Ulrich Nehmzow

Mobile Robotics: A Practical Introduction

Springer

3.5 Further Reading

- On robot sensors: Johann Borenstein, H.R. Everett and Liqiang Feng, *Navigating Mobile Robots*, AK Peters, Wellesley MA, 1996.
- On robot sensors and robot actuators: Arthur Critchlow, *Introduction to Robotics*, Macmillan, New York, 1985.
- For specific information on sonar sensors see [Lee 95], [Leonard *et al.* 90].
- For information regarding laser range finders see http://www.zf-usa.com.

4 Robot Learning: Making Sense of Raw Sensor Data

Summary. This chapter introduces fundamental concepts of robot learning and machine learning, discusses commonly used mechanisms such as reinforcement learning and connectionist approaches, and presents three case studies of mobile robots that can learn.

4.1 Introduction

Section 4.1 of this chapter discusses the fundamental principles that constitute the robot learning problem, and relates the robot learning problem to the machine learning problem. This section is fairly abstract, and introduces general definitions only.

Section 4.2 is more practical, and explains some commonly used mechanisms of reinforcement learning in mobile robots. This section also gives references, where more detailed information about the discussed algorithms can be found.

The chapter concludes with three case studies of learning robots and an exercise. The case studies give examples of self-supervised learning, supervised learning and unsupervised learning. The exercise is for your own enjoyment — see whether you can do it!

4.1.1 Motivation

Learning establishes in an agent a procedure, a capability or knowledge that was not available before, at the design stage. Learning is unnecessary for implementing capabilities that are completely understood, and can be implemented using a fixed, "hardwired" structure. However, if such a fixed structure cannot be identified at the design stage, learning is a way to establish the desired competence later, through interaction with the environment.

There are many reasons why it may be impossible to implement a competence at the design stage. There may be incomplete knowledge of task, agent or environment. We may not know the environment the agent is to operate in, or the task it is to perform. We may also not know the precise properties of the agent's sensors and actuators, any slight defects or idiosyncrasies.

Another reason for not being able to implement a competence at the design stage would be that environment, task or agent are known to change over time, in unpredictable ways. In environments inhabited by humans, for instance, objects can change their position at any time (moving furniture), object properties may change (painting the walls), or general environmental conditions may change (switching light on and off). The task may change as the agent is pursuing a high level goal that involves achieving various different subgoals. And finally, the agent may change in unpredictable ways, too — sensor and actuator characteristics of robots, for instance, change with decreasing battery charge, or changes in temperature and humidity. *ALDER* (see figure 4.14) had the irritating habit of curving slightly left when instructed to move straight, only to start curving right once the batteries ran low. It was impossible therefore to compensate for its inability to move straight, using a hardwired control strategy.

Finally, we (that is the human designers) may be unable to implement a suitable fixed control structure at the design stage, because we perceive the world differently to a robot. We refer to this as the *problem of perceptual discrepancy*. We sometimes simply don't know what the best control strategy would be.

Here is an example for the problem of perceptual discrepancy. Most people would think that good landmarks for a mobile robot navigating in an office-type environment would be doors. Doors are highly visible from long distances, and at varying angles. They are designed to be detected easily. However, they are designed to be detected easily *by humans* — experiments show that for a robot it is far easier to detect the slightly protruding door *frames*, rather than the doors themselves. The frames reflect sonar bursts very well, and shine like beacons in the darkness; and yet for a human door *frames* are completely inconspicuous.

Nothing ever looks exactly the same the second time. Some objects change their appearance over time (like living beings), sometimes environmental conditions are different (e.g. lighting), or the viewing perspective is different. *Generalisation* — a form of learning — can help to identify the salient features of an object, whilst ignoring the changeable ones.

It is for these reasons that we are interested in adding a learning capability to a mobile robot. Learning will enable robots to acquire competences that cannot be implemented by the human designer at the design stage, for any of the reasons given above.

4.1.2 Robot Learning versus Machine Learning

The Metal Skin Metaphor Consider a mobile robot like *FortyTwo*, shown in figure 3.17. This 16-sided robot possesses a metal skin which separates the interior of the robot from the "world out there", the "real world" (referred to simply as the "world" from now on). Embedded in this metal skin are the robot's sensors, for example the sonars and IR sensors.

Some properties of the world are perceived by the robot's sensors, and made available to the inside — that was lies inside the metal skin of the robot — as electrical signals. For example, some property of the world that made a sonar burst return to the robot after 4 milliseconds arrives at the inside in the form of a signal saying "66 *cm*". The latter signal is not equivalent to the former, it is merely *one* of many possible sensations arising from the state of the world.

The metal skin separates the world from the inside of the robot, and the metal skin metaphor allows us to distinguish between the robot learning problem and the machine learning problem.

The *robot learning problem* addresses the question of making a robot perform certain tasks in the world successfully. The *machine learning problem*, on the other hand, addresses the problem of how to obtain one well defined goal state from the current well defined system state. The machine learning problem is what goes on *inside* the metal skin, whereas the robot learning problem sees robot, environment and task as one inseparable entity (see figure 4.1).

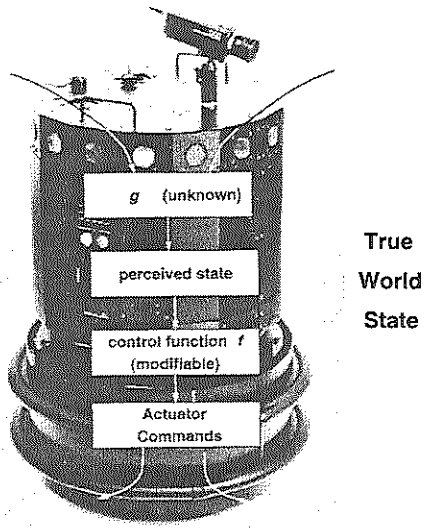

FIG. 4.1. THE ROBOT LEARNING PROBLEM. THE TRUE STATE OF THE WORLD IS INACCESSIBLE TO THE ROBOT - ONLY THE PERCEIVED STATE IS AVAILABLE TO THE LEARNING PROCESS

The true state of the world is transformed, by some unknown function g, into the perceived state of the world. This perceived state of the world can be associated with actions through a control function f, which is modifiable through learning. Changing f is the machine learning problem, which is the subject of the following discussion.

A More Formal Description of Machine Learning Every learning process has some "goal", and the first consideration is how to define that goal.

Because we only have access to the perceived state of the world, rather than the true state of the world "out there", the most precise definition for goals is in terms of perceived states, i.e. sensor states.

However, it is not always possible to do so, because often there is no one-to-one mapping between some desired behaviour of the robot in the world and the perceived state of the world. Typically, during the execution of actions the same perceived states may occur many times, so that it is impossible to detect the goal behaviour by detecting a goal perceived state of the world. In those cases, qualitative descriptions of the goal have to suffice, which makes the analysis of the learning system more difficult.

Assuming that G is an identifiable goal state, and X and Y are two different perceived states of the world, the machine learning problem can be formally described by

$$G : X \to Y; R\,, \tag{4.1}$$

interpreted as "if the robot finds itself in a state satisfying condition X, then the goal of reaching a state satisfying condition Y becomes active, for which a reward R is received."

For example, the goal of recharging a low battery can be represented in this way, by setting

X = "Battery level low",
Y = "Robot is connected to battery charger", and
R = 100.

Given a set of such goals, we can define a quantitative measure of robot performance such as the proportion of times that the robot successfully achieves condition Y given that condition X has been encountered, or the sum of the rewards it receives over time. If we wish, we might further elaborate our measure to include the cost or delay of the actions leading from condition X to condition Y.

Given this definition of *robot performance* relative to some set of goals G, we can say that the robot learning problem is to improve robot performance through experience. Thus, robot learning is also relative to the particular goals and performance measure. A robot learning algorithm that is successful relative to one set of goals might be unsuccessful with respect to another. Of course we are most interested in general-purpose learning algorithms that enable the robot to become increasingly successful with respect to a wide variety of goal sets.

Characterisation of the Machine Learning Problem by Target Function

We said above that the machine learning problem can be described as the problem of acquiring the policy that will yield the maximum cumulative reward, as the agent moves from system state to system state. Using the descriptors s for the system's current state ($s \in S$, with S being the finite set of all possible states), a the action selected ($a \in A$, with A being the finite set of all possible actions) and V being the expected discounted future reward, different machine learning scenarios can be described as follows.

The simplest situation would be to learn a control function f directly, from training examples corresponding to input-output pairs of f (equation 4.2).

$$f : S \to A. \tag{4.2}$$

One example of a system that learns f directly is Pomerleau's ALVINN system ([Pomerleau 93]). It learns the control function f for steering a vehicle. The system learns from training data obtained by watching a human driver steer the vehicle for a few minutes. Each training example consists of a perceived state (a camera image of the road ahead), along with a steering action (obtained by observing the human driver). A neural network is trained to fit these examples. The system has successfully learned to drive on a variety of public roads.

Another example of this situation is given in case study 2 on p. 74, where *FortyTwo* learns several sensor-motor competences by observing the actions of a human trainer.

In other cases, training examples of the function f might not be directly available. Consider for example a robot with no human trainer, with only the ability to determine when the goals in its set G are satisfied and what reward is associated with achieving that goal. For example, in a navigation task in which an initially invisible goal location is to be reached, and in which the robot cannot exploit any guiding information present in the environment for navigation, a sequence of many actions is needed before the task is accomplished. However, if it has no external trainer to suggest the correct action at each intermediate state, its only training information will be the delayed reward it eventually achieves when the goal is satisfied through trial and error. In this case, it is not possible to learn the function f directly because no input-output pairs of f are available.

One common way to address this problem is to define an auxiliary function $Q : S \times A \to V$, an evaluation function of the expected future reward V, that takes into account system state s and the action taken (a). This method, Q-learning, is discussed in detail in section 4.2 of this chapter.

Using Q, the immediate feedback (which is unavailable) is replaced by the internal predictions of how well the agent is going to do.

There are further general categories of the machine learning problem: The robot could learn to predict the next perceived state s' of the world, given it takes an specific action a in state s: $NextState : s \times a \to s'$, where s' is the state resulting from applying action a to state s. One very useful property of $NextState$ is that it is task-independent and can support any learning process. An example where prediction is successfully applied is given in [O'Sullivan *et al.* 95].

Finally, the robot could learn to map from raw sensory input and actions to a useful representation of the robot's state, $Perceive : Sensor^* \times a^* \to s$. This corresponds to learning a useful state representation, where the state s is computed from possibly the entire history of raw sensor inputs $Sensor^*$, as well as the history of actions performed a^*. Again, $Perceive$ is task-independent and can be used to support any learning application.

Characterisation of the Machine Learning Problem by Training Information

There are three major classes of training information that can be used to control the learning process: supervised, self-supervised and unsupervised.

In *supervised learning*, training information about values of the target control function are presented to the learning mechanism externally. A prototypical example is the supervised training data obtained from the human trainer for the ALVINN system described above. The *backpropagation algorithm* for training artificial neural networks (see p. 63) is one common technique for supervised learning. The training information provided in supervised control is usually an example of the action to be performed, as in ALVINN or case study 2 (p. 74).

In *self-supervised* learning, the actual learning mechanism is the same as in supervised learning. However, the *external* feedback given by the trainer in the supervised case is replaced by *internal* feedback, supplied from an independent, internal control structure. In reinforcement learning, discussed in section 4.2 below, this is the "critic". In case study 1 (p. 69) this is the "monitor".

Finally, *unsupervised learning* clusters incoming information without using input-output pairs for training, but instead exploiting the underlying structure of the input data. Kohonen's self-organising feature map is a well known example of an unsupervised learning mechanism ([Kohonen 88]). Unsupervised learning, i.e. cluster analysis, has an important role in optimising the robot's organisation of sensory (training) data. For example, unsupervised learning can identify clusters of similar data points, enabling the data to be represented in terms of orthogonal (highly characteristic) features, without the user knowing nor specifying what those features are. This reduces the effective dimensionality of the data, enabling more concise data representation and supporting more accurate supervised learning. An example of such an application is case study 3 on p. 80.

4.1.3 Further Reading on the Robot Learning Problem

- Ulrich Nehmzow and Tom Mitchell, The Prospective Student's Introduction to the Robot Learning Problem, *Technical Report UMCS-95-12-6*, Manchester University, Department of Computer Science, Manchester, 1995 (available at `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-95-12-6.ps.Z`.

4.2 Learning Methods in Detail

4.2.1 Reinforcement Learning

Introduction

> "[The term reinforcement learning covers techniques of] learning by trial and error through performance feedback, i.e. from feedback that evaluates the behaviour, ...but does not indicate correct behaviour" ([Sutton 91]).
> "Examples of optimal solutions are not provided during training" ([Barto 95]).

Usually, the form this performance feedback takes is a simple "good/bad" signal at the end of a sequence of actions, for example after finally having reached a goal location, grasped an object, etc. This type of situation is common in robotics, and reinforcement learning can therefore be applied to many robotics tasks.

Reinforcement learning can be viewed as an optimisation problem, whose goal it is to establish a control policy that obtains maximum reinforcement, from whatever state the robot might be in. This is shown in figure 4.2.

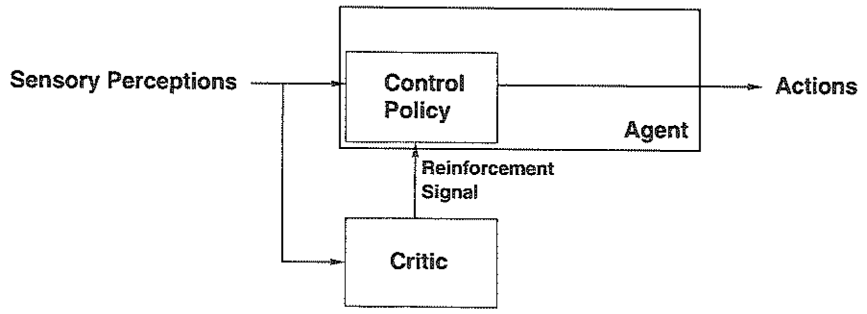

FIG. 4.2. THE REINFORCEMENT LEARNING SCENARIO. PERFORMING ACTIONS IN THE WORLD, THE AGENT SEEKS TO MAXIMISE CUMULATIVE REWARD.

Reinforcement learning techniques are particularly suitable for robotic applications in which mistakes of the robot are not immediately fatal and where some sort of evaluation function of the robot's performance exists. Reinforcement learning uses an overall performance measure (the reinforcement) to control the learning process ([Barto 90] and [Torras 91]), in this it differs from supervised learning schemes (for example certain kinds of connectionist computing architectures), which use specific target values for individual units. This property can be particularly useful for robotics, where often only the overall desired behaviour of the robot is known; however at the same time this can also be a problem, as it can be difficult to establish which parameter within the controller to alter in order to increase the reinforcement.

It is through this performance feedback that a mapping from state (the representation of a particular situation) to action is learned.

[Sutton 91] gives the following overview of reinforcement learning architectures for intelligent agents:

- *Policy only* architecture, which is the simplest architecture. Here, the policy of the agent is the only modifiable structure. These architectures work well only if the rewards are distributed around a baseline of zero (that is positive reinforcement is a positive number, negative reinforcement a negative number, they can't both be positive, with the former being bigger than the latter).
- *Reinforcement comparison techniques* use a prediction of the reward as the baseline and are thus able to cope with rewards distributed around a non-zero baseline.

- The *Adaptive Heuristic Critic* architecture (see below for a detailed discussion) uses a predictor of return (the long-term cumulative reward), not reward, to take non-immediate rewards into account. Neither of the previous two architectures can do this.
- In *Q-learning* (again, see below) the predicted return is a function not only of state, but also of the action selected. Finally,
- *Dyna architectures* are reinforcement learning architectures that contain an internal world model. For each single step of selecting an action and performing it in the real world, Dyna architectures perform another k learning steps using the world model (k is an integer number).

Reinforcement Learning Can Be Slow [Sutton 91] reports that in a simulation on path finding where start and goal location are 16 squares in a 9 x 6 grid apart, it takes a Dyna-adaptive heuristic critic four steps in the simulated world and another 100 steps ($k = 100$) per each of those four steps to find the path. If a new obstacle is placed in the way, the new path is found in a "very slow" process. He also presents a simulation of a Dyna-Q system that has to find a path of length 10 (squares). This takes 1000 time steps, and another 800 after an obstacle is moved in the way. At $k = 10$ this means 100 steps in the simulated environment, and another 80 to find a new path. For a real robot this can be too slow a learning rate. The only cost to be paid in simulation is that of computing time; in robotics however the cost function is a different one: apart from the fact that due to their battery capacity robots only operate for a certain length of time, certain competences such as obstacle avoidance *have* to be acquired very quickly in order to ensure safe operation of the robot. The conclusion is: for mobile robotics it is crucial that the learning algorithm is fast (on the slow speed of reinforcement learning see also [Brooks 91a]).

The fact that reinforcement learning can be extremely slow is shown by other researchers, too. [Prescott & Mayhew 92] simulate the *AIVRU* mobile robot and use a reinforcement learning algorithm similar to the one described by Watkins ([Watkins 89]). The sensor input space of the simulated agent is a continuous function, simulating a sensor that gives distance and angle to the nearest obstacle. The simulated world is 5 m x 5 m in area, the simulated robot 30 cm x 30 cm. Without learning, the agent runs into obstacles in 26.5% of all simulation steps, and only after 50,000 learning steps this rate drops to 3.25%.

[Kaelbling 90] compares several algorithms and their performances in a simulated robot domain. The agent has to stay away from obstacles (negative reinforcement is applied if it hits an obstacle), it receives positive reinforcement if it moves near a light source. Kaelbling reports that all reinforcement learning algorithms investigated (Q-learning, interval estimation plus Q-learning and adaptive heuristic critic plus interval estimation) suffered from the fact that the agent often acquired an appropriate strategy only very late in the run, because it did not come near the light source in the early stages of the learning process and thus did not receive positive reinforcement. After 10,000 runs, the different algorithms obtained average reinforcement values of 0.16 (Q-learning), 0.18 (interval estimation plus Q-learning) and 0.37 (adaptive heuristic critic plus interval estimation). A hand-coded "optimal" controller obtained 0.83. As in the case mentioned earlier ([Prescott & Mayhew 92]), learning took a long time, and the achieved performance was far below optimal performance.

Slow learning rates, finding the appropriate critic for the reinforcement learning architecture and determining how to alter controller outputs in order to improve performance are the main problems when implementing reinforcement learning on robots ([Barto 90]). Another "problem that has prevented these architectures from being applied to more complex control tasks has been the inability of reinforcement learning algorithms to deal with limited sensory input. That is, these learning algorithms depend on having complete access to the state of the task environment" ([Whitehead & Ballard 90]). For robotics applications this is unrealistic and extremely limiting, and there are far more simulations of reinforcement learning architectures than there are implementations on real robots. Two examples of robots using reinforcement learning are given below.

Two Examples of Robots Using Reinforcement Learning The mobile robot *OBELIX* [Mahadevan & Connell 91] uses reinforcement learning (Q-learning) to acquire a box-pushing skill. In order to overcome the credit assignment problem[1], the overall task of box-pushing is divided into three subtasks: box-finding, box-pushing and unwedging. These three tasks are implemented as independent behaviours within a subsumption architecture, box-finding being the lowest level and unwedging being the highest level. Obelix has eight ultrasonic sensors and one infrared sensor, in addition to that the robot can monitor the motor supply current (which gives an indication whether the robot is pushing against a fixed obstacle). Instead of using the raw data, Mahadevan and Connell quantise it into an 18-bit-long vector which is then reduced to nine bits by combining several bits. This 9-bit input vector is used as an input to the Q-learning algorithm. The possible motor actions of Obelix are restricted to five: forward, left turn, right turn, sharp left turn and sharp right turn. Input information to *OBELIX'* learning controller is small, and uses preprocessed range data, where sonar scans are coded into "range bins".

Their experimental results confirm that Q-learning may require a large number of learning steps: After a relatively long training time of 2000 learning steps, the find-box behaviour obtained an average value of reward of 0.16, whereas a hand-coded box-finder obtained ca. 0.25.

The second example is that of the walking robot *GENGHIS* that learns to coordinate its leg movements so that a walking behaviour is achieved ([Maes & Brooks 90]). Unlike [Brooks 86a], who determines the arbitration between behaviours by hand, in *GENGHIS* the "relevance" of a particular behaviour is determined through a statistical learning process. The stronger the correlation between a particular behaviour and positive feedback, the more relevant it is. The more relevant a behaviour is in a particular context, the more likely it is to be invoked. In *GENGHIS'* case positive feedback signals are received from a trailing wheel that serves as a forward motion detector, and nega-

[1] How does one correctly assign credit or blame to an action when its consequences unfold over time and interact with the consequences of other actions ([Barto 90])?

tive feedback is received from two switches mounted on the bottom of the robot (the switches detect when the robot is not lifted from the ground).

The speed of learning is strongly influenced by the size of the search space (in this case the space of possible motor actions), that is the amount of search that is required before the first positive reinforcement is obtained. [Kaelbling 90] writes that in experiments with the mobile robot *SPANKY*, whose task it was to move towards a light source, the robot only learned to do this successfully if it was helped in the beginning, so that some positive feedback was received. Similarly, in *GENGHIS'* case the search space is small enough to give the robot positive feedback at an early stage. This is equally true for the robot learning example given later in case study 1.

Reinforcement Learning Architectures

Q-Learning In many learning applications the goal is to establish a control policy that maps a *discrete* input space onto a *discrete* output space such that maximum cumulative reinforcement (reward) is obtained. Q-learning is one mechanism that is applicable to such learning situations.

Assuming that we have a set of discrete states $s \in S$ ("state" here refers to a specific constellation of relevant parameters that affect the robot's operation, e.g. sensor readings, battery charge, physical location, etc., with S being the finite set of all possible states), and a discrete set of actions $a \in A$ (A is the finite space of all possible actions) that the robot might perform ("action" here refers to possible responses of the robot, e.g. movement, acoustic output, visual output, etc), it can be shown that Q-learning converges to an optimal control procedure ([Watkins 89]).

The basic idea behind Q-learning is that the learning algorithm learns the optimal evaluation function over the entire state-action space $S \times A$. The Q function provides a mapping of the form $Q : S \times A \rightarrow V$, where V is the value, the "future reward" of performing action a in state s. Provided the optimal Q function is learned, and provided the partitioning of action space and robot state space does not introduce artefacts or omits relevant information, the robot then knows precisely which action will yield the highest future reward in a particular situation s.

The function $Q(s, a)$ of the expected future reward, obtained after taking action a in state s is learned through trial and error according to equation 4.3.

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \beta(r + \lambda E(s) - Q_t(s,a) \;, \qquad (4.3)$$

where β is the learning rate, r is the reward or punishment resulting from the action a taken in state s, λ is a discount factor ($0 < \lambda < 1$) which reduces the influence of expected future rewards, and $E(s) = max(Q(s,a))$ the utility of state s resulting from action a, using the Q function that has been learned so far.

One example of the application of Q-learning to robot learning has been given above (*OBELIX*). Further examples can be found in [Arkin 98].

Adaptive Heuristic Critic One fundamental problem of reinforcement learning is the *temporal credit assignment problem*: because reinforcement is only received once the final goal state has been reached, it is hard to apportion credit to the actions that preceded the final, successful action.

The way this problem is often addressed is by learning an internal evaluation function which predicts the long-term reward of taking action a in state s. Systems that learn such internal evaluation function are called *adaptive critics* ([Sutton 91]).

Temporal Difference Learning One major drawback of Q-learning is that actions taken early on in the exploration process do not contribute anything towards the learning process – only when finally, by chance, a goal state is reached can the learning rule be applied. This makes Q-learning so time-consuming.

One way of dealing with this problem would be to make predictions of the outcome of actions as the robot explores its world, and to estimate the value of each action a in state s through these predictions. This kind of learning is called *Temporal Difference Learning* or *TD Learning*.

In effect, the learning system predicts reward, and uses this predicted reward for the learning system, until actual external reward is received. A discussion of this method can be found in [Sutton 88].

Further Reading on Reinforcement Learning

- Andrew Barto, *Reinforcement Learning* and *Reinforcement Learning in Motor Control*, in [Arbib 95, pp. 804-813].
- [Ballard 97, ch. 11].
- [Mitchell 97, ch. 13].
- [Kaelbling 90].

4.2.2 Probabilistic Reasoning

We discussed earlier the difference between the machine learning problem and the robot learning problem: the former attempts to find an optimal function with regard to some reward criterion, that maps a fully known input state on to a fully known goal state, while the latter has the additional complication that the true state of the world is unknown.

Uncertainty is a major issue in the interaction between a robot and the environment it is operating in. Sensor signals do not signify the presence of objects, for instance. They merely indicate that the environment is such that a sonar pulse returns to the robot after such and such a time, which indicates with some probability that there is an object out there, reflecting sonar bursts.

Likewise, a robot never knows precisely where it is (the problem of localisation is discussed in detail in chapter 5), and can only estimate its current position with some probability.

In some situations these probabilities are known, or can be estimated to sufficient accuracy to allow mathematical modelling. Markov processes are the most widely used instantiation of such probabilistic models.

Markov Processes A Markov process is a sequence of (possibly dependent) random variables $(x_1, x_2, \ldots x_n)$ with the property that any prediction of x_n may be based on the knowledge of x_{n-1} alone. In other words, any future value of a variable depends only on the current value of the variable, and not on the sequence of past values.

An example is the Markov process shown in figure 4.3, which determines whether a bit string has odd or even parity, i.e. whether it contains an odd or an even number of '1's.

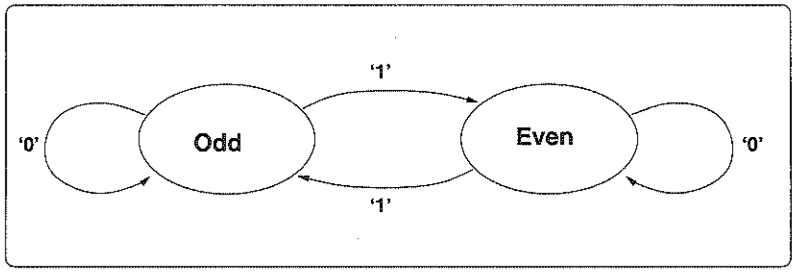

FIG. 4.3. A MARKOV PROCESS TO DETERMINE ODD OR EVEN PARITY OF A BIT STRING

If you start in position 'even' before processing the first bit, and then follow the transitions between the two states according to whether the current bit in the bit string is a '0' or a '1', the final state you end up in indicates whether the bit string has odd or even parity. All transitions are merely dependent on the currently processed bit, not on the sequence of previously processed bits. There is no counting of '1's at all.

Markov Decision Processes One can expand the definition of a Markov process (which was dependent only on the current state s and the transition process a — in the parity example the fact whether a '0' or a '1' was being processed) by adding a state transition model of the environment, and a reward function that assesses the agent's performance.

A Markov decision process is defined by a tuple $< S, A, T, R >$, where S is a finite set of system states, A a finite set of actions, T a state transition model that maps state-action pairs onto a probability distribution over S (indicating the probability of reaching state s' if action a is performed in state s), and R a reward function that specifies the reward the robot receives for taking an action $a \in A$ in state $s \in S$. As this is a Markovian process, knowledge of s and a is sufficient to determine the new state s', and to compute the reward r obtained by moving to state s'.

The value $V(s)$ is the expected sum of future rewards, discounted by a discount factor $0 < \lambda < 1$ that increases as the expected rewards are further away in the future.

It is possible to determine the optimal policy with respect to V in a Markov decision process, and therefore to determine how the agent should act in order to obtain that maximal value V ([Puterman 94]).

Partially Observable Markov Decision Processes A common problem in robotic applications is that usually the state is not fully observable, i.e. it is not always known what the current system state is. In this case, a model of observations must be added. This model specifies the probability of making observation o after having taken action a in state s.

The *belief state* then is a probability distribution over S (the set of all possible states), representing for each state $s \in S$ the belief that the robot is currently in state s.

The procedure for the Markov decision process discussed above can now be modified to partially observable environments, by estimating the robot's current state s, and by applying a policy that maps belief states onto actions. Again, the objective is to determine the policy that will maximise the discounted future reward.

Further Reading on Markov Decision Processes

- Leslie Pack Kaelbling, Michael Littman and Anthony Cassandra, Planning and Acting in Partially Observable Stochastic Domains, *Artificial Intelligence*, Vol. 101, 1998.

4.2.3 Connectionism

Connectionist computing architectures (also called "artificial neural networks") are mathematical algorithms that are able to learn mappings between input and output states through supervised learning, or to cluster incoming information in an unsupervised manner. Their characteristic feature is that many independent processing units work simultaneously, and that the overall behaviour of the network is not caused by any one component of the architecture, but is emergent from the concurrent working of all units. Because of their ability to learn mappings between an input and an output space, to generalise incoming data, to interpret (cluster) input information without supervision (i.e. without teaching signal), their resistance to noise and their robustness (the term *graceful degradation* describes the fact that the performance of such networks is not solely dependent on the individual unit — losing one unit will merely mean a degradation, not a total loss of performance), connectionist computing architectures can be used well in robotics. [Torras 91] gives an overview of (largely simulation) work that has been done in the field: supervised learning schemes have been applied to the generation of sequences, both supervised and unsupervised learning schemes have been used to learn non-linear mappings such as inverse kinematics, inverse dynamics and sensorimotor integration, and reinforcement learning has largely been used for tasks involving optimisation, such as path planning.

Artificial neural networks can be used in a number of applications in mobile robotics. For instance, they can be used to learn associations between input signals (e.g. sensor signals) and output signals (e.g. motor responses). Case studies 1 and 2 (p. 69 and p. 74) are examples of such applications.

They can also be used to determine the underlying (unknown) structure of data, which is useful to develop internal representations, or for data compression applications. In case study 3 (p. 80) *FortyTwo* uses a self-organising artificial neural network to determine the underlying structure of visual perceptions of its environments, and applies this structure to detecting target objects in the image.

The following sections and case studies give a brief overview of common artificial neural networks, and their applications to mobile robotics.

McCulloch and Pitts Neurons The inspiration for artificial neural networks is given by biological neurons, which perform complicated tasks such as pattern recognition, learning, focusing attention, motion control, etc. extremely reliably and robustly. A simplified biological neuron — the model for the artificial neuron — consists of a cell body, the *soma* which performs the computation, a number of inputs (the *dendrites*), and one or more outputs (the *axon*) which connect to other neurons. Signals in the simplified biological neuron model are encoded in electric spikes, whose frequency encodes the signal.

The connections between dendrites and soma, the *synapses*, are modifiable by so-called neurotransmitters, meaning that incoming signals can be amplified, or attenuated.

The simplified model assumes that the firing rate (i.e. the frequency of output spikes) of a neuron is proportional to the neuron's activity. In artificial neural networks, the output of the artificial neuron is sometimes kept analogue, sometimes thresholded to produce binary output. This is dependent on the application.

In 1943 McCulloch and Pitts proposed a simple computational model of biological neurons. In this model, the input spikes of biological neurons were replaced by a continuous, single-valued input signal, the "chemical encoding" of synaptic strength was replaced by a multiplicative weight, the threshold function of a biological neuron was modelled using a comparator, and the spiking output signal of a biological neuron was replaced by a binary value.

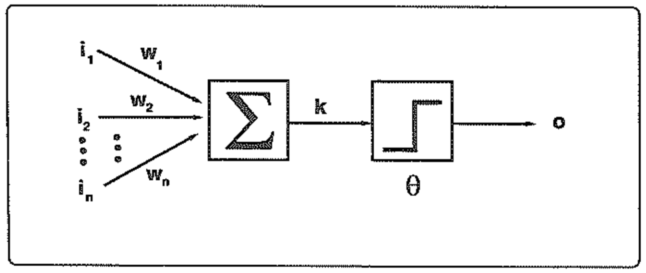

FIG. 4.4. THE MCCULLOCH AND PITTS NEURON

The McCulloch and Pitts neuron is shown in figure 4.4. Its functionality is as follows. The neuron computes the weighted sum k of all n inputs $\imath$, according to equation 4.4.

$$k = \sum_{j=1}^{n} \imath_j w_j \tag{4.4}$$

This weighted sum is then compared with a fixed threshold Θ to produce the final output o. If k exceeds Θ, the neuron is "on" (usually defined as '$o = 1$'), if k is below the threshold, the neuron is "off" (usually defined as either '$o = 0$' or '$o = -1$').

McCulloch and Pitts ([McCulloch & Pitts 43]) proved that, given suitably chosen weights, a synchronous assembly of such simple neurons is capable in principle of universal computation — any computable function can be implemented using McCulloch and Pitts neurons. The problem, of course, is to choose the weights "suitably". How this can be achieved is discussed later in this section.

Example: Obstacle Avoidance Using McCulloch and Pitts Neurons A robot as shown in figure 4.5 is to avoid obstacles when one or both of the whiskers trigger, and move forward otherwise. Whiskers LW and RW signal '1' when they are triggered, '0' otherwise. The motors LM and RM move forward when they receive a '1' signal, and backwards when they receive a '-1' signal.

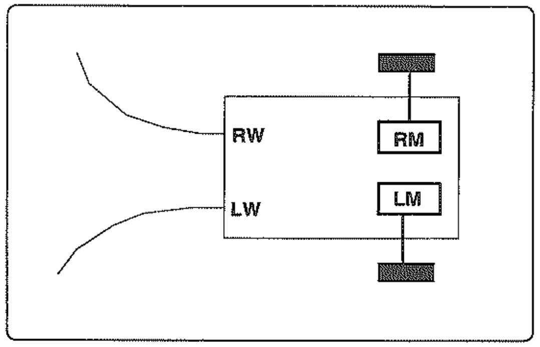

FIG. 4.5. A SIMPLE VEHICLE

The truth table for this obstacle avoidance behaviour is shown in table 4.1.

We can implement this function using one McCulloch and Pitts neuron for each motor, using neurons whose output is either "-1" or "+1". In this example, we will determine the necessary weights w_{RW} and w_{LW} for the *left* motor neuron only. The weights for the right neuron are determined in a similar fashion. We choose a threshold Θ of just below zero, -0.01 say.

The first line of truth table 4.1 stipulates that both motor neurons must be '+1' if neither LW nor RW fire. As we have chosen a threshold of $\Theta = -0.01$ this is fulfilled.

LW	RW	LM	RM
0	0	1	1
0	1	-1	1
1	0	1	-1
1	1	don't care	don't care

Table 4.1. TRUTH TABLE FOR OBSTACLE AVOIDANCE

Line two of table 4.1 indicates that w_{RW} must be smaller than Θ for the left motor neuron. We choose, say, $w_{RW} = -0.3$.

Line three of the truth table, then, indicates that w_{LW} must be greater than Θ. We choose, say, $w_{LW} = 0.3$.

As a quick check with table 4.1 shows, these weights already implement the obstacle avoidance function for the left motor neuron! The functioning network is shown in figure 4.6.

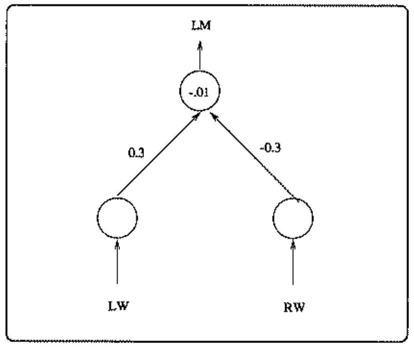

FIG. 4.6. LEFT-MOTOR NODE FOR OBSTACLE AVOIDANCE

The way we determined the weights here was by "common sense". Looking at the relatively simple truth table of the obstacle avoidance function, it is a straightforward exercise to determine weights that will give the desired output function.

However, for more complicated functions, determining weights by "common sense" is very hard, and it is desirable to have a learning mechanism that would determine those required weights automatically. There is a second reason why we want such a learning rule: it would allow us to build robots that learn. The *Perceptron* is a network consisting of McCulloch and Pitts neurons that fulfils this requirement.

Exercise 2: Full Obstacle Avoidance Using McCulloch and Pitts Neurons What will the robot do if *both* whiskers are touched simultaneously? Which artificial neural network, based on McCulloch and Pitts neurons, would implement the function indicated in truth table 4.1, *and* make the robot move backwards if both whiskers are touched? The answer is given in appendix 2.1 on page 218.

Perceptron and Pattern Associator The Perceptron ([Rosenblatt 62]) is a "single-layer" artificial neural network that is easy to implement, low in computational cost and fast in learning. It consists of two layers of units: the input layer (which simply passes signals on) and the output layer of McCulloch and Pitts neurons (which performs the actual computation, hence "single layer network" — see figure 4.7, right).

The function of input and output units is as follows: input units simply pass the received input signals $\vec{\imath}$ on to all output units, the output o_j of output unit j is determined by

$$o_j = f(\sum_{k=1}^{M} w_{jk} \imath_k) = f(\vec{w_j} \cdot \vec{\imath}), \tag{4.5}$$

where $\vec{w_j}$ is the individual weight vector of output unit j, M the number of input units, and f the so-called transfer function.

The transfer function f of the Perceptron is defined as a step function:

$$f(x) = \begin{cases} 1 & \forall\, x > \Theta \\ 0\,(or - 1) & \text{else.} \end{cases}$$

Θ again is a threshold value.

The Pattern Associator (see figure 4.7, left) is a variant of the Perceptron, with the only difference that here $f(x) = x$. The main difference between Perceptron and Pattern Associator, therefore, is that the Perceptron generates binary output, while the Pattern Associator's output is continuous-valued. Otherwise, these two networks are very similar ([Kohonen 88] and [Rumelhart & McClelland 86]).

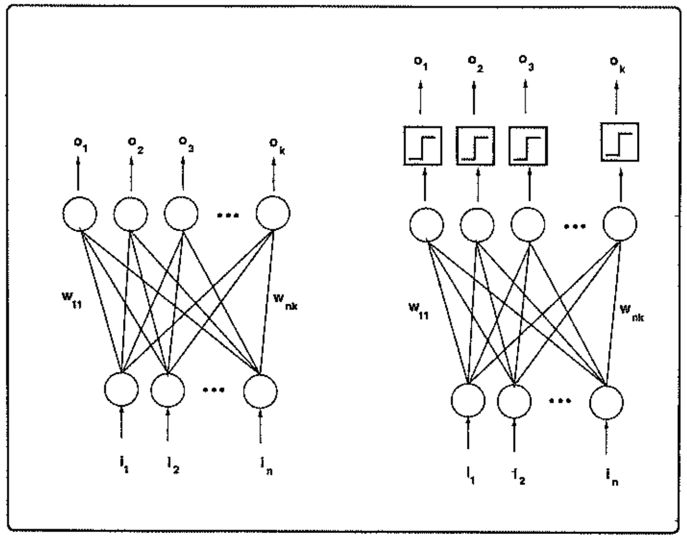

FIG. 4.7. PATTERN ASSOCIATOR (LEFT) AND PERCEPTRON (RIGHT)

The Perceptron Learning Rule In the case of the McCulloch and Pitts neuron, we determined the suitable weights by applying our common sense. For complex networks and complex input-output mappings, this method is not suitable. Furthermore, we would like to have a learning rule so that we can use it for autonomous robot learning.

The rule for determining the necessary weights is very simple[2], it is given in equation 4.6.

$$\Delta\vec{w}_k(t) = \eta(t)(\tau_k - o_k)\vec{\imath}, \tag{4.6}$$

$$\vec{w}_k(t+1) = \vec{w}_k(t) + \Delta\vec{w}_k, \tag{4.7}$$

with τ_k being the target value for unit k, i.e. the desired output of output unit k, and o_k the actually obtained output of unit k. The speed of learning is determined by the learning rate $\eta(t)$. A big η (for example 0.8) will result in a network that adjusts very quickly to changes, but which will also be "neurotic" (it will forget all it has learned and learn something new as soon as a couple of freak signals occur). A small η (for example 0.1), on the other hand, will result in a "lethargic" network that takes a long time before it learns a function. The learning rate η is usually chosen to be constant, but may be variable over time.

Example: Obstacle Avoidance Using a Perceptron We'll consider the the same example we have considered before: obstacle avoidance (see p. 57). The difference this time is that we use a Perceptron, and that we *determine* the required weights, using the Perceptron learning rule given in equations 4.6 and 4.7.

Let η be 0.3, and Θ again just below zero, say -0.01. The two weights of the left-motor node are also zero to start with. This initial configuration is shown in figure 4.8.

We now go line by line through the truth table 4.1, applying equations 4.6 and 4.7.

Line one of the truth table yields:

$$\begin{aligned} w_{LWLM} &= 0 + 0.3\,(1-1)\,0 = 0 \\ w_{LWRM} &= 0 + 0.3\,(1-1)\,0 = 0 \end{aligned}$$

Likewise, the other two weights also remain zero.

Line two of table 4.1 results in the following updates:

$$\begin{aligned} w_{LWLM} &= 0 + 0.3\,(-1-1)\,0 = 0 \\ w_{LWRM} &= 0 + 0.3\,(1-1)\,1 = 0 \\ w_{RWLM} &= 0 + 0.3\,(-1-1)\,1 = -0.6 \\ w_{RWRM} &= 0 + 0.3\,(1-1)\,1 = 0 \end{aligned}$$

Line three of the truth table gives:

[2] The derivation of this rule can be found in [Hertz *et al.* 91].

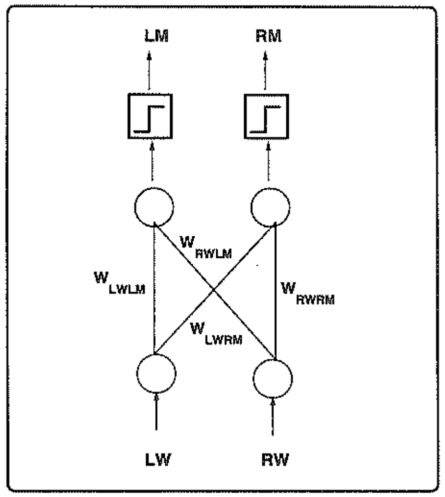

FIG. 4.8. Layout of a Perceptron for Obstacle Avoidance

$$\begin{aligned} w_{LWLM} &= 0 + 0.3\,(1-1)\,1 = 0 \\ w_{LWRM} &= 0 + 0.3\,(-1-1)\,1 = -0.6 \\ w_{RWLM} &= -0.6 + 0.3\,(1-1)\,0 = -0.6 \\ w_{RWRM} &= 0 + 0.3\,(-1-1)\,0 = 0 \end{aligned}$$

A quick calculation shows that this network already performs the obstacle avoidance function stipulated by table 4.1 perfectly! The final network is shown in figure 4.9. It is essentially the same network as the one obtained by hand earlier (shown in figure 4.6).

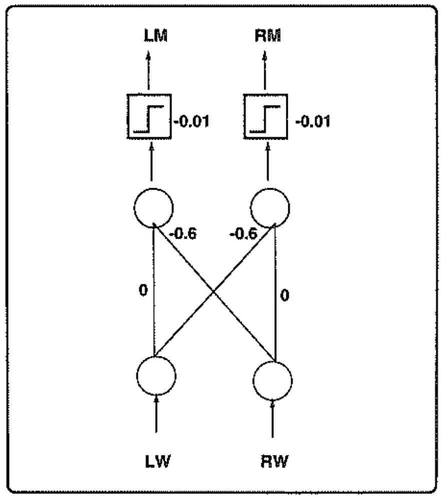

FIG. 4.9. Complete Perceptron for Obstacle Avoidance

Limitations of the Perceptron Consider a network like the one shown in figure 4.6, and the truth table 4.2 (the exclusive-or function).

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.2. EXCLUSIVE OR (XOR) FUNCTION

If the network shown in figure 4.6 was to perform this function correctly, the following inequalities would have to be true:

$$w_{LW} > \Theta$$
$$w_{RW} > \Theta$$
$$w_{LW} + w_{RW} < \Theta.$$

The first two expressions add up to $w_{LW} + w_{RW} > 2\,\Theta$, which contradicts the third inequality. It can't be done. In general Perceptrons are unable to learn functions that are not linearly separable, i.e. functions where two classes cannot be separated by a line, plane or hyperplane in the general case.

This means, of course, that a robot learning by using a Perceptron or a Pattern Associator can only learn functions that are linearly separable. Here is an example of a function the robot could not learn using a Perceptron: suppose we wanted the robot to escape from a dead end by turning left whenever either of two front whiskers is on, and by reversing whenever both whiskers are on simultaneously. For the 'turn-left output node' of the Perceptron this means that it has to be on if either of the two whiskers fires, and to be off in the other two cases. This is the *exclusive or* function, a function that cannot be separated linearly and is therefore unlearnable by a Perceptron output node.

Fortunately, many functions robots have to learn are linearly separable, which means that the very fast learning Perceptron can be used for robot learning (an example is given in case study 1 in section 4.4.1).

In fact, its speed is the major advantage of the Perceptron over networks such as the Multilayer Perceptron or Backpropagation Network (p. 63). A very small number of teaching experiences suffices to produce the correct associations between stimulus and response. A backpropagation network may typically require several hundred teaching experiences before a function is learned. Re-learning (for example when adjusting to new circumstances) then again takes typically several hundred training steps whereas the pattern associator re-learns as quickly as it learned in the first place. This property is important in robotics — certain competences, such as for example obstacle avoidance, have to be learned very quickly, because the robot's ability to stay operational crucially depends on them.

Further Reading on Perceptrons

- [Beale & Jackson 90, pp. 48-53].
- [Hertz *et al.* 91, ch. 5].

Multilayer Perceptron So far, we have seen that a network consisting of McCulloch and Pitts neurons is capable of universal computation, provided one knows how to set the weights suitably. We have also seen that a single layer network consisting of McCulloch and Pitts neurons, the Perceptron, can be trained by the Perceptron learning rule. However, we also realised that the Perceptron can only learn linearly separable functions.

The explanation for this fact is that each layer of a network consisting of McCulloch and Pitts neurons establishes one hyperplane to separate the two classes the network has to learn (the '1's and the '0's). If the separation between the two classes cannot be achieved with one hyperplane — as is the case in the XOR problem — a single layer network will not be able to learn the function.

A Perceptron with not only one layer of output units, but with one or more additional layers of hidden units (see figure 4.10), however, might be able to do just that! Indeed, it can be shown that the Multilayer Perceptron can implement any computable function. The problem is, as before, to determine the suitable weights.

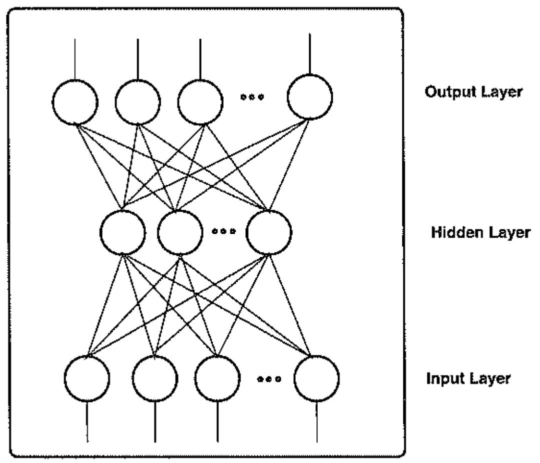

FIG. 4.10. MULTI-LAYER PERCEPTRON

The so-called *backpropagation algorithm* (so called because the update rule uses the backpropagation of output error to determine the required changes in the network's weights) can be used to determine these weights ([Rumelhart *et al.* 86] and [Hertz *et al.* 91]).

To begin with, all network weights are initialised to random values. The thresholds Θ are replaced by weights to an input that is always set to $+1$. This makes threshold updates a part of weight updates, which is computationally simpler.

Once the network is initialised, training commences by presenting input-target (desired output) pairs to the network. These training patterns are then used to adapt the weights.

The output o_j of each unit j in the network is now computed according to equation 4.8.

$$o_j = f(\vec{w}_j \cdot \vec{i}), \qquad (4.8)$$

where $\vec{w}$ is the weight vector of that unit j, and $\vec{i}$ is the input vector to unit j. The function f is the so-called activation function. In the case of the Perceptron, this was a binary threshold function, but in the case of the Multilayer Perceptron it has to be a differentiable function, and is usually chosen to be the sigmoid function given in equation 4.9.

$$f(z) = \frac{1}{1 + e^{-kz}}, \qquad (4.9)$$

where k is a positive constant that controls the slope of the sigmoid. For $k \to \infty$ the sigmoid function becomes the binary threshold function that was used earlier in the McCulloch and Pitts neurons.

Now that the outputs of all units — hidden units and output layer units — are computed, the network is trained. All weights w_{ij} from unit i to unit j are trained according to equation 4.10.

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_{pj} o_{pi}, \qquad (4.10)$$

where η is the learning rate (usually a value of around 0.3), δ_{pj} is the error signal for unit j (for output units, this is given by equation 4.11, for hidden units it is given by equation 4.12); o_{pi} is the input to unit j, coming from unit p.

Error signals are determined for output units first, then for the hidden units. Consequently, training starts with the output layer of the net, and then proceeds backwards through the hidden layer.

For each unit j of the output layer, the error signal δ_{pj}^{out} is determined by equation 4.11.

$$\delta_{pj}^{out} = (t_{pj} - o_{pj}) o_{pj} (1 - o_{pj}), \qquad (4.11)$$

where t_{pj} is the target signal (the desired output) for the output unit being updated, and o_{pj} the output actually obtained from the output unit being updated.

Once the error signals for the output units have been determined, the penultimate layer of the network — the final hidden layer — is updated by propagating the output errors backwards, according to equation 4.12.

$$\delta_{pj}^{hid} = o_{pj}(1 - o_{pj}) \sum_k \delta_{pk} w_{kj}, \qquad (4.12)$$

where o_{pj} is the output of the hidden layer unit currently being updated, δ_{pk} the error of unit k in the subsequent layer of the network, and w_{kj} the weight between hidden unit j and the subsequent unit k on the next higher layer.

This training process is repeated until, for example, the output error of the network drops below a user-defined threshold. For a detailed discussion of the Multilayer Perceptron see, for instance, [Rumelhart *et al.* 86], [Hertz *et al.* 91] [Beale & Jackson 90] and [Bishop 95].

Advantages and Disadvantages The Multilayer Perceptron can be used to learn non-linearly separable functions, and thus overcomes the problems of the Perceptron. The price to pay for this is, however, that learning usually is a lot slower. Whereas the Perceptron learns within a few learning steps, the Multilayer Perceptron typically requires several hundred learning steps to learn the desired input-output mapping. This is a problem for robotics applications, not so much for the computational cost, but for the fact that a robot would have to repeat the same sort of mistake hundreds of times, before it learns to avoid it[3]. For fundamental sensor-motor competences such as obstacle avoidance, this is usually not acceptable.

Further Reading on Multilayer Perceptrons

- [Rumelhart & McClelland 86, ch. 8].
- [Hertz *et al.* 91, pp.115-120].

Radial Basis Function Networks Like the Multilayer Perceptron (MLP), the Radial Basis Function Network (RBF net) can learn non-linearly separable functions. It is a two-layer network, in which the hidden layer performs a non-linear mapping based on radial basis functions, whilst the output layer performs a linear weighted summation on the output of the hidden layer (as in the Pattern Associator). The fundamental mechanisms of RBF net and MLP are similar, but whilst the MLP partitions the input space using linear functions (hyperplanes), the RBF net uses nonlinear functions (hyperellipsoids). Figure 4.11 shows the general structure of a radial basis function network ([Lowe & Tipping 96]).

The hidden units have a Gaussian capture region which serves to identify similarities between the current input vector and a hidden unit's weight vector — each weight vector is a prototype of one specific input signal. The hidden layer performs a nonlinear mapping of the input space, which increases the probability that classes can be separated linearly.

The output layer then associates the classification of the hidden layer with the target output signal through linear mapping, as in the Perceptron and the MLP.

The output $o_{hid,j}$ of RBF unit j in the hidden layer is determined by equation 4.13:

$$o_{hid,j} = exp(-\sum_j \frac{||\vec{i} - \vec{w}_j||}{\sigma}), \qquad (4.13)$$

[3] To use once perceived sensor patterns for repeated training is problematic, because of the high number of freak perceptions obtained with robot sensors.

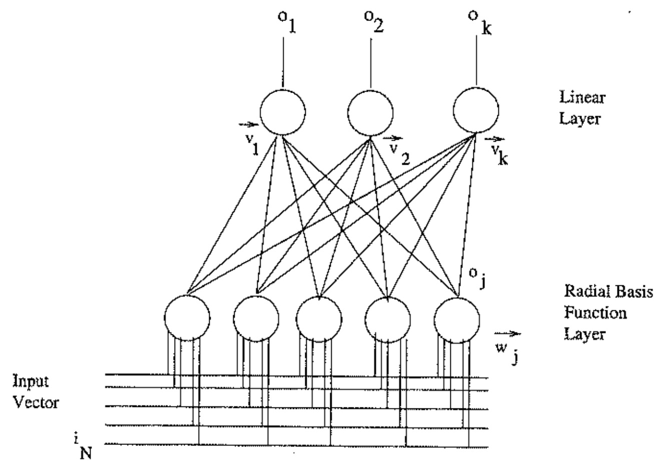

FIG. 4.11. RADIAL BASIS FUNCTION NETWORK

with $\vec{w}_j$ being the weight vector of RBF unit j, $\vec{\imath}$ being the input vector, and σ being the parameter controlling the width of the bell curved capture region of the radial basis function (e.g. $\sigma = 0.05$).

The output o_k of each unit of the output layer is determined by equation 4.14.

$$o_k = \vec{o}_{hid} \cdot \vec{v}_k, \tag{4.14}$$

where $\vec{o}_{hid}$ is the output of the hidden layer, and $\vec{v}_k$ is the weight vector of output unit k.

Training of the output layer is simple, and achieved by applying the Perceptron learning rule given in equations 4.6 and 4.7.

The RBF network works by mapping the input space onto a higher dimensional space through using a nonlinear function, for example the Gaussian function described above. The weights of the hidden layer, therefore, have to be chosen such that the entire input space is represented as evenly as possible. There are several methods for determining the weights of the units in the hidden layer. One simple method is to spread the cluster centres evenly over the input space ([Moody & Darken 89]). Alternatively, it is sometimes sufficient to position the cluster centres at randomly selected points of the input space. This will ensure that RBF cluster density is high where input space density is high, and low where the input density is low ([Broomhead & Lowe 88]). Finally, the weights of the hidden layer can be determined by a clustering mechanism such as the one used in the self-organising feature map (see next section).

Further Reading on Radial Basis Function Networks

- David Lowe, *Radial Basis Function Networks*, in [Arbib 95, pp. 779-782].
- [Bishop 95, ch. 5].

The Self-Organising Feature Map All artificial neural networks discussed so far are trained by "supervised training": learning is achieved by using a target value, i.e. the desired output of the net. This target value is supplied externally, from a "supervisor" (which could be another piece of code, as in case study 1 on p. 69, or a human, as in case study 2 on p. 74).

However, there are applications where no training signal is available, for example all applications that have to do with clustering some input space. It is often useful in robotics to cluster a high dimensional input space and to map it automatically — in an unsupervised manner — onto a lower dimensional output space. This dimensionality reduction is a form of generalisation, reducing the complexity of the input space whilst, hopefully, retaining all the "relevant" features in the input space. Case study 3 (p. 80) gives an example of the application of unsupervised learning in mobile robotics.

The self-organising feature map (SOFM), or Kohonen network, is one mechanism that performs an unsupervised mapping of a high dimensional input space onto a (typically) two-dimensional output space ([Kohonen 88]).

The SOFM normally consists of a two-dimensional grid of units, as shown in figure 4.12.

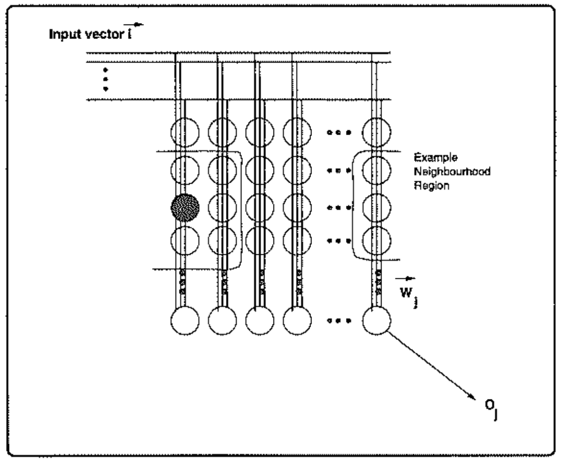

FIG. 4.12. SELF-ORGANISING FEATURE MAP

All units receive the same input vector $\vec{\imath}$. Initially, the weight vectors $\vec{w}_j$ are initialised randomly and normalised to unit length.

The output o_j of each unit j of the net is determined by equation 4.15.

$$o_j = \vec{w}_j \cdot \vec{\imath}. \tag{4.15}$$

Because of the random initialisation of the weight vectors, the outputs of all units will differ from one another, and one unit will respond most strongly to a particular input vector. This "winning unit" and its surrounding units will then be trained so that they respond even more strongly to that particular input vector, by applying the update rule of equation 4.16. After having been updated, weight vectors $\vec{w}_j$ are normalised again.

$$\vec{w}_j(t+1) = \vec{w}_j(t) + \eta(\vec{\imath} - \vec{w}_j(t)), \tag{4.16}$$

where η is the learning rate (usually a value around $\eta = 0.3$). The neighbourhood around the winning unit, within which units get updated, is usually chosen to be large in the early stages of the training process, and to become smaller as training progresses. Figure 4.12 shows an example neighbourhood of one unit around the winning unit (drawn in black). The figure also shows that the network is usually chosen to be torus-shaped, to avoid border effects at the edges of the network.

As training progresses, certain areas of the SOFM become more and more responsive to certain input stimuli, thus clustering the input space onto a two-dimensional output space. This clustering happens in a topological manner, mapping similar inputs onto neighbouring regions of the net. Example responses of trained SOFMs are shown in figures 4.19, 4.23 and 5.19.

Application of SOFMs to Robotics Generally, SOFMs can be used to cluster an input space to obtain a more meaningful, abstracted representation of that input space.

A typical application is to cluster a robot's sensory input space. Similarities between perceptions can be detected using SOFMs, and the abstracted representation can be used to encode policies, i.e. the robot's response to a particular perception. Case study 6 (section 5.4.3) gives one example of how SOFMs can be used for robot route learning.

Further Reading on Self-Organising Feature Maps

- [Kohonen 88, ch. 5].
- Helge Ritter, *Self-Organizing Feature Maps: Kohonen Maps*, in [Arbib 95, pp. 846-851].

4.3 Further Reading on Learning Methods

On Machine Learning

- Tom Mitchell, *Machine Learning*, McGraw Hill, New York, 1997.
- Dana Ballard, *An Introduction to Natural Computation*, MIT Press, Cambridge MA, 1997.

On Connectionism

- John Hertz, Anders Krogh, Richard G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City CA, 1991.
- R. Beale and T. Jackson, *Neural Computing: An Introduction*, Adam Hilger, Bristol, Philadelphia and New York, 1990.
- Christopher Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford, 1995.
- Simon Haykin, *Neural Networks : a Comprehensive Foundation*, Macmillan, New York, 1994.

4.4 Case Studies of Learning Robots

4.4.1 Case Study 1. *ALDER*: Self-Supervised Learning of Sensor-Motor Couplings

Having discussed the problem of robot learning in general, and the machine learning mechanisms that can be used to achieve competence acquisition in robots, we will now take a closer look at specific examples of mobile robots that learn to interpret their sensory perceptions to accomplish particular tasks.

The first case study presents a self-organising controller architecture which enables mobile robots to learn through trial and error, in a self-supervised learning process that requires no human intervention. The first experiments were conducted in 1989 ([Nehmzow *et al.* 89]), using the robots *ALDER* and *CAIRNGORM* (see figure 4.14), but the mechanism has since been used in many robots internationally to achieve autonomous acquisition of sensor motor competences (see, for instance, [Daskalakis 91] and [Ramakers 93]).

The fundamental idea of the controller is that a Pattern Associator (see section 4.2.3) associates sensory perception with motor action. Because this association is *acquired* through a learning process, rather than being pre-installed, the robot can change its behaviour and adapt to changing circumstances, if the appropriate training mechanism is provided. In *ALDER's* and *CAIRNGORM's* case performance feedback is received using so-called *instinct rules*. These instinct rules specify sensor states that must be maintained (or avoided) throughout the entire operation of the robot: behaviour is thus expressed in the form of sensor states, and the robot learns the appropriate behaviour in order to maintain (or avoid) the required states.

Figure 4.13 shows the general structure of the entire controller used in all experiments discussed in this section. The controller consists of fixed and plastic components, fixed components being the so-called instinct-rules, the robot morphology and various parameters within the controller; the plastic component being the Pattern Associator.

Instinct Rules As the Pattern Associator is trained under supervised learning, a target signal — the desired response to a particular input — must be provided. As we want the robot to learn without human intervention, an independent method has to be devised to obtain these target signals.

We use fixed rules for this purpose which we call *instinct-rules*. They are similar, but not identical to *instincts* as defined by [Webster 81]:

> "[An instinct is a] complex and specific response on the part of an organism to environmental stimuli that is largely hereditary and unalterable though the pattern through which it is expressed may be modified by learning, that does not involve reason, and that has as its goal the removal of a somatic tension or excitation".

This describes behaviour and is therefore different to the instinct-rules used in the experiments described here, as instinct-rules are not behaviour, but constants

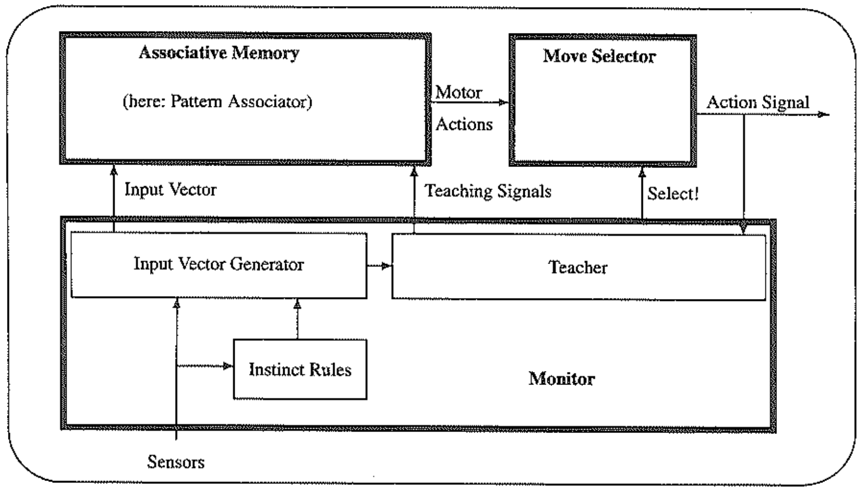

FIG. 4.13. COMPUTATIONAL STRUCTURE OF THE SELF-ORGANISING CONTROLLER

(sensor states) that guide the learning of behaviour. The goal of instinct and instinct-rules, however, is the same: the removal of a somatic tension (in the case of the robot such a "somatic tension" is an external sensor stimulus, or, in some experiments, the lack of it).

Each instinct-rule has a dedicated sensor in order that it can be established whether it is violated or not. This sensor can be a physical, external sensor (for example a whisker), or an internal sensor (for example a clock that is reset every time some external sensor stimulus is received).

To sum up, the instinct-rules are used to generate a reinforcement signal (the signal being the fulfilment of a previously violated instinct-rule), they do not indicate correct behaviour.

Input and Output Current and previous sensor signals constitute the input signals to the Pattern Associator. Typically, raw sensor signals are used, but sometimes it is useful to apply preprocessing to the sensor signals. Information about violated instinct-rules could similarly be used.

The output of the network denotes motor actions of the robot. Examples for such motor actions are swift left turn (i.e. right motor moving forward while left motor is moving backward), swift right turn, forward or backward movement. An alternative to using such "compound" motor actions is to use artificial motor neurons with analogue outputs to drive a differential drive system. Experiments of this kind are described in [Nehmzow 99c].

The idea behind this controller setup is that effective associations between sensor signals and motor actions arise over time through the robot's interaction with the environment, without human intervention.

Mechanism As said earlier, the Pattern Associator requires a teaching signal to develop meaningful associations between its input and its output. This teaching signal is provided by the *monitor*, a "critic" that uses the instinct rules to assess the robot's performance and teach the network accordingly: as soon as any of the instinct-rules become violated (i.e. as soon as a specified sensor status is no longer maintained), an input signal is generated by the *input vector generator*, sent to the associative memory and the output of the network is computed (fig. 4.13). The *move selector* determines which output node carries the highest output value and which motor action this output node stands for. That motor action is then performed for a fixed period of time (the actual length of time depends on the speed of the robot). If the violated instinct-rule becomes satisfied within this period of time, the association between original input signal and output signal within the Pattern Associator is taken to be correct and is confirmed to the network (this is done by the monitor). If, on the other hand, the instinct-rule remains violated, a signal is given from the monitor to the move selector to activate the motor action that is associated with the second strongest output node. This action is then performed for a slightly longer period of time than the first one to compensate the action taken earlier; if this motor action leads to satisfaction of the violated instinct-rule, the network will be taught to associate the initial sensor state with this type of motor action; if not, the move selector will activate the next strongest output node. This process continues until a successful move is found. Because the monitor is part of the robot controller, the process of sensor-motor competence acquisition is completely independent from an operator's supervision: the robot acquires its competences autonomously.

Figure 4.7 (left) shows the general structure of the Pattern Associator used. The actual input to the network may vary from experiment to experiment, the output nodes denote motor actions. The result of this process is that effective associations between input stimuli and output signals (motor actions) develop.

Experiments This controller has been implemented on a number of different mobile robots by different research groups. The consistent finding is that it enables mobile robots to acquire fundamental sensor-motor couplings very rapidly, using no more than 20 learning steps (and typically far less than that), taking a few tens of seconds in real time. Due to the ability to autonomously re-map sensory input to actuator output the robots have the ability to maintain task-achieving competences even when changes in the world, the task, or the robot itself occur.

Learning to Move Forward The simplest experiment to begin with is to make a robot learn to move forward. Assuming that the robot has some means of determining whether it is actually moving forwards or not[4], an instinct rule of the form

"Keep the forward motion sensor 'on' at all times"

[4] This can be achieved, for instance, by detecting whether a swivelling caster wheel is aligned with the main axis of the robot or not, using a suitably placed microswitch.

will lead to a mapping between sensory perception and motor actions that will result in the robot moving forward continuously.

Obstacle Avoidance The simple, acquired forward motion behaviour can be expanded to a forward motion and obstacle avoiding behaviour by adding one or more further instinct rules. For example, if we take the case of a simple mobile robot with just two whisker sensors mounted at the front (figure 4.5), the following two instinct rules would lead to that behaviour:

1. "Keep the forward motion sensor 'on' at all times!"
2. "Keep whiskers 'off' at all times!"

For robots with infrared or sonar sensors, the second instinct rule would have to be changed accordingly, for instance to "Keep all sonar readings above a certain threshold", or similar.

Experimental results with many different robots, using different sensor modalities show that the robots learn obstacle avoidance very quickly (less than a minute in most cases), and with very few learning steps (less than 10 in most cases).

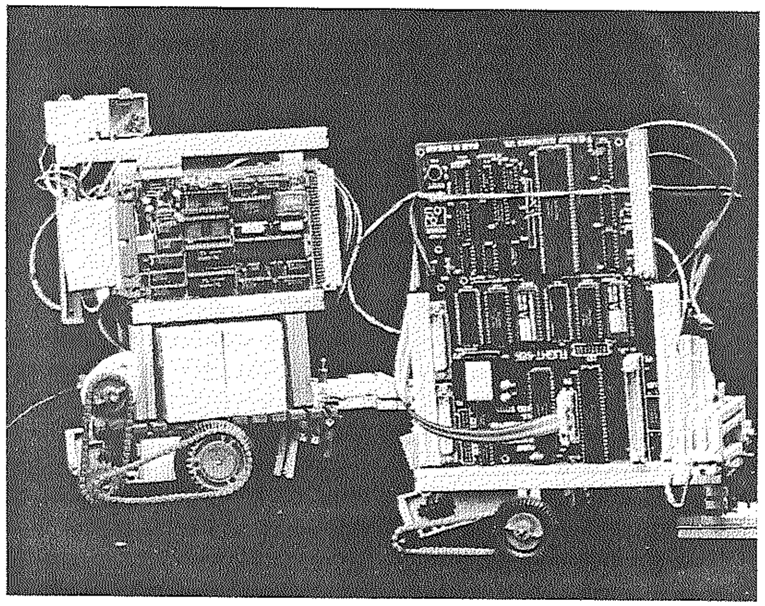

FIG. 4.14. *ALDER* (LEFT) AND *CAIRNGORM* (RIGHT)

Adapting to Changing Circumstances We stated earlier that the robot's ability to cope with unforeseen situations increases through its learning capability. Two experiments we conducted with *ALDER* and *CAIRNGORM* highlight this

([Nehmzow 92]). In the first experiment, the robots were placed in an environment containing convex obstacles (ordinary boxes), and quickly learned to turn away from a touched whisker. When encountering a dead end, however, the robots performed poorly to begin with (by turning back into the dead end, rather than towards the exit). Soon (within two minutes), though, they found the exit and had by then acquired a new behaviour: they were now turning in one and the same direction, regardless of whether the left or the right whisker was touched. This behaviour is best suited for leaving dead ends: the robots had adapted to their new environment.

In a second experiment, the robots' whiskers were physically swapped after they had acquired the obstacle avoidance competence. In between four and six learning steps they adapted the associations between sensor signals and motor actions within the artificial neural network, regained the ability to avoid obstacles and thus "repaired" the (in this case externally induced) fault.

We once found that one of our robots had been used for weeks, successfully learning many tasks, whilst inadvertently one of the eight infrared sensors had not been working due to a loose connection. Because of the redundancy in sensors (more than one sensor covering the area surrounding the robot), the robot had learned to avoid obstacles using the other sensors.

Contour Following By extending the set of instinct rules, the robots can acquire the ability to stay close to contours such as walls. In *ALDER's* and *CAIRNGORM's* case a third instinct-rule was added:

1. "Keep the forward motion sensor 'on' at all times!"
2. "Keep whiskers 'off' at all times!"
3. "Do touch something every four seconds!"

Using these three rules (or rules to the same extent), robots quickly learned to move forward, to steer away from obstacles, but to seek obstacles (i.e. walls) actively every four seconds. The resulting behaviour is that of wall following.

The following experiment demonstrates how quickly the robots can adapt to changing circumstances. When *ALDER* or *CAIRNGORM* were turned by 180° after having successfully learned to follow a wall on, say, their right hand side, they re-mapped sensory input to motor action within three to four learning steps such that they were able to follow a wall on their left hand side. If the robots' direction of motion is changed again, the re-learning process was even faster, because latent associations, acquired earlier, were still present within the network and only needed to be strengthened slightly to become active again.

Corridor Following By adding a fourth instinct rule, using short-term memory this time, the robots can learn to stay in the centre of a corridor by touching left and right walls in turn. The instinct rules for corridor following behaviour are:

1. "Keep the forward motion sensor 'on' at all times!"
2. "Keep whiskers 'off' at all times!"
3. "Do touch something every four seconds!"
4. "The whisker that was touched last time must not be touched this time!"

Phototaxis and Box-Pushing The possibilities of acquiring new competences are merely limited by a robot's sensors, and the fact that not all behaviours can be expressed in simple sensor states alone. The first case study will be concluded by looking at experiments with an IS Robotics R2 robot, which had infrared range sensors and light sensors.

Phototaxis was acquired within less than 10 learning steps, using an instinct-rule stipulating that the light sensors mounted at the front of the robot must return the highest value (i.e. face the brightest light).

Similarly, by using a instinct rule requiring the front facing infrared sensors to be "on" constantly (i.e. to return sensor values that indicate an object was situated in front of the robot), the robot acquired a box-pushing or object following competence in two learning steps, one for each of the two situations where the box is placed to the right or the left of the robot's centre respectively. The learning time to acquire a box-pushing competence is less than one minute.

4.4.2 Case Study 2. *FortyTwo*: Robot Training

The mechanism for autonomous competence acquisition, described in case study 1, can be adapted in such a way that the robot can be *trained* to perform certain sensor-motor tasks. This chapter explains how this can be done, and presents experiments conducted with *FortyTwo*.

Why Robot Training? For robotic tasks that are to be performed repeatedly and in structured environments, fixed installations (both hardware and software) for robot control are viable. Many industrial applications fall into this category, for example mass assembly tasks, or high-volume transportation tasks. In these cases, fixed hardware installations (robot assembly lines, conveyor belts, etc.) and the development of fixed, one-off control code are warranted. As robot hardware technology advances, sophisticated robots become available at constantly decreasing cost and even small workshops and service companies become interested in robotics applications ([Schmidt 95] and [Spektrum 95]), the development of control software becomes the governing factor in cost-benefit analysis. For low-volume tasks programming and re-programming the robot is not viable.

In this second case study we discuss experiments in which the robot is *trained* through supervised learning (training signals being provided by the operator) to perform a variety of different tasks. Simple sensor-motor competences such as obstacle avoidance, random exploration or wall following, as well as more complex ones such as clearing objects out of the way, area-covering motion in corridors ("cleaning") and learning simple routes are achieved by this method, without the need to alter the robot control code. During the training phase the robot is controlled by a human operator, and uses the received feedback signals to train an associative memory. After a few tens of learning steps, taking five to ten minutes in real time, the robot performs the required task autonomously. If task, robot morphology or environment change *re-training*, not *re-programming* is used to regain the necessary skills.

Related Work The method of providing external feedback to the learning process is as yet relatively rarely used in robotics. Shepanski and Macy use an operator-taught multilayer perceptron to achieve vehicle-following behaviour in a *simulated* motorway situation ([Shepanski & Macy 87]). The network learns to keep the simulated vehicle at an acceptable distance to the preceding vehicle after about 1000 learning steps. Colombetti and Dorigo present a classifier system with a genetic algorithm that enables a mobile robot to achieve phototaxis ([Colombetti & Dorigo 93]). *AUTONOMOUSE*, the robot used, began to show good light seeking behaviour after about 60 minutes of training time.

Unsupervised learning has also been used in robot control. Tasks such as obstacle avoidance and contour following ([Nehmzow 95a]), box-pushing ([Mahadevan & Connell 91] and [Nehmzow 95a]), coordination of leg movement in walking robots ([Maes & Brooks 90]) or phototaxis ([Kaelbling 92], [Colombetti & Dorigo 93] and [Nehmzow & McGonigle 94]) have successfully been implemented on mobile robots.

Teach-by-guiding ([Critchlow 85] and [Schmidt 95]) is still a common method of programming industrial robots. This method is different to work based on artificial neural networks, in that it does not show the generalisation properties of networks, but merely stores chains of locations to be visited in sequence.

Controller Architecture The central component in the controller used in the experiments is an associative memory, implemented as before through a Pattern Associator. The controller is shown in figure 4.15.

Inputs to the associative memory consist of preprocessed sensor signals. In the experiments presented here sensor signals from the robot's sonar and infrared sensors have been used, and preprocessing was limited to a simple thresholding operation, generating a '1' input for all sonar range signals of less than 150 cm distance (see figure 4.16). Experiments using visual input data are reported elsewhere ([Martin & Nehmzow 95]); in these experiments thresholding, edge detection and differentiation were used in the preprocessing stages.

The analogue output signals o_k of the associative memory are computed according to equation 4.17:

$$o_k = \vec{w}_k \cdot \vec{\imath}, \tag{4.17}$$

with $\vec{\imath}$ being the input vector containing the sensor signals, and $\vec{w}_k$ the weight vector of output node k (i.e. one of the two output nodes).

The two analogue output nodes of the associative memory drive the steering and translation motor of the robot, respectively (in the experiments presented here steering and turret rotation are locked, so that the front of the robot is always facing the direction of travel). This generates continuous, smooth steering and translational velocities, depending on the strength of the association between current sensory stimulus and motor response (see also [Nehmzow 99c]): the robot moves fast in situations which have been trained frequently and therefore have strong associations between sensing and action ("familiar" situations), and slowly in "unfamiliar" situations. If sensor signals are received which have

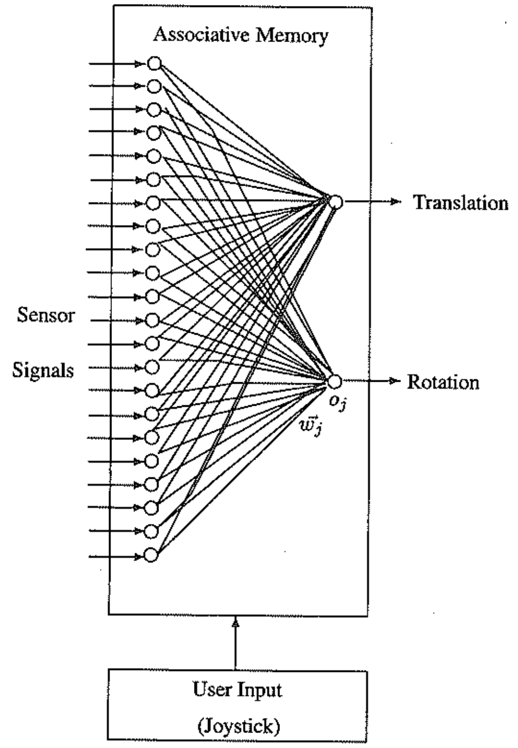

FIG. 4.15. THE CONTROLLER ARCHITECTURE

5 values	5 values	6 values	6 values
Left facing Sonars	Right facing Sonars	Left facing IRs	Right facing IRs

FIG. 4.16. THE INPUT VECTOR USED

never been encountered during the training phase, i.e. situations in which the network weights between sensory input and motor action output are zero, the robot does not move at all.

Training the Network Initially, there are no sensor-motor associations stored in the associative memory (i.e. all network weights are set to zero), and the robot is driven by the operator by means of the joystick. Information about the current environment (the input vector $\vec{\imath}$ to the controller) and the desired motor action in this situation (yielding the target output τ_k of equation 4.6 for each of the two output units of the network) is therefore available to the robot controller in the training phase.

Adjusting the weight vector $\vec{w}_k$ of output unit k is achieved by applying the Perceptron learning rule (equations 4.6 and 4.7).

The learning rate η was chosen at 0.3, and decreased by 1% every learning step. This eventually stabilises the associations stored in the associative memory of the controller. There are other ways conceivable of choosing the learning rate: an initially high, but rapidly falling rate will result in a robot learning only during the initial stages of the learning process, whilst a constant η will lead to continous learning behaviour. Thirdly, by means of a novelty detector a low η could be increased when drastic changes are detected.

Experiments The experiments described here were conducted in a laboratory of 5 m × 15 m, containing tables, chairs and boxes which were detectable by the robot's sensors.

Initially, there were no sensor-motor associations stored in the associative memory, and the robot was driven by the operator by means of the joystick. Using this information for training, meaningful sensor-motor couplings developed within the artificial neural network, and the robot improved rapidly in performing a specified task. After a few tens of learning steps the acquired information was usually sufficient to control the robot without any user intervention, the robot then performed the task autonomously, solely under network control.

Obstacle Avoidance Initially driving the robot with the joystick, *FortyTwo* was trained to avoid convex obstacles and cul-de-sacs. The robot learned this in less than 20 learning steps (one step being one application of equations 4.6 and 4.7), taking a few minutes in real time. The resulting obstacle avoidance motion was smooth, and incorporated translational and rotational movements at varying speeds, according to the strength of association between sensory perception and corresponding motor action (see also [Nehmzow 99c]).

Wall Following In the same manner as before, the robot was trained to stay within a distance of about 50 cm to the wall. As the only relevant part of the input vector for this task is the infrared one, the robot moved closer towards dark objects (little infrared reflection), and further away from light objects. Again, learning was achieved within 20 learning steps, and the acquired motion was smooth. The robot could be trained to follow left hand walls, right hand walls, or both.

Box-Pushing Both infrared sensor and sonar sensor range data is part of the input vector presented to the associative memory of the controller (see figure 4.16), and can therefore be used by the controller to associate action with perception. *FortyTwo's* infrared sensors are mounted low, about 30 cm above ground, whilst the sonar sensors are mounted high (ca. 60 cm above ground).

In a box-pushing experiment the robot was taught to move forward if a low object was detected straight ahead, and to turn left or right if an object was detected to the left or the right side of the robot respectively. During the training phase of about 30 learning steps, associations between the robot's infrared sensors (which are relevant to this task) developed, whilst sonar data delivered contradictory information in this experiment and strong associations between the "sonar part" of the input vector and the motor-driving outputs did not develop.

The robot acquired the ability to push a box quickly, in a few minutes of real time. After the initial training phase the robot was able to push and follow a box autonomously, staying behind the box even if the box moved sideways.

Clearing As the infrared sensors of the robot return identical signals for *any* object placed near the robot, it is impossible for the controller to differentiate between boxes, walls, or people. For example, the robot will attempt to push walls, as much as it will push boxes.

There is, however, a way of training *FortyTwo* to abandon boxes near tall obstacles: as the high mounted sonar sensors reach beyond a low box, the robot can be trained to pursue objects as long as they appear only in the "infrared part" of the input vector, but to abandon them as soon as tall obstacles appear in the "sonar part" of the input vector. Training for about 30 to 50 learning steps, taking between five and ten minutes in real time, the robot could be taught to acquire such a "clearing" ability. After the training phase, the robot pursued any object becoming visible to the infrared sensors only, and pushed it until tall obstacles were detected by the sonar sensors. *FortyTwo* then turned away from the object, looking for new boxes away from the tall obstacles. This resulted in a "clearing" behaviour: *FortyTwo* pushed boxes towards walls, left them there and returned to the center of the room to look for more boxes to move to the sides.

Surveillance Using the same method, and without any need to re-program the robot, *FortyTwo* could be trained to move in random directions whilst avoiding obstacles. During the training phase of approximately 30 learning steps, the robot was instructed to move forward when no objects were ahead, and to turn away from obstacles once they became visible to the robot's sensors. This developed associations between the sonar and infrared sensors and the motor-driving output units of the network. In this manner the robot acquired a general obstacle avoidance behaviour, also for input constellations that had not been encountered during the training phase (this is a result of the artificial neural network's capability to generalise).

The resultant obstacle avoidance behaviour was smooth and continuous; fast translational movement occured when familiar sensor states indicated free space. Smooth turning action was perceived when *distant* obstacles were detected, rapid turning action resulted from *nearby* obstacles being detected. The smooth motion of the robot was achieved through the direct association of the (analogue) outputs of the network with motor action, and the output itself being dependent on the strength of the input signal received (nearby objects will produce stronger input signals).

Such random exploration and obstacle avoidance behaviour forms the basis of a surveillance function of the robot. Using this behaviour, *FortyTwo* will, for example, follow corridors, avoiding moving and stationary obstacles as it moves.

Route Learning As the robot effectively learns to associate sensory perception directly with a motor response, it can be taught to perform certain motor actions at particular locations, thus performing navigation. This is navigation along "perceptual landmarks", in which the perceptual properties of the environment are used to perform the desired motion at each physical location[5].

We have trained *FortyTwo* to follow a route as shown in figure 4.17. After about 15 minutes of training time the robot was able to follow the wall in the way indicated in figure 4.17, leave the laboratory through the door (which had about twice the width of the robot's diameter), turn and return through the door back into the laboratory and resume the path.

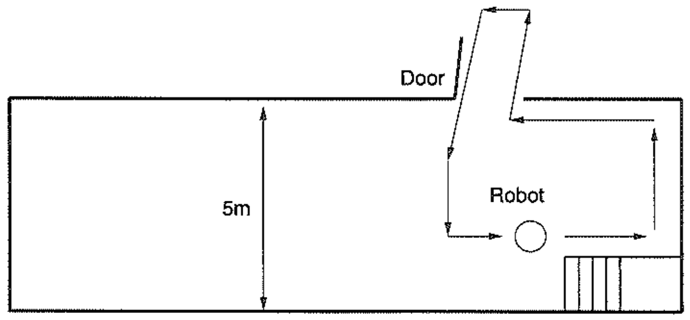

FIG. 4.17. ROUTE LEARNING

Cleaning Cleaning tasks require the robot to cover as much floorspace as possible. For large, open spaces random exploration might be sufficient to accomplish the task over time. However, it is possible to train *FortyTwo* to cover floorspace in a more methodical manner.

Using the same training method as before, the robot was trained to move along a corridor in the Computer Science Department in the manner shown in figure 4.18.

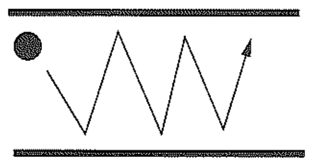

FIG. 4.18. CLEANING OPERATION

As before, learning is fast and the robot acquires the competence in a few tens of learning steps.

[5] The problem of "perceptual aliasing", which describes the fact that two different places may appear identical to the robot's sensors, is discussed in section 7.3.2.

Conclusions *Training*, rather than *programming* has a number of advantages. First of all, the Pattern Associator used here learns extremely fast; within a few learning steps meaningful associations between inputs and outputs develop.

Furthermore, it is able to generalise, i.e. to acquire input-output associations for input situations that have not been encountered explicitly. For large sensor spaces it is practically impossible that the robot encounters all possible sensor states during the training phase — the generalisation ability of artificial neural networks provides a solution to this problem.

Thirdly, as sensor stimuli are directly associated with (analogue) motor responses, the strength of an association determines the velocity of the action. This means that the robot moves fast in familiar territory, and slowly in unfamiliar territory; that the robot turns rapidly if nearby obstacles are detected, and gently in the presence of distant ones. This property is emergent, and not a design feature of the controller.

In addition to these points, "programming through teaching" provides a simple and intuitively clear man-machine interface. As the operator is able to instruct the machine *directly*, without the need of a middle man (the programmer), the risk of ambiguities is reduced.

And finally, through supervised training of artificial neural networks, an effective control strategy for a mobile robot can be established, even if explicit control rules are not known.

4.4.3 Case Study 3. *FortyTwo*: Learning Internal Representations of the World Through Self-Organisation

For many robotics applications, for example those of object identification and object retrieval, it is necessary to use internal representations of these objects. These models — abstracted (i.e. simplified) representations of the original — are to capture the essential properties of the original, without carrying unnecessary detail.

The "essential properties" of the original are not always directly available to the designer of an object recognition system, nor is it always clear what constitutes unnecessary detail. In these cases the only feasible method is that of model *acquisition*, rather than model *installation*.

In this third case study, therefore, we present experiments with a self-organising structure that acquires models in an unsupervised way through robot-environment interaction. User predefinition is kept at a minimum.

In this particular instance, *FortyTwo's* task was to identify boxes within its visual field of view, and to move towards them. No generic model of these boxes was used, instead a model was acquired through a process of unsupervised learning in an artificial neural network.

Experimental Setup *FortyTwo's* task was to determine whether or not a box was present in its visual field of view, and, if present, to move towards the box. The camera was the only sensor used. The boxes carried no particular distinguishing features.

The trainable object recognition system shown in figure 4.19, based on a self-organising feature map, was used to achieve this.

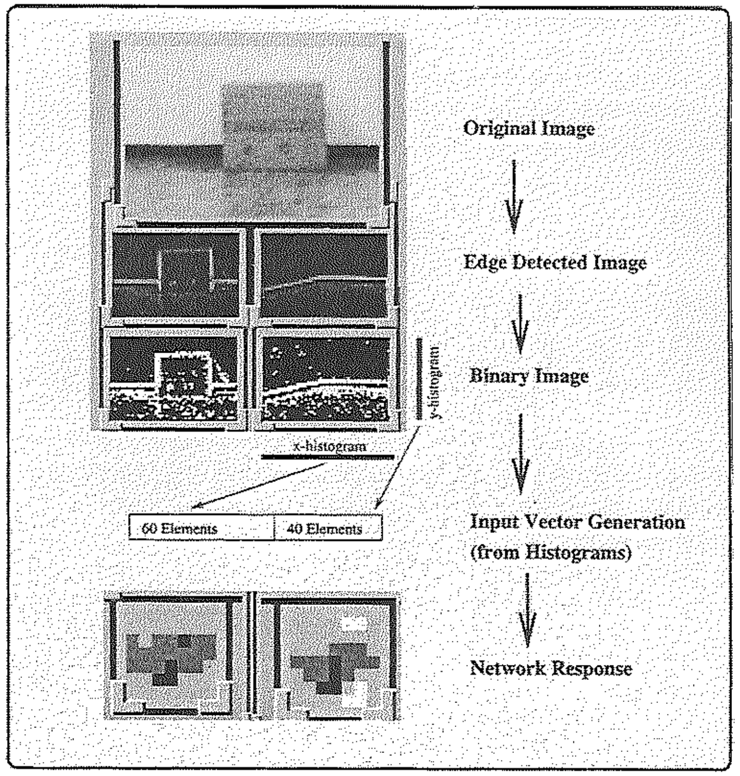

FIG. 4.19. The Box Recognition System

Vision Data Preprocessing System As a first processing step, the raw grey level image of 320 by 200 pixels was reduced to 120 by 80 pixels by selecting an appropriate window that would show the target object at a distance of about 3 m.

The reduced grey level image was then subjected to a convolution with the edge-detecting template shown in figure 4.20, where each new pixel value is determined by equation 4.18.

$$e(t+1) = |(c+f+i) - (a+d+g)| \, . \tag{4.18}$$

The edge-detected image was then coarse coded by averaging the pixel values of a 2 x 2 square, yielding an image of 60 by 40 pixels.

-1	0	1
-1	0	1
-1	0	1

a	b	c
d	e	f
g	h	i

FIG. 4.20. EDGE DETECTING TEMPLATE

Following the coarse coding stage, we computed the average pixel value of the entire image, and used this value to generate a binary image (such as the one shown in figure 4.21) by thresholding.

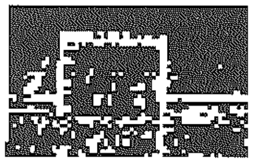

FIG. 4.21. BINARY IMAGE

Finally, by computing the histogram along the vertical and horizontal axis of the binary image, we obtained a 60+40 element long input vector, which was used as input to our box detection algorithm. This is shown in figure 4.22.

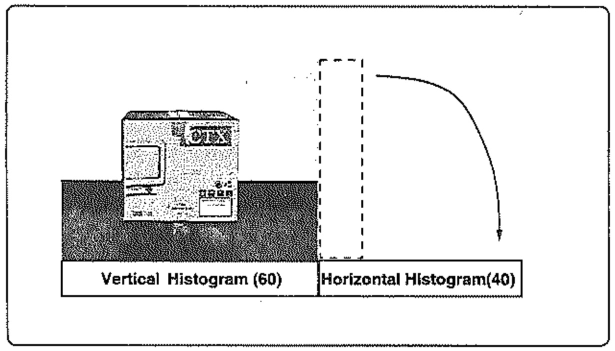

FIG. 4.22. INPUT TO THE BOX-DETECTING SYSTEM

The Box Detection Mechanism We then used this 100-element long input vector as an input to a self-organising feature map (SOFM) of 10 by 10 units (see section 4.2.3). The learning rate was set to 0.8 for the first 10 learning steps, after that to 0.2 for the entire remaining training period. The neighbourhood around the winning unit within which an update was performed remained static as ± 1 throughout the entire training period.

The idea behind this was, of course, to develop different network responses for images containing boxes than to images containing no boxes — based on self-organisation, and without any *a priori* model installation at all.

Experimental Results A test set of 60 images (30 with boxes, 30 without — see figure 4.23) was used to train the network, and evaluate the system's ability to differentiate between images containing boxes, and those not containing boxes.

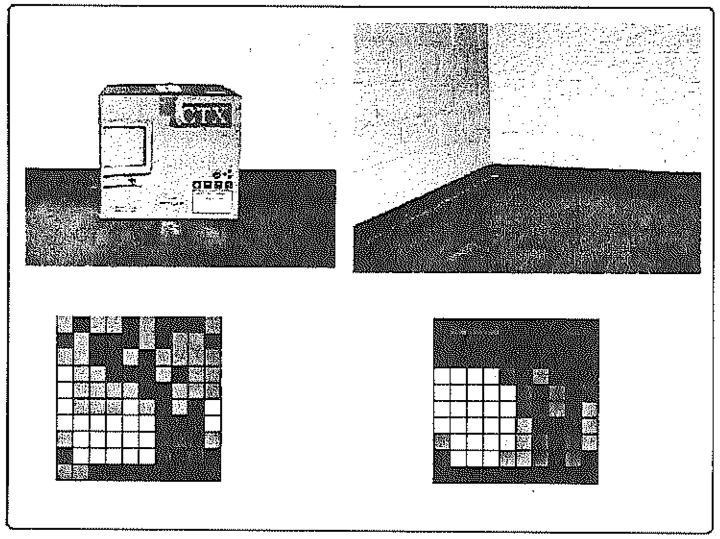

FIG. 4.23. TWO EXAMPLE IMAGES AND THE NETWORK'S RESPONSE TO THEM. THE GREY-LEVEL CODING INDICATES THE STRENGTH OF A UNIT'S ACTIVATION.

As can be seen in figure 4.23, the network's response in both cases is similar, but not identical. The difference in network response can be used to classify an image.

Fifty images of aligned boxes were used for the training phase of the network, of the remaining 10 test images all were correctly classified by the system.

In a second set of experiments, boxes were placed in *various* positions and angles. The training set consisted of 100 images, the test set contained 20 images. In the case of images showing boxes, 70% of all test images were classified correctly, 20% were wrong, and 10% were "not classified" (i.e. the excitation patterns of the SOFM neither resembled the "box" pattern, nor the "no box" pattern). Of the images not showing boxes, 60% were classified correctly, 20% were incorrectly classified and 20% were not classified.

To assess the ability of the system to classify images under more "realistic" situations, we conducted a third set of experiments, in which images were used that were similar to those in the previous experiment, i.e. they contained images of boxes in varying positions and angles. In addition to this, images of stairs, doors and other objects that had similarities with boxes were included ("difficult" images).

The training set consisted of 180 images, the test set comprised 40 images. Of the "box" images, 70% were classified correctly, 20% were incorrectly classified and 10% were not classified at all.

Of the "no box" images, 40% were classified correctly, 55% were incorrectly classified as "box" and 5% were not classified.

Associating Perception with Action Next, we were interested to use the system to guide the robot towards boxes, if any were identified within the image.

In SOFMs, this can be achieved by extending the input vector by adding an action component to it. The entire input vector (and therefore the weight vector of each unit of the SOFM) then contains a perception-action pair. In the training phase, the robot is driven manually towards a box in its field of view. Input vectors are generated by combining the preprocessed vision input and the user-supplied motor command.

In the recall phase, the robot is then able to move towards a box autonomously, by determining the winning unit of the SOFM (i.e. that unit that resembles the current visual perception most closely), and performing the motor action associated with that unit. Our observation was that *FortyTwo* was well able to move towards a single box in its field of view, regardless of orientation of the box, or initial orientation of the robot. As the robot approached the box, lateral movements of the robot decreased, and the approach became faster and more focused, until the box filled the entire field of view, and thus became invisible to the system. The robot approached boxes reliably under these conditions.

However, the robot could get confused by other box-like objects in the field of view (like the stairs in our robotics laboratory). In this case, the robot would approach the misleading object in question, to abandon it later when the error was detected. At this stage, however, the robot was often no longer able to detect the original box, because it had moved too far off the direct approach route.

Conclusions Developing internal representations is essential for many robotics tasks. Such models of the original objects simplify computation through their abstraction properties. In order to be useful, models need to capture the essential properties of the objects modelled whilst eliminating unnecessary detail.

As these characteristic properties are often not directly accessible to the designer, methods of (subsymbolic) model *acquisition*, rather than model *installation* are a possibility.

The box recognition system discussed here does not use any symbolic representations, and only very general information is supplied at the design stage (i.e. edge detection, thresholding and histogram analysis of images). Instead, models are acquired *autonomously* by clustering sensory perception, using a self-organising feature map.

The experiments showed that the acquired models could identify target objects with good reliability, provided the images contained no misleading (i.e. box-like) information.

Boxes are very regular objects. Whether a simple system such as the one described in this case study can construct representations of more complex objects (such as, for instance, people), is not clear. However, these experiments demonstrate that a robot *can* build internal representations of objects in its environment, without *a priori* knowledge, and without human support.

Case Study 3: Further Reading

- Ulrich Nehmzow, Vision Processing for Robot Learning, *Industrial Robot*, Vol. 26, No. 2, pp. 121-130, 1999.

4.5 Exercise 3: A Target-Following, Obstacle-Avoiding Robot

A mobile robot is equipped with two tactile sensors, one on its left side, and one on its right side. These sensors return "+1" when they touch something, otherwise "0". In addition to this, the robot has a centrally mounted beacon sensor, which can detect whether a beacon (which is placed somewhere in the environment) is to the right or to the left of the robot. The beacon sensor returns "-1" if the target is to the left, and "+1" if the target is to the right of the robot. Due to the asymmetries of the real world, the beacon sensor will never perceive the beacon as absolutely dead ahead, and therefore no third value for "ahead" exists.

The robot's motors will turn forward if a "+1" signal is applied, and backwards on a "-1" signal. The robot is shown in figure 4.24.

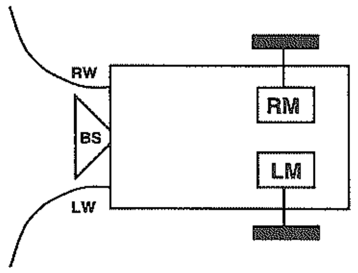

FIG. 4.24. A ROBOT WITH TWO TACTILE SENSORS AND ONE BEACON SENSOR

The task of the robot is to move towards the beacon, avoiding obstacles along the way.

- State the truth table required to achieve this function.
- Design an artificial neural network, using McCulloch and Pitts neurons, to implement the target seeking, obstacle avoiding robot.

The solution is given in appendix 2.2 on p. 218.

That the robot is moving towards a wall at the end of the run can be deduced from the fact that the distance readings of the front sensor are decreasing constantly after time step 40.

Whether or not the doors were open in the experiment is not quite so easy to decide. The depth of the doors is measured at about 10 cm, which indicates they were closed. However, the beam of the side-looking sonar may not be narrow enough to pass through the doorway, and therefore detect the door posts, even if the door itself is open.

2 Robot Learning

2.1 Full Obstacle Avoidance, using McCulloch and Pitts Neurons

The truth table to be implemented is shown in table 1.

LW	RW	LM	RM
0	0	1	1
0	1	-1	1
1	0	1	-1
1	1	-1	-1

Table 1. TRUTH TABLE FOR FULL OBSTACLE AVOIDANCE

Line one of this truth table indicates that the threshold Θ must be below zero. As before, we choose $\Theta = -0.01$. We determine the weights w_{LW} and w_{RW} for the left motor neuron here. The weights for the right motor neuron are found analogously.

Lines two, three and four of the truth table translate into the following three inequalities:

$$w_{RW} < \Theta$$
$$w_{LW} > \Theta$$
$$w_{LW} + w_{RW} < \Theta$$

These three inequalities can be satisfied with, for example, $w_{LW} = 0.3$ and $w_{RW} = -0.5$.

2.2 A Target Following, Obstacle Avoiding Robot

The truth table for the target seeking, obstacle avoiding robot described in section 4.5 is given in table 2. This robot will obviously never execute a forward movement, because the beacon sensor either indicates "steer left" or "steer right", which will be executed as a turn. Therefore, we can restrict the truth table to the function of the left motor, and later implement the exact opposite function of the left motor for the right motor.

LW	RW	BS	LM
0	0	-1	-1
0	0	1	1
0	1	-1	-1
0	1	1	-1
1	0	-1	1
1	0	1	1
1	1	-1	don't care
1	1	1	don't care

Table 2. TRUTH TABLE FOR TARGET SEEKING AND OBSTACLE AVOIDANCE

We will now attempt to implement this truth table using one McCulloch and Pitts neuron per motor — again looking at the left motor only. The structure of this network is shown in figure 2.

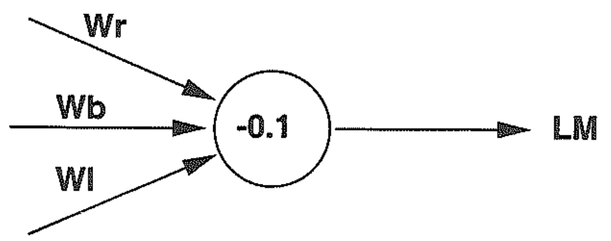

FIG. 2. STRUCTURE OF THE REQUIRED NEURON FOR THE LEFT MOTOR

Lines 1 and 2 of the truth table yield $w_B > \Theta$ (with an arbitrary selection of $\Theta = -0.01$ here).

Line 5 of the truth table yields $w_L - w_B > \Theta$, and from line 4 of the truth table we can determine the inequality $w_R + w_B < \Theta$.

This allows us to select three weights that satisfy the requirements, for example $w_L = 5$, $w_B = 3$ and $w_R = -5$. Going through all lines of the truth table confirms that these weights would implement the required function.

The final network structure is shown in figure 3. Weights for the right motor are the mirror image of those for the left motor (taking into account, of course, that the beacon sensor has a "+1/-1" encoding for direction).

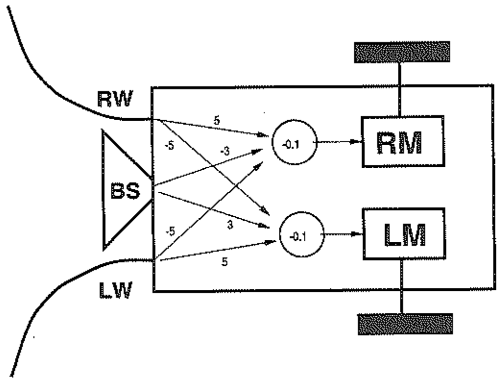

FIG. 3. A TARGET SEEKING MOBILE ROBOT THAT AVOIDS OBSTACLES, USING MCCULLOCH AND PITTS NEURONS.

3 Error Calculations and Contingency Table Analysis

3.1 Mean and Standard Deviation

A robot is stationary in front of an obstacle, and obtains the following range readings from its sonar sensors (in cm): 60, 58, 58, 61, 63, 62, 60, 60, 59, 60, 62, 58.

Using equation 7.2, we can compute the mean distance measured as $\mu = \frac{721}{12} = 60.08$ cm. Equation 7.3 can be used to compute the standard deviation as $\sigma = \sqrt{\frac{1}{11}30.92} = 1.68$. This means that 68.3% of all range values lie in an interval of (60.08 ± 1.68) cm.

The standard error is $\bar{\sigma} = 0.48$ (equation 7.4), indicating that with a certainty of 68.3% the true mean lies in the interval 60.08 cm $\pm$ 0.48 cm.

3.2 Classifier System

A mobile robot has been equipped with a fixed algorithm to detect doorways, using an on-board CCD camera. The probability of this system producing a correct answer is the same for each image.

The system is initially tested using 750 images, half of which contain a doorway, and half of which don't. The algorithm produces correct answers in 620 cases.

In a second series of experiments, 20 images are presented to the robot. What is the probability that there will be two classification errors, and which number of errors is to be expected in classifying those 20 images?

Answer: If a classifier system is used in n independent trials to classify data, it will produce n answers, each of which is either correct or incorrect. This is a binomial distribution.

We define p as the probability of producing an incorrect answer. In this case $p = 1 - \frac{620}{750} = 0.173$.

The probability p_2^{20} of making 2 mistakes in twenty classifications can be determined using equation 7.5: $p_2^{20} = \frac{20!}{(20-2)!2!}0.173^2(1-0.173)^{20-2} = 0.186$.

The number of errors μ_b to be expected in 20 experiments is given by equation 7.6 as $\mu_b = np = 20 \cdot (1 - \frac{620}{750}) = 3.47$.

3.3 T-Test

A robot control program is written to enable robots to withdraw from dead ends. In a first version of the program, the robot takes the following time in seconds to escape from a dead end: x=(10.2, 9.5, 9.7, 12.1, 8.7, 10.3, 9.7, 11.1, 11.7, 9.1). After the program has been improved, a second set of experiments yields these results: y=(9.6, 10.1, 8.2, 7.5, 9.3, 8.4).

Do these results indicate that the second program performs significantly better?

Answer: Assuming that the outcome of the experiments has a normal (Gaussian) distribution, we can apply the T-test to answer this question.

$\mu_x = 10.21, \sigma_x = 1.112, \mu_y = 8.85, \sigma_y = 0.977$.

Applying equation 7.12 yields:

$$T = \frac{10.21-8.85}{\sqrt{(10-1)1.112^2+(6-1)0.997^2}}\sqrt{\frac{10*6(10+6-2)}{10+6}} = 2.456\,.$$

As $k = 10+6-2$, $t_\alpha = 2.145$ (from table 7.1). The inequality $|2.456| > 2.145$ holds, the hypothesis H_0 (i.e. $\mu_x = \mu_y$) is rejected, which means that the second program performs significantly better than the first one, the probability for this statement to be erroneous is 0.05.

4 Analysis of Categorical Data

4.1 χ^2 Analysis

A mobile robot is placed in an environment that contains four prominent landmarks, A, B, C and D. The robot's landmark identification program produces four responses, α, β, γ and δ to the sensory stimuli received at these four locations. In an experiment totalling 200 visits to the various landmarks, the following contingency table is obtained (numbers indicate the frequency of a particular map response obtained at a particular location):

Is the output of the classifier significantly associated with the location the robot is at?

Answer: Following equation 7.14 $n_{A\alpha} = \frac{40*34}{200} = 6.8$, $n_{A\beta} = \frac{40*78}{200} = 15.6$, and so on (the table of expected values is table 4).