1. **Receive task specific goal behaviour $\mathbf{b}_{\text{goal}}$ from higher level of control.**

2. **Access the current world model to obtain a raw action $\mathbf{a}_{\text{raw}}$ which is predicted to be likely to achieve $\mathbf{b}_{\text{goal}}$.**

3. **Apply action $\mathbf{a}_{\text{raw}}$.**

4. **Observe actual behaviour $\mathbf{b}_{\text{actual}}$.**

5. **Update the world model with the information that $\mathbf{a}_{\text{raw}} \rightarrow \mathbf{b}_{\text{actual}}$.**

Table 4.1: The AB Control Cycle

knowledge that the relationship is a deterministic function from actions to behaviours let us call such a relationship the *Perception Function* (PF):

$$\text{PF} : \textbf{Action} \rightarrow \textbf{Behaviour} \tag{4.1}$$

The arm position task is an example of a PF because we have the knowledge that (i) given a set of joint angles the arm's real world endpoint position is determined and (ii) the cameras are fixed and so the perceived hand position on the camera is determined by the real world hand position.

To *learn* the PF would mean that the learning controller would have no knowledge about the relationship other than that it was a PF. A very general way of implementing such a learning controller is the *AB* Control Cycle, shown in Table 4.1.

Because we are dealing with raw action signals and raw perception signals (after crude image processing), the only substantial computation is in Steps 2 and 5. Requirements analysis and design of this computation is the part of main work of this dissertation. But before we consider this, in order to learn *dynamic* control, we must introduce the concept of perceived state.

4.2 The Perceived State Transition Function

A dynamic system has a *state*. Conventionally, when mathematical analysis is to be applied, the choice of the state representation is crucial to the tractability of the mathematics. The conventional

form of the differential equation derived from mathematical analysis is

$$\dot{\mathbf{x}} = g(\mathbf{x}, \mathbf{u}) \tag{4.2}$$

where $\mathbf{x}$ is the system's state, and $\mathbf{u}$ is the control input to the system [Burghes and Graham, 1980]. It is hoped that this equation will be simple, for example linear:

$$g(\mathbf{x}, \mathbf{u}) = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{4.3}$$

where $\mathbf{A}$ and $\mathbf{B}$ are constant matrices. The choice of state representation will strongly influence the simplicity, and hence tractability, of Equation 4.2. Here, let us call a state representation chosen to aid mathematical simplicity a *dynamic-model* state. The state is usually observed indirectly, by taking a series of measurements of the system, for example visual. Let us call the vector of these direct measurements the *perceived* state, $\mathbf{p}$, of the system (this is analogous to the definition of perceived behaviour in the previous section). The perceived state is transformed to and from the dynamic-model state by further mappings, the perception function PF and its inverse PF^{-1}.

$$\mathbf{x} = \mathrm{PF}(\mathbf{p}) \text{ and } \mathbf{p} = \mathrm{PF}^{-1}(\mathbf{x}) \tag{4.4}$$

The transformations PF and PF^{-1} are often at least as complex to model as the dynamics themselves. For example, in the case of a robot arm observed visually, PF is the composition of the inverse perspective transform and the inverse kinematics. The mathematical analysis derives the change in the perceived state $\mathbf{p}$ by transforming it to the dynamic-model state, then performing the dynamics model (Equation 4.2) and then finally transforming back to perceived space:

$$\dot{\mathbf{p}} = \mathbf{J}^{-1} g(\mathrm{PF}(\mathbf{p}), \mathbf{u}) \text{ where } \mathbf{J}_{ij} = \frac{\partial \mathrm{PF}_i}{\partial p_j} \tag{4.5}$$

$\mathbf{J}$ is the Jacobian matrix of the perception function PF.

In this investigation, we are not using mathematical analysis to perform the modelling, and so it is not necessary to perform these transformations explicitly, nor work out the inverse Jacobian matrix of the kinematics (called the inverse differential kinematics), nor indeed even necessary to invent a dynamic-model state representation. Instead, the perceived state dynamics are denoted by one mapping, the *perceived state transition function* (PSTF):

$$\dot{\mathbf{p}} = \mathrm{PSTF}(\mathbf{p}, \mathbf{u}) \tag{4.6}$$

The PSTF is a deterministic function from the perceived *state* and raw *action* applied to the perceived *behaviour*.

$$\mathrm{PSTF} : \mathbf{State} \times \mathbf{Action} \rightarrow \mathbf{Behaviour} \tag{4.7}$$

One common example of perceived behaviour is the change in perceived state, but in Chapter 10 there are examples of other kinds of perceived behaviour.

There are computational advantages to using the PSTF, but the greatest benefit, which will be described later, comes from the abstractness of perceived space, and the fact that it is in perceived space that the task descriptions themselves are likely to occur.

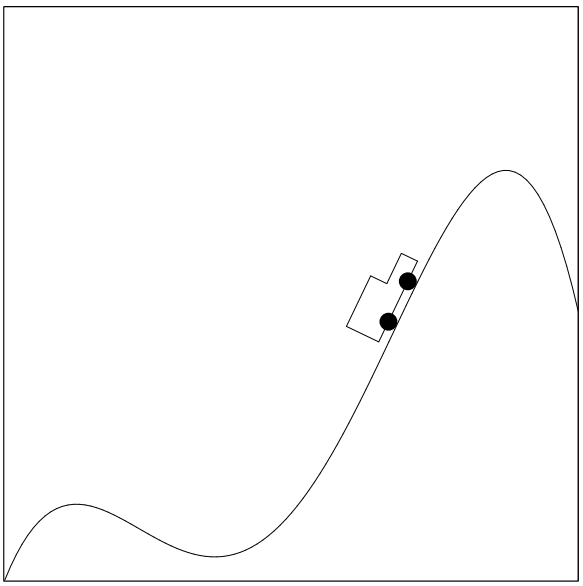

Figure 4.1

The "Mountain Car": a very simple dynamic system.

4.3 The Mountain Car Example

This is a very simple example which will be used for illustration during subsequent chapters. It is contrived so that the state space and control space are both one-dimensional. An automatic car drives along a one-dimensional track over a mountain range, as depicted in Figure 4.1.

The controlling station can sense the horizontal distance s that the car is from the start of the road. The controlling station can also send a signal u which specifies how far down the car's pedal should be pressed. Instantaneous changes to the pedal cause instantaneous changes to the car's speed. The speed depends on the pedal height, the gradient of the mountain where the car is, and also the altitude. Since these latter two features only depend on the car's location, we can deduce that the speed depends indirectly only on the pedal height and the distance along the road. The road is 1000 metres long and the pedal height varies between 0 and 10 centimetres. In the simulations that follow, this world will be used (none of this information is available to the controller):

- The height of the road (in metres) varies with distance in metres according to the function

$$h(s) = 1000P(s/1000) \text{ where } P(x) = -8x(x^3 - 1.82x^2 + 0.95x - 0.16) \tag{4.8}$$

- The speed of the car along the road's surface is increased by decreasing the pedal height u. It is also increased by a downwards gradient or high altitude. The surface speed in metres per minute is

$$v_{\text{surface}}(s,u) = K_0\sqrt{1 - \frac{u}{10}} - K_1\frac{dh}{ds} + K_2 h \tag{4.9}$$

where $K_0 = 140$, $K_1 = 10$ and $K_2 = 0.05$.

- The horizontal speed of the car depends on the surface speed and the current hill gradient:

$$v_{\text{horiz}}(s,u) = \frac{v_{\text{surface}}(s,u)}{\sqrt{1+(\frac{dh}{ds})^2}} \tag{4.10}$$

  The horizontal speed varies between approximately -8m/min and 150m/min.

- The perceived state (distance travelled) is scaled uniformly in the range 0 to 10. The perceived action (pedal height) and perceived behaviour (horizontal speed) are similarly scaled. This arbitrary scaling exemplifies the fact that SAB learning need have no notion of the meaning of these variables. Writing s_p as the perceived state, a_p as the perceived action and b_p as the perceived behaviour:

$$b_p = \text{PSTF}(s_p, u_p) = 10\frac{v_{\text{horiz}}\,(10s/s_{\max}, 10u/a_{\max}) - b_{\min}}{b_{\max} - b_{\min}} \tag{4.11}$$

  where $b_{\max} = 150$m/min, $b_{\min} = -8$m/min, $s_{\max} = 1000$m and $a_{\max} = 10$cm.

The function $\text{PSTF}(s,u)$ can be graphed. The behaviour (speed) is a function of two variables: the state s (position) and the action u (pedal height). This is graphed in Figure 4.2. In this diagram the brightness denotes the behaviour—the brighter the tone, the closer the behaviour to the speed 5.7 units. This somewhat odd convention is adopted because we will later be studying a task in which the closer the speed to 5.7 units, the better. Thus, for example the graph shows that when $s = 6.5$ and $u = 3$ then the speed produced is 5.7. The bands below the central white band show increasing speed due to the pedal being closer to the floor. The bands above the white band show decreasing speed caused by the pedal being further from the floor.

4.4 The SAB Control Cycle

We are now ready to propose an enlargement of the learning controller of Section 4.1 which takes account of state. This is called the SAB control cycle, shown in Table 4.2.

It can now be seen that AB learning is a specific example of SAB learning in which the observed behaviour depends entirely on the chosen action, and there is no state to take into account.

Steps 3 and 6 are the crucial computations for the learning system. The performance of the robotic system will depend on the following:

- **Fast Learning.** It is desirable to minimize the length of time that the robot is training (that is, performing inadequately due to lack of knowledge). This is because of the expense of wasted robot time, and the possible hazard (or more reasonably the expense of precautions to avoid hazard) during this time of uncontrolled activity. Another reason for requiring a high rate of learning is that if changes occur in the environment, we wish the robot, through learning, to adapt to these changes reasonably quickly.

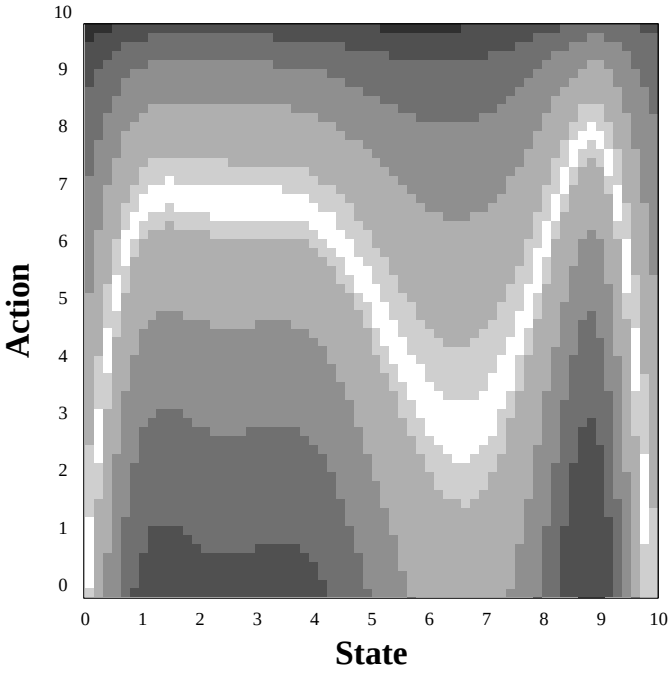

Figure 4.2

The Perceived State Transition
Function of the Mountain car. Brightness denotes the difference in speed from the goal speed of 5.7.

1. **Observe current perceived state $s_{current}$.**
2. **Receive task specific goal behaviour b_{goal} from higher level of control. The requested behaviour might depend partially on $s_{current}$.**
3. **Access the current world model to obtain a raw action a_{raw} which is predicted to be likely to achieve b_{goal} acting in the current state.**
4. **Apply action a_{raw}.**
5. **Observe actual behaviour b_{actual}.**
6. **Update the world model with the information that $(s_{current}, a_{raw}) \rightarrow b_{actual}$.**

Table 4.2: The *SAB* Control Cycle

- **Cheap Learning.** Step 3 needs to take place within the timescale of the robot's dynamics, which is assumed to be a fraction of a second. As can be seen in Step 6 of the control cycle defined above, model updating has also been placed within the constraint in this investigation. There was an alternative to this decision: the update data could simply have been recorded during the execution of a dynamic task. Afterwards, when the robot was stable, the data would be used to update the world model. This alternative has two disadvantages: (i) It impacts the Fast Learning requirement during the initial trials, where we might have hoped to learn rough models within the very first moments of execution and (ii) it assumes that resting times will be available— in practice a large class of robotic tasks are under constant execution. An example is a balancing task.

- **Variable Resolution of Interest.** The state, action and behaviour spaces are multi-dimensional. This means that the number of significantly different states is enormous, and the number of significantly different state-action pairs is a magnitude greater still. A simple example would be a six-dimensional state space and a three-dimensional action space. Each dimension contains a continuum of values, but let us suppose, optimistically, that in each dimension only ten values are significantly different: then there are 10^9 different state-action combinations. At, say, six observations a second it would still take over five years to try each significantly different state-action pair even once. This illustrates an important restriction: we should not try to learn everything. Instead we should try to learn well and accurately in the regions of state-action space which are found useful for a task, and only learn a rough general picture for other areas.

The initial questions include: How should the world model be represented? How should generalization take place when asked to make predictions about things not experienced? Can the learning be all of fast, cheap and general? In the next chapter we examine a simple powerful generalization and in subsequent chapters we will consider its efficiency and its ability to cope with disorder.

Chapter 5

Nearest Neighbour: Quick, Cheap Learning?

This chapter begins with a definition and description of the Nearest Neighbour generalization. Its history is also described. A survey of a wide variety of alternative learning techniques is then provided, along with a comparison to nearest neighbour. Later in the chapter the generality of nearest neighbour is formalized and proved, the speed of learning is analysed, and finally a variety of further aspects of nearest neighbour are discussed.

5.1 The Nearest Neighbour Generalization

The important components of a learning system are (i) an underlying representation of the concept being learned, called the *performance element*, (ii) an updating function which takes a new piece of data with which it updates the performance element and (iii) an accessing function which interprets the performance element to provide predictions, most importantly about data it has not seen.

The pieces of data which are presented are called, in this work, *exemplars*. For *SAB* learning they are triplets of data $(\mathbf{s}, \mathbf{a}, \mathbf{b})$ where $\mathbf{s} \in$ **State** is a perceived state, $\mathbf{a} \in$ **Action** is a raw action signal and $\mathbf{b} \in$ **Behaviour** is a perceived behaviour. Each is a vector of real numbers.

In the case of *SAB* learning there are actually two accessing functions which will prove useful. These are called *prediction* and *partial inversion*.

1. **Prediction:** Given a value $(\mathbf{s}, \mathbf{a}) \in$ **State** $\times$ **Action**, what is PSTF$(\mathbf{s}, \mathbf{a})$?
2. **Partial Inversion:** Given a value $\mathbf{s} \in$ **State** and a target value $\mathbf{b} \in$ **Behaviour**, what value of $\mathbf{a} \in$ **Action** (if any) will give PSTF$(\mathbf{s}, \mathbf{a}) = \mathbf{b}$?

The accessing function must *generalize*. This is what distinguishes a learning system from a rote-memorizing system, which is not feasible in a high dimensional or continuous control space

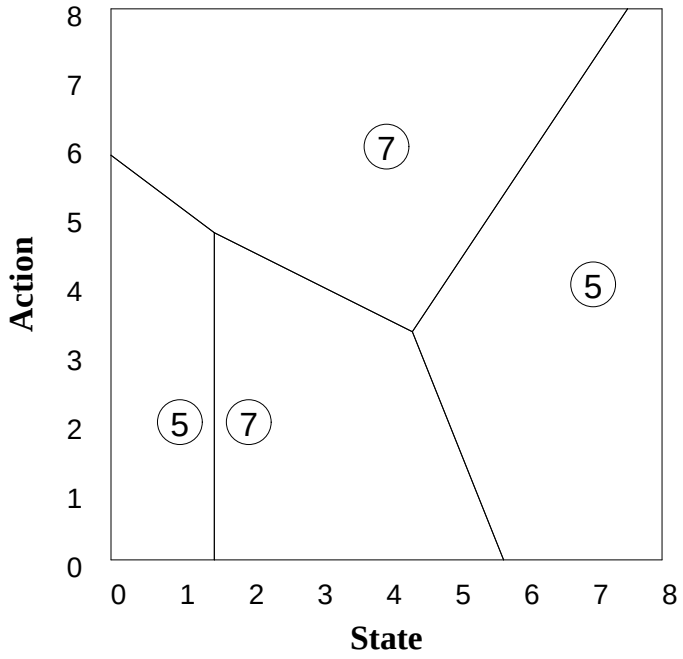

Figure 5.1

The nearest neighbour generalization predicting behaviour as a function of state and action.

because there is not enough time to have every possible experience even once. The type of generalization depends on the learning application, and in particular can vary in strength. A very strong generalization, for example, might be to assume that the underlying function being learned is linear. Then the generalization could consist of linear regression from all the data received so far. A strong generalization can learn very quickly, but only provided that the underlying assumption is accurate: otherwise learning will not occur.

The nearest neighbour lies at the other extreme: it is a very weak generalization, based only on the assumption that the function is generally continuous (this is formalized later in this chapter). For the nearest neighbour the performance element is very simple: it is the explicit set of all the exemplars which have been received. In the following text this set is called $\mathbf{E}$. After n observations it has value

$$\mathbf{E} = \{(\mathbf{s}_1, \mathbf{a}_1, \mathbf{b}_1), (\mathbf{s}_2, \mathbf{a}_2, \mathbf{b}_2), \ldots, (\mathbf{s}_n, \mathbf{a}_n, \mathbf{b}_n)\} \tag{5.1}$$

The update rule is also very simple: $\mathbf{E} := \mathbf{E} \cup \{(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})\}$.

The access rule uses the nearest neighbour generalization: To predict PSTF$(\mathbf{s}, \mathbf{a})$, find the $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i) \in \mathbf{E}$ for which $(\mathbf{s}_i, \mathbf{a}_i)$ is closest to $(\mathbf{s}, \mathbf{a})$. The predicted behaviour is $\mathbf{b}_i$. The notion of "closeness" is provided by the Euclidian distance metric, with the components of $\mathbf{s}_i$, $\mathbf{a}_i$ and $\mathbf{b}_i$ all scaled uniformly from their maximum ranges to the range $[0, 1]$. This is discussed shortly.

Let us consider an example. Figure 5.1 shows the interpretation after just four exemplars have arrived. In this example all spaces are one-dimensional and

$$\mathbf{E} = \{(1, 2, 5), (2, 2, 7), (4, 6, 7), (7, 4, 5)\}. \tag{5.2}$$

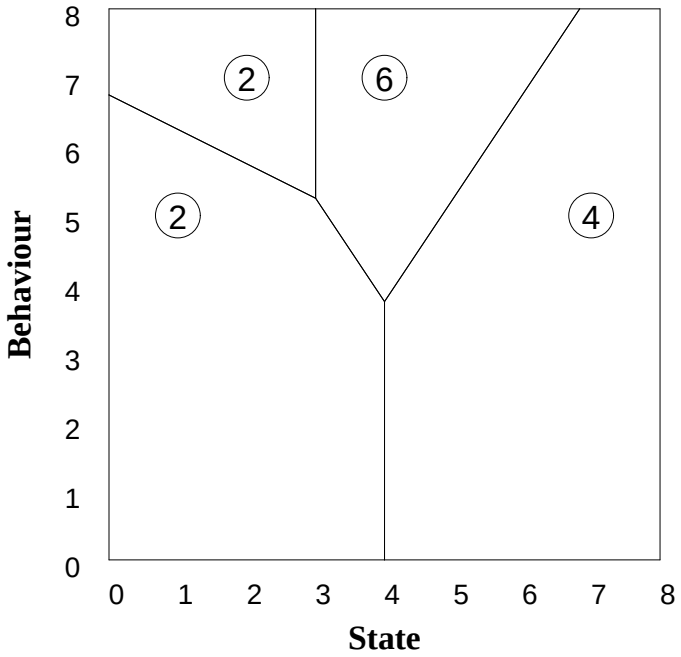

Figure 5.2

The nearest neighbour generalization performing partial inversion: the recommended action is derived as a function of state and desired behaviour.

The figure graphs "behaviour predicted" against "perceived state" on the horizontal axis and "action" on the vertical axis. For example, all the points in the polygon surrounding the leftmost '5' correspond to state-action pairs which will be predicted, according to the nearest neighbour generalization, to result in behaviour **5**.

Given $\mathbf{s}$ and a desired $\mathbf{b}$, partial inversion could be accomplished by finding a $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i) \in \mathbf{E}$ such that $\mathbf{s} = \mathbf{s}_i$ and $\mathbf{b} = \mathbf{b}_i$. However, in general such an exemplar will not be available. Instead, the exemplar with the nearest $(\mathbf{s}_i, \mathbf{b}_i)$ to $(\mathbf{s}, \mathbf{b})$ can be expected to have a good $\mathbf{a}_i$, *provided* that the predicted value of PSTF$(\mathbf{s}, \mathbf{a}_i)$ is $\mathbf{b}_i$. This proviso is important. Figure 5.2 provides an alternative view of the same exemplar set with "recommended action" plotted against "perceived state" on the horizontal axis and "required behaviour" on the vertical axis. First, let us consider an example where partial inversion is successful:

> If the current state is **0** and we require behaviour **5** then the nearest neighbour is the leftmost '2' exemplar. This recommends action **2**. Indeed, looking at the earlier prediction figure, this is a sensible recommendation because it predicts that applying action **2** in state **0** will give the desired behaviour **5**.

However partial inversion is not generally so successful:

> If we were in state **2** and required behaviour **5** then again, examining the lower diagram, we would be recommended action **2**. But if we examine the prediction we can see that we already know what happens if we apply action **2** in state **2**: we get behaviour **7**. Thus partial inversion used without prediction makes a serious mistake: and worse, it won't learn anything from the mistake, it will simply re-record the exemplar (2,2,7).

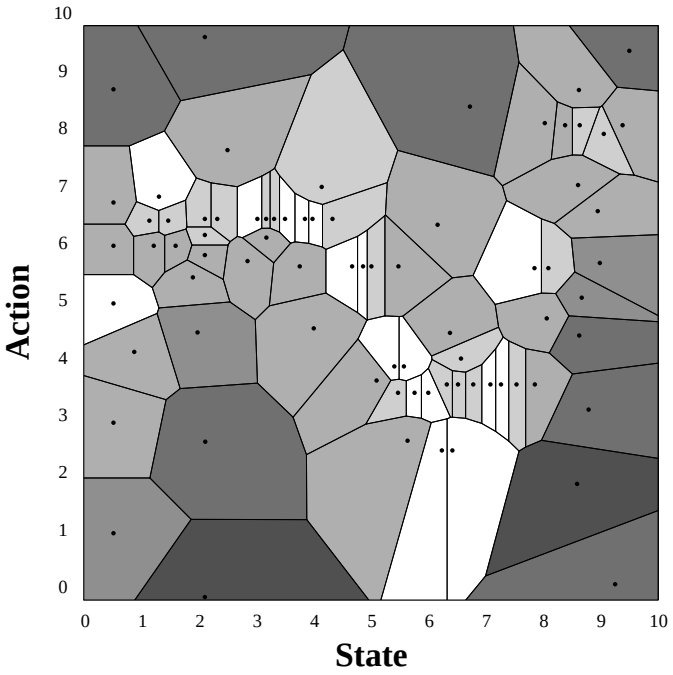

Figure 5.3

The nearest neighbour interpretation of the Mountain car PSTF after 100 experiences.

The conclusion is that partial inversion, which is closely related to associative memory lookup, is not necessarily valid for control choice using the nearest neighbour generalization (further, it seems likely that the same problem would occur for other generalizations). Chapter 8 will examine in depth how we should choose actions.

Figure 5.3 shows the approximation to the Mountain car's PSTF after approximately 100 experiences of the world. These are not random experiences—they were obtained using the SAB action chooser of Chapter 8. The agreement between the function values shown in this figure and the real PSTF depicted in Section 4.3 is generally good.

5.1.1 The Euclidian Metric

The Euclidian metric, L_2, defines the distance between two k-dimensional vectors $\mathbf{c}_1$ and $\mathbf{c}_2$ as

$$| \mathbf{c}_1 - \mathbf{c}_2 | = \sqrt{\sum_{i=1}^{k} ([\mathbf{c}_1]_i - [\mathbf{c}_2]_i)^2} \tag{5.3}$$

where $[\mathbf{c}]_i$ is the ith component of vector $\mathbf{c}$. The choice of a Euclidian metric is natural, because it allows a measure of closeness of states and actions which is independent of the direction of the axes. However, the choice of metric is not important providing the neighbourhoods are of a reasonable shape. Other possibilities include the L_∞ metric (maximum component difference) or L_1 metric (Manhattan distance).

The components of the vectors are scaled within their maximum ranges to provide equal weighting. In robotics, the ranges of both sensors and actuators are generally explicitly known to the system designer. If not they can be discovered. In the absence of further information the decision

to weight equally is arguably the most sensible. Empirical study in Chapter 10 shows that performance is not sensitive to the weighting. If the "fair" ranges were not known it *would* be possible to discover them from the data by *cross-validation*. This technique removes a small proportion (e.g. 10 %) of the exemplars and then tests itself by predicting their values using nearest neighbour lookup on the remaining exemplars. This is repeated for a variety of weight-sets, and the weight-set which results in the highest prediction accuracy is selected for use.

5.1.2 History of the Nearest Neighbour

The use of nearest neighbour for pattern classification was noted by [Duda and Hart, 1973] and [Friedman *et al.*, 1977; Bentley, 1980]. That it can also be used for prediction of real-valued functions has been noted by many investigators, for example [Cleveland, 1979; Omohundro, 1987; Kibler *et al.*, 1988; Farmer and Sidorowich, 1988]. It has generally been passed over in favour of local regression (described in Section 5.2) which is asymptotically more accurate, but less well suited to applications requiring high speed predictions.

The nearest neighbour can be found by means of a *k*d-tree representation in $O(\log N)$ time by an algorithm first proposed in [Friedman *et al.*, 1977]. This is described and evaluated in detail later in this thesis.

The use of fast nearest neighbour search has been recommended by [Omohundro, 1987] in a survey of a great variety of methods for efficient implementation of learned mappings and other learning behaviours. The $O(\log N)$ performance predicted by both the above references is somewhat optimistic in high dimensionality where, as has been pointed out by [Maclaren, 1989], the computational cost may be dominated by a very large constant term, exponential in the dimension. It is proved in [Preparata and Shamos, 1985] that nearest neighbour searching can be no faster than $O(\log N)$.

Omohundro also suggests that associative nearest neighbour search can be obtained by learning the function

$$\mathbf{In} \times \mathbf{Out} \rightarrow \{\text{True}\} \tag{5.4}$$

instead of

$$\mathbf{In} \rightarrow \mathbf{Out} \tag{5.5}$$

In [Maclaren, 1989] it is shown how a *k*d-tree-based nearest neighbour search can still be implemented in a non-Euclidian distance metric.

There have been a number of recent investigations which have actually used the nearest neighbour. For learning text to phoneme classification it has been studied by [Stanfill and Waltz, 1986], and for classifying objects according to shape it has been used by [Gottschalk *et al.*, 1989]. For general classification it has been investigated by [Salzberg, 1988] and [Aha *et al.*, 1990]. For learning robot control, it has been used by [Atkeson and Reinkensmeyer, 1989; Atkeson, 1989] (described in

Section 3.4) and also in the investigations described in this thesis (see also [Clocksin and Moore, 1989; Moore, 1990]).

Salzberg extends nearest neighbour by trying to reduce the number of exemplars stored. This is achieved by only storing exemplars which had been incorrectly predicted prior to observation. The advantages of reducing the number of exemplars are the decreased memory cost, search cost and also a more concise, and thus more "explanatory", mapping representation. The disadvantage is a reduction in classification accuracy. Salzberg also adds to the classification regions caused by the nearest neighbour generalization with hyper-rectangular classification regions. He also discusses an important feature of nearest neighbour learning—it is *one-shot*: a concept can be learnt with only one presentation of an item of data.

The work of [Aha *et al.*, 1990] also modifies the nearest neighbour generalization with a similar feature to reduce the number of points stored. It also shows that for pattern classification with noisy exemplars this method can lead to poor predictions, and so supplements the algorithm with a mechanism for detecting and removing noisy exemplars. The work also proves the important result that all mappings with finitely bounded classification regions are PAC-learnable under the nearest neighbour generalization. PAC-learnability is introduced in [Valiant, 1984]; see also Sections 5.4 and 5.3, which will make use of some of their results.

5.2 Alternative Generalizations

I will now examine some alternative methods of generalizing from observed data in order to learn a mapping **In** $\rightarrow$ **Out**. The set **In** is called the *domain* of the mapping and **Out** is the *range*. This investigation is strongly biased by the requirements for a robot learning system, discussed fully in Section 4.4: the learning should be fast, cheap and with the ability to adapt to different resolutions of interest.

The various representations originate from three families. **Neural** representations are inspired by the behaviour of the nervous systems of biological organisms. **Statistical** representations have also been used. Finally, **Algorithmic** methods have evolved from mainstream computer science. These labels are arbitrary and several representations could be argued to lie in more than one field, but the distinctions are useful because they reflect different motivations.

Another, orthogonal, distinction is whether the mapping uses real valued, numeric-discrete or symbolic inputs and outputs. For learning control we will generally be most interested in the former, but some representations described below can only use discrete or symbolic variables. A learning system in which the output is symbolic is generally referred to as a *classifier*.

In this section the following mapping learning methods will be considered:

- **Array** *(Algorithmic)*
- **Global Polynomial Regression** *(Statistical)*

- **Local Regression** *(Statistical)*
- **Perceptron** *(Neural classifier)*
- **Multiple Layer Neural Network** *(Neural classifier)*
- **CMAC** *(Neural)*
- **BACON** *(Algorithmic)*
- **Decision Trees** *(Algorithmic classifier)*
- *Q* **Nearest Neighbours** *(Algorithmic)*
- **Shepard's Interpolation** *(Algorithmic)*
- **Radial Basis Functions** *(Algorithmic)*
- **Genetic Classifiers** *(Algorithmic classifier)*

At the end of the section there is a comparative summary.

5.2.1 The Multidimensional Array

Real valued components of the domain vectors are quantized to bounded ranges of integers. The cell (or box) corresponding to each region of the domain typically contains statistics compiled from all the experiences in that region. In [Michie and Chambers, 1968] each cell contains the expected life-time and number of times each action was tried. In [Raibert, 1978b] it contains the average parameters of the local simplified (assumed constant state) arm dynamics. In [Christiansen *et al.*, 1990] it contains the statistics of probability distributions of state transitions.

The representation is computationally cheap because update is a simple array update and access is a simple array access. This has the advantage of simplicity of implementation but has several problems. First is the enormous memory needed if the control space is to have more than five or so dimensions and the variables be quantized to more than ten or so levels. Secondly there is the problem of generalization: if there are a very large number of partitions then to learn the whole state space it is necessary to record at least one value from each of these partitions. Finally, we have no ability to concentrate on interesting areas of the control space, or ignore uninteresting areas.

5.2.2 Polynomial Regression

This is a *parametric* method for generalizing functions, which means that it starts with an assumption that the function which we are approximating is a member of a family of functions which can be characterised by a finite number of parameters. A simple example of such a family of functions are the set of linear functions $\Re \rightarrow \Re$. Each function in this family is of the form $y(x) = mx + c$

and so can be characterized by two parameters m and c. The generalization from the data is to find the member of the family which best fits the data. The definition of "best fit" is typically the fit which minimizes the squares of the errors at the exemplars. For simple families this definition of best fit can lead to closed form solutions to obtain the best fit parameters from the data.

One such simple family is the polynomials of degree N for small N. A polynomial of degree N from $\Re^{k_d} \rightarrow \Re^{k_r}$ has

$$P(N, k_d, k_r) = k_r \begin{pmatrix} k_d + N \\ k_d \end{pmatrix} \tag{5.6}$$

parameters, which grows very fast with N and k_d. Would this be a suitable generalizing method? If the underlying function were a polynomial in the family then learning would require $P(N, k_d, k_r)$ observations before the function were learned perfectly. If the observations were noisy then the learning would still be robust, provided the number of observations significantly exceeds the number of parameters in the polynomial. The problems, however, are severe: for a robotic system the assumption that the underlying function is a polynomial is certainly wrong and so we can only hope that the best fit is good enough. This means large errors are likely in some areas of state space, and repeated data observations will not improve this. Furthermore, there is no opportunity to vary resolutions of interest. Update and access involve large matrix multiplications and inversions.

5.2.3 Local Regression

This is a method which explicitly remembers and uses all the data points which are presented to it. Learning consists merely of recording a set of observed data points $\{(x_1, y_1), \ldots (x_n, y_n)\}$. The method is based on the idea that locally a smooth function behaves like a low degree polynomial. Thus, to find the value of a point which has not been experienced, the local function is approximated by using those data points which are close to the query point. These local points have a low degree polynomial fitted to them. This scheme has been suggested in a variety of ways by a number of researchers:

- **Cleveland and Delvin's locally weighted regression.** It is suggested by [Cleveland, 1979; Cleveland and Delvin, 1988] who are particularly concerned with *noisy* data points. Each explicitly stored data point is assumed to be of the form

  $$y_i = g(x_i) + \epsilon_i \tag{5.7}$$

  where ϵ_i is the noise signal with mean zero and g is a smooth function. To evaluate the smoothed value at x_i the method uses the nearest N_0 neighbours, which are then fitted to a low degree (e.g. degree one) polynomial. There are extra mechanisms to bias the contribution of nearby local points more strongly than further ones, and to remove distortions caused by anomalous outlying data points.

- **Grosse's LOESS.** A more recent extension of this is [Grosse, 1989] which is concerned with making the calculation more computationally tractable, even in high dimensions. However, the calculation is optimized for a fixed set of points, and no cheap way is suggested for adding individual new pieces of data.

- **Omohundro's "efficient neural net behaviour".** This is a slightly simpler scheme, proposed in a major survey of computational techniques for classification and regression [Omohundro, 1987]. For a k-dimensional domain, the $k+1$ nearest neighbours of a query point are obtained, and a linear model based on these neighbours is then used. The $k+1$ nearest neighbours can be obtained fairly quickly, using a kd-tree, described in detail in Chapter 6.

- **Other suggestions and applications** Similar schemes are proposed by [Kibler *et al.*, 1988] and [Farmer and Sidorowich, 1988]. Such local interpolation schemes have been used by [Clocksin and Moore, 1989] in learning manipulator perception and kinematics and [Atkeson, 1989] in learning dynamic manipulator trajectories. The latter reference also provides a more thorough overview of further literature in the field of local regression.

Possible advantages of this method are extreme accuracy with relatively few exemplars. The problem is the expense of finding suitable local exemplars. It is clear that at least $k_d + 1$ are needed, where k_d is the dimensionality of the function domain. Unless the system is to be extremely vulnerable to noise, many more are required. It is also necessary to find local points which *surround* the query point, otherwise the process is extrapolation instead of interpolation, which can lead to large errors. The regression needs the exemplars to be local before accurate interpolation is obtained. This produces a slower *initial* rate of learning than simple nearest neighbour.

Local interpolation is sufficiently expensive that our implementation for kinematics experiments had to be performed off-line from the robot task execution, which prevents learning occurring *during* activity. The work of [Atkeson, 1989] also performed interpolation off-line. This work used a repetitive trajectory, meaning local data points were accumulated around a one dimensional path through state space, which allows the local interpolation to be particularly effective.

5.2.4 Perceptrons

This, and the following two mapping learners are popular connectionist methods which have been used in successful learning control experiments. There is a very great literature of alternative neural network architectures. It should be remembered that neural networks are also used for purposes other than learning mappings such as dimension reduction, associative memory and probability equalization.

Perceptrons were described extensively in the book [Minsky and Papert, 1969]. They provide a simple way of learning classification mappings from real valued vectors to a finite set of symbolic

values. It is an iterative learning method, in which internal weights are adjusted to reduce classification errors for each new input. If the classification regions are separable by a hyperplane then the mapping can be learned. Perceptrons can also be used to learn linear real-valued functions. The linearity requirement renders perceptrons unsuitable for general robot learning.

5.2.5 Multiple-layer Networks

These were popularized by [Rumelhart and McClelland, 1984]. They perform the same function as perceptrons but can learn non-linear discriminations. Given a fixed number of inputs and outputs there are a wide range of possible network configurations from which to choose. A configuration consists of N nodes linked together by M connections. Each connection (and occasionally nodes also) have a weight. Each node has a value associated with it—its *activation energy*. When an input is presented to the net, a selected subset of the nodes, called *input nodes* have their activation energies set according to the input. Adjacent nodes are affected according to combinations of the activations of their neighbours. According to these local rules the activation energies of all nodes change until the activations of another distinguished set of nodes, the *output nodes*, have been evaluated. This is the output of the system. During learning, when the wrong output is produced from the given input the weights can be adjusted to reduce the error on future similar presentations. One popular algorithm to do this is *back propagation* which can be shown to provide a local best fit of the weights to the data. "Best" is used here in the sense of minimizing the least squares prediction error [Angus, 1989].

Neural nets are another example of a parametric method (the parameters are the connection strengths) and so, like low degree polynomials, suffer from the danger that given the net configuration no solution could possibly represent the mapping. The update and access of neural nets is cheap—in each case it is no worse than proportional to the number of weights. As well as needing to hope the best fit is good enough, the disadvantages include slow learning times. Slow learning occurs because a piece of data may have to be presented many times to direct the weights to a set of values which accurately predicts the data. There are also reported cases of the weight adjusting method becoming stuck.

5.2.6 CMAC

The Cerebellar Model Articulation Controller [Albus, 1975a; Albus, 1975b; Albus, 1981] learns real-valued mappings. Its functionality is similar to a multi-dimensional array, but with the following advantages:

- Updates are distributed over a neighbourhood to provide a generalization.
- Array cells are hashed to reduce storage requirements.

CMAC relies on the assumption that behaviours within a neighbourhood will tend to be similar, that is that the function is smooth. This is a far less restrictive assumption than that used for the parametric methods.

The functional behaviour of CMAC is similar to that obtained by radial basis functions, described below. In Section 11.1 of this dissertation the similarity is explained and there is a detailed description of how CMAC's behaviour can be efficiently captured using an algorithmic method similar to that used in this thesis.

5.2.7 BACON

BACON is a method in which the space of simple algebraic formulae is searched in order to find the algebraic formulae which best fits the data [Langley *et al.*, 1983]. This uses a heuristic search, in which small sub-formulae which display some regularity in some contexts, are combined until larger regularities are discovered. This has been very successful in learning some elementary laws of Physics and Chemistry.

It is necessary that there exists a simple formula to model the data. For our purposes this seems unlikely. The behaviour of real robotic systems is complex, and much complexity is derived from components which are far from mathematically ideal. Even the linear analytic models with a great deal of assumed simplicity have a complex closed form, indicating that it would be difficult for a BACON-like search to find the correct formula. Another problem for our purposes is that it is non-incremental: there is no clear way of modifying the current theory as extra pieces of information arrive. However, for any non-incremental scheme, an incremental version is possible: simply remember all the data explicitly and have a continually running background process which processes all data so far. When it finishes executing it replaces the current performance element with the new one it has discovered and immediately starts executing again on the old data and the new data which arrived during the previous execution.

5.2.8 Decision Trees

These are classifiers, which learn functions from multi-dimensional numeric or binary domains to symbolic classifications. A well known example is [Quinlan, 1983]. The input domain is recursively partitioned. At each level the partition is binary, and splits the domain into the two subspaces in as informative a manner as possible. For example, a function of two binary variables, in which the result was greatly correlated with the first, and not correlated with the second, would be split in the first variable. The partitioning algorithm is formalized using the theory of information content. The notion behind decision trees is that they tend to find regular patterns in data. The classification regions correspond to rectangular hyperregions of the domain space. In their simplest form decision trees are not incremental, but recent work by [Utgoff, 1989] rectifies this.

The CART (Classification and Regression Trees) system [Breiman *et al.*, 1984] performs a

similar function to both Local Regression and Decision Trees.

5.2.9 Q Nearest Neighbours

An extension to using the value associated with the nearest neighbour is to process the values associated with some small number of the nearest neighbours. Its advantage is protection against noisy data. This has been suggested in a wide variety of places, for example [Duda and Hart, 1973]. The most common suggestions are

- Take a vote among the values of the Q nearest neighbours, and use the most common. This is only appropriate for symbolic valued mappings.
- To use the median value of the neighbours. This is particularly appropriate for discrete-numeric values. It protects against occasional wildly inaccurate exemplars.
- To use the mean value of the neighbours. This is appropriate for numeric domains and provides some degree of protection against low-amplitude noise.

In each case, the learning rate is slowed somewhat in comparison to the rate of single nearest neighbour, because predictions continue to be influenced by non-local exemplars for a long time. The search for the Q nearest neighbours also takes significantly longer than for the single nearest neighbour.

5.2.10 Shepard's Interpolation

This method, described among others in a survey paper [Franke, 1982], records all data points explicitly, and when a query point is supplied, a weighted average of all the stored points is returned as the interpolated value. The contributions to the average are weighted so that stored points close to the query point make more of a contribution than those far from it. A typical weighting function is to have the weight of a stored point inversely proportional to the distance between the stored point and the query point. This method is depicted in Figure 5.4 in which three data points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ are stored, and Shepard's Interpolation (SI) for all other x coordinates gives

$$y_{\text{shep}}(x) = \frac{\sum_{i=1}^{3} (x - x_i)^{-1} y_i}{\sum_{i=1}^{3} (x - x_i)^{-1}} \tag{5.8}$$

For comparison, the interpolation produced by the nearest neighbour is also given.

Shepard's interpolation was used in [Connell and Utgoff, 1987] to learn an evaluation function for the pole balancing problem. The method has the problem of expensive access, because the contribution of every exemplar yet experienced needs to be included. Another disadvantage is that

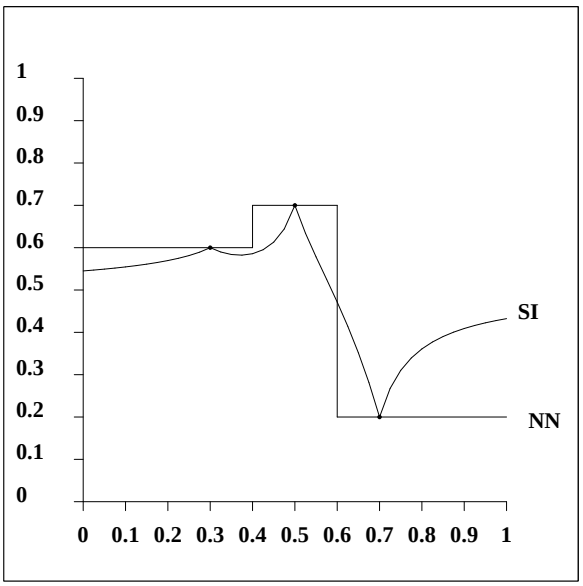

Figure 5.4

Interpreting a one-dimensional function from three data points ($0.3 \rightarrow 0.6$), ($0.5 \rightarrow 0.7$) and ($0.7 \rightarrow 0.2$). This diagram compares the nearest neighbour (NN) generalization with a Shepard's Interpolation (SI) generalization.

it does not generalize effectively at a large distance from any exemplars. Instead the interpolation tends to the average of all observed points.

5.2.11 Radial Basis Functions

A similar proposal to Shepard's interpolation is the use of radial basis functions, also described and evaluated in [Franke, 1982]. In this case contributions from all the exemplars are added together, instead of being averaged. Given a set of exemplars $\mathbf{E} = \{(x_1, y_1), \ldots (x_N, y_N)\}$ the prediction is defined as

$$y_{\text{rbf}}(x) = \sum_{i=1}^{N} A_i G(\mid x - x_i \mid) \tag{5.9}$$

The values $\{A_1, \ldots, A_N\}$ are weights which are computed from the exemplar set $\mathbf{E}$. A typical choice of the function G is the normal curve $G(x) = \exp(-x^2/c^2)$ for some constant c. The values of the A_i's are chosen to minimize some measure of error. This measure is typically the combination of the sum squared errors at the known exemplars and the "unsmoothness" of the resulting function (defined as the mean square second derivative of y_{rbf}) [Poggio and Girosi, 1989]. The computation of the A_i values makes update expensive, increasing linearly with the number of exemplars. In [Poggio and Girosi, 1989] a neural net based method is designed which has interpolative behaviour approximating that of Radial Basis Functions while only requiring a fixed amount of computation.

5.2.12 Genetic Classifiers

Genetic Algorithms [Holland *et al.*, 1987] are a recent optimization method in which spaces can be searched in a novel fashion which has been demonstrated to be particularly robust against capture by local minima. The search is most commonly in the space of fixed length strings of bits, but other syntactic domains can be used. The method keeps a fixed size population of partial solutions which are crossed over and mutated on the basis of their performance. This is analogous to some theories of biological evolution of genomes.

Among the many things which can be optimized are populations of *classifiers*. A classifier system is a set of *rules* which between them process an input vector to produce an output vector. The thing which is optimized is the accuracy of the set of rules. The optimization method consists of computing which rules should have credit for good classifications and which should be blamed for bad classifications. This computation is achieved by Holland's *bucket brigade* algorithm, described in the same paper. Individual rules tend to survive and be crossed-over if they are associated with successful predictions.

The power of genetic algorithm based optimization has been observed for a variety of domains (for example [Whitley, 1989; Gordon and Grefenstette, 1990]) but is not well understood. The vocabulary of the classifiers themselves is restrictive and more suited to discrete multi-input domains.

5.2.13 Summary

The mapping learning methods are shown in Table 5.1, which provides a rather terse description of the important features. The table is not meant to constitute an absolute judgement about the methods. Instead it provides a guide to how well each feature of each method was judged to suit the aims described earlier in this dissertation. An important detail is that the "Accuracy" field describes the expected accuracy, assuming that the function which we are attempting to learn is indeed learnable by the method under consideration.

Nearest neighbour is comparatively simple, and this is what permits real time access. Despite its simplicity it was selected as a robust learning method which best meets the goals for *SAB* learning. This decision is further justified in Sections 5.3—5.5 in which some remaining problems with nearest neighbour are also discussed.

5.3 The Class of Learnable Functions

Nearest neighbour is very general. It makes only a weak assumption about the mapping being learned. This can be described informally as the assumption that it is generally continuous. Here are examples of functions which will be seen to be nearest-neighbour learnable:

1. A continuous function over a closed interval.

	Accu-racy	Learning Rate	Gener-ality	Variable Resol'n	Noise Resist.	Update Cost	Access Cost	Memory Use
Array	Fair	Bad	Good	Bad	Fair	VG	VG	VB
Global Regr'n	Good	Good	Bad	Bad	VG	Poor	Poor	Good
Local Regr'n	Good	VG	VG	Good	Bad	Good	VB	Poor
Perceptron	Good	Fair	VB	Bad	VG	Good	Good	Good
Neural Net	Good	Bad	Poor	Poor	VG	Fair	Fair	Good
CMAC	Fair	Fair	Good	Poor	Fair	Poor	Poor	Fair
BACON	VG	Fair	Bad	Fair	Good	Bad	Good	Poor
Dec'n Tree	Good	Good	Fair	VG	Poor	Poor	Good	Poor
Q Nearest N'b'r	Fair	Fair	Good	Good	Fair	Good	Bad	Poor
Shepard Interp	Good	Good	Fair	Poor	Fair	Good	VB	Poor
Radial Basis	Good	Good	Fair	Fair	Fair	Good	VB	Poor
Genetic Class'r	Fair	Poor	Fair	Poor	Good	Poor	Fair	Fair
Nearest N'b'r	Fair	Good	VG	VG	Poor	Good	Poor	Poor

Table 5.1: Brief summary of a variety of generalizations discussed in the text.

2. A function with one or more asymptotes.

3. A piecewise uniformly continuous function with finite discontinuity.

In this section it is proved that nearest neighbour can achieve this, using the style of proof of [Aha *et al.*, 1990]. Let the mapping to be learned be $f : [0,1]^k \rightarrow \Re$. I shall define a broad class of function which includes all three given above. It will include functions which are generally continuous and generally smooth, but not entirely so. The rest of this section uses elementary, but somewhat detailed, mathematics and may be omitted if the reader is prepared to simply accept these claims of generality. It should also be noted that this discussion, for simplicity, will assume that exemplars and query points are drawn from a uniform probability distribution. In fact, the results will still hold for any other distribution.

5.3.1 Generally Continuous Functions

Given a function $f : [0,1]^k \rightarrow \Re$, and a value $\epsilon > 0$, define the *continuous domain* $D_\epsilon \subseteq [0,1]^k$ as the set of points in $[0,1]^k$ which are greater than or equal to distance ϵ from any discontinuity of the function f. Notice that from this definition the function f is continuous over all closed connected regions of D_ϵ. Define f to be *generally continuous* (GC) if

1.
$$\text{As } \epsilon \rightarrow 0 \text{ then } \mid D_\epsilon \mid \rightarrow 1 \tag{5.10}$$

where $\mid D_\epsilon \mid$ is the volume of the set D_ϵ (more formally, $\mid D_\epsilon \mid = \int_{D_\epsilon} 1 dx$).

2. The partition of $[0,1]^k$ induced by the discontinuities contains a finite number of disjoint regions.

Informally, a function is generally continuous if the portion of space "close" to a discontinuity diminishes to zero as the notion of "closeness" tends to zero. It can be shown that the three classes of function mentioned above are all generally continuous.

The set of GC functions is very large, and safely includes all analytic functions which have been used to model robot perspective, kinematics or dynamics, both quantitatively and qualitatively. It is not easy to imagine functions which are not GC, but one example is

$$f(x) = \begin{cases} 0 & \text{if } x \text{ is rational} \\ 1 & \text{if } x \text{ is irrational} \end{cases} \tag{5.11}$$

5.3.2 Nearest Neighbour can Learn Generally Continuous Functions

Any piecewise continuous mapping which satisfies the definition of GC can almost all, almost surely, be learned by nearest neighbour to any required accuracy. This statement can also be formalized in the manner of Valiant's work:

> Imagine we perform nearest neighbour by drawing N random exemplars. Then for any probability $\alpha > 0$ and any proportion $\beta > 0$ and any accuracy $\theta > 0$ we can find an N such that:
>
> With probability greater than $(1 - \alpha)$
> For all points x in the domain except a proportion β
> The nearest neighbour prediction of $f(x)$ differs from the
> value of $f(x)$ by less than θ.

This is proved using a result called the *coverage lemma* which we do not prove here. The coverage lemma is stated as follows [Aha *et al.*, 1990]:

> **The Coverage Lemma.** For any $\epsilon > 0$, $\alpha > 0$ and $\gamma > 0$, if you draw
>
> $$N(\epsilon, \gamma, \alpha) = \left(\left\lceil \sqrt{k}/\epsilon \right\rceil^k / \gamma \right) \times \log \left(\left\lceil \sqrt{k}/\epsilon \right\rceil^k / \alpha \right) \tag{5.12}$$
>
> k-dimensional exemplars from any fixed probability distribution, then with probability greater than $(1-\alpha)$ all but proportion less than γ of the domain will lie within distance ϵ of an exemplar.

A proof of the coverage lemma for $k = 2$ is given in [Aha *et al.*, 1990]. The learnability of nearest neighbour can now be proved. We need to achieve accuracy θ on all but a proportion β of the domain. To achieve this we will begin by fixing $\epsilon_1 > 0$ to create some continuous domain D_{ϵ_1} (later

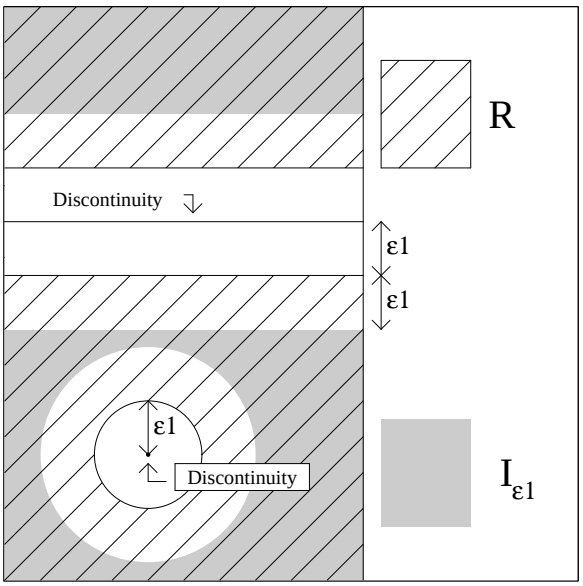

Figure 5.5

A two-dimensional domain with two discontinuities: one is along a horizontal line and the other is at a single point. The R regions are shown by cross hatching and the I_{ϵ_1} regions by gray shading.

on we will find a suitable value of ϵ_1 for our purposes). D_{ϵ_1} is a set of one or more disjoint regions, none of which have discontinuities within the region or at the edge of the region. It is a standard result of mathematical analysis [Burkhill, 1978] that a function which is continuous within such a closed interval R is *uniformly continuous* within R:

$$\begin{aligned} &\forall \theta > 0 \quad \exists \tau > 0 : \\ &\qquad (\forall x, x' \in R \quad | \, x - x' \, | < \tau \;\Rightarrow | \, f(x) - f(x') \, | < \theta) \end{aligned} \tag{5.13}$$

So for each region R in D_{ϵ_1}, let τ_R be the value necessary to ensure that any points within distance τ_R of each other differ in function value by at most θ. We can choose a non zero value of ϵ which is less than τ_R for all regions (it is guaranteed non-zero because there are only a finite number of regions). We can also ensure that $\epsilon < \epsilon_1$. Thus if any two points of R are within distance ϵ then their values differ by less than θ.

For each region R let I_{ϵ_1} be an *internal connected region*: the set of points in R which are not within distance ϵ_1 of the edge of R.

$$I_{\epsilon_1} = \{x \in R : \forall x' \in [0,1]^k, | \, x - x' \, | \leq \epsilon_1 \Rightarrow x' \in R\} \tag{5.14}$$

Figure 5.5 provides an illustration. The domain has two discontinuities: one is along a horizontal line and the other is at a single point. There are two disjoint R regions (sets of points further than ϵ_1 from a discontinuity) and thus two disjoint I_{ϵ_1} regions: each a subset of an R region.

From the definition of I_{ϵ_1}, if $x \in I_{\epsilon_1}$ then, for x' within distance ϵ of x, x' must be in the same region R (because it is closer than ϵ which has been defined to be less than ϵ_1), and so $| \, f(x) - f(x') \, | < \theta$.

We would like to take enough exemplars that we can ensure that all points in D_{ϵ_1} lie within distance ϵ of an exemplar, which would mean that for any point in any internal connected region I_{ϵ_1}, the nearest neighbour would have a value which differed by less than θ. Unfortunately, for any number of exemplars it is not guaranteed that all points lie within distance ϵ of an exemplar. However, by the coverage lemma we *can* ensure that with probability $1-\alpha$, for all but proportion $\gamma = \frac{1}{2}\beta$, all points will lie within distance ϵ.

Now a suitable value of ϵ_1 can be chosen. It should be chosen to be small enough that all but proportion $\frac{1}{2}\beta$ of points lie within an internal connected region I_{ϵ_1} (the gray areas on the diagram). We are guaranteed that such an ϵ_1 exists because

$$\lim_{\epsilon_1 \to 0} \mid I_{\epsilon_1} \mid = \lim_{\epsilon_1 \to 0} \mid D_{2\epsilon_1} \mid = 1 \tag{5.15}$$

The necessary value of ϵ_1 is thus determined by β and the value of ϵ is determined by ϵ_1 and θ. This is adequate to complete the proof, because now, with probability $1-\alpha$, all but proportion $\frac{1}{2}\beta$ of points within internal connected regions are predicted to accuracy better than θ by nearest neighbour. Furthermore, all but proportion $\frac{1}{2}\beta$ of all points in $[0,1]^k$ are in internal connected regions and so in total all but proportion β are sufficiently accurate.

5.4 The Accuracy of Learning

The previous section proved that a very wide range of functions could be learned to arbitrary tolerance with nearest neighbour. But a question which is crucial for practical use is how many exemplars are required to achieve the tolerance? Conversely, how does the error decrease as the number of exemplars increases? The number of exemplars required to learn, with probability $(1-\alpha)$, all but proportion β of the domain to accuracy θ depends directly on the values of α, β and θ and indirectly on the function f itself. If, for a given function this dependency is exponential in any of $\frac{1}{\alpha}$, $\frac{1}{\beta}$ or $\frac{1}{\theta}$ then it can be argued that the function is not practically learnable [Valiant, 1984]. In Appendix B, in a proof slightly modified from those in [Kibler *et al.*, 1988] and [Omohundro, 1987], it is shown why, for uniformly continuous functions, the number required is polynomial in all three values.

Appendix B shows that the number of exemplars $n(\alpha, \beta, \theta)$ required is

$$n = \left(\frac{2}{\beta} \left\lceil \frac{G\sqrt{k}}{\theta} \right\rceil^k \right) \times \log \left(\frac{1}{\alpha} \left\lceil \frac{G\sqrt{k}}{\theta} \right\rceil^k \right). \tag{5.16}$$

Thus when learning uniformly continuous functions the number of k-dimensional exemplars required is better than $(k+1)$-nomial in $\frac{1}{\theta}$, linear in $\frac{1}{\beta}$ and sublinear in $\frac{1}{\alpha}$.

Appendix B also outlines why most other generally continuous functions should be expected to be learnable in polynomial time. However, even if the learning rate is polynomial, it can still be asked if after a practical amount of time the generalization will be to sufficient accuracy. The

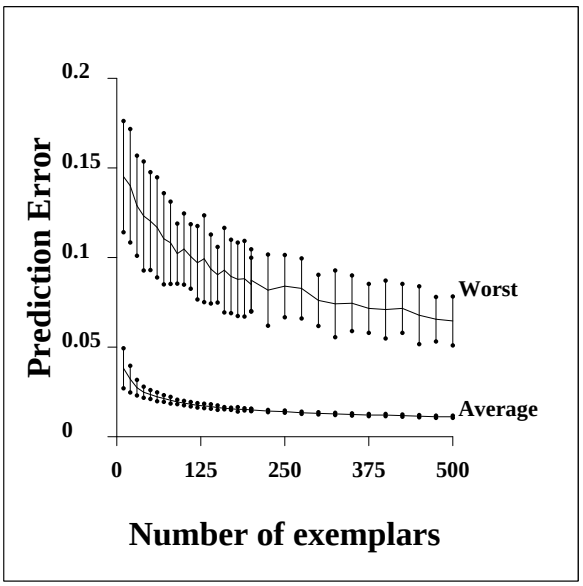

Figure 5.6

The worst case error and the average error plotted against the number of three-dimensional exemplars. These empirical trials were performed forty times; the vertical bars indicate the standard deviations of the data points (see Appendix A).

above analysis is worst case, and trying some numbers in Equation 5.16 indicates how pessimistic this worst case analysis is. Let us consider an example. The function f is three-dimensional and continuous with a maximum slope G of 1. We wish to learn the function over $[0,1]^3$. We are satisfied if, with probability 95%, over proportion 95% of the unit cube, nearest neighbour achieves an accuracy of $\frac{1}{3}$. Equation 5.16 tells us that the number of exemplars needed is no more than 72326. It should be immediately clear that this bound, although valid, is not close to the actual number one would expect to be needed. This feeling is supported by empirical evidence. The graphs in Figures 5.6 and 5.7 indicate that for one example of such a function,

$$f(x,y,z) = \frac{\sin(\pi x)\sin(\pi y)\sin(\pi z)}{\pi\sqrt{3}}, \tag{5.17}$$

the number of exemplars needed to achieve much better worst case accuracy over the whole domain is much less. These empirical tests were carried out for a variety of sizes of randomly generated exemplar sets. For each size, forty independent trials were run. Each trial estimated the average and worst case error by the average and worst case of 10,000 random nearest neighbour queries respectively. For a target accuracy of $\theta = 0.1$ the required number of exemplars is, in 95% of cases, less than 200 exemplars. Even this figure is grimmer than necessary, because we will be particularly interested in the *expected error*. In this case the empirical results indicate that only 10 exemplars are needed to bring the expected error to less than 0.1. Can formal analysis provide some measure of the expected error?

The answer is that it can, by estimating the expected distance to a nearest neighbour after n exemplars have been received. The following paragraphs review how this value is derived in terms of n and k, the dimensionality of the relation, under the assumption that the exemplars are

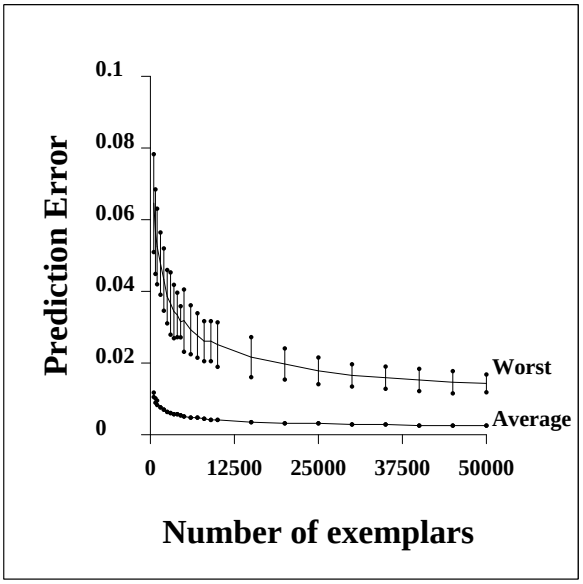

Figure 5.7

Plot of the same experiment as in Figure 5.6, but for larger exemplar sets.

distributed uniformly.

Imagine that we are looking for the nearest neighbour of a point at the centre of a k-dimensional sphere of radius 1 which contains n uniformly distributed random exemplars. We wish to evaluate $E^{n,k}$, the expected distance of the centre to the nearest neighbour. Write $Q^{n,k}(x)$ as the probability that all of the exemplars are further than distance x from the center of the sphere. The exemplars' locations are distributed independently, so

$$Q^{n,k}(x) = (\mathbf{Prob}(\text{Any one given exemplar is further than } x))^n \tag{5.18}$$

Because the distribution is uniform, the chance that one exemplar is further than x is the proportion of the sphere which is not in the smaller sphere radius x. This proportion is $1^k - x^k$. Thus $Q^{n,k} = (1 - x^k)^n$.

Let $P^{n,k}$ be the probability distribution function

$$\begin{aligned} P^{n,k}(x) &= \mathbf{Prob}(\text{Given } n \text{ } k\text{-dimensional exemplars, dist. of nearest} \leq x) \\ &= 1 - Q^{n,k}(x) \\ &= 1 - (1 - x^k)^n \end{aligned} \tag{5.19}$$

To derive the expected value of the nearest distance we need the probability density function

$$\rho^{n,k}(x) = \frac{dP^{n,k}}{dx} = nkx^{k-1}(1 - x^k)^{n-1} \tag{5.20}$$

The expected distance is

$$E^{n,k} = \int_0^1 x\rho^{n,k}(x)dx = nk\int_0^1 x^k(1 - x^k)^{n-1}dx \tag{5.21}$$

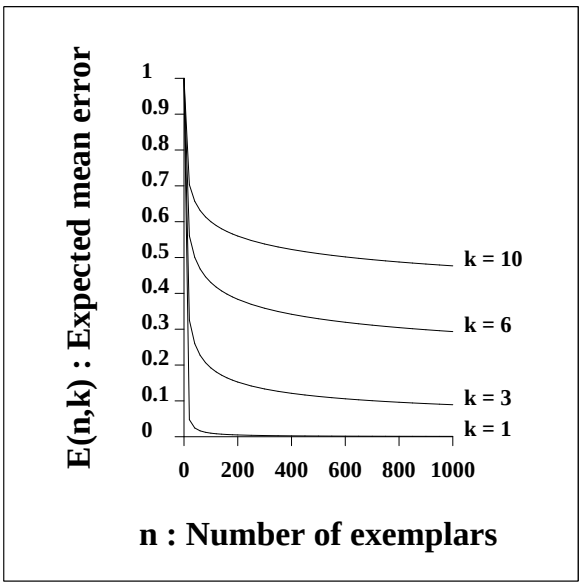

Figure 5.8

The expected accuracy of the nearest neighbour prediction for a uniformly continuous function with average slope 1. The dimensionality is k.

After performing this definite integration the result is

$$E^{n,k} = \frac{n!k^n}{(k+1)(2k+1)\cdots(nk+1)} \tag{5.22}$$

Let us assume that this estimate holds throughout the space $[0,1]^k$, rather than at the center of a sphere. This approximation becomes more accurate when there are more exemplars in the interval, because the probability of edge-effects is then reduced. Let us assume the function is uniformly continuous. The expected error in predicting the value of $f(x)$ depends on how far away the nearest exemplar is to x and on the magnitude of the slope G_x around point x. If we write $E^{n,k}_{\text{pred}}(x)$ for the expected prediction error at x, then we have $E^{n,k}_{\text{pred}}(x) = G_x E^{n,k}$. The expected error if we make a prediction of a random point chosen uniformly in the space $[0,1]^k$ is

$$A(n,k) = \int_{[0,1]^k} E^{n,k}_{\text{pred}}(x)dx = \int_{[0,1]^k} G_x E^{n,k} dx = E^{n,k} G_{avg} \tag{5.23}$$

Where G_{avg} is the average slope over the function. Assuming an average slope of 1, Figure 5.8 graphs the expected error based on this analysis.

It is clear that the error decreases, with a particularly fast initial improvement. It is also clear that large values of k have a devastating effect on performance. If we write $n(\frac{1}{\theta}, k)$ as the n required to achieve $A(n,k) = \theta$, then n is exponential in k, but for any fixed k can be shown to be polynomial in $\frac{1}{\theta}$.

$$n > \left(\frac{1}{\theta G_{avg}}\right)^{k+1} \Rightarrow A(n,k) < \theta \tag{5.24}$$

The low learning rate with large k would be a disaster if we were intending to try to learn everything. As noted earlier, we are assuming the data is received uniformly and accessed uniformly.

If the function f were k-dimensional, but the data were all obtained from a subspace S of dimension k_{distrib} then it is clear we are really learning a function $f_S : [0,1]^{k_{\text{distrib}}} \rightarrow \Re$. The subspace need not be a linear projection for this to remain true. It is by using this that SAB-learning will attempt to dodge the curse of dimensionality, because in practice SAB-learning will *never* be trying to learn the entire space uniformly, simply because there is never enough time to do so. Instead, it will tend to remain close to a low dimensional subspace of the control space. This low dimensional subspace might be 0,1,2 or 3 dimensional, but as we shall see later it need not generally be greater. Evidence for this, and examples of tasks where each of these underlying dimensionalities occur, will appear in Chapter 10.

With $k = 3$ or less, the accuracy is seen, both in the analysis and in the empirical experiment, to be adequate. The asymptotic convergence is slower than most of the other methods suggested in Section 5.2, but the other methods achieve greater speed by placing restrictions on the class of learnable functions which are far stronger than those for nearest-neighbour learnability. Furthermore, although they out-perform asymptotically, the merits of *initial* learning speed are still open to debate.

5.5 Nearest Neighbour: Discussion

Section 5.2 discussed some candidate generalizations which were considered as an alternative to nearest neighbour. This section describes in more detail why nearest neighbour was chosen, bearing in mind the performance criteria for SAB learning given in Section 4.4. Sections 5.3 and 5.4 discussed the generality and asymptotic learning speed of nearest neighbour. Here we will see that nearest neighbour indeed has further desirable features although it is not perfect. The acceptable imperfections will be identified. It will be the work of the two chapters following this one to overcome the other imperfections.

5.5.1 Initial Learning Speed

We shall see here that nearest neighbour is extremely quick at obtaining an initial rough model of important areas of the domain. There are three reasons:

1. For initial SAB-learning all that is needed is rough, qualitative control—very informally enough knowledge that the system's behaviour can be sent in the right direction instead of in the wrong direction. This means that the accuracy we need to attain rough control is perhaps a half or third of the maximum deviation over the function.

2. For a robot learning system the initial learning will all, necessarily, take place in a small region of the state space, although a large region of the action space might be explored. This is because there will not be time, in the first few moments of learning to travel very far from

the start state. So, if we can consider the state effectively constant at the start of learning, then instead of learning on a

$$[Dim(\mathbf{State}) + Dim(\mathbf{Action})] \tag{5.25}$$

-dimensional domain we are learning on a

$$[Dim(\mathbf{Action})] \tag{5.26}$$

-dimensional domain.

3. The rate of learning is relatively much higher at the start of learning than at any other point, especially for high dimensionality.

An example is the control of the two-jointed arm considered in Section 10.2. The control space is six-dimensional, but as learning begins, the state is roughly constant. This leaves a two-dimensional action space over which the exemplars can vary, so the learning rate for $k = 2$ applies. Imagine the expected slope is 1, and we require an initial accuracy of 30%. Considering Equation 5.23, the number of exemplars needed is n where

$$0.3 > \frac{n!2^n}{(2+1)(2\times 2+1)\cdots(2n+1)} \tag{5.27}$$

which is satisfied by $n = 8$. The results chapter will detail how good the initial learning was for an empirical investigation of this system.

The initial learning is very fast because the learning is *one-shot*. This is in contrast to a connectionist or regression method in which several (and in some cases very, very many) pieces of data are needed before any meaningful predictions can be made.

5.5.2 Variable Resolution

The nearest neighbour generalization can vary its resolutions of interest very well. This means that areas of the control space which are especially important to a task can be learned to high accuracy, and uninteresting areas to low accuracy. It has already been described in several different contexts why this is so desirable:

- (Mentioned in Section 3.3) A robot will wish to concentrate on particular tasks, instead of learning generally.
- (Mentioned in Section 5.2) There is not enough time to visit each general area of the control space even once.
- (Mentioned in the previous section) The number of exemplars needed to reach a fixed accuracy is exponential in the underlying dimensionality, k_{distrib}, of the data points.

- (Will be discussed in Section 6.5) The computational cost of nearest neighbour is too high if we search in a uniform distribution of data.

The importance of varying resolutions was noted by [Simons *et al.*, 1982]. This work used a multi-dimensional array as the performance element, but to adapt resolutions it recursively broke selected boxes of the array into higher resolution sub-arrays.

Of all the generalizing methods discussed in Section 5.2, nearest neighbour is the best at varying its resolution of interest, and this comes for free. No extra implementation effort is required.

Local regression and radial basis functions also vary their resolutions, but because much more local data than just one neighbour is needed, the "rate of increasing locality" is very much reduced.

5.5.3 Simplicity

One advantage of a learning method being conceptually simple is that we can have confidence in its being robust. Its behaviour can be understood by its users. Nearest neighbour is a particularly simple method. It is intuitively clear how it works, and it has been demonstrated here to be sufficiently simple to be amenable to analysis in only a dozen or so pages. Here I will detail both advantages of simplicity.

- **Robustness.** A learning system initially can be expected to make errors, but if it continues to make the same error then, informally, one could claim that it is not learning. This can happen to some parametric methods. A common reason would be that the class of functions being parameterized did not include the actual function being learned. For some iterative methods, such as *back-propagation* [Rumelhart and McClelland, 1984], a second reason is that the iteration can get stuck in a local minimum. For nearest neighbour learning of any generally continuous function, this cannot happen because the representation is derived directly, instead of indirectly through iteration. Depending on the implementation of local regression, this can also make some very large prediction errors in regions of large curvature, especially if the local distribution of points is not dense or if it accidently extrapolates.

- **Understandability.** There are many occasions when one needs to know *why* a learning method made a particular prediction, particularly if there has been a significant error. Nearest neighbour can provide a fairly direct, understandable answer:

  > *Sorry. You see, in this very similar previous situation that decision seemed to work, so I thought it probably would here too. I'll know better next time.*

  Other methods based on statistical or connectionist models can only give a far less direct explanation:

  > *Sorry. You see, this is the algorithm and this is the data received so far, and so the execution resulted in this decision.*

This comparison is somewhat unfair as some work is now being done to allow connectionist networks to explain their behaviour [Diederich, 1990].

In [Michie, 1989] it is argued that until learning systems can explain their decisions clearly they are unlikely to be accepted into commercial use. It should also be noted that the nearest neighbour explanation is inferior to that from a decision tree or from BACON, which can both give particularly succinct accounts of their current theory of the world.

5.5.4 Computational Resource Costs

There is a significant fear that demands on the computer will increase linearly with n, the number of exemplars. The resources which might be badly affected are (i) time to update, (ii) time to access and (iii) memory requirements. Before discussing these cases it is worth considering just how large n will become. It is constrained by the length of time in which the robot learns. For example an extreme learning task might last three days with five SAB-cycles per second, in which case n would be $1,296,000$ exemplars. But it is questionable whether would there be much point in continuing learning when n is in the millions. Each exemplar has, on average, only a one in a million chance of being accessed, and so there will not be much payoff from continued learning. This is borne out by Figure 5.9, which graphs the expected increase in accuracy obtained by doubling the current number of exemplars. It shows, for example, that if we have 100,000 three-dimensional exemplars from a function with average gradient 1, then taking a further 100,000 will decrease the expected error by approximately 0.01. For larger dimensionality there is a higher payoff in increasing the number of exemplars, but given the enormous time to double the number of exemplars, it will rarely seem worth it. However, instead of discontinuing learning, it is preferable to add new exemplars and delete old ones. This is also important to cope with a changing environment, and the topic is fully discussed in Section 7.3.

Thus, for these two reasons, we will not hold more than the order of a quarter of a million exemplars in store. We can now make an estimate of the amount of memory which will be required. Assuming two bytes per field in the state, action and behaviour vectors, and assuming a twelve-dimensional state space, six-dimensional control space and six-dimensional behaviour space, with an overhead of another 30 bytes per exemplar (Chapter 7 will explain overhead requirements), the memory requirement to store $\frac{1}{4}$ million exemplars would be $19\frac{1}{2}$ megabytes. This is an amount of memory well within the virtual memory of modest computers in 1990, and is already within the on-board memory of powerful machines. It must, however, be mentioned that most parametric methods require a magnitude less memory. Multi-dimensional arrays require a magnitude more.

There is also the question of the time for update and access. Chapter 6 describes an implementation to make this sufficiently small. For low degree parametric methods, update and access are also cheap, but nearest neighbour is faster than the other suggested methods.

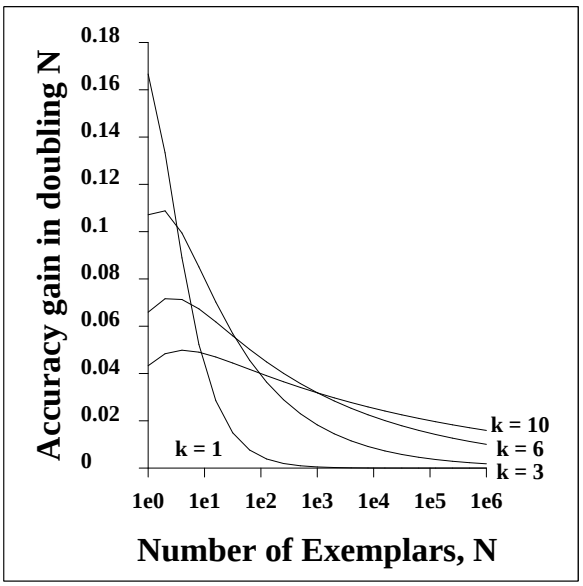

Figure 5.9

The horizontal (logarithmic) scale lists varying values of n, the number of exemplars. We plot the increase in accuracy, $\Delta\theta$, which would be obtained from doubling the number of exemplars.

$$\Delta\theta = E^{n,k} - E^{2n,k}$$

5.5.5 Disorder

I have not as yet mentioned a major advantage of parametric methods: they do not learn noise. Imagine we receive exemplars of the form

$$(\mathbf{c}_i + \boldsymbol{\delta c}_i \rightarrow f(\mathbf{c}_i) + \boldsymbol{\delta b}_i) \tag{5.28}$$

where $\boldsymbol{\delta c}_i$ and $\boldsymbol{\delta b}_i$ are noise signals distributed independently of i with mean zero. Parametric methods will, if they converge, converge to generalizations which give the correct, noise-free prediction. Nearest neighbour will do very badly: it will learn the noisy values exactly as they occur. Radial basis functions and a suitable implementation of local regression will cope, but they will need to include a large number of exemplars. In particular, local regression with only a few noisy exemplars is likely to have severe errors.

Chapter 7 explains how nearest neighbour is modified to average out noise, in a similar manner to Shepard's regression, but optimized for access efficiency. In doing so, it will be necessary to put an upper bound on the accuracy to which the function can be learned around discontinuities.

This investigation assumes that there is no information in the noise, or in the distribution of the noise: we simply want to get rid of it. This assumption is a consequence of the earlier assumption that the Perceived State Transition Function is deterministic. Some work has been done by [Christiansen *et al.*, 1990] on a robotic tray tilting system in which the observed world is non-deterministic and in which the distributions of observed behaviours need to be learned.

5.5.6 Knowing what one Knows

The nearest neighbour estimate can provide a measure of how accurate a given prediction is expected to be. This estimate is a function of the actual distance to the nearest neighbour. As we will discover later, this knowledge is important for deciding intelligently when experimentation is desirable. Knowledge of knowledge can also be obtained directly from an array representation, and somewhat less directly from the other non-parametric methods reviewed earlier. It cannot be obtained from a parametric representation, although it is possible that the system could attempt to learn knowledge of knowledge explicitly.

Chapter 6

Kd-trees for Cheap Learning

This chapter gives a specification of the nearest neighbour algorithm. It also gives both an informal and formal introduction to the kd-tree data structure. Then there is an explicit, detailed account of how the nearest neighbour search algorithm is implemented efficiently, which is followed by an empirical investigation into the algorithm's performance. Finally, there is discussion of some other algorithms related to the nearest neighbour search.

6.1 Nearest Neighbour Specification

Given two multi-dimensional spaces **Domain** = $\Re^{k_d}$ and **Range** = $\Re^{k_r}$, let an *exemplar* be a member of **Domain**$\times$**Range** and let an *exemplar-set* be a finite set of exemplars. Given an exemplar-set, $\mathbf{E}$, and a target domain vector, $\mathbf{d}$, then a *nearest neighbour* of $\mathbf{d}$ is any. exemplar $(\mathbf{d}', \mathbf{r}') \in \mathbf{E}$ such that *None-nearer*$(\mathbf{E}, \mathbf{d}, \mathbf{d}')$. Notice that there might be more than one suitable exemplar. This ambiguity captures the requirement that any nearest neighbour is adequate. *None-nearer* is defined thus:

$$\textit{None-nearer}(\mathbf{E}, \mathbf{d}, \mathbf{d}') \quad \Leftrightarrow \quad \forall(\mathbf{d}'', \mathbf{r}'') \in \mathbf{E} \quad | \mathbf{d} - \mathbf{d}' | \leq | \mathbf{d} - \mathbf{d}'' | \tag{6.1}$$

In Equation 6.1 the distance metric is Euclidean, though any other L_p-norm could have been used.

$$| \mathbf{d} - \mathbf{d}' | = \sqrt{\sum_{i=1}^{i=k_d} (d_i - d'_i)^2} \tag{6.2}$$

where d_i is the ith component of vector $\mathbf{d}$.

In the following sections I describe some algorithms to realize this abstract specification with the additional informal requirement that the computation time should be relatively short.

Algorithm:	Nearest Neighbour by Scanning.
Data Structures:	
domain-vector	A vector of k_d floating point numbers.
range-vector	A vector of k_r floating point numbers.
exemplar	A pair: (**domain-vector**, **range-vector**)
Input:	**exlist**, of type **list of exemplar**
	dom, of type **domain-vector**
Output:	**nearest**, of type **exemplar**
Preconditions:	**exlist** is not empty
Postconditions:	if **nearest** represents the exemplar $(\mathbf{d}', \mathbf{r}')$,
	and **exlist** represents the exemplar set $\mathbf{E}$,
	and **dom** represents the vector $\mathbf{d}$,
	then $(\mathbf{d}', \mathbf{r}') \in \mathbf{E}$ and *None-nearer*$(\mathbf{E}, \mathbf{d}, \mathbf{d}')$.
Code:	
1.	**nearest-dist** := infinity
2.	**nearest** := undefined
3.	for **ex** := each exemplar in **exlist**
3.1	**dist** := distance between **dom** and the domain of **ex**
3.2	if **dist** $<$ **nearest-dist** then
3.2.1	**nearest-dist** := **dist**
3.2.2	**nearest** := **ex**

Table 6.1: Finding Nearest Neighbour by scanning a list.

6.2 Naive Nearest Neighbour

This operation could be achieved by representing the exemplar-set as a list of exemplars. In Table 6.1, I give the trivial nearest neighbour algorithm which scans the entire list. This algorithm has time complexity $O(N)$ where N is the size of $\mathbf{E}$. By structuring the exemplar-set more intelligently, it is possible to avoid making a distance computation for every member.

6.3 Introduction to kd-trees

A *kd-tree* is a data structure for storing a finite set of points from a k-dimensional space. It was examined in detail by J. Bentley [Bentley, 1980; Friedman *et al.*, 1977]. Recently, S. Omohundro has recommended it in a survey of possible techniques to increase the speed of neural network learning [Omohundro, 1987].

A kd-tree is a binary tree. The contents of each node are depicted in Table 6.2. Here I provide an informal description of the structure and meaning of the tree, and in the following subsection I

Field Name:	Field Type	Description
dom-elt	**domain-vector**	A point from k_d-d space
range-elt	**range-vector**	A point from k_r-d space
split	**integer**	The splitting dimension
left	**kd-tree**	A kd-tree representing those points to the left of the splitting plane
right	**kd-tree**	A kd-tree representing those points to the right of the splitting plane

Table 6.2: The fields of a kd-tree node

give a formal definition of the invariants and semantics.

The exemplar-set **E** is represented by the set of nodes in the kd-tree, each node representing one exemplar. The **dom-elt** field represents the domain-vector of the exemplar and the **range-elt** field represents the range-vector. The **dom-elt** component is the index for the node. It splits the space into two subspaces according to the splitting hyperplane of the node. All the points in the "left" subspace are represented by the **left** subtree, and the points in the "right" subspace by the **right** subtree. The splitting hyperplane is a plane which passes through **dom-elt** and which is perpendicular to the direction specified by the **split** field. Let i be the value of the **split** field. Then a point is to the left of **dom-elt** if and only if its ith component is less than the ith component of **dom-elt**. The complimentary definition holds for the right field. If a node has no children, then the splitting hyperplane is not required.

Figure 6.1 demonstrates a kd-tree representation of the four **dom-elt** points $(2,5)$, $(3,8)$, $(6,3)$ and $(8,9)$. The root node, with **dom-elt** $(2,5)$ splits the plane in the y-axis into two subspaces. The point $(3,8)$ lies in the lower subspace, that is $\{(x,y) \mid y < 5\}$, and so is in the left subtree. Figure 6.2 shows how the nodes partition the plane.

6.3.1 Formal Specification of a kd-tree

The reader who is content with the informal description above can omit this section. I define a mapping

$$\textit{exset-rep} : k\text{d-tree} \rightarrow \text{exemplar-set} \tag{6.3}$$

which maps the tree to the exemplar-set it represents:

$$\begin{aligned} &\textit{exset-rep}(\text{empty}) &&= \phi \\ &\textit{exset-rep}(< \mathbf{d}, \mathbf{r}, -, \text{empty}, \text{empty} >) &&= \{(\mathbf{d}, \mathbf{r})\} \\ &\textit{exset-rep}(< \mathbf{d}, \mathbf{r}, \text{split}, \text{tree}_{\textit{left}}, \text{tree}_{\textit{right}} >) &&= \\ &\qquad \textit{exset-rep}(\text{tree}_{\textit{left}}) \cup \{(\mathbf{d}, \mathbf{r})\} \cup \textit{exset-rep}(\text{tree}_{\textit{right}}) \end{aligned} \tag{6.4}$$

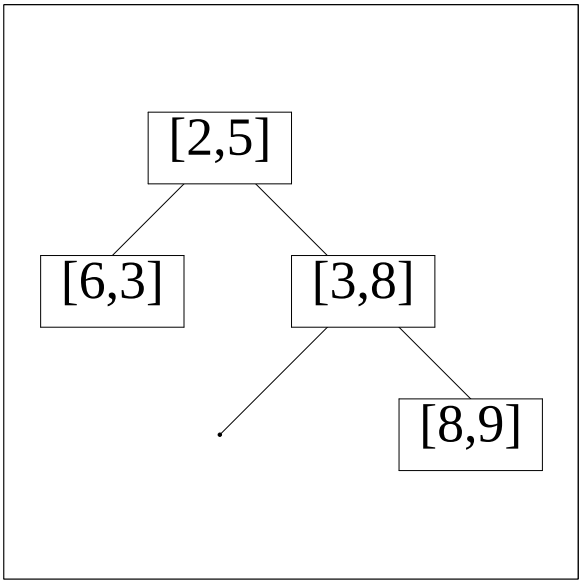

Figure 6.1

A 2d-tree of four elements. The splitting planes are not indicated. The [2,5] node splits along the y = 5 plane and the [3,8] node splits along the x = 3 plane.

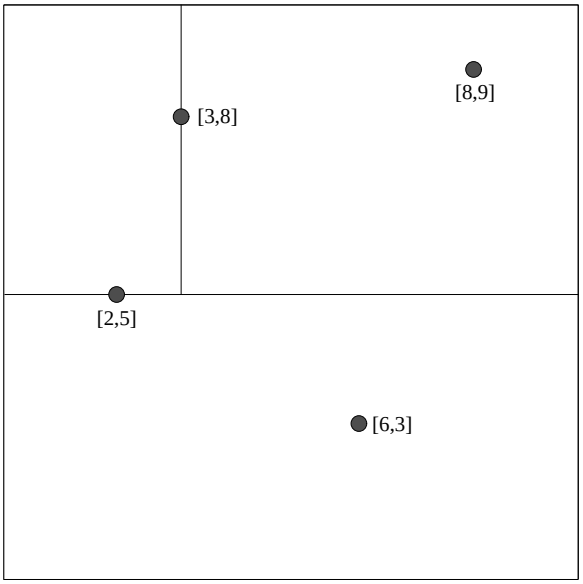

Figure 6.2

How the tree of Figure 6.1 splits up the x,y plane.

The invariant is that subtrees only ever contain **dom-elt**s which are on the correct side of all their ancestors' splitting planes.

$$
\begin{aligned}
&\textit{Is-legal-kdtree}(\text{empty}).\\
&\textit{Is-legal-kdtree}(<\mathbf{d},\mathbf{r},-,\text{empty},\text{empty}>).\\
&\textit{Is-legal-kdtree}(<\mathbf{d},\mathbf{r},\text{split},\text{tree}_{\textit{left}},\text{tree}_{\textit{right}}>) \quad \Leftrightarrow\\
&\qquad \forall(\mathbf{d}',\mathbf{r}') \in \textit{exset-rep}(\text{tree}_{\textit{left}})\ \ {d'}_{\text{split}} \leq d_{\text{split}} \wedge\\
&\qquad \forall(\mathbf{d}',\mathbf{r}') \in \textit{exset-rep}(\text{tree}_{\textit{right}})\ \ {d'}_{\text{split}} > d_{\text{split}} \wedge\\
&\qquad\qquad \textit{Is-legal-kdtree}(\text{tree}_{\textit{left}}) \wedge\\
&\qquad\qquad \textit{Is-legal-kdtree}(\text{tree}_{\textit{right}})
\end{aligned}
\tag{6.5}
$$

6.3.2 Constructing a kd-tree

Given an exemplar-set **E**, a *k*d-tree can be constructed by the algorithm in Table 6.3. The pivot-choosing procedure of Step 2 inspects the set and chooses a "good" domain vector from this set to use as the tree's root. The discussion of how such a root is chosen is deferred to Section 6.7. Whichever exemplar is chosen as root will not affect the correctness of the *k*d-tree, though the tree's maximum depth and the shape of the hyperregions *will* be affected.

6.4 Nearest Neighbour Search

In this section, I describe the nearest neighbour algorithm which operates on *k*d-trees. I begin with an informal description and worked example, and then give the precise algorithm.

A first approximation is initially found at the leaf node which contains the target point. In Figure 6.3 the target point is marked X and the leaf node of the region containing the target is coloured black. As is exemplified in this case, this first approximation is not necessarily the nearest neighbour, but at least we know any potential nearer neighbour must lie closer, and so must lie within the circle centred on X and passing through the leaf node. We now back up to the parent of the current node. In Figure 6.4 this parent is the black node. We compute whether it is *possible* for a closer solution to that so far found to exist in this parent's other child. Here it is not possible, because the circle does not intersect with the (shaded) space occupied by the parent's other child. If no closer neighbour can exist in the other child, the algorithm can immediately move up a further level, else it must recursively explore the other child. In this example, the next parent which is checked will need to be explored, because the area it covers (i.e. everywhere north of the central horizontal line) *does* intersect with the best circle so far.

Table 6.4 describes my actual implementation of the nearest neighbour algorithm. It is called with four parameters: the *k*d-tree, the target domain vector, a representation of a hyperrectangle in **Domain**, and a value indicating the maximum distance from the target which is worth searching. The search will only take place within those portions of the *k*d-tree which lie both in the hyper-

Algorithm:	Constructing a kd-tree
Input:	**exset**, of type **exemplar-set**
Output:	**kd**, of type **kdtree**
Pre:	None
Post:	**exset** = *exset-rep*(**kd**) $\wedge$ *Is-legal-kdtree*(**kd**)
Code:	
1.	If **exset** is empty then return the empty kdtree
2.	Call pivot-choosing procedure, which returns two values: **ex** := a member of **exset** **split** := the splitting dimension
3.	**d** := domain vector of **ex**
4.	**exset'** := **exset** with **ex** removed
5.	**r** := range vector of **ex**
6.	**exsetleft** := $\{(\mathbf{d}', \mathbf{r}') \in$ **exset'** $\mid d'_{\text{split}} \leq d_{\text{split}}\}$
7.	**exsetright** := $\{(\mathbf{d}', \mathbf{r}') \in$ **exset'** $\mid d'_{\text{split}} > d_{\text{split}}\}$
8.	**kdleft** := recursively construct kd-tree from **exsetleft**
9.	**kdright** := recursively construct kd-tree from **exsetright**
10.	**kd** := $<$ **d**, **r**, **split**, **kdleft**, **kdright** $>$
Proof:	By induction on the length of **exset** and the definitions of *exset-rep* and *Is-legal-kdtree*.

Table 6.3: Constructing a kd-tree from a set of exemplars.

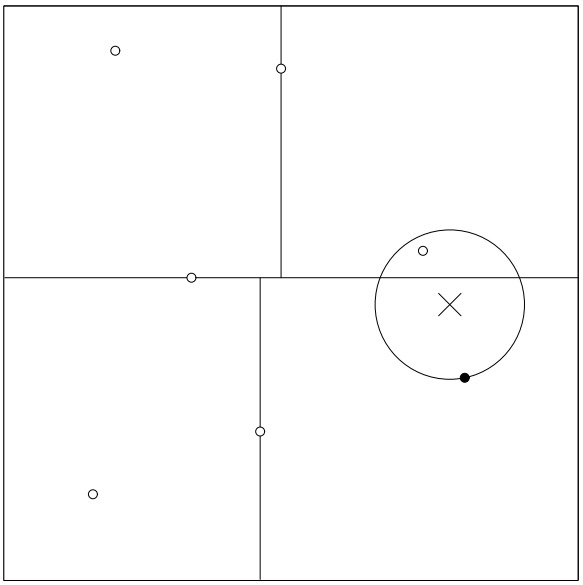

Figure 6.3

The black dot is the dot which owns the leaf node containing the target (the cross). Any nearer neighbour must lie inside this circle.

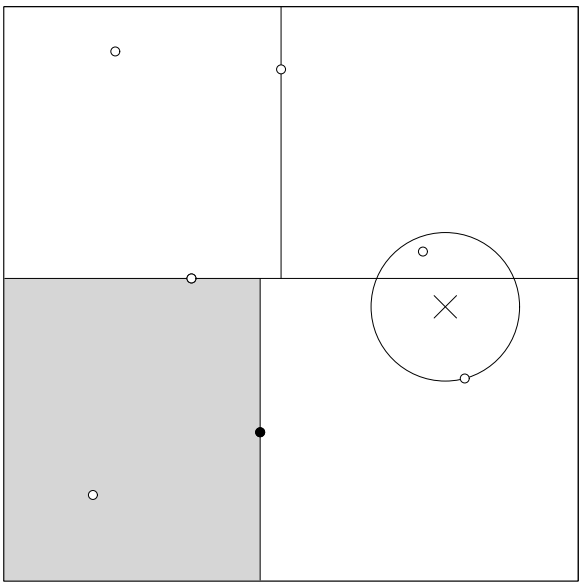

Figure 6.4

The black dot is the parent of the closest found so far. In this case the black dot's other child (shaded grey) need not be searched.

rectangle, and within the maximum distance to the target. The caller of the routine will generally specify the infinite hyperrectangle which covers the whole of **Domain**, and the infinite maximum distance.

Before discussing its execution, I will explain how the operations on the hyperrectangles can be implemented. A hyperrectangle is represented by two arrays: one of its minimum coordinates, the other of its maximum coordinates. To *cut* the hyperrectangle, so that one of its edges is moved closer to its centre, the appropriate array component is altered. To check to see if a hyperrectangle **hr** intersects with a hypersphere radius r centered at point **t**, we find the point **p** in **hr** which is closest to **t**. Write $hr_i^{\min}$ as the minimum extreme of **hr** in the ith dimension and $hr_i^{\max}$ as the maximum extreme. p_i, the ith component of this closest point is computed thus:

$$p_i = \begin{cases} hr_i^{\min} & \text{if } t_i \leq hr_i^{\min} \\ t_i & \text{if } hr_i^{\min} < t_i < hr_i^{\max} \\ hr_i^{\min} & \text{if } t_i \geq hr_i^{\max} \end{cases} \tag{6.6}$$

The objects intersect only if the distance between **p** and **t** is less than or equal to r.

The search is depth first, and uses the heuristic of searching first the child node which contains the target. Step 1 deals with the trivial empty tree case, and Steps 2 and 3 assign two important local variables. Step 4 cuts the current hyperrectangle into the two hyperrectangles covering the space occupied by the child nodes. Steps 5–7 determine which child contains the target. After Step 8, when this initial child is searched, it may be possible to prove that there cannot be any closer point in the hyperrectangle of the further child. In particular, the point at the current node must be out of range. The test is made in Steps 9 and 10. Step 9 restricts the maximum radius in which any possible closer point could lie, and then the test in Step 10 checks whether there is any

Algorithm:	Nearest Neighbour in a *k*d-tree
Input:	**kd**, of type **kdtree** **target**, of type **domain vector** **hr**, of type **hyperrectangle** **max-dist-sqd**, of type **float**
Output:	**nearest**, of type **exemplar** **dist-sqd**, of type **float**
Pre:	*Is-legal-kdtree*(**kd**)
Post:	Informally, the postcondition is that **nearest** is a nearest exemplar to **target** which also lies both within the hyperrectangle **hr** and within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. $\sqrt{\textbf{dist-sqd}}$ is the distance of this nearest point. If there is no such point then **dist-sqd** contains infinity.
Code:	
1.	if **kd** is empty then set **dist-sqd** to infinity and exit.
2.	**s** := **split** field of **kd**
3.	**pivot** := **dom-elt** field of **kd**
4.	Cut **hr** into two sub-hyperrectangles **left-hr** and **right-hr**. The cut plane is through **pivot** and perpendicular to the **s** dimension.
5.	**target-in-left** := $\textbf{target}_\textbf{s} \leq \textbf{pivot}_\textbf{s}$
6.	if **target-in-left** then
6.1	**nearer-kd** := **left** field of **kd** and **nearer-hr** := **left-hr**
6.2	**further-kd** := **right** field of **kd** and **further-hr** := **right-hr**
7.	if **not target-in-left** then
7.1	**nearer-kd** := **right** field of **kd** and **nearer-hr** := **right-hr**
7.2	**further-kd** := **left** field of **kd** and **further-hr** := **left-hr**
8.	Recursively call **Nearest Neighbour** with parameters (**nearer-kd**,**target**, **nearer-hr**,**max-dist-sqd**), storing the results in **nearest** and **dist-sqd**
9.	**max-dist-sqd** := minimum of **max-dist-sqd** and **dist-sqd**
10.	A nearer point could only lie in **further-kd** if there were some part of **further-hr** within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. if this is the case then
10.1	if $(\textbf{pivot} - \textbf{target})^2 < \textbf{dist-sqd}$ then
10.1.1	**nearest** := (**pivot**,**range-elt** field of **kd**)
10.1.2	**dist-sqd** := $(\textbf{pivot} - \textbf{target})^2$
10.1.3	**max-dist-sqd** := **dist-sqd**
10.2	Recursively call **Nearest Neighbour** with parameters (**further-kd**,**target**, **further-hr**,**max-dist-sqd**), storing the results in **temp-nearest** and **temp-dist-sqd**
10.3	If **temp-dist-sqd** < **dist-sqd** then
10.3.1	**nearest** := **temp-nearest** and **dist-sqd** := **temp-dist-sqd**
Proof:	Outlined in text

Table 6.4: The Nearest Neighbour Algorithm

space in the hyperrectangle of the further child which lies within this radius. If it is *not* possible then no further search is necessary. If it *is* possible, then Step 10.1 checks if the point associated with the current node of the tree is closer than the closest yet. Then, in Step 10.2, the further child is recursively searched. The maximum distance worth examining in this further search is the distance to the closest point yet discovered.

The proof that this will find the nearest neighbour within the constraints is by induction on the size of the *k*d-tree. If the cutoff were not made in Step 10, then the proof would be straightforward: the point returned is the closest out of (i) the closest point in the nearer child, (ii) the point at the current node and (iii) the closest point in the further child. If the cutoff *were* made in Step 10, then the point returned is the closest point in the nearest child, and we can show that neither the current point, nor any point in the further child can possibly be closer.

Many local optimizations are possible which while not altering the asymptotic performance of the algorithm will multiply the speed by a constant factor. In particular, it is in practice possible to hold almost all of the search state globally, instead of passing it as recursive parameters.

6.5 Theoretical Behaviour

Given a *k*d-tree with N nodes, how many nodes need to be inspected in order to find the proven nearest neighbour using the algorithm in Section 6.4?. It is clear at once that on average, at least $O(\log N)$ inspections are necessary, because any nearest neighbour search requires traversal to at least one leaf of the tree. It is also clear that no more than N nodes are searched: the algorithm visits each node at most once.

Figure 6.5 graphically shows why we might expect considerably fewer than N nodes to be visited: the shaded areas correspond to areas of the *k*d-tree which were cut off.

The important values are (i) the worst case number of inspections and (ii) the expected number of inspections. It is actually easy to construct worst case distributions of the points which will force nearly all the nodes to be inspected. In Figure 6.6, the tree is two-dimensional, and the points are scattered along the circumference of a circle. If we request the nearest neighbour with the target close to the centre of the circle, it will therefore be necessary for each rectangle, and hence each leaf, to be inspected (this is in order to ensure that there is no point lying inside the circle in any rectangle).

Calculation of the expected number of inspections is difficult, because the analysis depends critically on the expected distribution of the points in the *k*d-tree, and the expected distribution of the target points presented to the nearest neighbour algorithm.

The analysis is performed in [Friedman *et al.*, 1977]. This paper considers the expected number of hyperrectangles corresponding to leaf nodes which will provably need to be searched. Such hyperrectangles intersect the volume enclosed by a hypersphere centered on the query point whose surface passes through the nearest neighbour. For example, in Figure 6.5 the hypersphere (in this

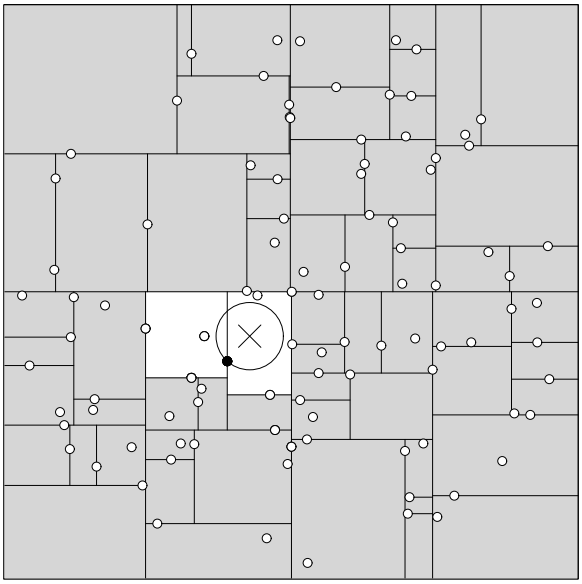

Figure 6.5

Generally during a nearest neighbour search only a few leaf nodes need to be inspected.

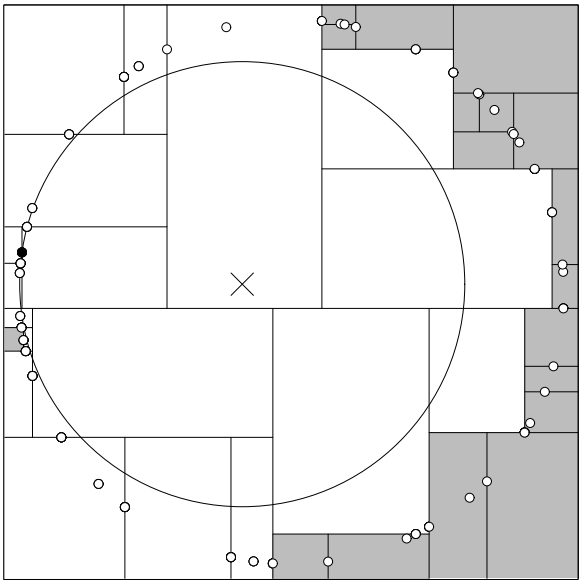

Figure 6.6

A bad distribution which forces almost all nodes to be inspected.

case a circle) is shown, and the number of intersecting hyperrectangles is two.

The paper shows that the expected number of intersecting hyperrectangles is independent of N, the number of exemplars. The asymptotic search time is thus logarithmic because the time to descend from the root of the tree to the leaves is logarithmic (in a balanced tree), and then an expected constant amount of backtracking is required.

However, this reasoning was based on the assumption that the hyperrectangles in the tree tend to be hypercubic in shape. Empirical evidence in my investigations has shown that this is not generally the case for their tree building strategy. This is discussed and demonstrated in Section 6.7.

A second danger is that the cost, while independent of N, is exponentially dependent on k, the dimensionality of the domain vectors[1].

Thus theoretical analysis provides some insight into the cost, but here, empirical investigation will be used to examine the expense of nearest neighbour in practice.

6.6 Empirical Behaviour

In this section I investigate the empirical behaviour of the nearest neighbour searching algorithm. We expect that the number of nodes inspected in the tree varies according to the following properties of the tree:

- N, the size of the tree.
- k_{dom} , the dimensionality of the domain vectors in the tree. This value is the k in kd-tree.
- d_{distrib} , the distribution of the domain vectors. This can be quantified as the "true" dimensionality of the vectors. For example, if the vectors had three components, but all lay on the surface of a sphere, then the underlying dimensionality would be 2. In general, discovery of the underlying dimensionality of a given sample of points is extremely difficult, but for these tests it is a straightforward matter to *generate* such points. To make a kd-tree with underlying dimensionality d_{distrib}, I use randomly generated k_{dom}-dimensional domain vectors which lie on a d_{distrib}-dimensional hyperelliptical surface. The random vector generation algorithm is as follows: Generate d_{distrib} random angles $\theta_i \in [0, 2\pi)$ where $0 \leq i < d_{\text{distrib}}$. Then let the jth component of the vector be $\prod_{i=0}^{i=d-1} \sin(\theta_i + \phi_{ij})$. The phase angles ϕ_{ij} are defined as $\phi_{ij} = \frac{1}{2}\pi$ if the jth bit of the binary representation of i is 1 and is zero otherwise.
- d_{target} , the probability distribution from which the search target vector will be selected. I shall assume that this distribution is the same as that which determines the domain vectors. This is indeed what will happen when the kd-tree is used for learning control.

In the following sections I investigate how performance depends on each of these properties.

[1] This was pointed out to the author by N. Maclaren.

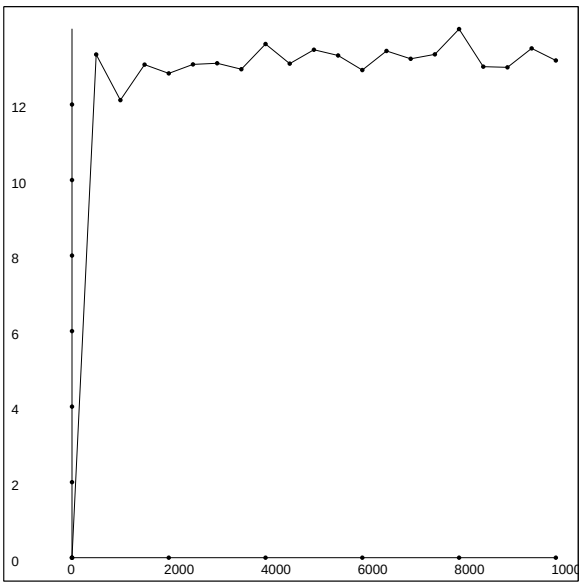

Figure 6.7

Number of inspections required during a nearest neighbour search against the size of the kd-tree. In this experiment the tree was four-dimensional and the underlying distribution of the points was three-dimensional.

6.6.1 Performance against Tree Size

Figures 6.7 and 6.8 graph the number of nodes inspected against the number of nodes in the entire kd-tree. Each value was derived by generating a random kd-tree, and then requesting 500 random nearest neighbour searches on the kd-tree. The average number of inspected nodes was recorded. A node was counted as being inspected if the distance between it and the target was computed. Figure 6.7 was obtained from a 4d-tree with an distribution distribution $d_{\text{distrib}} = 3$. Figure 6.8 used an 8d-tree with an underlying distribution $d_{\text{distrib}} = 8$.

It is immediately clear that after a certain point, the expense of a nearest neighbour search has no detectable increase with the size of the tree. This agrees with the proposed model of search cost—logarithmic with a large additive constant term.

6.6.2 Performance against the "k" in kd-tree

Figure 6.9 graphs the number of nodes inspected against k_{dom}, the number of components in the kd-tree's domain vectors for a 10,000 node tree. The underlying dimensionality was also k_{dom}. The number of inspections per search rises very quickly, possibly exponentially, with k_{dom}. This behaviour, the massive increase in cost with dimension, is familiar in computational geometry.

6.6.3 Performance against the Distribution Dimensionality

This experiment confirms that it is d_{distrib}, the distribution dimension from which the points were selected, rather than k_{dom} which critically affects the search performance. The trials for Figure 6.10 used a 10,000 node kd-tree with domain dimension of 14, for various values of d_{distrib}. The

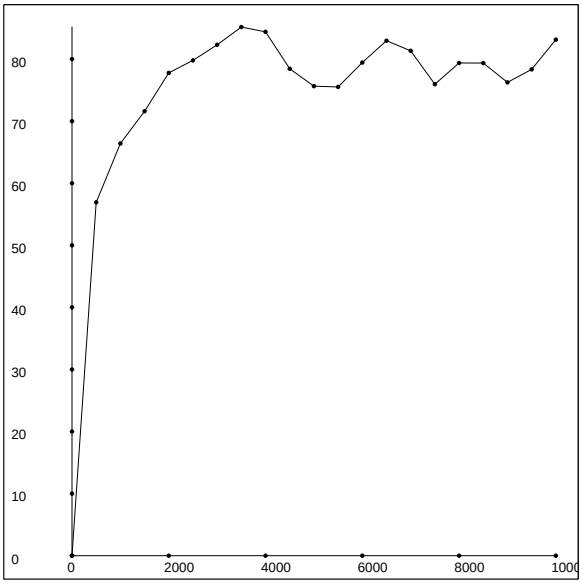

Figure 6.8

Number of inspections against kd-tree size for an eight-dimensional tree with an eight-dimensional underlying distribution.

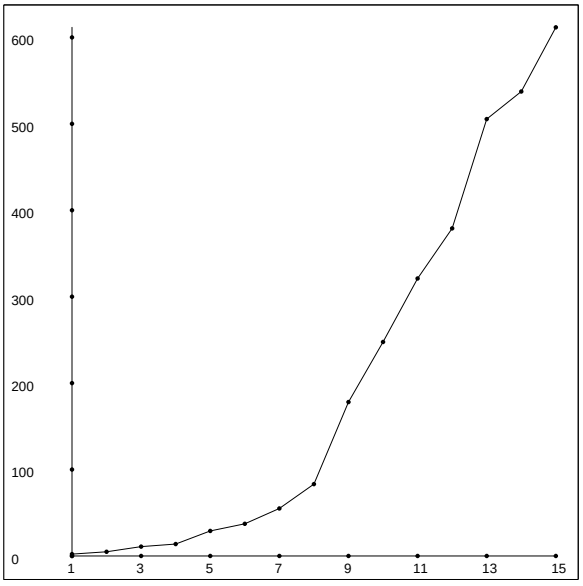

Figure 6.9

Number of inspections graphed against tree dimension. In these experiments the points had an underlying distribution with the same dimensionality as the tree.

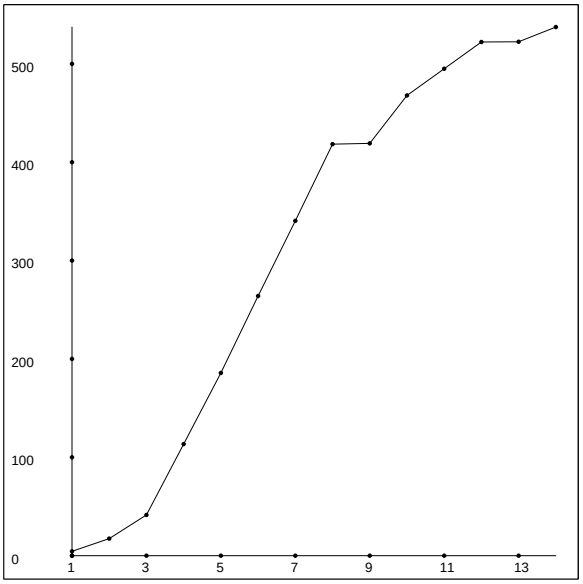

Figure 6.10

Number of inspections graphed against underlying dimensionality for a fourteen-dimensional tree.

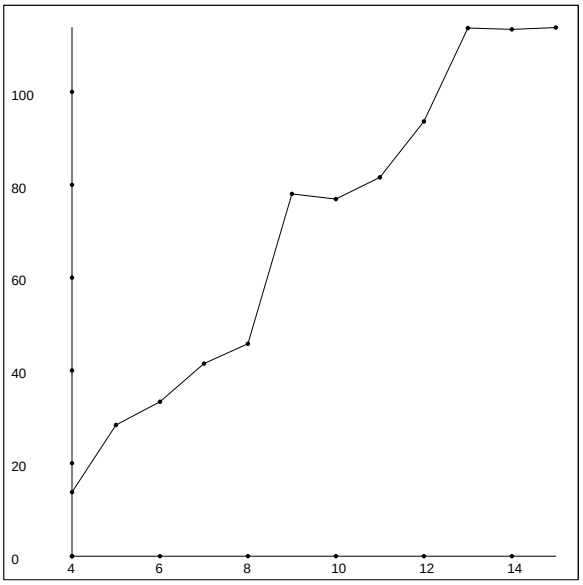

Figure 6.11

Number of inspections graphed against tree dimension, given a constant four dimensional underlying distribution.

important observation is that for 14d-trees, the performance does improve greatly if the underlying distribution-dimension is relatively low. Conversely, Figure 6.11 shows that for a fixed (4-d) underlying dimensionality, the search expense does not seem to increase any worse than linearly with k_{dom}.

6.6.4 When the Target is not Chosen from the kd-tree's Distribution

In this experiment the points were distributed on a three dimensional elliptical surface in ten-dimensional space. The target vector was, however, chosen at random from a ten-dimensional distribution. The *k*d-tree contained 10,000 points. The average number of inspections over 50 searches was found to be 8,396. This compares with another experiment in which both points and target were distributed in ten dimensions and the average number of inspections was only 248. The reason for the appalling performance was exemplified in Figure 6.6: if the target is very far from its nearest neighbour then very many leaf nodes must be checked.

6.6.5 Conclusion

The speed of the search (measured as the number of distance computations required) seems to vary ...

- ... only marginally with tree size. If the tree is sufficiently large with respect to the number of dimensions, it is essentially constant.
- ... very quickly with the dimensionality of the distribution of the datapoints, d_{distrib}.
- ... linearly with the number of components in the *k*d-tree's domain (k_{dom}), given a fixed distribution dimension (d_{distrib}).

There is also evidence to suggest that unless the target vector is drawn from the same distribution as the *k*d-tree points, performance can be greatly worsened.

These results support the belief that real time searching for nearest neighbours is practical in a robotic system where we can expect the underlying dimensionality of the data points to be low, roughly less than 10. This need not mean that the vectors in the input space should have less than ten components. For data points obtained from robotic systems it will not be easy to decide what the underlying dimensionality is. However Chapter 10 will show that the data does tend to lie within a number of low dimensional subspaces.

6.7 Further kd-tree Operations

In this section I discuss some other operations on *k*d-trees which are required for use in the *SAB* learning system. These include incrementally adding a point to a *k*d-tree, range searching, and selecting a pivot point.

6.7.1 Range Searching a kd-tree

$$\textit{range-search} \quad : \quad \text{exemplar-set} \times \textbf{Domain} \times \Re \rightarrow \text{exemplar-set}$$

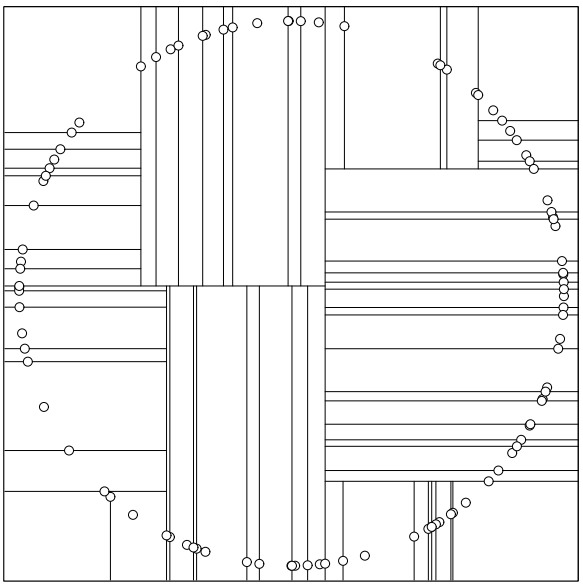

Figure 6.12

A 2d tree balanced using the 'median of the most spread dimension' pivoting strategy.

The abstract range search operation on an exemplar-set finds all exemplars whose domain vectors are within a given distance of a target point:

$$\textit{range-search}(\mathbf{E}, \mathbf{d}, r) \quad = \quad \{(\mathbf{d}', \mathbf{r}') \in \mathbf{E} \mid (\mathbf{d} - \mathbf{d}')^2 < r^2\}$$

This is implemented by a modified nearest neighbour search. The modifications are that (i) the initial distance is not reduced as closer points are discovered and (ii) all discovered points within the distance are returned, not just the nearest. The complexity of this operation is shown, in [Preparata and Shamos, 1985], to still be logarithmic in N (the size of $\mathbf{E}$) for a fixed range size.

6.7.2 Choosing a Pivot from an Exemplar Set

The tree building algorithm of Section 6.3 requires that a pivot and a splitting plane be selected from which to build the root of a kd-tree. It is desirable for the tree to be reasonably balanced, and also for the shapes of the hyperregions corresponding to leaf nodes to be fairly equally proportioned. The first criterion is important because a badly unbalanced tree would perhaps have $O(N)$ accessing behaviour instead of $O(\log N)$. The second criterion is in order to maximize cutoff opportunities for the nearest neighbour search. This is difficult to formalize, but can be motivated by an illustration. In Figure 6.12 is a perfectly balanced kd-tree in which the leaf regions are very non-square. Figure 6.13 illustrates a kd-tree representing the same set of points, but which promotes squareness at the expense of some balance.

One pivoting strategy which would lead to a perfectly balanced tree, and which is suggested in [Omohundro, 1987], is to pick the splitting dimension as that with maximum variance, and let

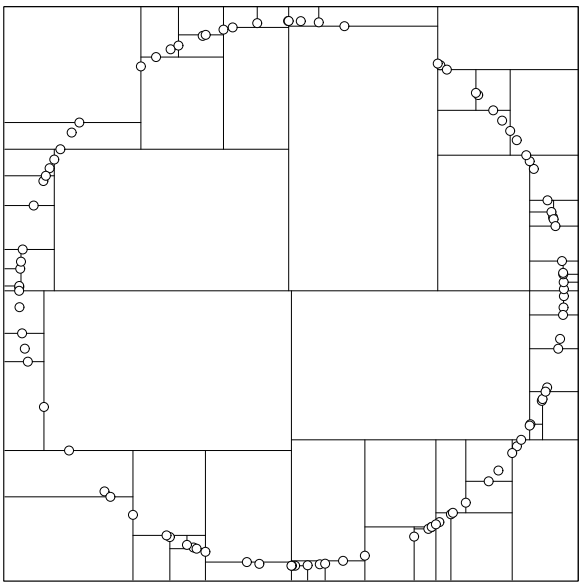

Figure 6.13

A 2d tree balanced using the 'closest to the centre of the widest dimension' pivoting strategy.

the pivot be the point with the median split component. This will, it is hoped, tend to promote square regions because having split in one dimension, the next level in the tree is unlikely to find that the same dimension has maximum spread, and so will choose a different dimension. For uniform distributions this tends to perform reasonably well, but for badly skewed distributions the hyperregions tend to take long thin shapes. This is exemplified in Figure 6.12 which has been balanced using this standard median pivot choice.

To avoid this bad case, I choose a pivot which splits the exemplar set in the middle of the range of the most spread dimension. As can be seen in Figure 6.13, this tends to favour squarer regions at the expense of a slight imbalance in the *k*d-tree. This means that large empty areas of space are filled with only a few hyperrectangles which are themselves large. Thus, the number of leaf nodes which need to be inspected in case they contain a nearer neighbour is smaller than for the original case, which had many small thin hyperrectangles.

My pivot choice algorithm is to firstly choose the splitting dimension as the longest dimension of the current hyperrectangle, and then choose the pivot as the point closest to the middle of the hyperrectangle along this dimension. Occasionally, this pivot may even be an extreme point along its dimension, leading to an entirely unbalanced node. This is worth it, because it creates a large empty leaf node. It is possible but extremely unlikely that the points could be distributed in such a way as to cause the tree to have one empty child at every level. This would be unacceptable, and so above a certain depth threshold, the pivots are chosen using the standard median technique.

Selecting the median as the split and selecting the closest to the centre of the range are both $O(N)$ operations, and so either way a tree rebalance is $O(N \log N)$.

6.7.3 Incrementally Adding a Point to a kd-tree

Firstly, the leaf node which contains the new point is computed. The hyperrectangle corresponding to this leaf is also obtained. See Section 6.4 for hyperrectangle implementation. When the leaf node is found it may either be (i) empty, in which case it is simply replaced by a new singleton node, or (ii) it contains a singleton node. In case (ii) the singleton node must be given a child, and so its previously irrelevant split field must be defined. The split field should be chosen to preserve the squareness of the new subhyperrectangles. A simple heuristic is used. The split dimension is chosen as the dimension in which the hyperrectangle is longest. This heuristic is motivated by the same requirement as for tree balancing—that the regions should be as square as possible, even if this means some loss of balance.

This splitting choice is just a heuristic, and there is no guarantee that a series of points added in this way will preserve the balance of the kd-tree, nor that the hyperrectangles will be well shaped for nearest neighbour search. Thus, on occasion (such as when the depth exceeds a small multiple of the best possible depth) the tree is rebuilt. Incremental addition costs $O(\log N)$.

6.7.4 Q Nearest Neighbours

This uses a modified version of the nearest neighbour search. Instead of only searching within a sphere whose radius is the closest distance yet found, the search is within a sphere whose radius is the Qth closest yet found. Until Q points have been found, this distance is infinity.

6.7.5 Deleting a Point from a kd-tree

If the point is at a leaf, this is straightforward. Otherwise, it is difficult, because the structure of both trees below this node are pivoted around the point we wish to remove. One solution would be to rebuild the tree below the deleted point, but on occasion this would be very expensive. My solution is to mark the point as deleted with an extra field in the kd-tree node, and to ignore deletion nodes in nearest neighbour and similar searches. When the tree is next rebuilt, all deletion nodes are removed.

Chapter 7

SAB-trees: Coping with Disorder

The nearest neighbour (see Chapter 5) has been selected for learning world models which relate "state", "action" and "behaviour" (see Chapter 4). The previous chapter explained in detail how the nearest neighbour can be found efficiently. This chapter explains how the standard nearest neighbour algorithm is tailored to the needs of this investigation. It begins by explaining how kd-trees are used. It then shows how resistance to noise is implemented in a way which means that each access still only requires a single nearest neighbour search. After that, the problem of adapting to a changing world model is addressed. The next section explains the detailed implementation and asymptotic efficiency of these operations. The final section describes garbage collection of the kd-tree, to remove old, inaccurate, exemplars.

7.1 SAB Relations, and their Representation

The Perceived State Transition Function has this form:

$$\text{PSTF} : \mathbf{State} \times \mathbf{Action} \rightarrow \mathbf{Behaviour} \tag{7.1}$$

All experiences are stored explicitly in a *k*d-tree. The choice of which components of the PSTF form the tree's domain (i.e. index), and which form the range is influenced by the fields which need a nearest neighbour search. Chapter 5 described how both *Dynamics Prediction*, searching on **State** $\times$ **Action**, and *Partial Inversion*, searching on **State** $\times$ **Behaviour**, would be useful.

The two different indexing requirements could be satisfied by various means. One method would be to index the *k*d-tree on all three components. During the search, when a split is found in a dimension which is *not* being used as an index, then both subtrees are searched. This is slower than if the search had been on a tree only indexed with the relevant components, but it is nevertheless faster than a complete linear search.

The solution to multiple indexes which I use is simpler. There are two kd-trees, one indexed by **State**$\times$**Action** and the other by **State**$\times$**Behaviour**. This uses more memory but is faster, because nearest neighbour search takes place in a lower dimension. The extra memory is significantly less than twice as much, because the second tree can have pointers to the vectors of the first tree, rather than its own copy of each **s**, **b** and **a** vector.

7.2 Resisting Noise

Each exemplar is obtained by observing a state, applying an action and then observing the behaviour. The assumption that the sensor readings are sufficient to determine the state implies that if the system is ever in the same perceived state again and the same raw action is applied then the same behaviour will be observed. More simply we assume that the PSTF is a deterministic function. A problem is that non-deterministic random noise is likely to corrupt any observations or raw action signals. This means that on different occasions, the same state-action pair may produce different behaviour.

Let $\mathbf{c} \in$ **State** $\times$ **Action** be a state-action pair which the controller believes it has experienced. In fact this belief could be wrong for two reasons:

1. The state observation could have an error.
2. The action signal which the controller requested could have been corrupted before it arrived at the actuator.

Imagine that the actual state action pair had been $\mathbf{c} + \boldsymbol{\delta c}$. Let $\mathbf{b}$ be the behaviour which then occurred. Due to noisy sensors the controller observes behaviour $\mathbf{b} + \boldsymbol{\delta b}$. Then the world model is updated with the observation that $\mathbf{c} \rightarrow \mathbf{b} + \boldsymbol{\delta b}$ when in fact the truth is that

$$\mathrm{PSTF}(\mathbf{c} + \boldsymbol{\delta c}) = \mathbf{b} \tag{7.2}$$

By the Taylor expansion

$$\mathrm{PSTF}(\mathbf{c} + \boldsymbol{\delta c}) = \mathrm{PSTF}(\mathbf{c}) + J\boldsymbol{\delta c} + O(\delta^2) \text{ where } J_{ij} = \frac{\partial \mathrm{PSTF}_i}{\partial c_j} \tag{7.3}$$

where J is the Jacobian matrix of the PSTF at $\mathbf{c}$. If $|\,\boldsymbol{\delta b}\,|$ and $|\,\boldsymbol{\delta c}\,|$ are sufficiently small the $O(\delta^2)$ term can be discarded. Thus the controller has wrongly recorded

$$\mathbf{c} \rightarrow \mathrm{PSTF}(\mathbf{c}) + \underbrace{J\boldsymbol{\delta c} + \boldsymbol{\delta b}}_{\text{error}} \tag{7.4}$$

The solution to the problem is to use the local average value for prediction. If we make a series of N observations at what the controller believes to be constant state-action $\mathbf{c}$ and then average

the results we can hope to reduce the error. Let the ith observation have errors $\boldsymbol{\delta b}_i$ and $\boldsymbol{\delta c}_i$. Then the average estimate for PSTF($\mathbf{c}$) is

$$\frac{1}{N}\sum_{i=1}^{N}(\mathbf{b}+\boldsymbol{\delta b}_i). \tag{7.5}$$

$$= \frac{1}{N}\sum_{i=1}^{N}(\text{PSTF}(\mathbf{c})+J\boldsymbol{\delta c}_i+\boldsymbol{\delta b}_i). \tag{7.6}$$

$$= \text{PSTF}(\mathbf{c})+J\left(\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{\delta c}_i\right)+\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{\delta b}_i. \tag{7.7}$$

If the $\boldsymbol{\delta b}_i$s and $\boldsymbol{\delta c}_i$s are both drawn from random distributions with means $\mathbf{m}_b$ and $\mathbf{m}_c$ then as N increases the value converges to

$$\text{PSTF}(\mathbf{c})+J\mathbf{m}_c+\mathbf{m}_b. \tag{7.8}$$

If these means are zero then the estimate converges to the correct value as required. What would it imply if the error distributions did not have zero mean? An example of this in the Mountain Car domain would be where the pedal on average was pushed 10mm further down than the controller requested. As a result, the speed obtained by requesting a pedal of 80mm for example would be predicted wrongly as that obtained from an actual pedal value of 70mm. But is this wrong? In fact it is not, because of the fundamental fact that *we are trying to learn the relationship between what we request and what we perceive* rather than what the actual control was and what the actual result was. Another illustration is for supposed regular errors in perception. If a vision controlled arm regularly thinks things on the retina are an inch to the left of where they "should" be (for example due to the camera having been set up slightly askew) and then is shown a goal position to move to, it will then try to move the "wrong" image of the hand to the "wrong" image of the goal and will thus succeed. There will only be a problem if the task is specified in a different coordinate system to the state description. In this case a regular error would necessarily fool any controller using any form of sensory feedback. This seems reasonable: a person performing a reaching task should still manage it wearing distorting glasses, but would find it very hard to cope if the goal's apparent position were distorted but the hand not.

The mean errors are zero and so the averaging estimate converges in the presence of noise to the correct value. The rate of convergence as learning progresses will depend on the amount of noise (the variance of the noise distributions) and the number of experiences of the state-action in question. The important state-actions which are frequently needed will converge more quickly than those which are rarely used (and these are state-action pairs for which less accuracy is likely to be needed).

I will now describe in more detail how local averaging is used in this *SAB* implementation. It takes place over all points within a certain distance D_{range} (the *range distance*) of the query point. This speeds the convergence because there are more data points to examine. Using spatially local

points for averaging is analogous to a common image processing operation, in which image noise is removed by replacing all pixels by the local weighted average of pixel values. The disadvantage is the need to choose the range distance D_{range}. Too small a distance will provide little averaging of the noisy values. Too large a distance will smooth too harshly and genuine details in the underlying function will be lost. The system designer must choose the range distance based on the expected noise and expected smoothness of the function. In the experimental studies this has not been a difficult choice and the behaviour has not been sensitive to variation in the range width. If necessary, it is possible that the range width could be obtained automatically by more sophisticated statistical analysis or a cross-validation technique.

The weight function gives more weight to points which lie close to the query point. The smoothed prediction from a set of exemplars

$$\mathbf{E} = \{(\mathbf{s}_1, \mathbf{a}_1, \mathbf{b}_1), \ldots, (\mathbf{s}_N, \mathbf{a}_N, \mathbf{b}_N)\} \tag{7.9}$$

is

$$eval_{\mathbf{E}}(\mathbf{s}, \mathbf{a}) = \frac{\sum_{i=1}^{N} (weight(dist_i) \times \mathbf{b}_i)}{\sum_{i=1}^{N} weight(dist_i)} \tag{7.10}$$

where $dist_i = |\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}, \mathbf{a})\ |$ and the weight strength decreases to zero beyond the range distance D_{range} according to the bisquare function

$$weight(d) = \begin{cases} \left[1 - (d/D_{\text{range}})^2\right]^2 & \text{if } d < D_{\text{range}} \\ 0 & \text{if } d \geq D_{\text{range}} \end{cases} \tag{7.11}$$

It is graphed in Figure 7.1.

In Section 7.4 I explain how $eval_{\mathbf{E}}(\mathbf{s}, \mathbf{a})$ is computed with adequate efficiency. At this point I note simply that to avoid having to perform this local averaging every time a query occurs, the smoothed value at each exemplar is actually stored with the exemplar as a fourth component $\mathbf{b}^{\text{smooth}}$:

$$\mathbf{E} = \{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i, \mathbf{b}_i{}^{\text{smooth}}) : \mathbf{b}_i{}^{\text{smooth}} = eval_{\mathbf{E}}(\mathbf{s}_i, \mathbf{a}_i)\} \tag{7.12}$$

When a prediction is required for a point which has not yet been experienced, the smoothed prediction is not computed. Instead the $\mathbf{b}^{\text{smooth}}$ component of the nearest neighbour is used. As well as reducing computation this has an important property for sparsely populated regions. The prediction will be more useful than the "zero" prediction provided by $eval_{\mathbf{E}}$ when there are no exemplars within the range distance. The $\mathbf{b}^{\text{smooth}}$ component of an exemplar is initially computed when the exemplar is added to the *SAB*-tree. It is updated only when necessary to preserve the invariant $\mathbf{b}_i{}^{\text{smooth}} = eval_{\mathbf{E}}(\mathbf{s}_i, \mathbf{a}_i)$.

This method has been introduced to reduce the effects of noisy exemplars. *It is not intended to, nor expected to, provide extra interpolative accuracy.* Here is an illustrative example. Suppose

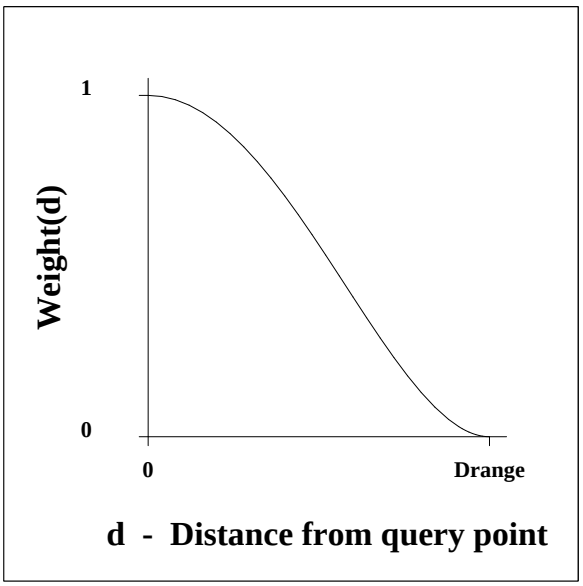

Figure 7.1

The smoothing function. The weight of a near neighbour decreases with distance to the query point. It has no influence beyond distance D_{range}.

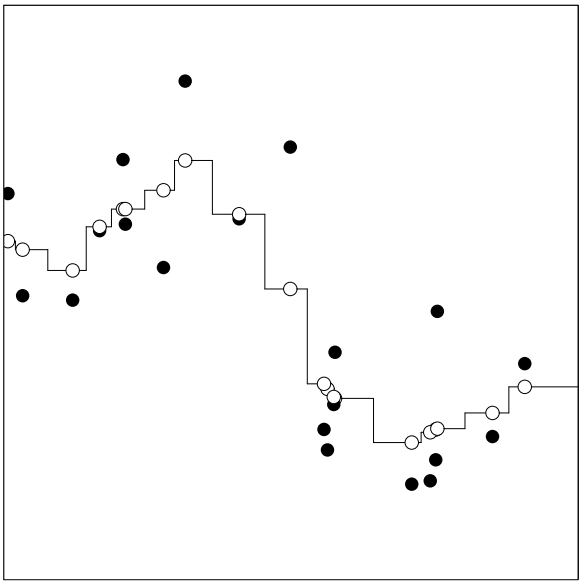

Figure 7.2

Learning from noisy exemplars. The y-coords of the black dots are the **b** components; those of the white dots are the $\mathbf{b}^{\text{smooth}}$ components. The line is the interpretation induced by nearest neighbour. D_{range} is 10% of the diagram width.

S is a zero-dimensional space, **A** is one-dimensional and **B** is one-dimensional and suppose that we have observed the noisy exemplars which are the solid dots in Figure 7.2. The white dots are the smoothed value at each point. The line shows the function produced by using the smoothed nearest neighbour. Figure 7.3 shows what happens if too small a value of D_{range} is used (equivalent to not using smoothing), and Figure 7.4 shows what happens if too large a value is used. Visually, the smoothing appears adequate for values of D_{range} between 5% and 40% of the diagram width.

Finally, Figures 7.5 and 7.6 show the Mountain Car PSTF respectively not using and using a

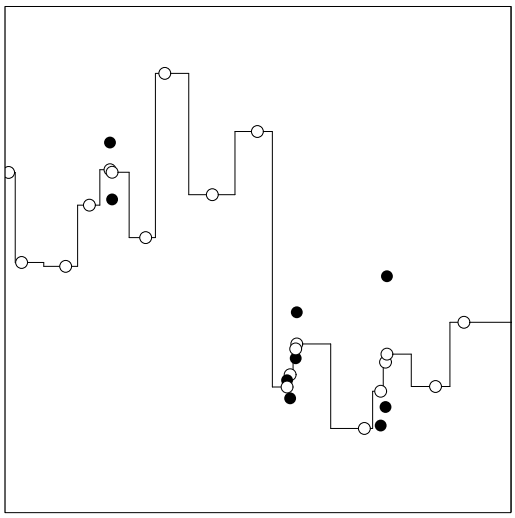

Figure 7.3: As Figure 7.2 except that D_{range} is 2% of the diagram width.

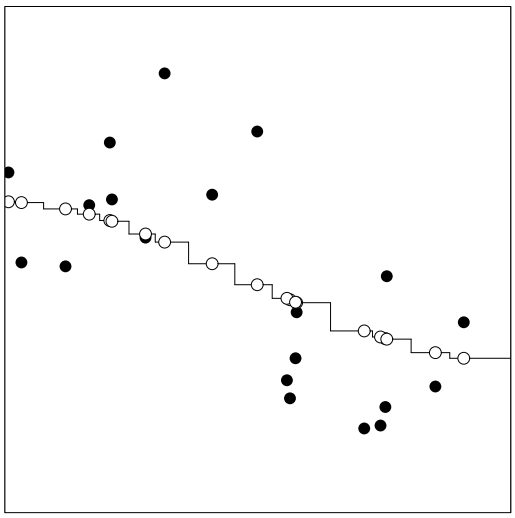

Figure 7.4: As Figure 7.2 except that D_{range} is 60% of the diagram width.

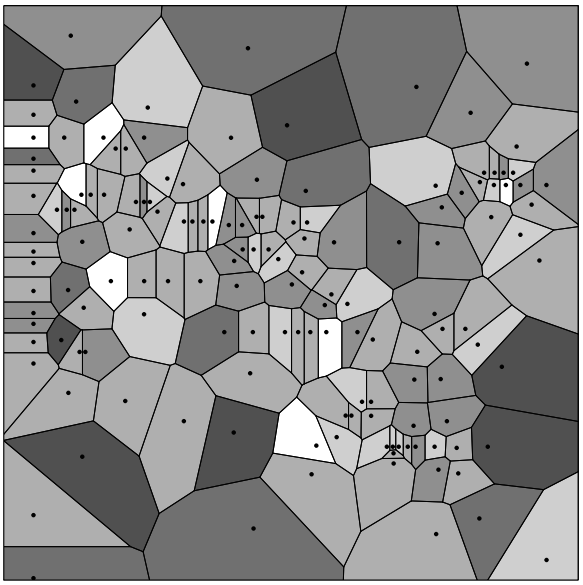

Figure 7.5

The mountain car PSTF (see Section 4.3). Noise of 50% was added to the perceived behaviour. This is the nearest neighbour interpretation using the **b** components of exemplars.

smoothing kernel in the presence of simulated noise.

Before concluding this section it is worth noting another interpretation of the $eval_{\mathbf{E}}$ smoothing operation. It is an example of Shepards Interpolation [Franke, 1982]. Our analogous operation, averaging, is performed locally. We have a tighter restriction on the shape of the smoothing function than would a user of Shepard's Interpolation: we require it reach zero within a relatively short range of the query point: Shepard's interpolation weight functions are usually non-zero over the entire

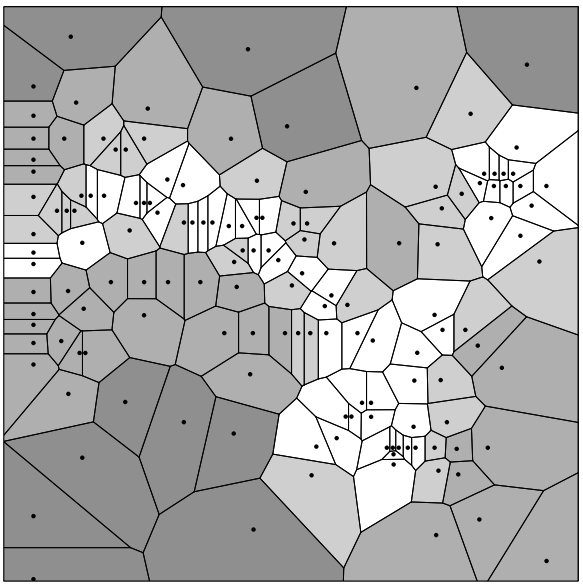

Figure 7.6

As Figure 7.5, but using the $\mathbf{b}^{\text{smooth}}$ components of exemplars. The range width, D_{range}, was 1.5 units.

domain.

7.3 Adapting to a Changing Environment

As well as disorder in the form of noise there is another problem that a learning controller should be able to cope with: an environment that changes unpredictably with time. Here are some examples of how this might come about:

1. **Gradual Change.** A new robot arm is put to work on a job. As time passes the stiff joints become less stiff due to wear, and so gradually over the first week of use the joint friction is reduced, changing the system dynamics.

2. **Sudden Change.** A vision-controlled robot is picking objects from a conveyor belt. After several months of operation the camera is jogged and instantly all observations are perceived to be two inches to left of their original locations.

An important feature of these changes is that they occur unpredictably. They could not in principle have been modelled, nor could their occurrence have even been announced.

Adjusting to this sort of event is an important problem for any learning controller, but for the nearest-neighbour learning described so far, it is particularly serious. The old, wrong exemplars will never disappear. During a gradual change, the old exemplars will initially be more useful than no knowledge at all, because their prediction will be close to correct though not perfect. But after a time it would have been better to be able to ignore the old exemplars. After a sudden change the old exemplars will all be misleading in the region of control space in which the change has occurred.

One possible solution would be to try to spot when a change had occurred by monitoring the recent history of errors. When this exceeded a threshold the controller would experience enforced amnesia, forgetting everything, and then begin relearning. This extreme approach would have two disadvantages. After a fairly small change (or a short period of a gradual change) the old, slightly wrong knowledge that the controller has is still more useful than no knowledge at all. Furthermore we should hope to be able to use the imperfect model to help guide the search for new, accurate knowledge. The second reason that it would be unfortunate to throw away all the knowledge is that there might be cases where the change only badly affects particular regions of the control space.

Instead, we are able to detect when it would be useful to throw away old knowledge, and when it is still useful to retain it. This is by comparing suspect old exemplars with their local neighbours. Because of the $\mathbf{b}^{\text{smooth}}$ modification to *SAB*-trees, this can be done reasonably cheaply.

Whenever a new point is added, all its local neighbours (within the range width, D_{range}, of Section 7.2) are inspected. An exemplar is deemed undesirable if...

- ...it disagrees with the local consensus of the value of the behaviour in its neighbourhood. For suspect exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i, \mathbf{b}_i{}^{\text{smooth}})$ this can be detected by noticing a large difference between $\mathbf{b}_i$ and $\mathbf{b}_i{}^{\text{smooth}}$. The test is whether $\mid \mathbf{b}_i - \mathbf{b}_i{}^{\text{smooth}} \mid > \Delta_{\text{old}}$ where Δ_{old} is the tolerable difference threshold.
- ...and it is relatively old. This is detected by storing the date of birth of each exemplar, and counting an exemplar as old if it is older than the mean age of the local exemplars.

Old, inaccurate points are simply deleted from the *SAB*-tree.

Examples of adapting to change for a noisy, one-dimensional mapping can be seen in Figure 7.7.

7.3.1 Discussion

Empirically, this adaptation mechanism is seen to work sufficiently well (c.f. Chapter 10). However, several improvements are possible, based on more detailed statistical analysis of the local points, for example by comparing the variance of the behaviours of (i) all local exemplars and (ii) recent local exemplars. This would help in the case where the exemplars are noisy, and the old exemplars' inaccuracies are not because of environmental changes. It is not clear that the avoidance of these occasional unjust exemplar removals is sufficient justification for the extra computation and hand-crafted statistical thresholds which would be required.

7.4 Updating the Relation

This section details the computation and the cost of the smoothing and adapting operations of the previous sections. They are implemented using the algorithms described in Chapter 6. The reader who is content to simply understand the behaviour without reading about the details may omit the rest of this chapter with no impact on later sections.

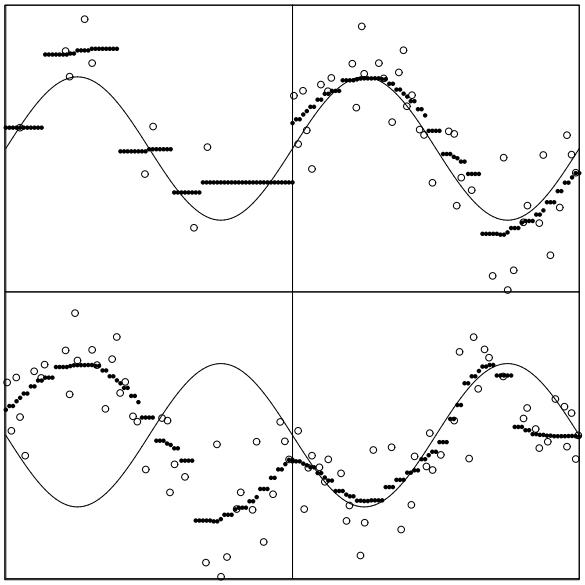

Figure 7.7

One-dimensional *SAB*-tree, attempting to learn a sine wave from noisy exemplars, shown as white dots. Black dots are the smoothed interpretation. Top left: After 8 random exemplars. Top right: After 50. Bottom Left: The underlying function is changed. Bottom right: After further 50 exemplars.

7.4.1 Duplicate Points

An additional, rather uninteresting, optimization occurs in the case when a new exemplar $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})$ is to be stored, but it is noticed that there is already an exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i)$ in the set $\mathbf{E}$ which is close enough in all three components to be considered identical. An example of where this might happen is when a robot arm is being successfully held stationary by *SAB* control. In this case the new point is not added, but the *mass* field of the exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i)$ is increased by one. The mass field is a small integer associated with each exemplar, which is initially one when a new exemplar is added to the tree. The mass field is necessary to preserve the definition of the $eval_{\mathbf{E}}$ operation, which now takes the mass of the exemplars into account in the local averaging operation (as in Equation 7.14 below). When an exemplar's mass is increased its date of birth is set to the current date (it is born again).

The obvious motivation for this optimization is that there is some saving in memory for very little extra work (provided that the memory for all the exemplars not duplicated exceeds the extra memory to include a mass field in each exemplar). However the main reason is that duplicate points stored in a *k*d-tree hinder nearest neighbour search which would inevitably have to search all of them if it would need to search one.

The mass feature is not included to alter the meaning of the exemplar set representation—it is a modification to the representation which preserves meaning.