a collision with the world boundary (if a forward action is chosen after a transition has taken place), but the action `right` is safe since it does not bound the robot to the world boundary. Therefore, this form of bias places at cell #0 of the belief matrix a higher preference to the `right` action than to the `left` action, figure 4.7(c).

Insight Bias, $\mathbf{B}_2$

This type of bias tries to exploit the unique characteristics of the problem. Every problem has its own unique characteristics that could be of a great help, if discovered, in solving the task. For the described problem, since the goal is placed in the third column, the main strategy of the agent should be to arrive at this column first before heading to the right destination cell. A close look at the task reveals that for some states there is more than one choice of action that the agent can choose from, but all leading to the same end effect. For instance, if the agent is at cell #3, its immediate strategy must be to reach cell # 1 so that it can choose the forward action and leave that column. This can be accomplished by choosing either of these two sequences of actions: {`left left`} or {`right right`},



(a)

state \ action	left	forward	right
↑ 0	1	-1	1
→ 1	1	1	1
↓ 2	1	-1	1
← 3	1	-1	1
↑ 4			

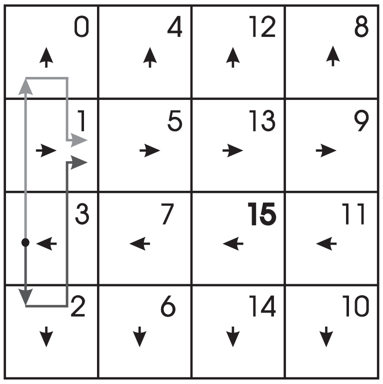
(b)

state \ action	left	forward	right
↑ 0	0.5	-1	1
→ 1	0.5	1	0.5
↓ 2	1	-1	0.5
← 3	1	-1	1
↑ 4			

(c)

Figure 4.7: *Environment bias: (a) The agent is informed about its environment - hence does not take a forward action at cell #0. (b) Actions that have immediate consequences are eliminated from the belief matrix. (c) In addition, actions that have potential consequences are less preferred.*

figure 4.8(a). So the agent need not execute both actions as they have the same end result. Therefore, one of these sequences is eliminated by putting a negative value in the belief matrix, figure 4.8(b). It is worth noting that leaving the insight bias and letting the reinforcement learning algorithm discover on its own these redundant state-action pairs enormously weakens the learner.



(a)

state \ action	left	forward	right
↑ 0	1	1	1
→ 1	1	1	1
↓ 2	1	1	1
← 3	1	1	-1
↑ 4			

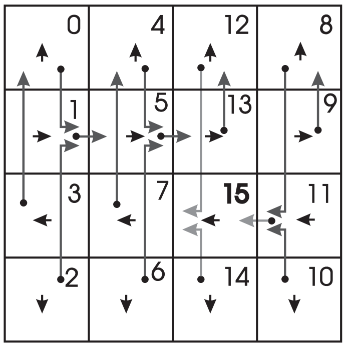
(b)

Figure 4.8: *Insight bias: (a) Two different sequences of actions (dark arrowed path) and (light arrowed path) applied at cell #3, lead to the same end state hence, (b) The second action sequence is excluded by putting a high negative value in the partial belief matrix.*

Goal Bias, $\mathbf{B}_3$

This is a goal directed bias. Since the destination is known, it is possible to bias the learner with vector fields that will ultimately lead the agent to the goal. However, if all vector fields are supplied, there is nothing left for the agent to learn. Therefore, similar to environment bias, we have identified two goal directed biases, namely `near` $\mathbf{B}_{30}$, and `far` $\mathbf{B}_{31}$, biases. In the near bias case, the right vector fields are supplied only to those states that are near to the goal and the remaining states are left for the agent to discover through reinforcement learning. The far bias case is the opposite of near bias in which all far vector fields are supplied and the agent learns only the near vector fields, figure 4.9.

Note here the word `far` and `near` are not used in their literal meaning to represent spatial distance. Far more, they carry a semantic meaning that represents the *reachability* of a state. The reachability of a state is defined as the minimum sequence of actions required to reach the target starting from that state. An agent can be near to the goal spatially, however, it may require a series of actions before it reaches the goal; e.g., if a robot position and its goal are very near but separated by, say a wall. In the task described, cell #7 is spatially nearer to the goal than cell #12, however, an agent that starts from this state requires at least five actions before it reaches the goal; whereas, if it starts from cell #12 it requires only one.



(a)

state \ action	left	forward	right
↑ 0	1	1	1
→ 1	1	1	1
↓ 2	1	1	1
← 3	1	1	1
↑ 4			

(b)

state \ action	left	forward	right
↑ 0	-1	-1	1
→ 1	-1	1	-1
↓ 2	1	-1	-1
← 3	-1	-1	1
↑ 4			

(c)

Figure 4.9: *Goal bias: (a) Near (light arrowed) and far (dark arrowed) vector fields of states. States from which the goal can be reached by one action are considered as near. (b) Near partial belief matrix. (c) Far partial belief matrix.*

4.7 Q-learning

In order to study the effect of bias in the learning time, Watkins' Q-learning is employed. Q-learning is a reinforcement learning algorithm that can be used whenever there is no explicit model of the system and the cost structure (see section 3.6). The algorithm works by maintaining an estimate of the expected reinforcement for each state-action pair (called Q-values), and adjusting these values based on the action taken and the reward received. This is done by using the difference between the immediate reward received plus the discounted value of the next state and the Q-value of the current state-action pair, i.e.,

$$Q(x,u) \leftarrow Q(x,u) + \beta(r + \gamma \max_{u} Q(y,u) - Q(x,u)) \tag{4.10}$$

where y is the next state of the system after applying action u in state x.

The choice of β and γ, the key parameters in Q-learning, affects the efficiency of the learner. The parameter β determines the learning rate, $\beta = 1$ results in an update rule which disregards all history accumulated in the current Q-values. It resets Q to the current sum of the received and expected reward at every time step, which usually causes the algorithm to oscillate. The other parameter γ is the discount factor for future reward. Ideally, γ should be close to 1, but in the general case $0 \leq \gamma \leq 1$.

The initial Q-values can also affect the speed of convergence. Intuitively, initializing the Q-values close to the optimal policy will speed up the learning process. But, the optimum Q-values are either not known a priori or can only be approximated partially. If the Q-values are initialized to zero in a problem whose optimal policy has positive final Q-values, the algorithm will converge to the first positive value, never exploring other possibilities [57]. This problem can be dealt with by occasionally performing a random action to guarantee that the entire action space is eventually explored (see next section).

Choice of Parameters

The values for the two learning parameters and the initial Q-values are shown in table 4.2. Since the world is deterministic, a unity discount factor is chosen so that the relevance of future reward is maximized. For each type of bias, the initial Q-values are initialized by their corresponding belief matrices, $\boldsymbol{b}$.

Update Steps

In order to propagate the Q-values for each state-action combination, they must be updated in one of the two ways. Either a state is updated as it is visited by the agent or the changes are propagated backwards for a chosen number of states at each time step. For example, [76] used a five step update process and [89] updated all of the states at each time step. In this work, we choose the former implementation, where only visited states are updated.

β	γ	initial Q
0.9	1.0	$\boldsymbol{b}$

Table 4.2: *Q-learning parameters initialization.*

4.8 Softmax Action Selection

During the learning process, two opposing objectives have to be combined. On the one hand, the environment must be sufficiently explored in order to find a (sub)optimal controller. On the other hand, the environment must also be exploited to minimize the cost of learning (see section 3.5). The simplest and most popular means of balancing exploration and exploitation is the ϵ-greedy action selection mechanism, where an action with the best expected reward is frequently taken but with the probability ϵ of choosing the action at random. The main drawbacks of this simple strategy is that when it experiments with a non greedy action, it is no more likely to try a promising alternative than a clearly hopeless action.

Instead of this simple strategy a slightly more complex action selection mechanism called the *softmax* rule is chosen. The method varies the action probabilities as a graded function of the evaluation values. The greedy action is still given the highest selection probability, but all others are ranked and weighted according to their expected evaluation values. The Boltzmann distribution [122] is used to compute the action probabilities from the evaluation values. At state x the probability that the controller executes action $u \in U(x)$ is given by:

$$p(u) = \frac{e^{-Q(x,u)/\tau}}{\sum_{v \in U(x)} e^{-Q(x,v)/\tau}} \tag{4.11}$$

where $Q(x, u)$ is the current evaluation and τ is a positive valued function called the temperature that controls the exploration by adjusting how sharply the probability peaks at the greedy policy. While high temperatures cause all the actions to be nearly equiprobable, low temperatures cause a greater difference in probabilities for actions that differ in their evaluation values.

4.9 Experimental Results

Before we examined the influence of the various types of biases on the speed of learning, a particular cell in the labyrinth grid, cell #7, was chosen as a start state. With this setting, the agent's specific task was to seek the shortest path from cell #7 to cell #15. Since the start and target states were fixed in advance, the optimal number of actions required to reach the goal was computed by hand and turned out to be 5. As we mentioned in the preceding chapter, there are many optimal policies that take the agent to the target state. Such two policies, for example, are {`left left forward left left`} and {`left left forward right right`}. But in the experiment we were interested in finding out only one of these policies.

Six Q-learning experiments each using different types of biases were carried out. An experiment is an entire learning process that consists of a set of episodes. An episode in turn consists of a set of 100 trials. Figures 4.10-4.11 show the resulting learning curves of each of the biases. All points on the curves are cumulative averages of ten episodes; each episode has a different seed to generate the random number used to select an action. Further, the vertical axes of all the plots are the average number of actions required by the agent at each trial. For the purpose of comparison, plot of the unbiased performance is included in all the figures.

On these curves the "time" needed to learn the shortest path can be defined in two different ways. One approach is to equate the learning time to the number of trials needed by the agent until it achieves the optimal performance. This approach, however, is misleading, because it does not take into account the actual time elapsed at each trial. For example, referring to figure 4.10 left the agent when it was biased by $\mathbf{B}_{10}$ took almost the same number of trials as when it was unbiased. But, it is clear from the figure that the actual time is longer when it was unbiased than when it was biased; since the number of actions taken at each trial is larger for the unbiased case than for the biased.

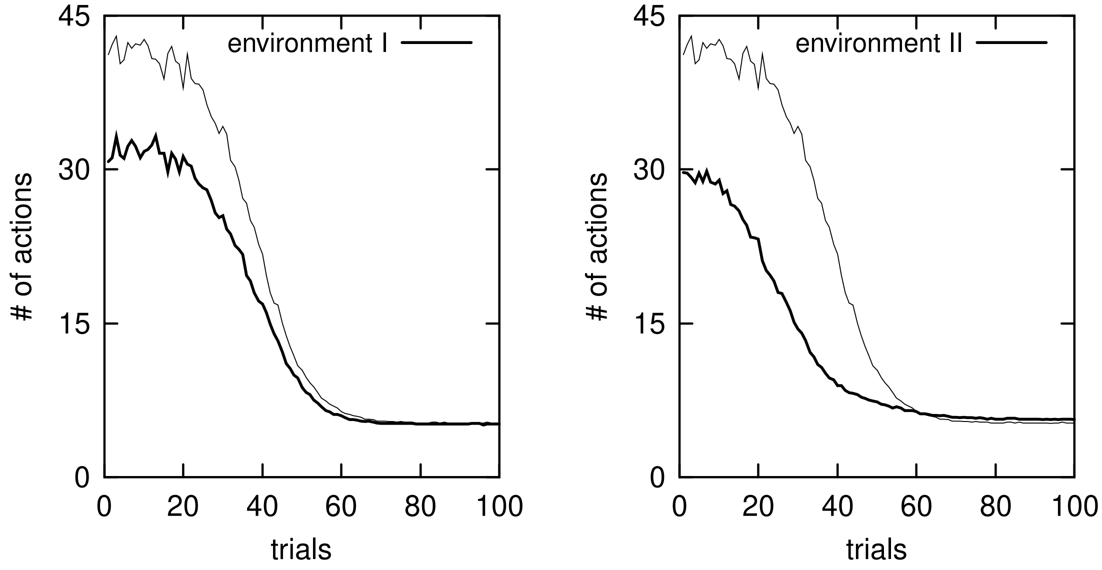


Figure 4.10: *Performance curve with two environment biases.*

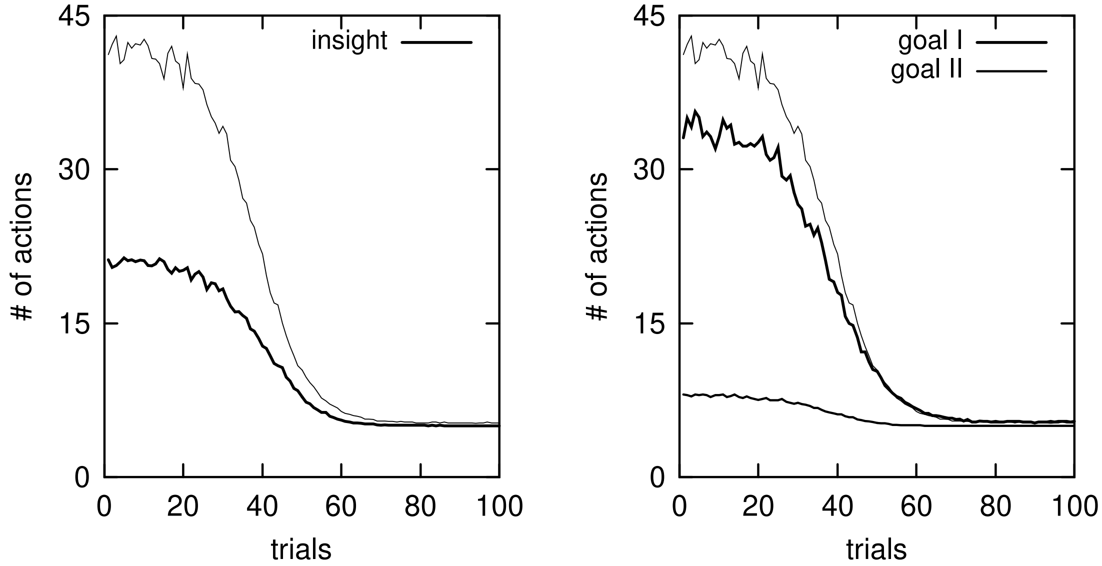


Figure 4.11: *Performance curve with insight and goal biases.*

A more accurate measure that considers both the number of trials and the time elapsed at each trial is,

$$\mathrm{J} = \frac{1}{T}\int_0^T \varphi(t)\, dt \tag{4.12}$$

where $\varphi(.)$ is the number of actions taken as a function of trial, t. That is, the function $\varphi(.)$ is any one of the plots shown in figures 4.10 or 4.11. In other words, instead of defining the time in terms of trials needed, it is now defined in terms of the *average number of actions* the agent needs until it settles to the optimum action. Since trials are discrete in nature, equation (4.12) is reformulated to its discrete equivalent, $\mathrm{J} = \frac{1}{N}\sum_i \varphi_i$, where N is the total number of trials made by the agent and φ_i is the number of actions taken at each trial.

To evaluate the various biases, an index that measures the effectiveness of each bias relative to the unbiased performance is useful. One such index is equation (4.13) which computes the *gain in the reduction* of the average number of action. Based on this index, table 4.3 shows the indices of the various biases.

$$r = 1 - \frac{\mathrm{J}(\mathbf{B}_i)}{\mathrm{J}(\mathbf{B}_0)} \quad i = 1, 2, 3 \tag{4.13}$$

Biases	unbiased $\mathbf{B}_0$	environment $\mathbf{B}_{10}$	environment $\mathbf{B}_{11}$	insight $\mathbf{B}_2$	goal $\mathbf{B}_{30}$	goal $\mathbf{B}_{31}$
Indices	-	22.7	41.7	45.3	23.9	75.9

Table 4.3: *The gain in the reduction of the average number of actions of different biases.*

Analysis

Incorporating the two environment biases $\mathbf{B}_{10}$ and $\mathbf{B}_{11}$ produced a gain, in the reduction of the average number of actions, of 22.7% and 41.7%, respectively. Comparing only the indices, bias $\mathbf{B}_{11}$ (that discriminates actions by looking one step ahead) clearly performed better than bias $\mathbf{B}_{10}$. Unfortunately, its steady state policy was even poorer than the unbiased policy, figure 4.10 right. That is to say, with bias $\mathbf{B}_{11}$ the agent failed to learn the optimal policy. But why?

In general, biased reinforcement takes advantage of the cleverness of the designer to reduce the state space manually. During this process most irrelevant inputs are eliminated, but potentially useful inputs can be overlooked resulting in an incomplete state space and a sub-optimal solution. On the other hand, unbiased reinforcement is complete in the state space and guarantees that the agent will, given sufficient time and reinforcement, produce complete (optimal) policy. This is exactly the bias-variance dilemma discussed in section 4.2. Since $\mathbf{B}_{11}$ has a higher bias than $\mathbf{B}_{10}$, it has a high bias error reflected in its final policy which differs from the optimal one.

Comparing the gain in the reduction of the average action of biases $\mathbf{B}_2$, and $\mathbf{B}_{10}$, the former bias (insight bias) is far better than the latter—this is astonishing. We first define the effective search space of the agent as the state-action pairs left after those irrelevant state-action pairs supplied by the bias are removed. Based on this definition, the agent's effective search space with bias $\mathbf{B}_{10}$ was 38, whereas with bias $\mathbf{B}_2$ it was 40. But despite its large search space, the insight bias enabled the agent to learn the task faster than the environment bias. This is a significant result in the sense that reducing merely the search space is not the only means of cutting the learning time. Biases derived from the problem insight are sometimes efficient and do much of the work.

Unfortunately, neither the environment nor the insight bias are rich enough to brutally cut the average number of actions in real and complex systems. Even for this simple and well defined problem, the best bias, $\mathbf{B}_2$, brought a gain of $1/2$ in the reduction of the average number of actions relative to the unbiased. This suggests that the system still needs an efficient bias to lever the learning process significantly. Figure 4.11 right is the learning curves of the two goal based biases $\mathbf{B}_{30}$ and $\mathbf{B}_{31}$. As seen from the figure, the near goal bias $\mathbf{B}_{30}$, performed even worse than $\mathbf{B}_2$; therefore, it is not worth discussing. With the far goal bias $\mathbf{B}_{31}$, the gain in the reduction of the average number of actions was $3/4$ relative to the unbiased one. Hence, among all the biases introduced, only bias $\mathbf{B}_{31}$ produced a significant leap in the learning process; leaving only a quarter of the knowledge to be discovered through reinforcement.

4.10 Continual Learning

The adaptation property of learning agents is indisputably useful. However, current reinforcement learning algorithms are very slow to converge to a policy and consequently slow to adapt. Perhaps more importantly, most reinforcement learning starts from scratch and adapts only to a single policy. Here, we carried out an experiment to examine if the agent was able to use previously learned knowledge to speed up the learning of an entirely new policy.

The procedure followed was as follows. First, we trained the agent for the previously described task, i.e., for the task where the agent's initial cell was #7 and target cell was #15, without using any of the biases. After the agent learned the task, the final learned Q-values were stored for later use. Next, a new task was constructed; namely, the target state was moved from cell #15 to cell #2. Notice, this new task was an extreme choice, since it corresponds to a shift in the goal location from the right to the left relative to the start cell, figure 4.5(a). Then for this new task, we conducted two reinforcement learning experiments. In the first experiment, all Q-values were initialized to identical values, which corresponds to learning from scratch, while in the second experiment, the Q-values were initialized with the stored values of the previously learned task.

Figure 4.12 shows the learning curves of the two experiments. Note that the optimum number of actions of the new task was two, and at steady state both experiments arrived at this same value. However, learning by initializing the Q-table with the previously learned Q-values (continual learning) was far superior than learning from scratch. Its superiority was two fold: first the number of actions taken at each trial was reduced and second, the agent required only a few additional trials to adapt to the new task.

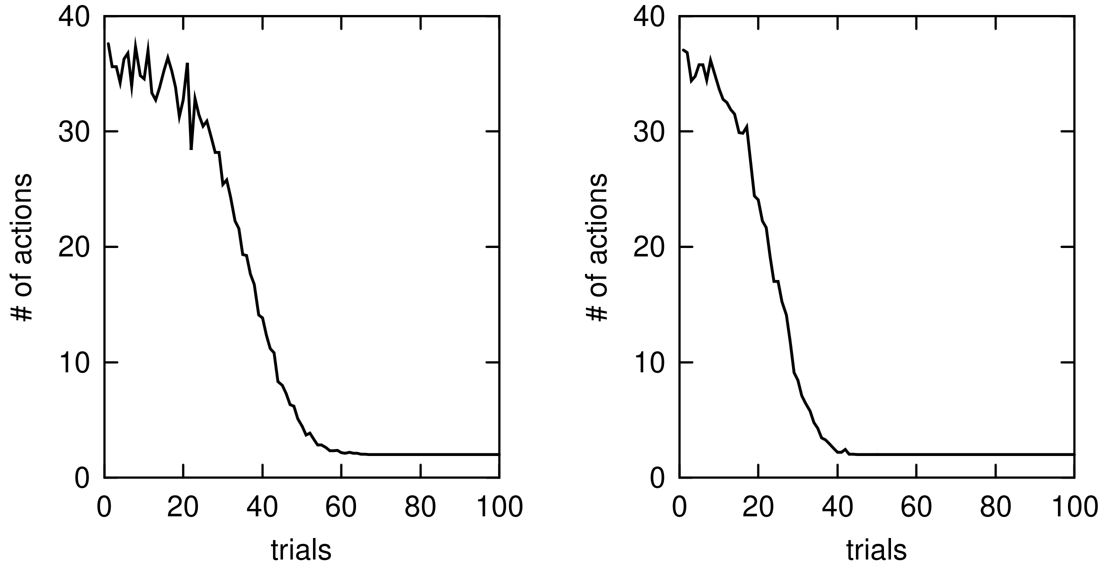


Figure 4.12: *Learning curves in scratch (left) and continual (right) learning modes.*

As mentioned before, the goal of the previously learned task was at the right relative to the start cell, while the goal of the new task was at the extreme opposite. But even for this extreme choice, continual learning was able to accelerate learning by adapting the Q-values faster than learning from scratch. Furthermore, the relative gain in the reduction of the average number of action of the continual learning was 31.9%. We checked also if the speed up in continual learning was independent on the choice of the new goal cell by carrying out the experiment for different goal cells in the labyrinth grid. Surprisingly, in no instance did earlier training interfered and caused continual learning to perform inferior to learning from scratch.

4.11 Summary

The tradeoff between the type and amount of built-in versus learned information is the key issue in machine learning. The less structure is built in, the more is left to the algorithm to discover. Minimizing built in structure eases the programming task; but often makes the learning process slower, the space and time complexity larger, and the result more task specific. There are many things an agent can be told that will make its learning task easier. The key is finding out what those things are and then devising ways of incorporating them into the agent.

In this chapter, we have first identified several types of biases that are able to influence the agent's learning. We then compared their effectiveness on the basis of the time the agent needed to learn a given task. The task has been a typical robot navigation problem in a labyrinth world. The main motive in using this hypothetical world has been its simplicity in generating and quantifying different kinds of biases. Six different forms of bias that tell the learner some kind of information about the environment or the task have been examined. Each bias is incorporated into the learning system through the use of a belief matrix that initializes the Q-table. The entries of the belief matrices are our initial inference of the importance of a particular action in a particular state.

Six reinforcement learning experiments, each corresponding to the different types of biases have been conducted. For each of the experiments, a performance curve was plotted and from the plot an index that measured the effectiveness of each bias as a ratio of the `unbiased` performance has been extracted. It has been found out that the `far` bias (where the agent was biased for all state spaces except for those spaces "near" the target) performed the best among all the other forms of biases. Furthermore, despite the large search space of the `insight` bias as compared to the `environment` bias, the former converged to the optimal policy after a relatively less average number of actions than the latter. This reaffirms the claim that biases drawn from the distinct characteristics of the problem are more powerful than biases derived from the global information.

Earlier we said that the more built-in knowledge, the less time the agent needs to learn the task. This was demonstrated by biases $\mathbf{B}_{10}$ and $\mathbf{B}_{11}$. In the latter bias form more built-in knowledge was embedded than in the former, and as a result the average number of actions relative to the `unbiased` was reduced almost by half. This reduction, however, was brought at the cost of the final steady state performance. Paradoxically, it is the fully exponential nature of the unbiased reinforcement learning that gives it one of the main positive properties: asymptotic completeness. A highly biased agent is not complete, since some relevant state spaces can possibly be removed to result in a suboptimal policy. That is why the steady state performance of the agent biased by $\mathbf{B}_{11}$ was poorer than that of the unbiased one.

The chapter has also presented results showing the advantage of continual learning over learning from scratch. Continual learning, as the name implies, is a mechanism, where an agent compiles and stores past learned experiences in Q-values to help it learn in a new environment. In almost all cases, even in cases where the new environment is quite different from what the agent had learned before, continual learning performed better, requiring only few extra training sessions, than learning from scratch.

Before we raised the issue of the amount and quality of bias, we began the chapter with the bias-variance dilemma. The estimation error can be decomposed into two parts—bias and variance. The bias variance dilemma arises when we try to minimize both terms by adjusting the network weight and size. A small network, with say one hidden unit, is likely to be biased, since the repertoire of functions spanned by the available weights are quite limited. If the regression is poorly approximated within the network, there will necessarily be a substantial bias. On the other hand, if we over parameterize, via a large number of hidden units and associated weights, the bias will be reduced, but the approximation will be highly sensitive to the data, which causes a significant variance contribution to the mean square error [14].

Then, we went on discussing the four common biases that have been useful in speeding up reinforcement learning. Modularization involves decomposing the global task into several distinct sub-tasks and provides a separate reinforcement signal for each sub-task. Instead of a monolithic learning architecture, the learning architecture is distributed among behaviors that learn individual sub-tasks in parallel. Since the individual sub-tasks' state representations are both manageable and learnable, the learning algorithms on the sub-tasks converge faster than on the global task. But modularization of the learning architecture is a mixed blessing; often a new task requires a different decomposition with little sharing of the constituent behaviors [75].

Learning agents that accept some form of assistance from outside significantly outperform agents that only learn from reinforcement. Advice and reflex are two forms of assistance mechanisms discussed in the chapter. While advice is normally derived from an external observer, often a human being, reflex is derived from an expert system. In both cases, the agent does not accept the advice absolutely nor permanently. Rather based on subsequent experience, the agent can refine and even discard the external assistance.

In monolithic reward function, the robot gets a reward very infrequently and learns very slowly. Domain-rich reward tries to impart prior knowledge through the reward signal to increase the *frequency* of rewards. Heterogenous reinforcement maintains multiple goals and generates an intermediate reward that reinforces the agent whenever subgoals are achieved. Progress estimator is a partial advice providing an indicator on the performance that tells the agent when it is doing well even if it has not achieved the goal yet. Both methods, however, upset the purpose of the reward function. Generally, the reward signal is not the right place to impart prior knowledge. As pointed by [124] the reward signal is our way of communicating to the agent what we want to achieve, not how we want it to achieve. Better places for imparting biases are the initial policy or value function [48, 69, 71].

Chapter 5

Learning a Minimum Cost Path

... Nonetheless, as information theorists, neuroscientists, and computer experts pool their talents, they are finding ways to get some life like intelligence from physical robots.

Suplee

5.1 Motivations

Programming an autonomous robot so that it reliably acts in a dynamic environment is difficult. This is mainly due to the unavailability of information at the design time, the unpredictability of the environment dynamics, and the inherent noise of the robot's sensors and actuators. For example, a robot may have to move in a cluttered environment with an unknown topology. Moreover, people may be moving around, and the robot may be equipped with noisy sensors, like sonars and dead reckoning, to sense its surroundings.

An autonomous robot, which can acquire knowledge by interacting with the environment and subsequently adapt its behavior in the course of its life, could greatly simplify the work of the designer. A learning robot needs not be given all the details of the environment in which it is going to act; it will acquire them by direct interaction. Also, its sensors and actuators need not be finely tuned, they will adapt to the specific requirements and environmental conditions [18].

So far, however, reinforcement learning has mainly been studied in well defined Markovian domains, such as games [127, 128] and point robot navigations [123]. With the exceptions of [76, 88, 98], reinforcement learning has not yet been demonstrated on a realistic agent domain, particularly when applied to physical robots. Werbos [140] has reported John Mayhew from Sheffield University controlling an autonomous vehicle using AHC architecture. But as far as our literature survey goes, we never came across such a work; if such work exists it has not yet been published. There are of course some works, such as [35], that look-like reinforcement learning. However, despite their success, we are not interested in these works, because their reinforcement learning methods are not structured around the value function, i.e., they do not use DP or TD to update the value of taking actions in states nor extract actions from the state values.

Our main goal in this chapter is to build a self competence acquisition system on the real B21 robot. What is demanded and expected from the robot is to determine autonomously the effective wiring between its sensors and actuators in order to achieve a specified task. The task is searching for the minimum cost path problem. Reinforcement learning is used to teach the robot on-the-fly to accomplish this task. Following [49, 88] competence is first acquired from various bias components by associating sensors (input stimuli) with actuators (response). The specific associator used here is the radial basis function network (section 1.5). Except the robot type (point vs. real) and the state representation (discrete vs. continuous), this work is strongly linked with the work of the preceding chapter in its use of bias and problem formulation. Particularly, it borrows the two biases, modularization and reflex discussed in section 4.3, and deals with similar navigation task. We begin the chapter by defining the minimum cost path problem.

5.2 The Minimum Cost Path Problem

Reinforcement learning technique has been used to solve the shortest path problem in a maze like structures [7, 80, 123]. Furthermore, algorithms have been developed to extend it to a non-maze like structures [90, 95]. However, these algorithms find a solution path without ever attempting to optimize the path.

This chapter addresses a search path problem in a non-maze like structure, but with additional constraint of optimizing the path—we are searching for a path that has a minimum cost. Formally, the problem consists of a robot, a world with randomly distributed obstacles, and two Cartesian coordinate locations, within the world, specifying the start and the goal positions. The specification of the goal is essential, since it is the key feature of reinforcement learning, which explicitly considers the whole problem as a goal directed interactive learning (section 3.2).

The task faced is to build a self-adaptive controller that improves its performance by learning from experience. The performance is the trajectory that, when followed by the robot, would lead to a minimum cost. Generally, the cost of a path $\mathcal{P}$ is defined as,

$$J(\mathcal{P}) = C_D + C_S \tag{5.1}$$

where the first term is the cost associated with the duration of the path, and the second term is the cost associated with the safety of the path. Thus, among all the feasible paths $\mathcal{Q}$ that lead to the target, the minimum cost path problem searches for a path $\mathcal{P}$ that is both safe and short, i.e., $\mathcal{P} = \arg\min_{\mathcal{L}\in\mathcal{Q}} J(\mathcal{L})$.

5.3 Experimental Set-up

A real robot laboratory provides a test of the efficacy and usefulness of learning algorithms. It is an important source of inspiration for deciding which components of the reinforcement learning framework are of practical importance and for cross validation of simulation results. In this section we present the environment and the physical robot with which we have conducted the reinforcement learning experiment.

The Physical Robot

The B21 robot from the Real World Interface (RWI) (figure 5.1) is used as our experimental platform. The robot is a cylindrical four-wheeled synchronous drive with two parts: a base and an enclosure. The base carries 32 infra-red (IR) and 32 tactile sensors, whereas the enclosure has a belt of 24 tactile, 24 IR, and 24 sonar sensors each placed evenly around the robot's perimeter. On the top of the enclosure a two finger manipulator with 6 DOF and a binocular CCD camera are mounted. In addition, the robot is equipped with an encoder that provides the Cartesian coordinates of the robot with an accuracy of 0.254 cm, though the actual accuracy is dependent on the slippage between the robot's wheels and the floor.

The sensor repertoire is capable to detect obstacles and to explore the environment, which are the basic abilities required to undertake a minimum cost path learning experiment. The 24 sonar sensors define the robot's view of the environment and form a part of the input space of the learning system. The infrared and tactile sensors are used in a low-level asynchronous emergency routine to detect real or virtual collisions. Whereas real collision is detected by the tactile sensors, virtual collision is detected by the infrared sensors, see section 5.5 for detail.

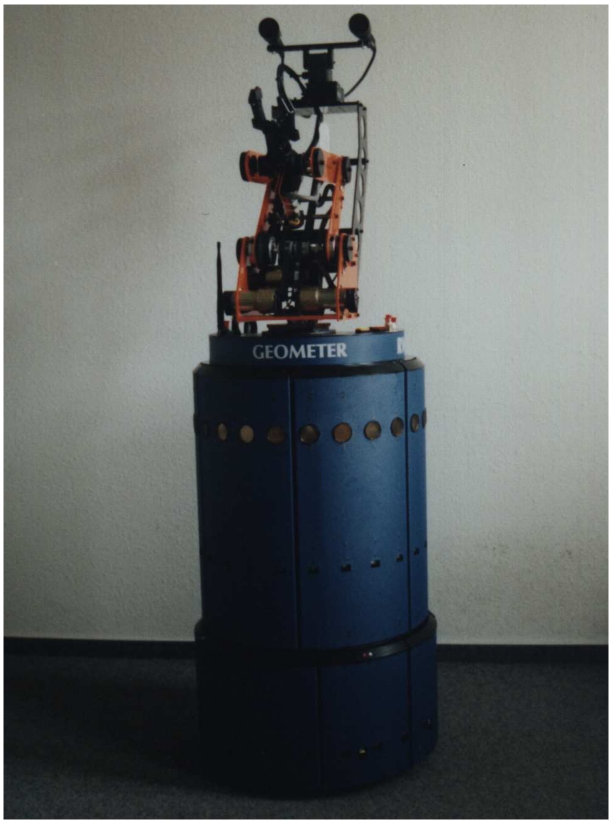


Figure 5.1: *The B21 experimental mobile robot equipped with 24 sonar, 56 infra-red, 32 tactile sensors, and an encoder. The 6 DOF manipulator and the binocular CCD camera are reserved for visual processing.*

The Robot Environment

Figure 5.2 shows the top view of the robot environment. The home position of the robot is the black dot inside the room and the big circle in the corridor is the target location. The robot can neither move nor see through obstacles, i.e., obstacles are opaque. As mentioned in section 5.2, the controller task is to take the robot from the home location $\boldsymbol{p}_r(t=0)$, to the goal location $\boldsymbol{p}_g$. The positions of the robot and the goal are specified in a Cartesian coordinate system,

$$\boldsymbol{p}_r(t) = \begin{pmatrix} x_r(t) \\ y_r(t) \end{pmatrix} \quad \boldsymbol{p}_g = \begin{pmatrix} x_g \\ y_g \end{pmatrix} \tag{5.2}$$

At first glance, both the task and the environment seem relatively simple compared to what we would like our robots be able to do. However, when one tries to implement it on present day robots, it becomes clear that this seemingly simple task is no longer easy. First and foremost, this task is performed using the *local* sensory information—the robot does not have either access to the global view of the environment or a comprehensive world model. This is controversial; there is no universal consensus if biological systems also learn from local sensory information. But, this will not hinder us; since in the same way it is possible to build machines that fly but do not flutter their wings [67], machine learning is aimed at designing machines that show intelligent behavior but *lack* the full perceived process found in biological systems. Second, it is a high-dimensional continuous learning task and successful goal reaching requires a nonlinear mapping from this space to the space of continuous real valued actions. In general, it is not easy to train networks on large spaces.

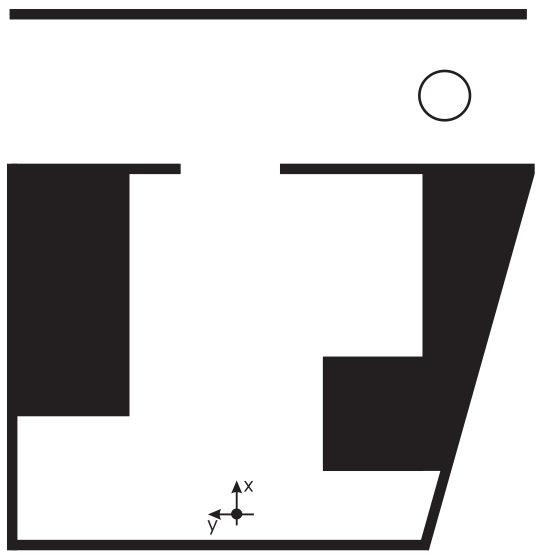


Figure 5.2: *The real-world environment is our robotics laboratory consisting of an indoor area of* 25 m^2 *and a corridor of width* 1.8 m. *The coordinate of the center of the target relative to the origin, black dot, is* $\boldsymbol{p}_g = \left(\begin{smallmatrix} 4.00 \ m \\ -2.00 \ m \end{smallmatrix} \right)$.

5.4 Inputs and Outputs

The learning system† builds its own input space from the external, as well as the internal sensors of the robot. Among the three types of external sensors, only the sonar sensors form a part of the input space, i.e., the controller learns an action map from these sensory spaces. Of course, it uses the other types of sensors (infrared and tactile) too, but solely in emergency conditions that require a fixed and prior mapping.

Before the sonar sensors are fed to the controller their values are normalized so that each lies in the interval $[0, 1]$, i.e.,

$$\boldsymbol{s} \rightarrow \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{23} \end{pmatrix} \tag{5.3}$$

where $s_i \in [0, 1]$. It is worth noting, that apart from this simple normalization process, no attempt is made, as we have done in section 2.3, either to process or collapse the continuous high dimensional sensory data. The main reason to work with this high dimensional input space is to minimize the perceptual aliasing problem, which is caused by the many-to-one mapping between the external world states and the internal perceived states [142]. But working in this large space becomes difficult, because the controller now has to learn not only the action map but also the state abstraction, i.e., the mapping from sensory space to feature space, section 5.7.

In addition to the sonars, the controller also uses the relative distance of the robot from the target, $\| \boldsymbol{p}_r(t) - \boldsymbol{p}_g \|$. Albeit, in order to make the dimension of this input comparable to that of the sonars, Millan's codification scheme [88] is applied on this scalar input. The scheme consists of eight processing units, figure 5.3, whose activation values depend on how far the normalized relative distance is away from the respective center positions of the units, i.e., $\phi_i = \exp(-(\mu_i - \rho)^2)$, where $\mu_i \in \boldsymbol{\mu} = (0.0625\ 0.1875\ 0.3125\ 0.4375\ 0.5625\ 0.6875\ 0.8125\ 0.9375)^T$, and ρ is the normalized distance between the robot and the goal. The elements of $\boldsymbol{\mu}$ are the pre-assigned center positions of the units that are placed evenly along the abscissa and spanning the interval [0,1].

†Hereafter, instead of learning system the term *controller* or *learner* are interchangeably used.

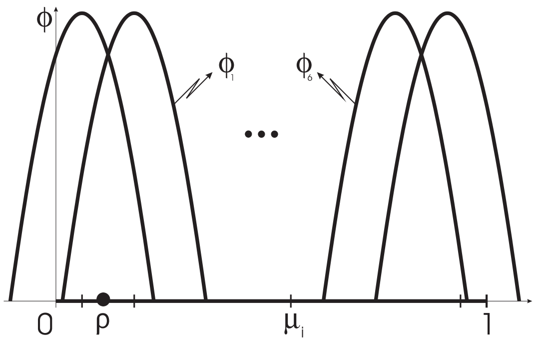


Figure 5.3: *Gaussian processing units that have identical variance but different means. The unit's total receptive field covers the interval* $[0, 1]$. *The scalar relative distance,* $\| \boldsymbol{p}_r(t) - \boldsymbol{p}_g \|$ *is first normalized so that, it falls within the unit's interval.*

It is these activation values forming a vector,

$$\boldsymbol{\rho} \rightarrow \begin{pmatrix} \phi_0 \\ \phi_1 \\ \vdots \\ \phi_7 \end{pmatrix} \tag{5.4}$$

that are used as part of the input space of the controller. Thus, the overall input vector to the controller is a vector of 32 real valued elements,

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{s} \\ \boldsymbol{\rho} \end{pmatrix} \tag{5.5}$$

The robot has multiple motor parameters that one can choose to efficiently control the robot's motion. However, learning to control multi-parameter in reinforcement problem setting is extremely difficult than that of a single parameter. First and foremost, the reward does not tell us which of these parameters are at fault. Furthermore, if common internal representations (such as common neurons or weight vectors) are used, adapting the internal representations for all the parameters can not be achieved without exceeding the time constraint. Therefore, we find it is more effective if the repertoire of motor parameters, which the controller can alter, are restricted. Consequently, the angular rotation α *rad*, which determine the robot's next direction of motion, is the only motor parameter chosen to control the robot. Nevertheless, for every rotation, two motor actions are initiated; the robot first completely rotates by the specified angle, after rotation has ceased it will move to a new location by translating forward a fixed distance, $l = 20$ *cm*.

5.5 Reinforcement Functions

In reinforcement learning, the goal of the agent is formalized in terms of a special reward signal passing from the environment to the agent. The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning. Although this way of formulating a goal might at first appear limiting, it has proved to be flexible and widely applicable [124]. For example, to make a robot learn to walk, a reward on each time step proportional to the robot's forward motion is provided. In making the robot learn how to escape from a maze, the reward is often zero until it escapes, at which time it becomes $+1$. From these simple examples we can see that if we want the robot do some thing for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goal. It is thus critical that the rewards we set up truly indicate what we want to accomplish.

The agent-environment interface boundary of figure 4.2 shows that the reward is computed externally or outside of the agent. In natural systems, however, rewards are computed inside the physical bodies. Animals recognize their ultimate goals by computations occurring inside their bodies, for example by sensors that recognize food, hunger, pleasure, and pain. In artificial learning systems the agent boundary is placed not at the limit of its physical boundary, but at the limit of its control. In other words, anything that can not be changed arbitrarily by the agent is considered to be outside of agent, even if it physically resides inside the agent. Thus, we always consider reward computation to be external to the agent because it defines the task facing the agent and must be beyond its ability to change it arbitrarily [124].

Our environment generates an immediate reinforcement or penalty signal after the robot has executed an action. This does not mean that the controller optimizes only the immediate reinforcement. Rather on the contrary, the controller is formulated to optimize state values that indicate the long term desirability of states after taking into account the states that are likely to follow and the reward available in those states. The reinforcement function consists of two components.

The first component penalizes the robot whenever it collides with or comes near an obstacle. To detect these two events, it uses two flags: `collision` and `close`. The collision flag is set either when the body of the robot is in contact with obstacles (real collision) or when the reflectance of any of the IR sensors[‡] is

[‡]The IR sensors detect objects that are shorter than $0.5\ m$ by emitting light and measuring the

greater than 70% (virtual collision). As pointed out in [3, 88], the inclusion of a virtual collision makes the learning process safe, since it always occurs prior to the occurrence of a real collision. When a collision occurs the controller receives a fixed penalty -3, the robot immediately stops, and it moves backwards until it clears itself from the collision.

The close flag is set when any of the sonars measures a distance less than a given specified value, d_c. In this case, instead of a fixed penalty, the trainer generates a variable penalty that increases as the distance between the robot and the obstacle decreases, i.e., $-1 + \min_j(d_j/d_c)$, where d_j is the reading of sonar j. Altogether, the component of the trainer that teaches the robot to avoid obstacles lies between [-3,0] and has the form,

$$f_1 = \begin{cases} -3 & : \text{ if collision} \\ -1 + \min_j(d_j/d_c) & : \text{ if } \min_j d_j < d_c \\ 0 & : \text{ otherwise} \end{cases} \tag{5.6}$$

The other component teaches the robot how to approach the goal point by first computing the acute angle between the robot heading θ_h, and the line connecting the goal and the robot location θ_{gr}. The acute angle is used as a measure of the divergence of the robot from the goal—if this angle is increasing then the robot is moving away from the goal. Hence, the second component of the reward function is proportional to the acute angle and has a value lying between [-1 0], i.e.,

$$f_2 = -\frac{\text{acute}(\ |\theta_h - \theta_{gr}|\ )}{\pi} \tag{5.7}$$

The total immediate reinforcement r, is the sum of the two components, $r = f_1 + f_2$. Note that, the reinforcement function does not teach the robot directly how to reach the goal. It only trains the robot how to approach the goal without collision. Approaching and reaching are quite different things. The robot can approach a goal, but never reach it (e.g., if the goal is enclosed). Therefore, the above reinforcement function presupposes that the environment satisfies the constraint that it has at least one free way or path through which the robot can reach the goal without collision.

intensity of the reflection bounced from the objects. The reflectance values range from zero (no reflection) to 100 (full reflection) and they are highly dependent on the color of the objects.

Inconsistent Reinforcements

Properties of a physical hardware of a robot, which are often constrained by various sensory, mechanical, and computational limitations impose restrictions not only on the control strategies that can be applied but also on the type of tasks and experiments that can be used. One of these hardware limitations is the robot's dead reckoning system. Most mobile robot controllers rely on reasonably accurate dead reckoning for localization or spatial learning. Over time, however, slippage between the robot's wheels and the floor results in errors, both in the position and orientation of the robot. The robot's rotational error tends to be more serious than the translational error, since small errors in rotation lead to large errors in translation at a location far from the origin of the coordinate frame.

In this thesis, the robot's position is used to decode the relative distance between the robot and the goal, section 5.4, as well as to provide a part of the reinforcement function, equation (5.7). From the two, the latter one is more sensitive to the inaccuracy of the robot position, because it leads to an *inconsistent* reinforcement function that makes learning difficult or even impossible. Noting this Millán [89] has eliminated the dependence of the reinforcement value on the odometry reading by building "goal sensors" that are capable of detecting the goal explicitly. In our work, the odometry reading is still used to implicitly compute the goal angle. But, in order to guarantee that the measured position is close to the true robot position, we have exploited the only crucial property of the robot's dead reckoning system. Dead reckoning performs satisfactorily provided that the robot does not move for an extended periods of time without reaching the goal. This characteristic has directly restricted how far and how hidden the goal should be placed away from the initial robot position.

5.6 Built-in Knowledge

Machine learning in general and reinforcement learning in particular is aiming at shifting the burden of programming a robot from human to the robot itself, so that the robot acquires and adapts the knowledge about the task automatically, as well as prepares itself to deal with unforeseen changes in the environment [18]. This property of reinforcement learning coupled with its biological relevance makes it a methodology of choice for learning in a variety of different domains. Unfortunately, reinforcement learning is an extremely slow learning process—the time it requires to converge toward the desired performance is extremely too long for all but the simplest problem. There are many reasons [6, 18, 82] that contribute to the slow convergence of reinforcement learning. The major ones are:

- Whenever a new situation is presented, reinforcement learning is unable to decide quickly and rationally where in the input space the situation belongs [45, 57, 97], and
- Reinforcement learning does not know where to search for suitable reactions in the action space for the new situation encountered [88].

These problems are originated from the naive definition that states reinforcement based learning robots learn by directly interacting with their environment; thus they do not require a model of their environment nor an external teacher. We showed in the previous chapter, however, that learning even in the labyrinth world requires a sufficient goal directed built-in knowledge. Considering the complexity and uncertainty of the physical system, the need for built-in knowledge is even stringent on physical robots. This section, therefore, is devoted to the two forms of built-in knowledge used in the minimum cost path problem. But, before introducing the built-in knowledge, the need for mixing bottom-up and top-down design technique for feature extraction will be discussed.

Feature Extraction–an Important Subproblem

To a greater degree the reason for the weakness of reinforcement learning is hinged on the reinforcement feedback, which gives little guidance for feature extraction, section 1.6. One reason is that if the system fails by choosing the wrong action, the feedback does not specify which of the output node was wrong. In a system which chooses its action by selecting the most active output node, an error can be caused either by having a node to be too active for a given input or by other nodes not being active enough. If the system has a hidden layer of feature detectors, another reason for poor feature extraction is that acting properly depends on both identifying the current context, as well as selecting an action appropriate to the context. A scalar feedback signal does not indicate which of these processes is at fault. The feedback does not distinguish between the case where the system rightly identified its context but select the wrong response, and the case where the system learned responses are correct, but its feature detectors misidentified the context. In terms of a typical neural network implementation, the system needs to know whether it should tune its feature detector, or the weights placed on the outputs of those feature detector, or both. Thus, as advocated by Sommer [117], learning method for reinforcement problems need bottom-up information or some type of domain knowledge to supplement the top-down reward feedback.

Environment Model

Reinforcement learning solves the temporal credit assignment problem, but *not* the structural credit assignment problem of inferring rewards in novel states. In the absence of environment knowledge, the problem with the structural credit assignment is two fold. First, during the course of learning a state space has to be constructed that is appropriate to the environment. Second, rewards have to propagate spatially across states so that similar states cause the agent to take similar actions.

On-line adaptive state construction has been addressed with statistical clustering method using recursive partitioning of the state space [24, 76, 95]. It is also addressed in [21, 63, 78] that uses radial basis function networks as state descriptors. These state construction algorithms have booked successful results when applied to problems that have low dimension of manifolds. Nevertheless, applying these methods directly to learning robots that have a large and continuous search space is a daunting task.

On a few robots, like the Nomad 200, where the robot has a separate motor called a *turret*, which is used to orient the sensors independently from the robot direction, sensor readings are made independent of the robot heading; i.e., $\boldsymbol{s}(\boldsymbol{p}_r(t), \theta_r) \rightarrow \boldsymbol{s}(\boldsymbol{p}_r(t))$. On this robot, Millán [88] has successfully *contained* the state space and constructed appropriate states adaptively from the raw sensory data. Unfortunately, these types of robots, as pointed even in [88], are the exception rather than the rule. Most robots do not align their sensors independent of the base. Consequently, the perceived sensory data would be different every time the robot visits a given location at different headings. In this case, the adaptive state construction algorithm would fall pray to the curse of dimensionality.

Without some prior knowledge of the environment it is, therefore, unlikely that any algorithm would split key regions quickly and learn the appropriate granularity of the space. Choosing features appropriate to the task is an important way of adding prior domain knowledge to a learning system. The features should correspond to the natural features of the task along which generalization is most appropriate.

Therefore, prior to the learning process, we extract some key features which aid in constructing a coarse model of the environment by decomposing the environment into four disjoint regions, the union of which covers all the state space. In effect, instead of viewing the entire robot environment as a uniform space, it is viewed as a repertoire of distinct regions that are considered to be the same for the purpose of learning and generating actions. The symbolic labels: `concave`, `door`, `corridor`, and `room` are used to identify each of the regions, figure 5.4 left. Based on the x and y intercepts of the three lines that delineate the regions and the current robot position $\boldsymbol{p}_r(t)$, a set of heuristic rules are written, which serve as an interface between the low level sensory signals and the high level cognitive knowledge. The heuristic (algorithm 8) basically singles out a unique symbol that indicates the region where the robot is currently found. Is the provision this domain-specific knowledge a large sacrifice to the robot autonomy? We argue no, because this is only a coarse partition that is not adequate, unless the agent is lucky, to learn the task. The interesting aspects of the task faced by the robot are still to be learned by the inductive learning, which splits further the initial partitioning.

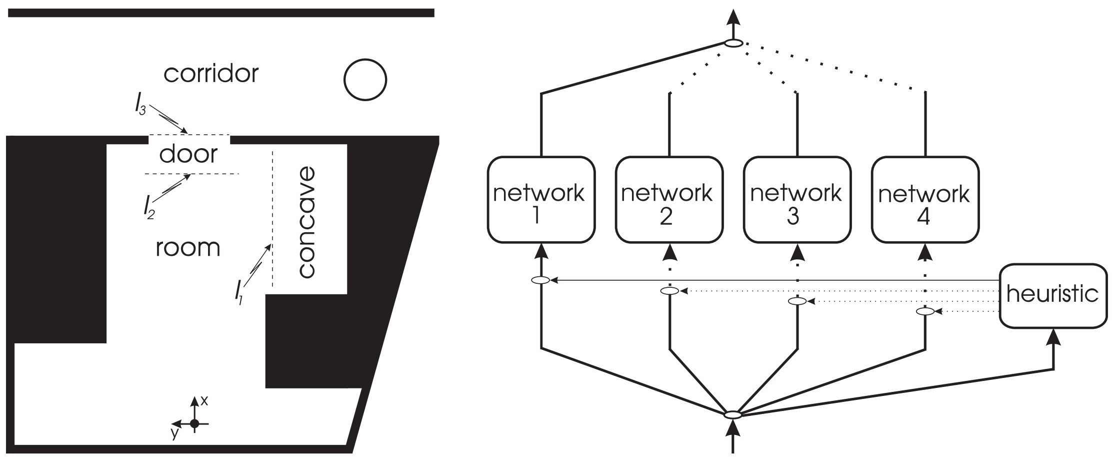


Figure 5.4: *Left: For the purpose of extracting high level cognitive knowledge, the world is cut at its joints and the lines ($l_1 : y = -1.0\ m, l_2 : x = 3.50\ m, l_3 : x = 3.75\ m$) at the boundaries delineate the regions. Right: By analyzing the input vector, the heuristic channels the input to a specific network that learns the action map of the region.*

In the heuristic below, l_{1y} is the y intercept of the line l_1, l_{2x}, and l_{3x} are the x intercepts of lines l_2 and l_3 respectively.

corridor=$l_{3x} < p_x(t)$
concave=$p_x(t) < l_{3x}$&&$p_y(t) < l_{1y}$
room=$p_x(t) < l_{2x}$&&!concave
door=!corridor&&!concave&&!room

Algorithm 8: *Heuristic signal-to-symbol mapper.*

The global behavior of this bias is quickly understood from figure 5.4 right. Once the bias singles out a particular symbol, it channels the sensory inputs to a particular network. In so doing, it early carves up the state space into mutually exclusive and exhaustive regions, and each network learns the action map of the subset of the input variables relevant to its specfic region. This bias renders the overall controller a *flat architecture* [26] which is built by a number of networks, all directly connected to the robot's sensory interface. The architecture looks like the gated network, figure 4.1, with the heuristic bias replacing the gate. However, there is an essential difference between them. Whereas the gated network is applied at the output and usually computes a weighted sum of the individual networks, the heuristic bias is applied at the input and completely cuts off all the networks, except the one that is chosen.

The bias has two main advantages. First, it minimizes perceptual aliasing that is caused by agent inability to discriminate among all the world states. In the flat architecture, each network stores its own state history information; therefore, the same state can infer two different actions from different networks without causing any ambiguity. Second, instead of tackling the problem by one monolithic network, four networks, each dedicated to a particular region of the entire environment, share the problem. Since the manifold of each region is much less than the total manifold, adaptive state space construction on the individual network can work without falling pray to the curse of dimensionality.

Finally, although we are aware that such a bias is generally ad hoc, causes sub-optimality in performance, and trades with autonomy, all these drawbacks are outweighted by the benefit it brings to the learning process.

Fuzzy Behaviors as Reflex

One thing that keeps agents that know nothing from learning anything is that they have a hard time even finding the interesting parts of the space. Most of the time, they either collide with obstacles and die or wander randomly without ever approaching the goal. Observing this, Millán [87, 88] has introduced and applied a new prior knowledge called reflex, which not only ameliorate the above problem but also brutally cut the learning trials. Subsequently, Hailu et al. have applied the reflex bias both on a simulated [47] and physical [45, 46] robots. The reflex actions, though eventually overridden by more detailed and accurate learned actions, keep the agent safe and direct it in the right direction while it is trying to learn. Added advantage of the reflex is its silency, it intervenes only when the learning system needs help [49].

Our reflex consists of two fuzzy behaviors, figure 5.5. The first one is a reactive `obstacle avoidance` behavior and the other one is a purposive `goal following` behavior. The behaviors are implemented by a set of fuzzy rules that have fuzzy sets in the antecedent and conclusion parts, section 2.6. Since the outputs of the behaviors are combined one to one, the number of output fuzzy sets of each behaviors and the form of their membership functions are identical. The output fuzzy sets, which decode the next robot direction, are {`left`, `forward`, `right`} and span the interval $[-\pi,\pi]$ with overlapping triangular membership functions. The obstacle avoidance behavior receives the sonar data as input variables and outputs a three dimensional vector $\boldsymbol{\alpha}_a$, whose elements indicate the activation levels of the above output fuzzy sets. Likewise, the goal following behavior inputs the acute angle between the robot heading and the vector connecting the current robot and goal locations and outputs a similar three dimensional vector $\boldsymbol{\alpha}_g$. Note that, since the latter behavior seeks a particular goal that can not be sensed by the robot's perceptual sensors, it utilizes the robot's internal representation to indirectly sense the goal.

The outputs of the behaviors are fused by Payton et al. [100, 147] architecture of combining multiple behaviors. Since we desire the reflex to point always in the direction of the goal, this scheme of combining outputs guarantees that the final command is goal directed. The other approach, the command arbitrator scheme, where a single behavior is chosen based on behavior priorities, *may* not be goal directed, since information regarding, say goal following, would not be available once the command arbitrator selects the collision avoidance behavior.

Blending

The behaviors discussed above are active during the operation of the robot. Consequently, at any given moment the reflex behavior is the resultant of the two behaviors. The behaviors are combined by assigning a desirability function to each behavior [112]. Unfortunately, this function is complex and varies with the context—each behavior has its own context of applicability, and the desirability function has to be considered only when that context is appropriate. For example, a goal following behavior can sensibly be applied only in situations where the space in front of the robot is free. When an obstacle is detected, this behavior is outside its area of competence and it should at least be partially disregarded.

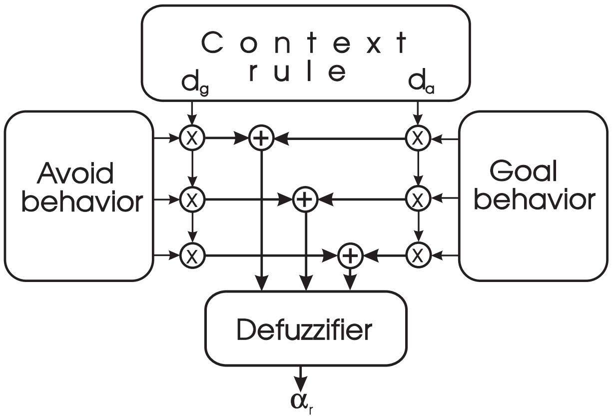


Figure 5.5: *The basic reflex, implemented as two fuzzy behaviors.*

However, since the reflex component is used to provide an initial search location in the action space, it is enough to consider a less rigorous behavior blending scheme. In particular, a scheme is sought where the behaviors have constant desirability functions irrespective of the robot situation in the environment. Following this, two constant desirability values, $\boldsymbol{d} = \begin{pmatrix} d_a & d_g \end{pmatrix}$, one for each behavior, are chosen. Fusion starts by combining the activation strengths of the corresponding nodes of the behaviors using a weighted sum of their respective desirability values,

$$\boldsymbol{\alpha}_f = \boldsymbol{d} \begin{pmatrix} \boldsymbol{\alpha}_a \\ \boldsymbol{\alpha}_g \end{pmatrix} = d_a \boldsymbol{\alpha}_a + d_g \boldsymbol{\alpha}_g \tag{5.8}$$

where $\boldsymbol{\alpha}_f$ is the fused vector. Following the fusion process, the fused vector is defuzzified [33, 50] to a crisp value α_r.

Evaluating the Reflex

The result of chapter 4 concludes that reinforcement learning requires a sufficient goal directed bias. Even though that conclusion is made for the task described in section 4.4, we insist that it still carries over here, because with the exception of the robot and the world representation the task remains the same—both tasks are basically concerned with a navigation problem.

Therefore, before the described reflex is transfered, it is necessary to select the two desirability values and test the appropriateness (both safety and goal directedness) of the resulting reflex behavior, so that time could be spared from experimenting with a useless or weak bias. As the global purpose of the reflex is to make the robot learning safe, a higher desirability function is assigned to the reactive behavior than to the goal seeking behavior. Four different desirability values have been chosen and tested, namely $\boldsymbol{d} = \left(d_a\ d_g\right) : (0.7\ 0.3)\,,\ (0.8\ 0.2)\,,\ (0.9\ 0.1)$, and (0.95 0.05). In all the cases, except the third, the robot either collided with obstacles or failed to reach the goal at all. For $\boldsymbol{d}$ values of (0.9 0.1), the resulting bias has produced a path which is both safe and goal directed, though folded.

5.7 Curse of Dimensionality

The labyrinth problem discussed in chapter 4 has discrete states and actions. Most potentially useful applications of reinforcement learning, however, take place in multidimensional continuous state space. The obvious way to transform such state space into discrete space involves quantization that partitions the state space into multidimensional grid and treats each cell within the grid as an atomic state. Although this can be effective in certain problem domains, such as [8], a simple grid approach generally leads to impractical memory requirements. The problem is not just the memory space, most importantly the time and data needed to fill them accurately. Bellman [9] has identified this stumbling block of learning in large spaces by coining the phrase *curse of dimensionality*.

In large space, it is seldom the case that the entire space requires a fixed resolution of partitioning, since there are significant sub-spaces of the state space that are either unimportant or for which the optimal response is the same throughout. However, it is quite often the case that some critical areas require high resolution. So, it would be inefficient to represent the all space at a high level of resolution. Therefore, there is a problem of apportioning available memory according to the perceived distribution of the inputs and the *variability* of the target function in order to build effective reinforcement learning systems.

In some cases, apportionment can be facilitated by choosing an appropriate representation for states and actions. For instance, if the sensors provide only two bits of relevant information, then it makes sense to represent them as a boolean and not as an integer [29]. But generally to deal with the problem of large state and action space, some *generalization techniques* that allow compact representation of learned information, which experience only a limited subset of the state space and yet produce a good approximation over a much larger subset, are required.

Function Approximators

A key element in the solution of reinforcement learning problem is the value function. The purpose of this function is to measure the long-term utility or value of a state, and it is important because an agent can use this value to decide what to do, section 1.6. The common problem in continuous space is that, the value function should represent the value of infinitely many states. For this reasons, parameterized function approximators such as neural networks are used to generalize between similar situations or states.

The use of multi-layer sigmoid neural networks for value function approximation has worked well [42, 69, 128], but there is no reason to believe that such network are well suited to reinforcement learning. First, they tend to forget episodes unless they are retrained for those episodes frequently. Second, the need to make small gradient descent steps makes learning slow, particularly in the early stages.

Therefore, instead of a global approximator, a sparse and coarse-coded local approximator known as radial basis functions (RBFs) network (section 1.5) is employed. This is similar to [39, 88] who applied the RBF network in the framework of reinforcement learning to approximate the value function locally by interpolating among the previously visited state values. The approximating function is a single layer RBF network, where the input neurons are fully connected to the high dimensional continuous input vector $\boldsymbol{x}$, via the excitatory connection vectors, figure 5.6. The basis functions determine their activation from the distance between the sensory input and the excitatory connection vector, i.e.,

$$\phi_j(\boldsymbol{x}) = \exp\left(-\frac{\| \boldsymbol{x} - \boldsymbol{c}_j \|^2}{\sigma_j^2}\right) \tag{5.9}$$

where $\boldsymbol{c}_j$ is the excitatory connection vector and σ_j is the receptive field of the basis function. We have experienced that adapting the width of a RBF neuron is a very tricky process; a change in the the neurons' receptive widths may results in an input coverage that totally upset the initial clusters, and causes the value function to vary intensively instead of smoothly. In order to avoid this damaging consequences, the widths of the receptive fields of all the neurons in the network are made identical and kept fixed, i.e., $\varsigma = \sigma_j = 0.1;\ \forall j$. It is only the neurons' excitatory connections which are variable and adapted.

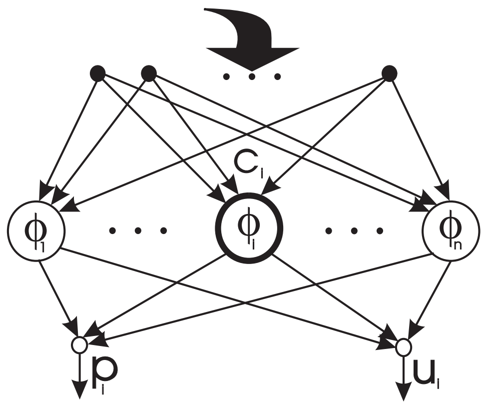


Figure 5.6: *Sparse and coarse-coded function approximator. The network learns to generalize state values from previously experienced states to ones that have never been seen before.*

Initially, the network is empty but grows gradually, similar to the works of [4, 21, 78, 89] as it starts tiling the sub-regions from the high dimensional sensory input. Remember that the entire environment has already been tiled manually into key sub-regions. When a new situation $\boldsymbol{x}$, arrives from the robot's sensory module, existing neurons (if any) compete to win the situation. The winner neuron i, is the one with the largest activation value or with the closest distance to the situation,

$$\mathrm{i} = \arg\max_j \phi_j(\boldsymbol{x}) = \arg\min_j (\boldsymbol{x} - \boldsymbol{c}_j)^T(\boldsymbol{x} - \boldsymbol{c}_j) \tag{5.10}$$

If the distance between the winning neuron and the situation is larger than the width of the receptors, i.e., $\| \boldsymbol{x} - \boldsymbol{c}_\mathrm{i} \| > \varsigma$, the situation can not be generalized; therefore, it is regarded as new or novel. In this case, a new neuron is introduced to the network and following the work of [88], the following three learning parameters, figure 5.6, are attached to the neuron: the position of the neuron in the continuous input space ($\boldsymbol{c}$), the expected discounted sum of reinforcement value, in short called utility (u), and the prototypical action (p). Each of these parameters are initialized first and evolve subsequently through reinforcement learning. This type of adding neurons is called *distance driven* [19] and we shall see later in section 5.9 another way of adding neurons in the network.

Otherwise, if the distance is less than the width of the receptors, the situation is generalized by the center location of the winning neuron, and its state value is approximated by the utility value of the winning neuron, i.e.,

$$\| \boldsymbol{x} - \boldsymbol{c}_{\mathrm{i}} \| < \varsigma \Rightarrow \boldsymbol{x} = \boldsymbol{c}_{\mathrm{i}} \quad \text{and} \quad V(\boldsymbol{x}) = u_{\mathrm{i}} \tag{5.11}$$

In short, based on the distance measure, the RBF network tiles the continuous high dimensional sensory space into coarse and sparse overlapping features space. Whereas the coarseness of the approximator is due to the number of basis functions employed (which typically is much less than the number of data size), the sparsity is due to the variable resolution of tiling the state space. Both these properties of the approximating function and the prior structuring of the state space by hand, section 5.6, enabled us to cope up with the high dimension.

5.8 Localized AHC Architecture

In the preceding section, we mentioned that the function approximating network gradually tiles the high dimensional real-valued state into *features*. It is important to clarify the potential confusion between a real-valued state and a feature. A real-valued state is a real-valued vector in a multi-dimensional space, whereas a feature is a finite discrete entity. Every real-valued state is in a feature space and each feature space contains a set of real-valued states. Just as we have abstracted the sensors in section 2.3 per hand, the approximating network builds its own state abstractions or features.

The learning architecture, figure 5.7, is a *localized* adaptive heuristic critic network that combines the function approximating network with a reinforcement-learning neuron. Notice that an additional output parameter μ_{i}, is introduced in the function approximating network. This parameter, like the other three parameters, is created at the same time and holds the location where the reinforcement-learning unit chooses to explore.

The operation of the network is similar to the way the global adaptive heuristic critic network, figure 3.4, operates; namely, the critic network guides how the action network is to be adapted. Architecturally, however, there is a subtle difference between them. In the latter case, two distinct networks (an actor and a critic) are adapted simultaneously; whereas, in the former case the actor and critic networks are lumped together and only a single network is adapted. Furthermore, in figure 3.4 the critic and action values are *indirectly* adjusted by adapting some parametric weights of their respective networks, but here the algorithms *directly* adapt the critic and action values.

Real Valued Stochastic Exploration

In section 3.5, we pointed out that pure exploitation is not enough, an agent has to explore its environment in order to discover a better policy than the one it is currently pursuing. So far we have seen exploration techniques that are applicable for discrete actions. But exploration in discrete action space, such as the one described in section 4.8, would provide no useful notion of actions selection mechanism for continuous real-valued actions.

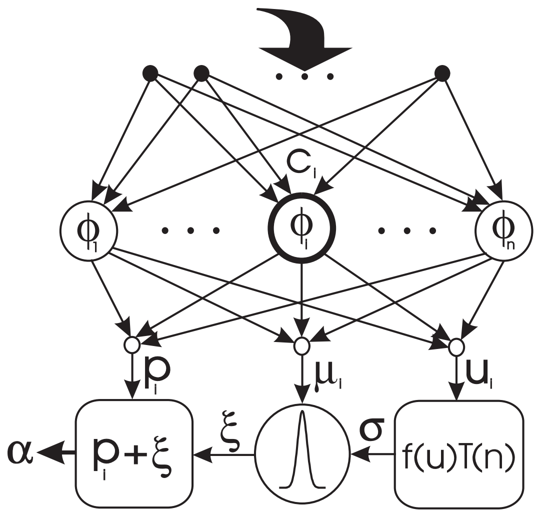


Figure 5.7: *A localized AHC network consisting of hidden and output layers. While the RBF hidden layer generalizes states and state values, the reinforcement-learning unit chooses action by exploring a restricted action space, taken (with modification) from [88].*

Gullapalli [41, 42] has developed a neural reinforcement-learning unit for use in continuous action space. The unit is a multi-parameter stochastic exploration unit that draws actions from a given distribution and adjusts the parameters based on the experience. Following Gullapalli's work, the approximator network is cascaded with a stochastic output layer, which computes the scalar continuous action stochastically from the normal distributed function given by,

$$\mathcal{N}(\xi; \mu, \sigma) = \exp\left(-\frac{||\xi - \mu||^2}{\sigma^2}\right) \tag{5.12}$$

A stochastic unit determining its output according to such distribution would first compute the values of the parameters μ and σ, that control the distribution process. As can be seen from the architecture, these values are directly derived from the learning parameters of the current winning neuron, i.e.,

$$\mu = \mu_{\mathrm{i}} \tag{5.13}$$

$$\sigma = f(u_{\mathrm{i}})T(n) = \frac{T(n)}{1 + \exp\left(u_{\mathrm{i}}\right)} \tag{5.14}$$

where $T(n)$ is a computational temperature. While the mean determines the location where the unit explores, the variance controls the exploratory behavior of the unit independent of where it chooses to explore.

Earlier, we intuitively mentioned that an agent can use the utility values to decide what to do. This intuition is translated into a working design by letting the utility values control the extent of exploration. In the experiment, we have related the extent of exploration to the utility value through a sigmoid function, equation (5.14). If the utility is small, i.e., the chosen action is not performing well, the variance will become high, resulting in exploration of the range of choice. On the other hand, when an action performs well, i.e, the utility is high, the mean moves in that direction and the variance decreases, resulting in a tendency to generate more action values near the successful one.

Once the network derives the parameters of the stochastic distribution from equations (5.13) and (5.14), it draws a random value ξ, according to equation (5.12). The random value modulates the prototypical action p, provided it lies with in $-T(n) < \xi < T(n)$, to generate the final stochastic action α that is send to the robot motor.

$$\alpha = \begin{cases} p_{\mathrm{i}} + T(n) & : \text{ if } \xi > T(n) \\ p_{\mathrm{i}} - T(n) & : \text{ if } \xi < -T(n) \\ p_{\mathrm{i}} + \xi & : \text{ otherwise} \end{cases} \tag{5.15}$$

Annealing

The parameter T is a kind of computational temperature, as used in [7], that balances the trade off between exploration and control. It appears in equations (5.14) and (5.15). While in the former case the temperature shrinks the extent of exploration around the mean, in the latter one it stabilizes the final action. Both are achieved by decreasing the temperature after each trial until it reached a preselected minimum value. A trial is defined as a time interval that begins as soon as the robot starts moving from the home location toward the goal and stops immediately after the robot reaches the goal.

Initially, the temperature is set at some high value, equation (5.16). This is the period when the controller makes no attempt to control the robot. As long as the temperature is high, the controller explores the environment in order to discover better actions. As the trial goes on, the temperature decreases, equation (5.17), and the controller gradually switches from pure exploration to control.

$$T(0) = T_{max} \tag{5.16}$$

$$T(n) = T_{min} + \beta(T(n-1) - T_{min}) \tag{5.17}$$

In equations (5.16) and (5.17) $T_{max} = 100$, $T_{min} = 10$ are the initial and final temperature values, n is trial index, and $\beta = 0.8$ is an adjustable parameter. The parameter β determines how fast the computational temperature is cooled down. It is chosen in such a way that it encourages enough exploration during the initial trials and discourages it afterwards.

5.9 Adaptations

It has to be recalled that for each feature state, represented by RBF neurons in figure 5.7, four learning parameters: p, u, μ, and $\boldsymbol{c}$ (subscripts omitted for simplicity) are associated with it. Further, the parameters are initialized as soon as neurons are created and they are adapted following the receipt of a reinforcement value. Since adaptation takes place after the receipt of a reinforcement value, the controller adapts the parameters of the past state before it generates an action for the present state. Different adaptation algorithms and error sources are used to evolve the four learning parameters. In the following section, we shall look at these algorithms and error sources.

Utility Update

The utility value of the past winning neuron is updated by the methods of temporal difference [122]. Let us introduce the subscript i, as the index of the past winning neuron and the subscript j, as the index of the present winning neuron. The utility value of the past winning neuron at time $t+1$ is estimated by, see section 3.6,

$$u_{\mathrm{i}}(t+1) = r(t+1) + \gamma u_{\mathrm{j}}(t+1) \tag{5.18}$$

where $r(t+1)$ is the reinforcement signal at time $t+1$ caused by the state and action choice made at time step t and $\gamma = 0.70$ is a discounting factor. During learning the stored utility $u_{\mathrm{i}}(t)$, and the estimated utility $u_{\mathrm{i}}(t+1)$, are not equal either because the stored utility does not yet converge or the controller chooses a non-optimal action. Therefore, the difference between the estimate and the stored utility values gives the temporal difference error,

$$\delta(t+1) = u_{\mathrm{i}}(t+1) - u_{\mathrm{i}}(t) = r(t+1) + \gamma u_{\mathrm{j}}(t+1) - u_{\mathrm{i}}(t) \tag{5.19}$$

If the TD error is not larger than a given threshold, say $\Delta = 50$, the past utility u_{i} is adapted by,

$$u_{\mathrm{i}}(t+1) = \begin{cases} u_{\mathrm{i}}(t) + \eta_r\, \delta(t+1) & : \text{ if } \delta(t+1) > 0 \\ u_{\mathrm{i}}(t) + \eta_p\, \delta(t+1) & : \text{ if } \delta(t+1) < 0 \end{cases} \tag{5.20}$$

The reason for using two learning rates $\eta_r = 0.5$ and $\eta_p = 0.05$, with $\eta_r > \eta_p$ is following Millán's reasoning. Since a negative TD error is probably caused by an incorrect action selection that results in the wrong estimate than the one stored, the utility is adapted to change only slightly toward the new estimate [88].

Exceptions¶

We estimate the utility value of a newly created neuron by computing the reinforce function, equations (5.6) and (5.7), for the present sonar values and the present state of the robot *only* in the very first trial. In subsequent trials, utility values of newly created neurons are *not* estimated immediately; instead, the learner waits until the robot reaches the goal and then assigns to any neuron created along the path during that trial the true utility value. The true utility value

¶Personal communication with Millán.

is the sum of the immediate reinforcement values received from the moment the learner used that neuron until the goal is reached.

Therefore, the update algorithm, equation (5.20), is not applicable if the past winning neuron is created during the current trial, because its utility value is not yet estimated and so can not be updated. The update algorithm uses the present winning neuron, too. By the same token, if the present neuron is new, updating the past winning neuron is again delayed until the robot reaches the goal.

Hidden States

A learning agent suffers from a hidden state if at any time the agent's state representation is missing information needed to determine the next correct action. The hidden state problem arises as a case of perceptual aliasing; the mapping between the states of the world and the sensations of the agent is not one-to-one (table 1.1).

There are some approaches to address the hidden state problem. The simplest approach is to simply ignore the hidden state problem and apply traditional reinforcement learning methods as if there were no aliased states. Experience has shown that in certain non-Markovian environments, such as [8], this approach works; however, there are many cases in which ignoring hidden states results in a complete failure [86]. A more careful solution to the hidden state problem is to avoid passing through the perceptual aliased states. This is the approach taken in Lion algorithm [141]. Whenever the agent finds a state that delivers an inconsistent reward, it sets that state's utility so low that the policy will never again visit it. The success of this algorithm depends on the deterministic world and on the existence of a path to the goal.

Our controller does not avoid aliased states, but does the best it can given the non-Markovian state representation while also evading the disasters that would result from ignoring the hidden state altogether. It involves the method first introduced and applied by Millán [88] to identify states that deliver inconsistence reward. The idea is, if the TD error is larger than a specified threshold Δ, then the past situation $x(t)$, is wrongly classified to the past winning neuron i. Because, even if the situation is close to the past winning neuron as measured by equation (5.10), after action is taken it is found out that the estimated utility is quite different from the stored one—as attested by large TD error. Therefore, the controller splits that situation from the past winning neuron by creating a new neuron at the location of the past perceptual input. In this way, sensory states that initially look similar and categorized in the same feature space, will gradually split according to their *consequences*. This is the other criteria of adding neurons in the network and is called *error driven*, where the error is the TD error of equation (5.19). All in all, with the use of the distance and error driven criteria of adding neurons in the network, only similar states with similar consequences are mapped into the same feature spaces.

Mean Update

The appropriate performance measure used to adapt the parameter μ_{i}, that controls the location of exploration is, $E\{u_{\mathrm{i}} \mid \mu_{\mathrm{i}}\}$. In other words, the expected update of the parameter μ_{i}, should lie along the gradient of the performance measure; i.e.,

$$E\{\Delta\mu_{\mathrm{i}} \mid \mu_{\mathrm{i}}\} \propto \Delta_{\mu_{\mathrm{i}}} E\{u_{\mathrm{i}} \mid \mu_{\mathrm{i}}\} \tag{5.21}$$

To determine $\Delta_{\mu_{\mathrm{i}}} E\{u_{\mathrm{i}} \mid \mu_{\mathrm{i}}\}$, we need to know $\partial u_{\mathrm{i}}/\partial \mu_{\mathrm{i}}$. Since the reinforcement signal does not provide any hint as to what the right answer should be in terms of the cost function, there is no gradient information. Hence, the gradient can only be estimated.

The intuitive idea behind the multi-parameter distribution, which is proposed by William [143], is used to estimate the gradient,

$$\frac{\partial u_{\mathrm{i}}}{\partial \mu_{\mathrm{i}}} = \delta(t+1) e_{\mathrm{i}}(t) \tag{5.22}$$

where $e_{\mathrm{i}}(t)$ is the *characteristic eligibility* of μ_{i} that measures how influential μ_{i} was in determining the stochastic action and is given by [143],

$$e_{\mathrm{i}}(t) = \frac{\partial \ln \mathcal{N}}{\partial \mu(t)} = \frac{\partial \ln \mathcal{N}}{\partial \mu_{\mathrm{i}}(t)} = \frac{\xi(t) - \mu_{\mathrm{i}}(t)}{\sigma_{\mathrm{i}}^2(t)} \tag{5.23}$$

and $\delta(t+1)$ is the TD error given in equation (5.19). The characteristic eligibility is the normalized difference between the actual and the expected stochastic actions and is displaced in time from the TD error to reflect the assumption that the reinforcement signal at time $t+1$ depends on the input and actions chosen at the earlier time step t.

Once the gradient information is estimated, the mean is updated so that it lies in the direction of the gradient; i.e.,

$$\mu_{\mathrm{i}}(t+1) = \begin{cases} \mu_{\mathrm{i}}(t) + \beta_r \delta(t+1) e_{\mathrm{i}}(t) & : \text{ if } \quad \delta(t+1) > 0 \\ \mu_{\mathrm{i}}(t) + \beta_p \delta(t+1) e_{\mathrm{i}}(t) & : \text{ if } \quad \delta(t+1) < 0 \end{cases} \tag{5.24}$$

where $\beta_r = 0.2$ and $\beta_p = 0.02$. Similar to utility update, the mean is updated less intensively when the TD error is negative than when it is positive, $\beta_r > \beta_p$.

Center Update

Depending on the performance, the center position of the past winning neuron is shifted toward the past sensation. Retaining as before the subscript i for the index of the past winning neuron,

$$\boldsymbol{c}_{\mathrm{i}}(t+1) = \begin{cases} \boldsymbol{c}_{\mathrm{i}}(t) + \epsilon_r(\boldsymbol{x}(t) - \boldsymbol{c}_{\mathrm{i}}(t)) & : \text{ if } \quad \delta(t+1) > 0 \\ \boldsymbol{c}_{\mathrm{i}}(t) + \epsilon_p(\boldsymbol{x}(t) - \boldsymbol{c}_{\mathrm{i}}(t)) & : \text{ if } \quad \delta(t+1) < 0 \end{cases} \tag{5.25}$$

where $\boldsymbol{c}_{\mathrm{i}}(t)$ is the center position before adaptation, $\epsilon_r = 0.01$ and $\epsilon_p = 0.001$ are two learning rates, and $\boldsymbol{x}(t)$ is the past sensation. Here again the center position is drawn closer to $\boldsymbol{x}(t)$ when the performance, equation (5.19), is positive than when it is negative.

Back Tracking

As the robot moves toward the goal, the controller maintains tuples of four elements $< \mathrm{i}(t), u(\mathrm{i}(t)), \alpha(t), r(t+1) >$, where $\mathrm{i}(t)$—index of the winning neuron, $u(\mathrm{i}(t))$—its associated utility value, $\alpha(t)$—the actual stochastic action, and $r(t+1)$—the resulting reinforcement value. To show the variations of the elements along the route, the elements have a time variable t, as an argument. As soon as the robot arrives at the goal, the controller backs up the utility values of all the neurons that lie along that current route. Backing begins from the goal location and propagates backwards until the home location is reached,

$$u(\mathrm{i}(t-1)) \leftarrow r(t) + u(\mathrm{i}(t)) \tag{5.26}$$

where $t \in \{1, \ldots, T\}$ and T is the time step taken by the robot to reach the goal.

The main advantage of propagating utility values backward is to speed up the learning process without affecting the final utility values [76]. But it also brings a computational advantage. Earlier, we mentioned that when a new neuron is created along a route its utility value is estimated by summing up the reinforcement values starting from the moment the neuron is first created until the goal is reached. But integrating the reinforcement values during the forward path requires a separate integral for each new neuron. Therefore, to avoid multiple integrals, we only mark the locations along the route where new neurons are first introduced. The actual estimation takes place after the robot has reached the goal by propagating the received reinforcements backwards, equation (5.26), and assigning, whenever a marked location is encountered, the current accumulated reinforcement to the utility of the neuron created at that location.

Up to now, we have seen only the techniques of adapting the parameters $\boldsymbol{c}, u$, and μ. Now we will look at how to adjust the prototypical action, p. We first define the total reinforcement of a trial n, as the sum of the immediate reinforcements the robot receives until it reaches the goal,

$$R(n) = \sum_{t=1}^{T} r(t) \tag{5.27}$$

If at a completion of a trial, this value is greater than the total reinforcement values obtained in earlier trials; that is, if $R(n) > R(q); \forall q \in 1, 2, \ldots, n-1$, then the prototypical actions of all the neurons lying along the current trajectory are replaced by their respective actual actions,

$$p(\mathrm{i}(t)) \leftarrow \alpha(t) \quad t = T, \ldots, 1 \tag{5.28}$$

In this way, the initial action acquired from the reflex, will be overridden by a more learned action. It is worth noting that since a particular neuron may win two or more times along a route, the final value stored in $p(\mathrm{i}(t))$ depends on the sequence of replacement—forward or backward. However, we observe that which ever sequence is employed, it has little influence on the overall result.

5.10 Learning Details

Algorithm 9 shows the pseudo code of the main inner loop of the learning algorithm that explores and maintains a variable resolution partitioning of the environment, while learning the minimum cost route.

The robot begins its trial at the home position $\boldsymbol{p}_r(t = 0)$, where it perceives a situation $\boldsymbol{x}$. Since the controller is initially empty (states are not yet encoded in the RBF network), it can not generalize the situation. Therefore, it invokes the reflex component to seek for an action α_r. After the learner receives the action, it creates a neuron and initializes the associated parameters as follows.

The center location of the neuron is initialized to the perceived situation, $\boldsymbol{c} = \boldsymbol{x}$. The prototypical action is equated with the action received from the reflex, $p = \alpha_r$. Since this is the very first trial of the robot, the utility is estimated by computing the reinforce function, equations (5.6) and (5.7), for the present sonar values and robot state. But remember that for latter trials, the utility value is estimated as described in section 5.9. Finally, the parameter that determines the location of exploration is set to zero, $\mu = 0$.

The controller then explores and generates an action that the robot executes in two phases, see section 5.5, and moves to a new location $\boldsymbol{p}_r(t+1)$. Again at this new location, the robot views a new situation $\boldsymbol{x}(t+1)$ and gets a reinforcement value $r(t+1)$ from the environment for the action that brought the robot from $\boldsymbol{p}_r(t)$ to the current location.

This situation is presented to the controller that first identifies the winning neuron, equation (5.10), among the existing neurons. If the distance of the situation is outside the receptive field of the winning neuron, a new neuron is added to the network at the present sensation as discussed in section 5.7. Otherwise, the situation is generalized, equation (5.11), and appropriate action is later generated by the winning neurons.

The parameters of the neuron that won the past situation $\boldsymbol{x}(t)$, are adapted by first computing the TD error, equation (5.19). Adaptation is skipped if the consequence of the past situation is higher than the expected value, in which case the association of the past situation is *undone* by creating a neuron at that situation.

After either adapting the parameters of the past winning neuron or creating a new neuron at the past situation, the controller explores and generates an action for the current situation. If this action takes the robot to the goal, the current trial is killed, the robot is manually guided to the home location, and a new trial is started. Otherwise, the controller continues the adaptation and exploration processes until the robot reaches the goal. Finally, the learning process is terminated after the robot has tried a specified number of trials.

In the pseudo code below, the indices i and j stand for the winning neurons at time step t, and $t+1$ respectively.

```
#define TRIAL
#define DELTA

begin
   step=0; trial=0; set T, eqn. (5.16);
   while(trial<TRIAL)
     perceive next situation x(t+1), eqn. (5.5);
     receive reinforcement, eqn. (5.6) & (5.7);
     if(goal)
       backup utility, eqn. (5.26);
       compute R(trial), eqn. (5.27);
       q=1; flag=0;
       while(q<trial&&!flag) then { flag=R(trial)<R(q); q++; }
       if(!flag) replace p, eqn. (5.28);
       cool T, eqn. (5.17);
       step=0; trial++;
       kill agent; reposition robot;
     else if(collision) // real or virtual collision
       rescue robot; // clear collision, continue trial
     else if(controller empty)
       call reflex; add neuron;
       explore and act eqn. (5.15); step++;
     else
       find winner i, eqn. (5.10);
       if(|| x - c_i ||> ς) // distance driven
         call reflex; add neuron;
       else
         compute δ(t+1), eqn. (5.19);
         if(δ(t+1) > DELTA) // error driven
           add neuron at x(t);
         else
           adapt u_i, eqn. (5.20);
           adapt μ_i, eqn. (5.24);
           adapt c_i, eqn. (5.25);
       explore and act eqn. (5.15); step++;
end
```

Algorithm 9: *Pseudo code of the inner loop of the learning algorithm.*

5.11 Learning Experiments

There is great appeal in using simulated robots for investigating robot learning. Besides being affordable, simulations simplify the logistics of experimentation. However, it can not be made sufficiently realistic, because in a simulated environment much of the richness and unpredictability of the real world is lost [76]. Nevertheless, working only with real robots seems to be too costly in terms of time and necessary resources. Simulation, therefore, retains an important role in eliminating infeasible control strategies at an early stage, performing many experiments, and varying parameter values prior to access to the real robot [32].

A reasonable compromise would then involve using the simulated robot to develop a first approximation of the final controller, which can be refined through direct training of the real robot. In this section, we will present results obtained by the path taken from an early development in the simulation to the actual testing on the actual robot.

Simulation Experiments

Simulation vs. Reality

The controller described so far is the one that was implemented on the physical environment. Prior to its implementation, however, another controller that worked on a simulated robot world was developed. Despite the fact that most parts of the simulated controller were carried over to the real world, the difference between the simulated world and the real world necessitated two modifications to be made before transferring to the physical robot.

First, the controller of the simulator [47] had only the reflex bias, section 5.6, and key states were disambiguated by the adaptive state construction algorithm alone. Nevertheless, when this controller was used on the physical robot, it was no longer possible, as it was on the simulator, to split the key world states quickly. Therefore, the controller had been modified to accommodate additional bias, which substantially altered the architecture from monolithic to hierarchical flat architecture, see figure 5.4.

Second, in the simulation work as soon as a real or virtual collision occurred on the robot path, the agent was immediately killed and a new trial was initiated by placing the robot back to its home location. Although this scheme worked well on the simulation, it was inefficient when applied to the real robot. Therefore, instead of killing the agent and aborting the trial altogether, an emergency routine was incorporated into the controller that rescued and enabled the robot to resume its trial.

A Note on the Simulation

At the time of simulation, the TRC robot (figure 2.1), was the only platform available in the our robotics laboratory. Therefore, the simulator was built keeping this robot in mind as the target robot. However, after we had completed the simulation work, we acquired a B21 robot that has far better sensor-motor characteristics and development software than the TRC robot. To exploit the hardware and software advantages of the new platform, instead of working on the TRC, for which the simulator was written, we began working on the B21. Therefore, while the simulation results were obtained from the TRC simulator, the real results were obtained from the B21 robot.

At this point, we want to stress that the reason why a major architectural adjustment was needed when transferring to the physical robot was *not* due to the change in the platform. The motivation for using simulation is not to finish the design stage there and transfer the components directly to the physical robot; rather to allow us to come up with a working (often coarse, however) version of the controller. Hence, architectural corrections were inevitable even when the platform had not been changed, see [146] for an example.

Results

While figure 5.8 top and bottom-left shows the ghost paths of the robot at different sampled trials, figure 5.9 shows the learning curves against the number of trials. As can be seen in figure 5.9 top-right and bottom-left, the robot totally failed to reach the target during the first eight trials. This was not surprising, because in the beginning the controller was empty and had to acquire enough situation-action pairs from the reflex component. Note that, since the total number of moves and the total reinforcement value were not defined during failed trials, no data was available to plot in these trial instances.

At trial nine the robot reached the goal for the first time. It is during this trial the controller received the lowest total reinforcement, figure 5.9 bottom-left. Furthermore, the path followed during this trail, figure 5.8 top-right, did not look like a planned trajectory. It was more of a haphazard motion rarely driven by instinct. After the robot had reached the goal by the ninth trail, it had chosen eight times non optimal actions that ultimately led to real or virtual collisions, figure 5.9 top-right and bottom-left. This is due to the stochastic nature of the learner in selecting action. In the twelfth trial and afterwards, the size of network, figure 5.9 top-left, was almost saturated, i.e., between trials only few neurons were added. This indicates that after the twelfth trial, the reflex component practically stopped intervening and the controller started operating in a pure reinforcement mode.

Beginning with trial 30, the robot visited the goal constantly. The total number of moves fluctuated only within the range of $143 - 148$, figure 5.9 top-right, the total reinforcement value remained stable within ± 3, except in trial 41, where the controller explored other actions from the currently known optimal values, figure 5.9 top-right and bottom-left. In subsequent trails, however, it had quickly discovered its past performance. The final result, figure 5.8 below-left, demonstrated that the robot had rapidly adapted the coarse and instinct skill acquired from the reflex component to get a smooth, short (10% shorter than figure 5.8, top-right) and planned like trajectory.

We also tested the behavior of the final controller by altering the start location of the robot. Figure 5.8 below-right shows the trajectories produced when the robot was started at four different start locations. The robot was still able to produce feasible paths even when it started from other locations for which the controller was not trained for. This was true provided that the new starting locations were within the region that the robot had already explored during its learning phase. When an initial robot locations outside the explored region was chosen, the controller generalization ability drastically deteriorated. This is quite natural, because the controller decoded the situation-action pairs only locally. As a result, when the robot was placed in a location that was not explored before, either the learned action of the active neuron was quite different from the one that was needed in that location or even worse no neuron was active.

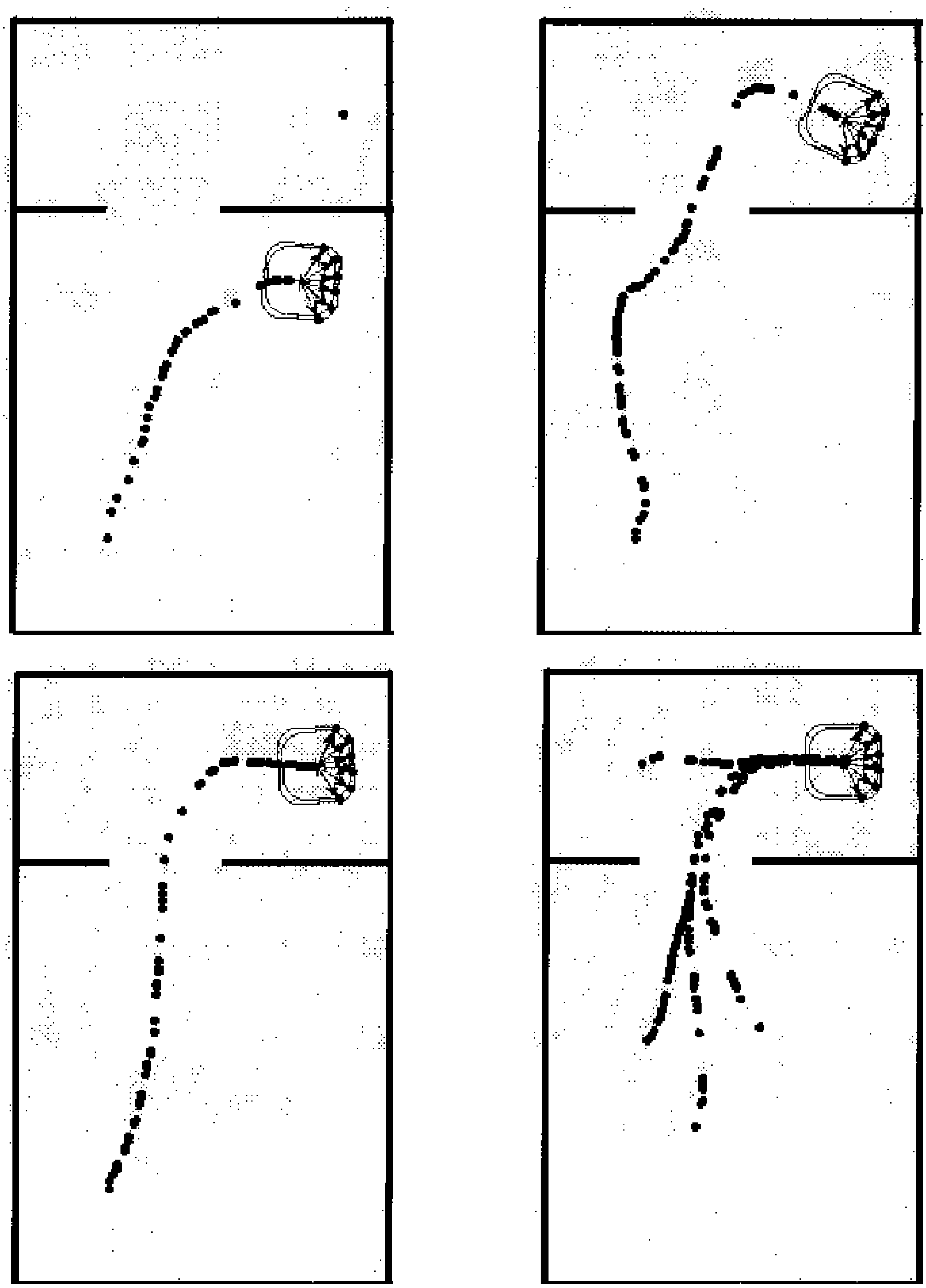
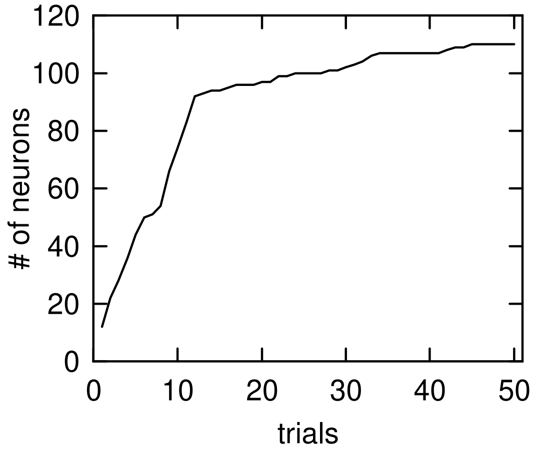
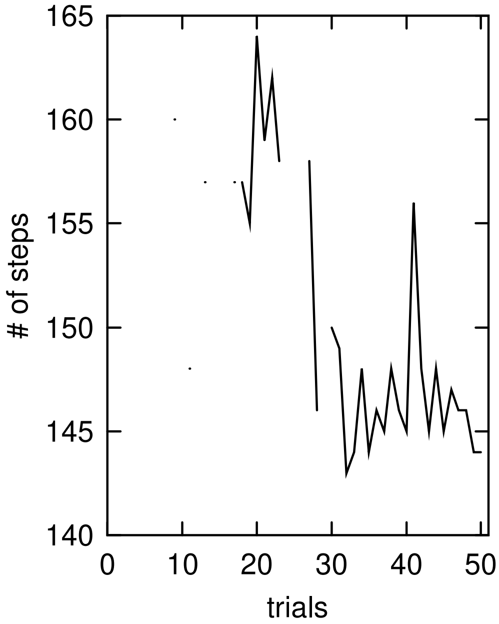


Figure 5.8: *Trajectories of the simulated robot at the first (top-left), ninth (top-right) and final (below-left) trials. A ghost was left as the controller toggles neurons to enable us pictorially represent where along the route neurons were concentrated. The behavior of the final controller for four sampled starting locations (below-right).*





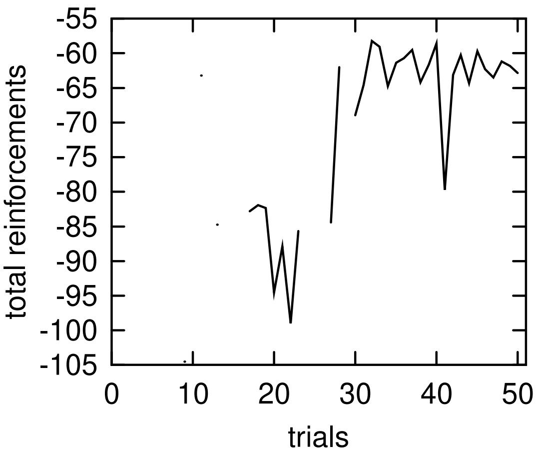


Figure 5.9: *Learning curves of a simulated robot. Top-Left: Size of the network vs. number of trials. Top-Right: Number of actions the robot has required to reach the goal. Bottom-Left: Total reinforcement (penalty) the robot has incurred.*

Real Experiments

Parallel with the results of the simulation work, results obtained on the physical robot will now be presented. As pointed out at the beginning of this section, all the results were obtained after the necessary architectural adjustments on the controller of the simulator had been made.

Figure 5.10 shows sample robot trajectories and figures 5.11-5.13 are plots of the learning curves against the number of trials. In the first trial, the controller produced a trajectory that was no better than what the basic reflex had produced. Like the earlier simulation results, the actual robot also incurred a high payoff in the first few trials, figure 5.13, and the neurons added to the network grew sharply, figure 5.11, signifying that the robot was in exploration phase.

As the trials went on, however, the robot gradually started to unfold its path. In addition, neurons were added to the network at a more reduced slope than the earlier trials. By the sixth trial and afterwards the robot had practically straightened its path, except by the eighth trial where the robot left the optimum path in search for a better one. Again, in subsequent trials the robot returned to its previous performance and followed the same path through out the remaining trials with out significant divergence. This same phenomena was observed in the work of [88], too.

To test the repeatability of the learning curves, ten sets of experiments, each consisting of 20 trials were carried out. The vertical error bars in figures 5.11-5.13 indicate the minimum and maximum variations of the respective values at each trial in the set of ten experiments. As can be seen from the curves, as the trial increases, the variation in the value decreases. Table 5.1 shows the mean and variance of the three quantities of the learning curves, computed from the last trial in the set of ten experiments.

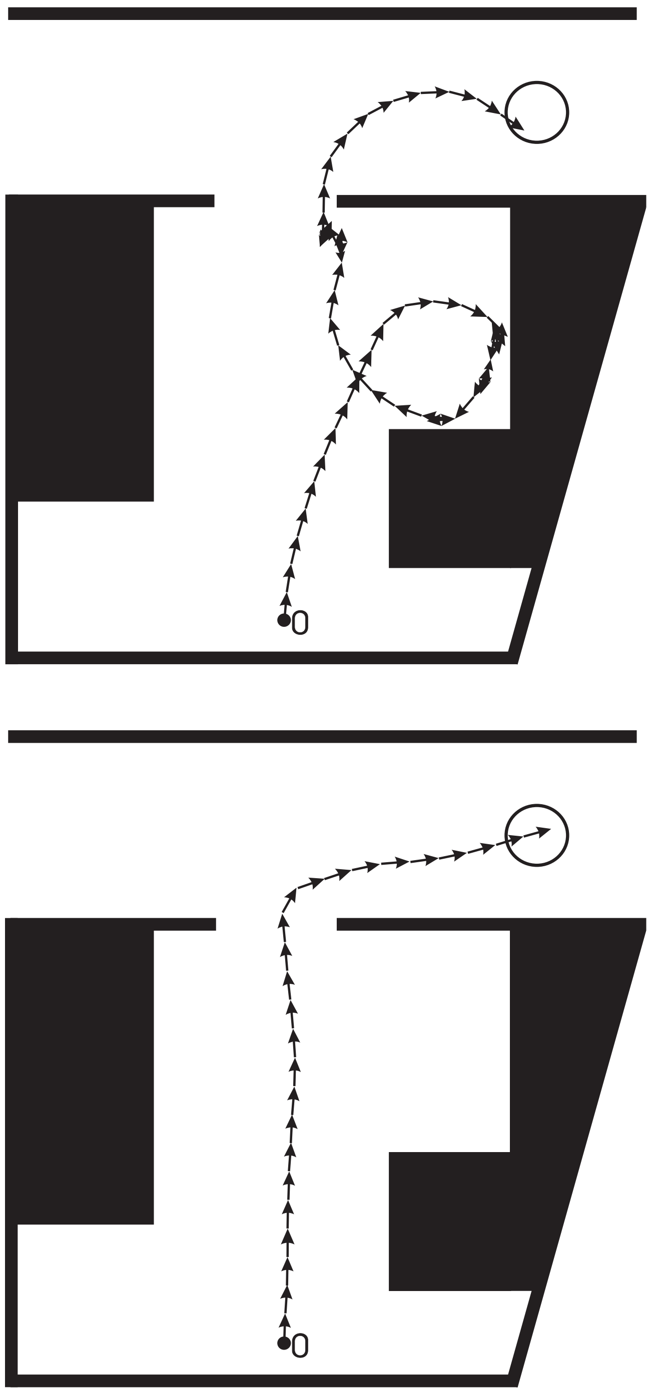


Figure 5.10: *Physical robot: The trajectories of the robot during the first (above) and final (below) trials. The robot has learned 1) to skip the concave region that causes the robot to fold its path, 2) to pass in the middle of the door, and 3) to head directly to the goal after it has passed the door.*

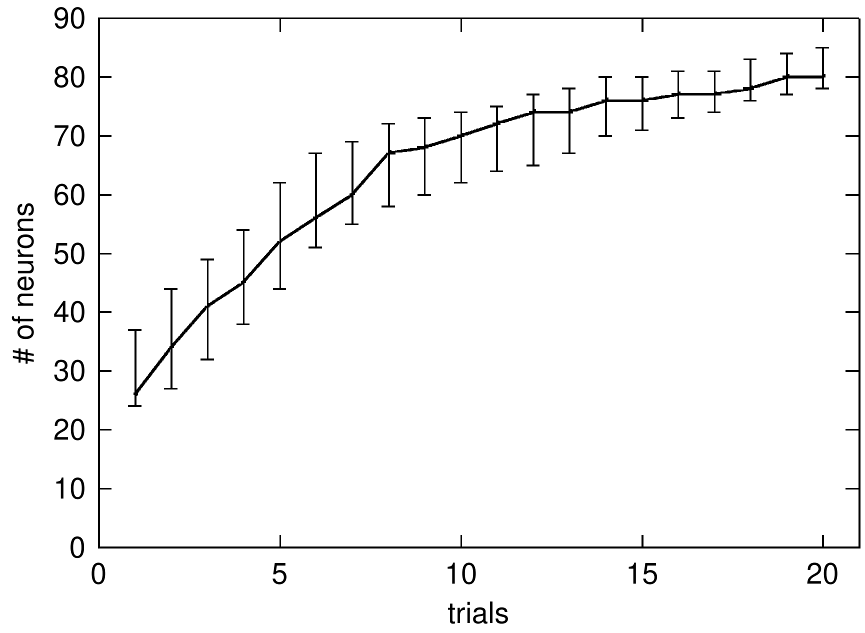


Figure 5.11: *Physical robot: Number of neurons or network size.*

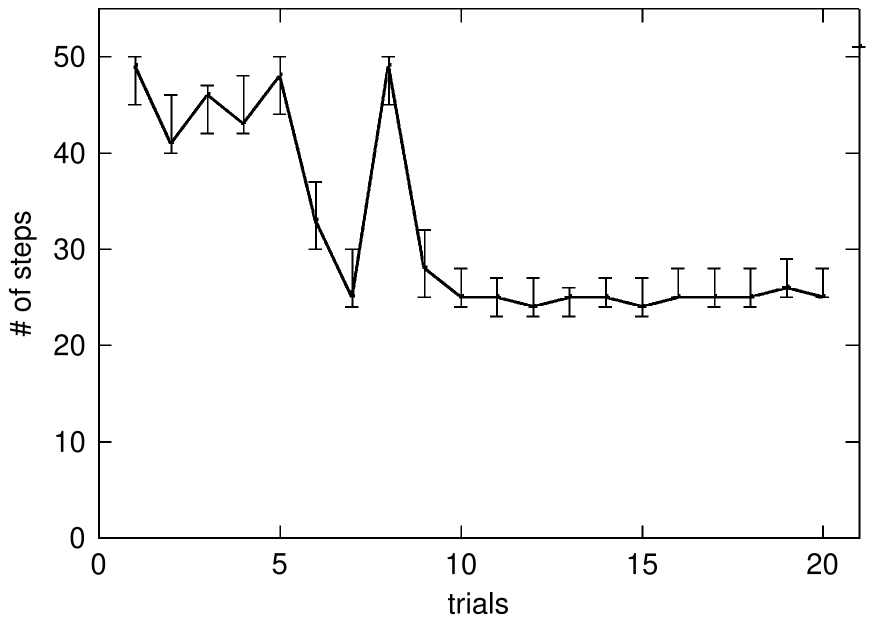


Figure 5.12: *Physical robot: Steps taken before the robot arrived at the goal.*

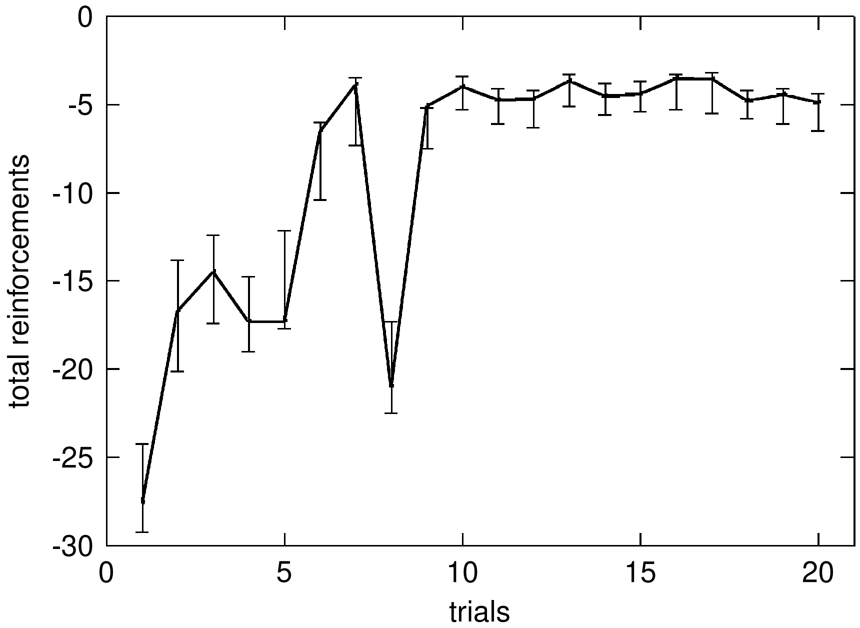


Figure 5.13: *Physical robot: The total reinforcement incurred at each trials.*

Table 5.1: *Final Network Performance.*

Quantities	*Mean*	*Variance*
Number of neurons	82.5	4.7286
Number of steps	27.7	1.9000
Total reinforcements	-6.24	0.8752

5.12 Related Work

Due to its slow learning process reinforcement learning is not yet widely applied like its counter part–supervised learning. Many factors, ranging from definition to algorithms, that contribute to the slow learning process have been discussed in this thesis. In the past, techniques have been proposed to speed up the learning process either by modifying the definition or the learning algorithm. We will survey some of the outstanding techniques and compare them with our work.

Mahadevan et al. [76] have employed Q-learning to learn a box pushing task. To speed up learning, they have broken the task into three behaviors each with pre-wired applicability conditions. In addition they have used statistical clustering technique for compact state space representation. The decomposition of the task into three behaviors and the use of pre-wired applicabilty conditions can be viewed as endowing the controller with some prior knowledge. However, their robot have learned Q-values that demand a tabular state and action representation, but our robot have learned a value function in continuous state and action space that requires a neural network for state representation and generalization.

Matarić [82] has applied reinforcement learning in a multi-agent environment. To accelerate learning, she has replaced states by conditions and actions by behaviors. Moreover, she has introduced heterogeneous reinforcement signal in the reward function. Conditions and behaviors are more suitable to subsumption like architecture [17] and are not directly applicable to connectionisit architecture that work close to the raw sensory data.

McCallum [6] has argued that existing reinforcement algorithms pass back rewards only to states along the current path and has modified the Q-learning algorithm to pass rewards along other paths, too. Even though we agree with the argument, the method he proposed is again valid when states and actions are represented in a look up table, hence hardly applicable to continuous space. Lin et al. [68] have used reinforcement learning to tune and create new fuzzy rules. Their architecture is a five layer neural network that employs, at least in some sense, the error back-propagation. Our architecture, on the other hand, has only one hidden layer with no back-propagation.

The work presented is close to Millán [88] who has implemented reinforcement learning on a Nomad robot for a similar task, but it is different from him in some ways. One major difference, as mentioned in section 5.6, is in the physical capability of the robot. While the Nomand has a turret motor that enables it to focus its sensors on the goal, we are experimenting here on a robot that lacks this ability. Domain specific knowledge about the environment, which is absent in [88], has been integrated into our learner to cope up with the dimension. The strategy, however, did not entirely eliminate the dependence of the sensor readings on the robot orientation. As a result, in the last few trials the network did not abate to grow. Consequently the variance of the final network performance is higher than that reported in [88]. The other difference is in the way exploration is done. Instead of a counter based search strategy the expected future reinforcement of the winning neuron is used to determine the exploration range.

5.13 Summary

In order to build a truly effective reinforcement learning system, we have to embed sufficient built-in knowledge or bias. In this chapter, two kinds of built-in knowledge that made reinforcement learning possible on B21 platform are presented.

The first one is a priori domain knowledge that partially structures the state space and imposes architectural constraints on the controller. As much as possible only essential domain-specific knowledge is embedded into the controller, so that the controller maximizes autonomy and minimizes parsimony. The approach is effective in pre-structuring key states, but the process of embedding semantics about the world is ad hoc.

The second one is a hard-wired controller that is independent of any particular environment and, therefore, does not have to be rewired in new environments. This pre-wired controller focuses the exploration by suggesting actions given the current view of the environment and the robot's internal representation. Although the need for exploration could not be entirely removed, the use of this pre-wired controller restricts the set of actions from which the learner composes a control policy. As a result, exploration is conducted in a space that excludes most of the unacceptable policies.

The learning architecture is a localized adaptive heuristic architecture that was proposed by [88]. Similar to [49], reaction rules are first acquired from the hard-wired controller by associating sensors (input stimuli) with actuators (responses). Subsequently, these acquired situation-action rules are adjusted through reinforcement learning. The associator is a radial basis function network [14, 51], which simultaneously functions as a coarse and sparse-code value function approximator.

Through the use of domain knowledge, the environment is initially partitioned in just four sub-regions, but unless the robot is lucky, this coarse partitioning would not be adequate to accomplish the learning task. Therefore, besides learning the appropriate actions and state values from the external reward values, the controller also learns to construct feature spaces, where the reinforcement learning take place, from the high dimensional continuous sensor spaces. Since it is not efficient to partition the whole space uniformly, the feature spaces are constructed using the distance and error criteria, which allow high resolution partition only to critical areas.

Behavior observation is one of the primary methods of validating a learning agent. It calls for experiments to be carried out on a real robot. Nevertheless, working only with real robots is found to be too expensive, specially in terms of time. The time it takes the robot just to move practically dominates the computational time of the learning algorithm! Therefore, prior to accessing the real robot, sufficient experiment is conduced with a simulated robot.

The reinforcement task faced by the simulated as well as the physical robots is the minimum cost path problem, where the robots should reach a pre-defined goal while fulfilling their externally motivated need. The need is such that the path followed by the robots shall be short, as well as have sufficient clearance from obstacles. In both domains the results show that the robots have indeed learned the required skill or behavior with an unprecedented learning speed. Both robots have quickly unfolded their initial trajectory and have consistently followed those trajectories that have the minimum payoff.

Although simulation results often hold, with minor changes, in the real world, in the minimum cost path problem, results obtained with the simulator did not hold in the real world. In particular, the perceptual aliasing problem which occurs occasionally during simulation but frequently in the real world, has necessitated a substantial modification to be made on the controller of the simulator before it is transfered to the physical robot. These findings conclusively validates the growing consensus in the field that to be considered useful for real robotic applications, a learning technique must be tested on a real robot [32].

Chapter 6

Conclusions

What is not today will be tommorrow.

Petronius

In this final chapter, we will tie together the main ideas raised throughout the previous chapters. At last, we will conclude the thesis with some closing thoughts about present day learning robots.

6.1 Discussion and Outlook

This dissertation has addressed three fundamental problems of learning robots, namely: the processing of sonar range returns, the role of the amount and quality of bias in learning, and the realization of delayed reinforcement learning on a physical robot. Let us take a brief look at what has been presented in the thesis concerning each of these subjects.

Mobile robot sensors have limited abilities; instead of providing a description of the world, they return simple properties such as the presence of and distance to objects within a fixed sensing region. In order to recover a robust spatial information of the robot world from the low-level sensing and to efficiently utilize this information in control, robots are equipped with multiple sensors. When multiple sensors are mounted on a robot, two sensor processing problems arise. First, the sensor space must be collapsed into few appropriate levels from which control strategy can be easily derived. Second, the inherent uncertainties present in the individual sensors must be removed. To accomplish these tasks a **hashing technique** and a **fusion method** have been proposed and validated in the thesis.

The hashing technique lifts up the input representation to a higher level by decomposing the large and unstructured sensor space into a much smaller and semantically meaningful set of spaces. The knowledge of the physical geometry of the robot has been used to regroup and relabel the sensors into five semantic regions, which are assumed reasonable for the intended task. This process of relabeling sensory information into forms that can be used by a controller for achieving a goal is already a step in the direction of bridging the signal-to-symbol gap. Hashing also allows parallelism—by having each processor execute the same set of control rules on different sensory space, full utilization of all the processors is achieved.

Once the problem of a huge sensor space has been by-passed by regrouping the sensors into reasonable regions, the true depth of the regions are estimated by a cascade of two filters; namely, a median and a Kalman filter. The former filter selects the median value of the sensors in a group and stores it in memory for further processing. While the latter filter propagates a Gaussian conditional probability density function through the stored measurements. The propagation is performed by iterating at each measurement the values of the parameters of the density function using the recursive Kalman filter algorithm. At the end, the maximum likelihood criterion is applied to the final density function to estimate the true region depth.

The method has been compared with other existing techniques by conducting both an off-line and an on-line experiments, which consist of a TRC robot, ten Polaroid ultrasonic sensors, a fuzzy logic controller and a noise promoting lab. environment. In all the experiments conducted, our method has performed far better than the existing techniques. Nevertheless, despite the success, the proposed method has its own deficiencies. First, the hashing technique heavily rests on pre-existing domain knowledge and the level of hashing is a matter of choice; hence, it is determined by trial and error. Second, the fusion technique requires a prior estimate of the noise distribution, which in general is difficult to estimate.

The basic subject behind the rest of the work has been reinforcement learning. It was our desire that the reinforcement learning system should be general enough to solve a variety of tasks with little or no built in knowledge. But this has been hindered by the bias-variance dilemma, which states that generality only comes with the demand for prohibitively large training data. Therefore, bias has to be built into the system. The problem is little is known about how to bias a learning system. If the bias is too much, there is nothing left to learn, on the other hand, if it is too small the system can not converge in a bounded time. Hence, the determination of the appropriate amount and quality of bias is crucial.

As the tradeoff between the built-in and learned knowledge varies across species, so also across physical problems and environments. It is unlikely to determine the right bias for a general problem and environment. Therefore, in order to investigate the amount and quality of bias in learning, attention has been restricted to a class of problem that is broadly known as the labyrinth problem. The main reason for choosing this domain is that it facilitates the coding of prior knowledge; different forms of biases can be easily encoded in a Q table by explicitly representing our belief of the effects of actions on states.

The task in this problem domain has been the learning of the minimum sequence of actions in order to reach a particular goal state. After running Q-learning experiments for all the introduced biases, a number of results have been deduced from the learning curves. 1) Not all biases have a similar influence on the learning speed; while some biases aid learning more intensively, others aid less. 2) No bias has been seen mitigating the learning process, however, care must be taken when constructing a bias. It is important to ensure that the introduced bias is harmless to the final learning performance. 3) The widely accepted method of biasing a learning system by cutting the search space may not always be the best choice. Certain biases, particularly derived from the unique characteristics of the problem, sometimes perform better than the former, in spite of their large space.

One desired feature of a learning system, consistent with the human skill acquisition, is the ability to utilize previously acquired knowledge when attempting to learn a similar but new task. To this end, experiments have been carried out to investigate the utility of using previous knowledge. The results have indicated that the learning algorithm is capable of speeding up learning by taking advantage of and building onto earlier training, even for those new tasks that substantially differ from the original trained task.

The lesson learned from the labyrinth world has been beneficiary in providing insight into what and how to incorporate bias in our real robot task. The real robot task has been a minimum cost path problem, where the robot learns a path that leads to the goal with minimum cost (penality). We emphasis that this task is learned in a *continuous high dimensional* domain, as this is the domain of a realistic robot task. Our quest has been for a learning system which to some extent is provided with a priori knowledge, but where most of its emergent behavior is acquired by learning. Therefore, the system has been endowed with two forms of prior knowledge, namely a rough symbolic representation of the world and a set of basic reflex rules. We have good reasons for choosing these two bias forms.

In a real system with large state space, it is difficult for the robot to quickly delineate the relevant world states from the sensor readings alone. Consequently, without some form of bias that pre-labels the world into its basic constituents or states, learning in large continuous domain is proven to be hopeless. A coarse symbolic representation of the world enables the learner to pre-structure its state space and encapsulates the entire learning process by making a connection between direct sensory experience and high level cognitive knowledge. This bias instantaneously captures the significant features of the world which otherwise would have taken a very long time if these features had been learned. The other bias form is a set of pre-wired rules that specify what the robot should do in a particular situation. These reflex rules enable the robot to be operational and safe right from the beginning; they are deemed to be instinctive because learning them has a high cost. Learning to avoid, for example, has a potentially damaging cost for the robot and, is not a natural learning task, as it appears to be innate in nature, and can be easily programmed on most systems [82].

When learning in large continuous space, most states encountered will never have been experienced exactly before. The only way to learn and successfully cope with such situations is when the system has the ability to generalize from previously experienced states to the one that has never been seen before. In this thesis, a growing radial basis neural network has been used both to generalize the value function, as well as to adaptively construct states. Furthermore, the hidden state problem has been successfully addressed by comparing the stored and the expected state values. Despite the provision of a pre-programmed abstract model of the world, the ability of the network to construct states and to address the hidden state problem from a large continuous state space (a vector of 32 real values) is particularly notable.

What is important about this thesis is that it is among the very few, notably [88] and few others, that have successfully conducted a delayed reinforcement learning experiment on a real robot in a non-Markovian environment, where the theory of reinforcement learning no longer applies. The final result has shown that, despite the lack of the support of the theory, with sufficient bias the robot has been able to learn the minimum cost path with a tremendous reduction in the number of trials. In conclusion, this dissertation asserts that **we can break through the bottleneck and realize reinforcement learning on real robots by endowing the robot with appropriate and systematized prior knowledge**.

6.2 Closing Thoughts

The quest for machines that learn from memorized experience is one of the greatest challenges of modern science. Robot learning is generally a hard problem. Robots, which today are claimed to posses an ability to learn, show a remarkable rigid behavior as compared to the simplest biological systems. So far, the most avant-garde laboratory can not begin to get a robot to do what a 12-month-old infant automatically does: teach itself to balance, walk erect, and instantly tell the difference between a dark shadow and a hole in the floor [120].

It is not likely that we will wake up tomorrow morning to find that learning robots have revolutionized our life. It is not even likely that this will have happened five years from now. Machine learning technology is still a tiny fragile seedling, struggling to survive. The reason most of us care about machine learning is our expectation of what they may be able to do for us in the real world—they hold great promise in making a naturally intelligent artificial system. It is our hope that the results made available in this thesis is a step forward in that direction.

Bibliography

[1] M. A. Abidi and R. C. Gonzalez. The Use of Multisensor Data for Robotic Application. *IEEE Transaction on Robotics and Automation*, RA-6(2):159–177, 1990.

[2] M. A. Abidi and R. C. Gonzalez. *Data Fusion in Robotics and Machine Intelligence*. Academic Press Inc., 1992.

[3] I. Ahrns. Ultraschallbasierte Navigation und adaptive Hindernisvermeidung eines autonomen mobilen Roboters. Master's thesis, Institut für Informatik und Praktische Mathematik, CAU zu Kiel, 1996.

[4] E. Alpaydin. Networks that Grow when they Learn and Shrink when they Forget. Technical Report 91-032, Computer Science Institute, Berkeley, CA, 1991.

[5] J. A. Anderson and E. Rosenfeld. *Neurocomputing, Foundations of Research*. MIT Press, Cambridge MA, 1988.

[6] M. R. Andrew. Using Transitional Proximity for Faster Reinforcement Learning. In *Proceedings of the Ninth International Machine Learning Conference*, pages 316–321, Aberdeen, 1992.

[7] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to Act using Real Time Dynamic Programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[8] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neurolike Elements that can Solve Difficult Learning Problems. *IEEE Transactions on Systems, Man and Cybernatics*, 5(13):834–846, 1983.

[9] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[10] D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London, UK, 1985.

[11] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[12] D. P. Bertsekas and S. E. Shreve. *Stochastic Optimal Control*. New York: Academic, 1978.

[13] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[14] C. M. Bishop. *Neural Networks for Pattern Recognition*. Kluwer Academic Publishing, 1995.

[15] J. Borenstein and Y. Koren. Error Eliminating Rapid Ultrasonic Firing for Mobile Robot Obstacle Avoidance. *IEEE Transactions on Robotics and Automation*, 11(1):132–138, 1995.

[16] J. A. Boyan and M. L. Littman. Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach. Unpublished.

[17] R. A. Brooks. A Layered Intelligent Control System for Mobile Robot. *IEEE Transactions on Robotics and Automation*, RA-2:14–23, 1986.

[18] R. A. Brooks and M. J. Mataric. Real Robots, Real Learning Problems. In *Robot Learning*, pages 193–213. Kluwer Academic Press, 1993.

[19] J. Bruske. *Dynamische Zellstrukturen - Theorie und Anwendung eines KNN Modells.* PhD thesis, Institut für Informatik und Praktische Mathematik, CAU zu Kiel, 1998.

[20] J. Bruske, M. Hansen, L. Riehn, and G. Sommer. Biologically Inspired Calibration-Free Adaptive Saccade Control of a Binocular Camera-Head. *Biological Cybernetics*, 77(6):433–446, 1997.

[21] J. Bruske and G. Sommer. Dynamic Cell Structure Learns Perfectly Topology Preserving Map. *Neural Computation*, 7(4):834–846, 1995.

[22] A. E. Bryson and Y. C. Ho. *Applied Optimal Control*. Blaisdell Publishing Co., Waltham, MA, 1969.

[23] D. Chapman. Planning for Conjuctive Goals. *Artificial Intelligence*, 32(3):333–378, 1987.

[24] D. Chapman and L. P. Kaelbling. Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparison. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1991.

[25] J. Clouse and P. Utgoff. A Teaching Method for Reinforcement Learning. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 92–101, Aberdeen, Scotland, 1992.

[26] M. Colombetti, M. Dorigo, and G. Borghi. Behavior Analysis and Training - A Methodology for Behaviour Engineering. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(3):365–380, 1996.

[27] R. H. Crites and A. G. Barto. Improving Elevator Performance using Reinforcement Learning. In *Advances in Neural Information Processing Systems*, San Francisco, CA, 1995.

[28] Y. L. Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1:541–551, 1989.

[29] T. Dean, K. Basye, and J. Shewchuk. Reinforcement Learning for Planning and Control. In *Machine Learning Methods for Planning and Scheduling*, 1992.

[30] G. Dejong and R. Mooney. Explanation-Based Learning: An Alternative View. *Machine Learning*, 1:145–176, 1986.

[31] B. L. Digney. *Emergent Intelligence in a Distributed Adaptive Control System*. PhD thesis, University of Saskatchewan, Department of Mechanical Engineering and Intelligent Systems Research Laboratory, 1994.

[32] M. Dorigo. Introduction to the Special Issue on Learning Autonomous Robots. *IEEE Transactions on Systems, Man and Cybernetics*, 26(3):361–364, 1996.

[33] D. Drankove, H. Hellendoorn, and M. Reinfrank. *An Introduction to Fuzzy Control*. Springer-Verlag, 1993.

[34] J. A. Franklin and O. G. Selfridge. Some New Direction for Adaptive Control Theory in Robotics. In *Neural Networks for Control*, pages 350–360, San Francisco, CA, 1992.

[35] T. Fujii, Y. Arai, H. Asama, and I. Endo. Multilayered Reinforcement Learning for Complicated Collision Avoidance Problems. In *IEEE Proceedings of Robotics and Automation*, pages 2186–2191, Belgium, Leuven, 1998.

[36] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[37] S. Geman, E. Bienenstock, and René Doursat. Neural Networks and Bias Variance Dilemma. *Neural Computation*, 4:1–58, 1992.

[38] J. C. Gittins. *Multi-armed Bandit Allocation Indices*. Wiley-Chichester, NY, 1989.

[39] G. J. Gordon. Stable Function Approximation in Dynamic Programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, 1995.

[40] R. P. Gorman and T. J Sejnowski. Learned Classification of Sonar Targets using a Massively-Parallel Network. *IEEE Transaction on Acoustics, Speech, and Signal Processing*, 36:1135–1140, 1988.

[41] V. Gullapalli. A Stochastic Reinforcement Learning Algorithm for Learning Real Valued Function. *Neural Networks*, 3:671–692, 1990.

[42] V. Gullapalli. Skillful Control under Uncertainty via Reinforcement Learning. *Robotics and Autonomous Systems*, 15:237–246, 1995.

[43] G. Hailu. Distributed Fuzzy and Neural Network Based Navigation Behaviors. Technical Lab. Report H-696, CAU, Cognitive Systems Laboratory, 1996.

[44] G. Hailu, J. Bruske, and G. Sommer. Fuzzy Logic Control of a Situated Agent. In *Seventh International Fuzzy System Association - World Congress*, pages 494–500, Prague, 1997.

[45] G. Hailu and G. Sommer. Embedding Knowledge in Reinforcement Learning. In *International Conference on Artificial Neural Networks*, pages 1133–1138, Skövde, Sweden, 1998.

[46] G. Hailu and G. Sommer. Integrating Symbolic Knowledge in Reinforcement Learning. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 1491–1495, San Diego, California, 1998.

[47] G. Hailu and G. Sommer. Learning by Biasing. In *IEEE Proceedings of Robotics and Automation*, pages 2168–2173, Belgium, Leuven, 1998.

[48] G. Hailu and G. Sommer. On Amount and Quality of Bias in Reinforcement Learning. In *IEEE International Conference of System, Man, and Cybernetics*, pages 728–733, Tokyo, Japan, 1999.

[49] D. A. Handelman and S. H. Lane. Fast Sensorimotor Skill Acquisition Based on Rule-Based Training of Neural Networks. In *Neural Networks in Robotics*, pages 255–270, 1996.

[50] C. J. Harris, C. G. Moore, and M. Brown. *Intelligent Control - Aspects of Fuzzy Logic and Neural Nets*. World Scientific, 1993.

[51] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theorey of Neural Computation*. Addison Wesley Publishing Company, 1991.

[52] G. E. Hinton. Connectionist Learning Procedures. In Jaime Carbonell, editor, *Machine Learning Paradigms and Methods*, pages 183–234, 1992.

[53] J. J. Holland. Properties of the Bucket Brigade Algorithm. In *Proceedings of an International Conference on Genetic Algorithm and their Applications*, Pittsburgh, PA, 1985.

[54] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[55] R. A. Jacob, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive Mixtures of Local Experts. *Neural Computation*, 3:79–87, 1991.

[56] C. Jou and N. Wang. Training a Fuzzy Controller to Backup an Autonomous Vehicle. In *IEEE International Conference on Neural Networks*, pages 923–928, San Francisco, 1993.

[57] L. P. Kaeblings and S. J. Rosenschein. Action and Planning in Embedded Agents. *Robotics and Autonomous Systems*, 6(1):35–45, 1990.

[58] L. P. Kaelbling. *Learning in Embedded Systems*. The MIT Press, Cambridge, 1993.

[59] L. P. Kaelbling. Reinforcement Learning: The Good, the Bad, and the Ugly. A talk given at the 2nd European Workshop on Reinforcement Learning, 1995.

[60] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement Learning: A Survey. *Artificial Intelligence Research*, 4:237–285, 1996.

[61] O. Khatib. Real Time Obstacle Avoidance for Manipulators and Mobile Robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

[62] T. Kohonen. Self-organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, 43:59–69, 1982.

[63] B. J. A. Kröse and J. W. M. van Dam. Adaptive State Space Quantisation for Reinforcement Learning of Collision-free Navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1327–1331, Raleigh, NC, 1992.

[64] R. Kuc and Di. Intelligent Sensor Approach to Differentiating Sonar Reflections from Corners and Planes. In *International Congress on Intelligent Autonomous Systems*, Amsterdam, The Netherlands, 1986.

[65] R. Kuc and M. W. Siegel. Physically Based Simulation Model for Acoustic Sensor Robot Navigation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(6):766–778, 1987.

[66] M. Kuperstein. Adaptive Visual-motor Coordination in Multi-joint Robots using a Parallel Architecture. In *IEEE International Conference on Robotics and Automation*, 1987.

[67] T. Landelius. *Reinforcement Learning and Distributed Local Model Synthesis*. PhD thesis, Linköping University, Department of Electrical Engineering, 1997.

[68] C. Lin and C. S. G. Lee. Reinforcement Structure/Parameter Learning for Neural-Network-Based Fuzzy Logic Control Systems. *IEEE Transactions on Fuzzy Systems*, 2(1):46–63, 1994.

[69] L. Lin. *Reinforcement Learning for Robots using Neural Networks*. PhD thesis, Carnegie Mellon Univeristy, Department of Computer Science, 1993.

[70] R. C. Luo and M. G. Kay. Data Fusion and Sensor Integration: State-of-the-Art 1990s. In Mongi A. Abidi and Rafael C. Gonzalez, editors, *Data Fusion in Robotics and Machine Intelligence*, 1992.

[71] R. Maclin and J. W. Shavlik. Incorporating Advice into Agents that Learn from Reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 694–699, 1994.

[72] R. Maclin and J. W. Shavlik. Creating Advice Taking Reinforcement Learner. *Machine Learning*, 22:251–282, 1996.

[73] P. Maes. The Dynamics of Action Selection. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 991–997, Detroit, MI, 1989.

[74] P. Maes and R. A. Brooks. Learning to Coordinate Behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 796–802, San Mateo, CA, 1990.

[75] S. Mahadevan. Enhancing Transfer in Reinforcement Learning by Building Stochastic Models of Robot Action. In *Machine Learning - Proceedings of the Ninth International Workshop (ML92)*, pages 290–299, Aberdeen, 1992.

[76] S. Mahadevan and J. Connell. Automatic Programming of Behavior-based Robots using Reinforcement Learning. *Artificial Intelligence*, 55:311–365, 1992.

[77] E. H. Mamdani and S. Assilian. An Experiment in Lingustic Synthesis with a Fuzzy Logic Controller. *International Journal of Man-Machine Studies*, 7:1–13, 1967.

[78] T. Martinetz and K. Schulten. Topology Representing Networks. *Neural Network*, 7(3):507–522, 1993.

[79] M. J. Matarić. Distributed Model for Mobile Robot Environment-Learning and Navigation. Technical Report 1228, MIT Artificial Intelligent Laboratory, 1990.

[80] M. J. Matarić. A Comparative Analysis of Reinforcement Learning Methods. Technical Report 1322, MIT Artificial Intelligent Laboratory, 1991.

[81] M. J. Matarić. Integration of Representation into Goal-Driven Behavior-Based Robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, 1992.

[82] M. J. Matarić. *Interaction and Intelligent Behavior*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1994.

[83] M. J. Matarić. Reward Functions for Accelerated Learning. In W. W. Cohen and H. Hirsh, editors, *Proceedings of the Eleventh International Conference on Machine Learning*, 1994.

[84] P. S. Maybeck. *Stochastic Models, Estimation, and Control*. Academic Press, NY, 1979.

[85] P. S. Maybeck. The Kalman Filter: An Introduction to Concepts. In I. J. Cox and G. T. Wilfong, editors, *Autonomous Robot Vehicles*. Springer-Verlag, 1994.

[86] A. R. McCallum. Overcoming Incomplete Perception with Utile Distinction Memory. In *Proceedings of the Tenth International Machine Learning Conference*, pages 190–196, Amherst, Massachusetts, 1993.

[87] J. R. Millán. Reinforcement Learning of Goal-Directed Obstacle-Avoiding Reaction Strategies in an Autonomous Mobile Robot. *Robotics and Autonomous Systems*, 15:275–299, 1995.

[88] J. R. Millán. Rapid, Safe and Incremental Learning of Navigation Stratagies. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(3):408–420, 1996.

[89] J. R. Millán. Incremental Acquisition of Local Networks for the Control of Autonomous Robots. In *Seventh International Conference on Artificial Neural Networks*, pages 739–744, Lausanne, Switzerland, 1997.

[90] J. R. Millán and C. Torras. A Reinforcement Connectionsit Approach to Robot Path Finding in a non Maze-like Environments. *Machine Learning*, 8(3–4):363–395, 1992.

[91] T. M. Mitchell. The Need for Biases in Learning Generalizations. In *Reading in Machine Learning*, pages 1114–1120, San Mateo, CA, 1990.

[92] T. M. Mitchell, M. T. Mason, and A. D. Christiansen. Towards a Learning Robot. Technical Report 89-106, CMU-CS, 1989.

[93] J. E. Moody and C. J. Darken. Fast Learning in Networks of Locally-tuned Processing Units. *Neural Computation*, 2(1):281–294, 1989.

[94] A. W. Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, Univeristy of Cambridge, 1990.

[95] A. W. Moore. Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-valued Spaces. In *Proceedings of the Eighth International Machine Workshop*, 1991.

[96] A. W. Moore and C. G. Atkeson. Prioritized Sweeeping: Reinforcement Learning with Less Data and Less Real Time. *Machine Learning*, 13:103–130, 1993.

[97] U. Nehmzow. *Experiment in Competence Acquisition for Autonomous Mobile Robots*. PhD thesis, University of Edinburgh, 1992.

[98] U. Nehmzow, T. Smithers, and J. Hallam. Steps Towards Intelligent Robots. Technical Report 502, University of Edinburgh, 1990.

[99] D. B. Parker. Learning Logic. Technical Report TR-47, Sloan School of Management, MIT, Cambridge, 1985.

[100] D. W. Payton, J. K. Rosenblatt, and D. M. Keirsey. Plan Guided Reaction. *IEEE Transaction on Systems, Man, and Cybernetics*, 20(6):1370–1382, 1990.

[101] J. Pearl. *Heuristics, Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, Inc., 1984.

[102] J. Peng and R. J. Williams. Efficient Learning and Planning within Dyna Framework. *Adaptive Behaviors*, 1(4):437–454, 1993.

[103] D. A. Pomerleau. *Neural Network Perception for Mobile Robot Guidance*. Kluwer Academic Publishers, 1993.

[104] P. Probert. Low Cost Sensor for Reactive Planning. In Stephen Cameron and Penelope Probert, editors, *Advanced Guided Vehicles*, pages 61–83. World Scientific, 1994.

[105] M. L. Puterman. *Markov Decision Processes - Discrete Stochastic Dynamic Programming*. John Wiley & Son, Inc., NY, 1994.

[106] P. Reignier. Fuzzy Logic Techniques for Mobile Robot Obstacle Avoidance. *Robotics and Autonomous Systems*, 12:143–153, 1994.

[107] P. Reignier. Supervised Incremental Learning of Fuzzy Rules. *Robotics and Autonomous Systems*, 16:57–71, 1995.

[108] F. Rosenblatt. *Principle of Neurodynamics*. Spartan, Chicago, 1962.

[109] S. M. Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, NY, 1983.

[110] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation. *Parallel Distributed Processing*, 1:318–362, 1986.

[111] S. J. Russell and P. Norvig. *Artificial Intelligence - a Modern Approach*. Prentice-Hall International, Inc., 1995.

[112] A. Saffiotti, E. H. Ruspini, and K. Konolige. Using Fuzzy Logic for Mobile Robot Control. In D. Dubois, H. Prade, and H.J. Zimmermann, editors, *Handbook of Fuzzy Sets and Possibility Theory*. Kluwer Academic, 1997.

[113] A. L. Samuel. Some Studies in Machine Learning using the Game Checkers. *IBM Journal of Research and Development*, 3:211–229, 1959.

[114] S. P. Singh. Transfer of Learning by Composing Solutions of Elemental Sequential Tasks. *Machine Learning*, 8:323–339, 1992.

[115] S. P. Singh, A. G. Barto, R. Grupen, and C. Connolly. Robot Reinforcement Learning in Motion Planning. In *Advances in Neural Information Processing Systems*, pages 655–662, San Mateo, CA, 1994.

[116] G. Sommer. Verhaltenbasierter Entwurf technischer visueller Systeme. *Künstlichen Intelligenz (KI)*, 3:42–45, 1995.

[117] G. Sommer. Algebric Aspects of Designing Behavior Based Systems. In G. Sommer and J. Koenderink, editors, *Algebraic Frames for Perception-Action Cycle, Lecture Notes in Computer Science 1315*, pages 1–28, Kiel, Germany, 1997.

[118] K. T. Song and J. C. Tai. Fuzzy Navigation of a Mobile Robot. In *Proceeding of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 621–627, Raleigh, 1992.

[119] M. Sugeno and M. Nishida. Fuzzy Control of a Model Car. *Fuzzy Sets and Systems*, 16:103–113, 1985.

[120] C. Suplee. Robot Revolution. *National Geographic Society*, 192(1):76–95, 1997.

[121] R. S. Sutton. *Temporary Credit Assignment in Reinforcement Learning*. PhD thesis, Univeristy of Massachusetts, Amherst, MA, 1984.

[122] R. S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[123] R. S. Sutton. First Result with Dyna, an Integrated Architecture for Learning, Planning and Reacting. In W. Thomas, R. S. Sutton, and P. J. Werbos, editors, *Neural Networks for Control*. Bradford Book, 1992.

[124] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. A Brandford Book, MIT Press, 1998.

[125] T. Takeuchi, Y. Nagai, and N. Enomoto. Fuzzy Control of a Mobile Robot for Obstacle Avoidance. *Information Science*, 45:231–248, 1988.

[126] J. Tani and N. Fukumura. Learning Goal-Directed Sensory-Based Navigation of a Mobile Robot. *Neural Networks*, 7(31):553–563, 1994.

[127] G. Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257–277, 1992.

[128] G. Tesauro. TD-gammon, a Self-teaching Backgammon Program, Achieves Master-level Play. *Neural Computations*, 6(2):215–219, 1994.

[129] C. K. Tham and R. W. Prager. A Modular Q-learning Architecture for Manipulator Task Decomposition. In *Proceedings of the Eleventh International Conference on Machine Learning*, 1994.

[130] S. B. Thrun. The Role of Exploration in Learning Control. In *Handbook of Intelligent Control; Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, NY, 1992.

[131] S. B. Thrun. *Explanation-based Neural Network Learning, A Lifelong Learning Approach*. Kluwer Academic, 1996.

[132] S. B. Thurn. Learning Metric-Topological Maps for Indoor Mobile Robot Navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[133] G. G. Towell and J. W. Shavlik. Knowledge-based Artificial Neural Networks. *Artificial Intelligence*, 70:119–165, 1994.

[134] Transition Research Corporation. *Labmate User Manual Version 5.21 L-F and Proximity Subsystem User Manual Release 4.6H*, Connecticut, USA, 1986.

[135] J. A. Walter. *Rapid Learning in Robotics*. PhD thesis, Technische Fakultät, Universität Bielefeld, 1996.

[136] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Kings's College, Cambridge, UK, 1989.

[137] C. J. C. H. Watkins and P. Dayan. Technical Note—Q Learning. *Machine Learning*, 8:279–292, 1992.

[138] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behaviroual Science*. PhD thesis, Harvard Univeristy, Cambridge, MA, 1974.

[139] P. J. Werbos. Back Propagation through Time: What It is and How to Do It. In *Proceedings of the IEEE*, pages 1550–1560, San Diego, CA, 1990.

[140] P. J. Werbos. A Menu of Design for Reinforcement Learning Over Time. In W. Thomas, R. S. Sutton, and P. J. Werbos, editors, *Neural Networks for Control*. Bradford Book, 1992.

[141] S. Whitehead and D. Ballard. Active Perception and Reinforcement Learning. In *Proceeding of the Seventh International Conference on Machine Learning*, pages 179–188, Austin, TX, 1990.

[142] S. D. Whitehead and D. H. Ballard. Learning to Perceive and Act by Trial and Error. *Machine Learning*, 7:45–83, 1991.

[143] R. J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256, 1992.

[144] R. J. Williams and L. C. Baird. Tight Performance Bound on Greedy Policies Based on Imperfect Value Functions. Technical Report NU-CCS 93-14, Northeastern Univeristy, College of Computer Science, Boston, MA, 1993.

[145] P. H. Winston. *Artificial Intelligence*. Addison Wesley Publisher, 1984.

[146] B. Yamauchi and R. Beer. Spatial Learning for Navigation in Dynamic Environments. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(3):496–504, 1996.

[147] J. Yen and N. Pfluger. A Fuzzy Logic Based Extension to Payton and Rosenblatt's Command Fusion Method for Mobile Robot Navigation. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(6):971–977, 1995.

[148] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

[149] L. A. Zadeh. A Fuzzy-Set-Theoretic Interpretation of Lingustic Hedges. *Journal of Cybernetics*, 3(2):4–34, 1972.

[150] W. Zhang and T. G. Dietterich. A Reinforcement Learning Approach to Job Scheduling. In *International Joint Conference on Artificial Intelligence*, pages 1114–1120, Cambridge, MA, 1995.

Reinforcement learning, in a nutshell, is a form of learning that enables the robot to construct a control law by a system of feedback signals that reinforce ”electrical path ways” that produce correct response, and conversely wipe-out connections that produce errors. Unfortunately, without biasing, it is a weak learning that presents unreasonable difficulty, especially when it is applied to real robots. The subject of this thesis is to study, for a particular class of problem, the effects of different form of biases on the speed of learning as well as on the quality of final learned policy, and to realize this learning paradigm on a physical robot by appropriately biasing the robot with domain knowledge that determines how much the robot knows about the different parts of its world.

Getachew Hailu was born in 1966 in Selalie. He received his M.Sc and B.Sc degrees in Electrical Engineering both from the Addis Ababa University (AAU). From 1990 to 1993 he was a Junior Associate at the Microprocessor Laboratory of the International Center for Theoretical Physics (ICTP), Trieste, Italy. From September 1994 to March 1995, he was a visiting researcher at Robotics Laboratory of the Toaki University, Kanagawa, Japan. From October 1995 to July 1999, he perused his Ph.D study in Computer Science at the Christian Albrechts University (CAU), Kiel, Germany.