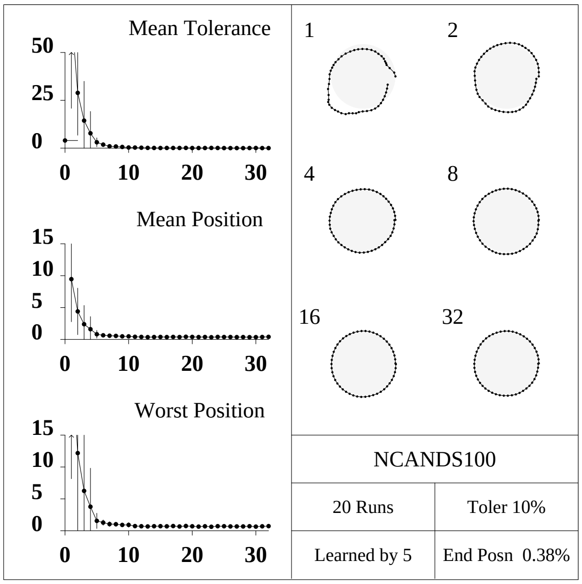


Figure 10.20

Experimental method described in the text. All parameters were default with the exception of the **Num-cands** parameter, which had value 100.

Num-cands	Learned By	End Position Err
4	Didn't Learn	11.37%
5	14	0.64%
6	13	0.45%
10 (BASIC)	8	0.41%
50	6	0.39%
100	5	0.38%
200	4	0.41%

Table 10.8: Trajectory tracking: Performance variation with number of candidate actions

approximately the same. This is because once success has been achieved, the hundred candidates do not need to be generated—the partial inversion action is always suitable.

Table 10.8 displays the performance of experiments with various values of the **Num-cands** parameter.

The importance of having more than a small number of candidates is demonstrated more clearly when the experiments are conducted with a very small value of **Tolerance**. Table 10.9 displays the performance of the system when the **Tolerance** is 0.2%. The 50 candidates experiment achieved the best final position error seen so far. This is because of the combination of aiming for behaviour very close to that desired by the ice puck controller and having sufficiently many candidates that this accuracy is achieved within the space of 32 trials.

Num-cands	Learned By	End Position Err
5	Didn't Learn	1.43%
10 (BASIC)	Didn't Learn	0.64%
50	24	0.17%

Table 10.9: Trajectory tracking: Performance variation with **`Num-cands`** with a very small **`Tolerance`** value (0.2%).

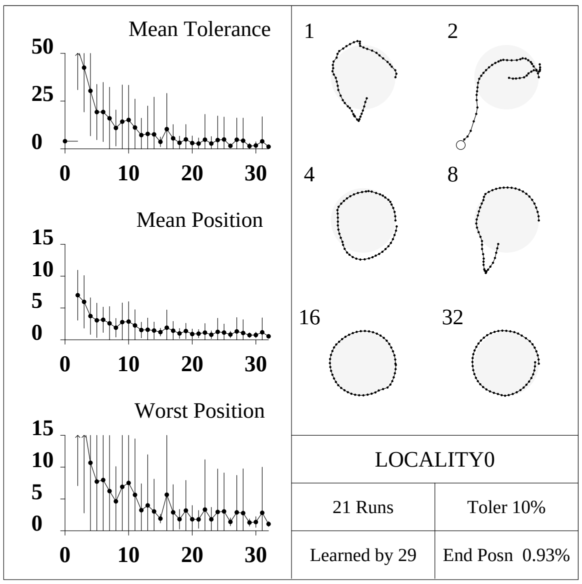


Figure 10.21

Experimental method described in the text. All parameters were default with the exception of the **`Locality`** parameter, which had value 0.

10.3.6 Other *SAB* action chooser Parameters

The **`Locality`** (P_{loc}) and **`Local-const`** (C_{loc}) parameters influence the generation of candidate actions. As discussed in Section 8.2, a candidate is chosen local to the best known action with probability **`Locality`**. If it *is* generated locally, then it is generated within a sphere which has radius proportional to the error of the current best known behaviour. The constant of proportionality is **`Local-const`**. The experiment with **`Locality`** of zero (in which candidates are always entirely random) is shown in Figure 10.21. The learning rate is significantly slower than that of BASIC. With **`Locality`** set to one the experimental results were almost identical to BASIC. These results are summarized in Table 10.10. The possible advantage of choosing a **`Locality`** less than one is not demonstrated here. This is the fear that only using local improvements might lead to local minima in the control choice.

The behaviour with the **`Local-const`** appears in Table 10.11. A low value of 0.1 slows down the learning because the local changes made are too small—the P_{succ} heuristic would score candidates higher which were further away, but with such a low **`Local-const`** such candidates are rarely generated. Values of **`Local-const`** between 0.5 and 3 do not greatly differ from the performance of

`Locality`	Learned By	End Position Err
0	29	0.93%
0.67 (BASIC)	8	0.41%
1	9	0.39%

Table 10.10: Trajectory tracking: Performance variation with `Locality`

`Local-const`	Learned By	End Position Err
0.1	18	0.67%
0.5	6	0.35%
2 (BASIC)	8	0.41%
3	15	0.45%

Table 10.11: Trajectory tracking: Performance variation with `Local-const`

the BASIC experiment.

The `Prob-const` parameter, C, alters how the P_{succ} heuristic judges the unreliability of nearest neighbours of differing distance. A low `Prob-const` causes the heuristic to score candidate actions as low if they are at a significant distance from the nearest exemplar. A high `Prob-const` treats the nearest neighbour estimate as reliable even if it is at a large distance. The results of a very low value (0.01) are shown in Figure 10.22. They are very bad! This is because nothing of any significant distance from the best known action is ever tried, and as a result the same action is repeatedly used throughout the entire learning run. A very high value (10), displayed in Figure 10.23, does improve, but takes longer than BASIC. The results of other values, tabulated in Table 10.12, indicate that performance is not altered greatly for values between 0.1 and 5.

10.3.7 *SAB*-tree parameters

The *SAB*-tree parameters are the relative weights given to the components of the state, action and behaviour vectors. In the BASIC experiment they were chosen roughly based on the observation

`Prob-const`	Learned By	End Position Err
0.01	Didn't Learn	20.55%
0.1	9	0.38%
1 (BASIC)	8	0.41%
2	9	0.49%
5	15	0.8%
10	Didn't Learn	1.88%

Table 10.12: Trajectory tracking: Performance variation with `Prob-const`

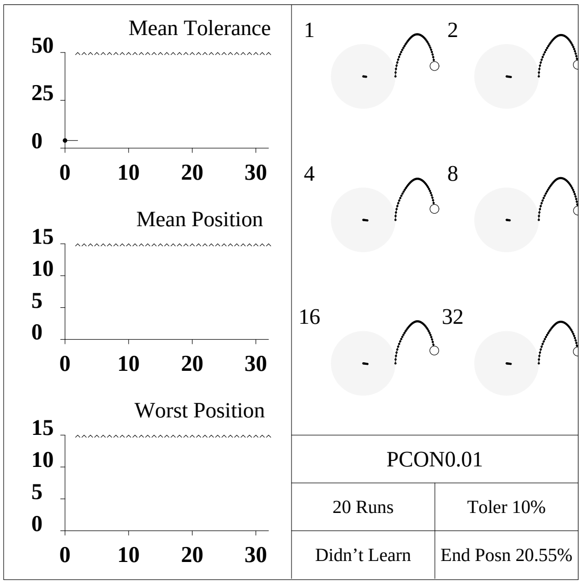


Figure 10.22

Experimental method described in the text. All parameters were default with the exception of the `Prob-const` parameter, which had value 0.01.

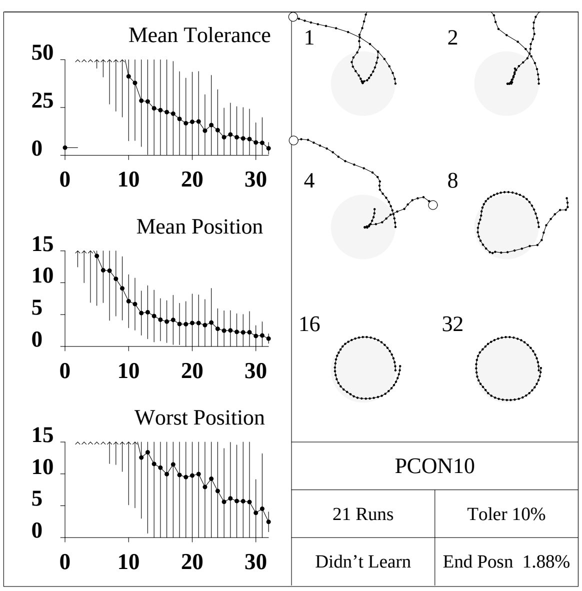


Figure 10.23

Experimental method described in the text. All parameters were default with the exception of the `Prob-const` parameter, which had value 10.

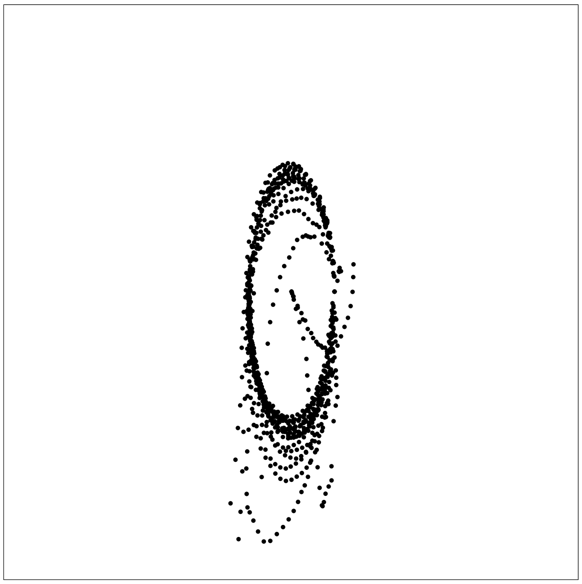


Figure 10.24

The distribution of exemplars in "position" space after a learning run of the WEIGHT.ALPHA Experiment.

of the maximum values of each aspect of behaviour. Four experiments were performed with different weight values. The weightings and results are tabulated in Table 10.13. For example, the WEIGHT.ALPHA experiment was told that the maximum possible perceived x-coordinate of the hand was 3m (BASIC had been told correctly that it was 1m). It was told that the maximum x-direction speed was three times greater than that of BASIC and it was similarly told that the maximum x-direction behaviour (i.e. acceleration) was three times greater than that of BASIC. As a result the nearest neighbour prediction was more sensitive to the y-direction. This is illustrated in Figure 10.24, in which the position components of the vectors stored in the *SAB*-tree are displayed.

The altered weighting can be seen to not have a great effect on performance. The reduced performance in the WEIGHT.GAMMA experiment is because the `Tolerance` is obtained from the scaled space of Behaviours. Thus, a tolerance of 10% when the maximal behaviour is 300 is equivalent to a tolerance of 30% when the maximal behaviour is 100.

The weights of the actions were not altered for any of the experiments. This is because their alteration would have also altered the strength of the arm.

The other *SAB*-tree parameter is the range width D_{range}. This parameter is called `Search-width`. The simulated noise is significant but not very large compared with the goal tolerance in the BASIC experiment and so a smoothing kernel of a lower width than the default can be expected to suffice. However, it is interesting to consider the result of increasing the smoothing kernel.

The results are shown in Table 10.14. They indicate that even with a large smoothing kernel, performance is not badly affected. However, the largest kernel, with width 0.2 of the scaled *SAB*-tree width never learns, despite achieving a fairly good final error position. This is because the very strong smoothing makes a constant regular error (similar to smudging an image). As a result

Experiment	Maximum Components			Learned	End Position
	State	Action	Behaviour	By	Err
BASIC	(1,1,10,10)	(6,6)	(100,100)	8	0.41
WEIGHT.ALPHA	(3,1,30,10)	(6,6)	(300,100)	6	0.59
WEIGHT.BETA	(3,3,30,30)	(6,6)	(100,100)	13	0.44
WEIGHT.GAMMA	(1,1,10,10)	(6,6)	(300,300)	7	1.74
WEIGHT.DELTA	(3,3,10,10)	(6,6)	(100,100)	7	0.37

Table 10.13: Trajectory tracking: Performance variation with a variety of weight sets

`Search-width`	Learned By	End Position Err
0.002	9	0.49%
0.02 (BASIC)	8	0.41%
0.06	11	0.44%
0.12	12	0.48%
0.2	Didn't Learn	0.6%

Table 10.14: Trajectory tracking: Performance variation with `Search-width`

the behaviour predictions have a regular constant error. In this case the error is fortunately not large enough to significantly impair the position.

The experiment with a search width of 0.2 took approximately 10 times longer than the BASIC experiment. This is because of the large range search required for each update of the *SAB*-tree.

10.3.8 Learning only after Execution

In this experiment I consider the performance if the computation involved with storing new exemplars is taken out of the *SAB* control cycle, and is performed off-line. Trajectory tracking falls into the class of task in which there are natural periods in which extra processing can take place—namely before each trial occurs. This experiment uses this period of time to update the *SAB*-tree with all the data observed in the previous trial. Thus learning does not occur *during* each individual task execution. The results are in Figure 10.25. Performance is reduced, compared with the performance of the BASIC experiment, but not very badly. This result might be useful if the expense of updating the *SAB*-tree were too large to perform within the timescale of the robot's dynamics.

10.3.9 Approximating the Nearest Neighbour

The nearest neighbour search described in Chapter 6 performs a branch and bound search of the *k*d-tree. An objection to this is that although computation is expected to be fairly cheap ($O \log N$), it is not guaranteed. A partial remedy, suggested in [Omohundro, 1987] is to perform normal

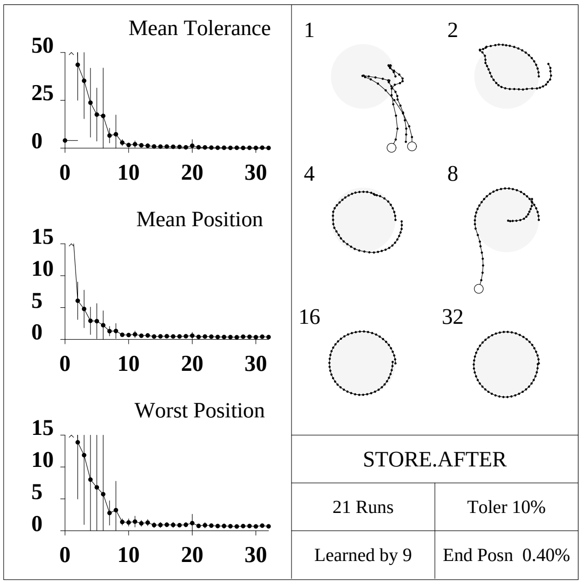


Figure 10.25

Experiment in which the *SAB*-tree is only updated after each trial instead of during each trial. Note the appalling behaviour in trial one—no knowledge of the world is available for the entire trial.

`Max-ninsp`	Learned By	End Position Err
1	13	0.52%
2	11	0.55%
5	10	0.50%
∞ (BASIC)	8	0.41%

Table 10.15: Trajectory tracking: Performance variation with `Max-ninsp`

nearest neighbour search, but after a fixed number, `Max-ninsp`, of nodes have been inspected, to halt the search with the nearest neighbour found to date.

This method was tested for `Max-ninsp` values of 1, 2 and 5. When `Max-ninsp` = 1 the search consists simply of finding the leaf node containing the query point. The results are given in Table 10.15. They show that learning does still occur well, with only slight degradation against the BASIC experiment.

The real time to perform the experiments was not noticeably less than for experiments involving normal nearest neighbour search. A disadvantage of approximating the nearest neighbour is the increased difficulty of analysis of the learning method and some danger of erroneous predictions becoming stuck.

10.3.10 Noise Tolerance

In this section the amount of noise in the environment is increased. We then inspect the extent to which the smoothing of Section 7.2 can help improve the performance. The noise in the BASIC experiment was produced by corrupting the requested torques by a random amount up to 0.2% of

Search Width	Final Position Error Noisy	Very Noisy
0.2%	1.45%	6.0%
2%	1.22%	6.7%
6%	0.93%	5.52%
12%	0.76%	3.12%

Table 10.16: Trajectory tracking in a noisy environment: Performance variation with `Search-width`

the maximum torque values. This can be observed to produce unpredictable behaviour corruptions of up to 1%. This default noise level is not severe, and does not impact greatly on the performance of the system. In eight experiments with much greater noise the smoothing kernel width was varied to discover how much effect it had on combating noise. The performance could never be expected to reach that of the noise free system, because the unpredictable noise was always being added. The controller has a similar problem to a person trying to do neat handwriting on a bumpy train journey.

The noisy environment had the action corrupted by up to 5% and the very noisy environment by up to 20%. The results are in Table 10.16. None of the above experiments achieved a "Learned By" value. This is predictable because the environment is so noisy. There is a noticeable improvement in the final position error as the search width is increased, but it is not sufficient to be classed as achieving the task well. Figure 10.26 shows the performance in the very noisy environment with a low search width, and Figure 10.27 shows performance with a large search width. The final animations were taken without noise. This helps indicate how accurate a model of the noiseless world the system has after 30 noisy trials.

10.3.11 Non-stationary Environment

Recalling Section 7.3, the *SAB* learning controller should be able to adapt to changes in the environment by noticing that old points are inaccurate and deleting them. Here, we test this behaviour. The standard BASIC experiment is used, except that unknown to the controller, just before trial 11, the arm's dynamics are changed severely. The motor for the elbow starts producing $1\frac{1}{2}$ times the requested torque. The results of Figures 10.28 and 10.29 show the effect on performance. The animations correspond to different trial numbers from the earlier figures. Trial ten, which is the last trial before the change, is shown first. A selection of subsequent trials are also shown, and indicate the behaviour after the change, both in the long and short terms. Figure 10.28 shows the effect with change adaptation switched off—points are never thrown away no matter how old or inaccurate they are. Thus, after trial 10, mistakes can be seen to be made for the rest of the run. With adaptation switched on (Figure 10.29), using a `Search-width` parameter of 6%, adjustment performance can be seen to be superior, with an eventual return to acceptable performance. The

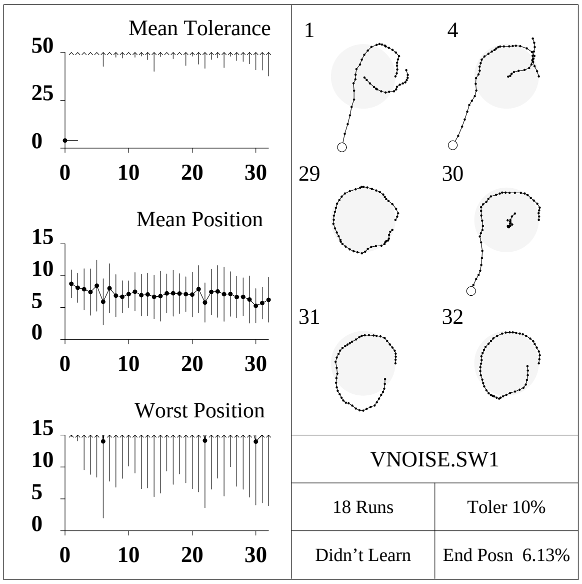


Figure 10.26

Experimental method described in the text. All parameters were default with the exception of the **`Search-width`** parameter, which had value 0.2%, and the **`Noise-level`** parameter which had value 20%.

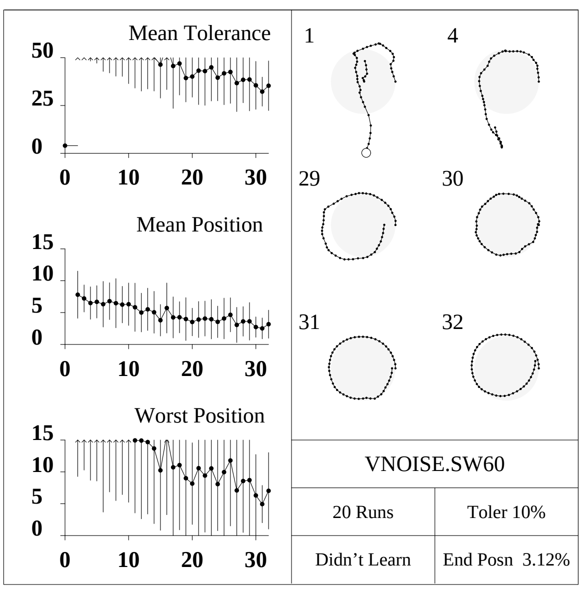


Figure 10.27

Experimental method described in the text. All parameters were default with the exception of the **`Search-width`** parameter, which had value 12%, and the **`Noise-level`** parameter which had value 20%.

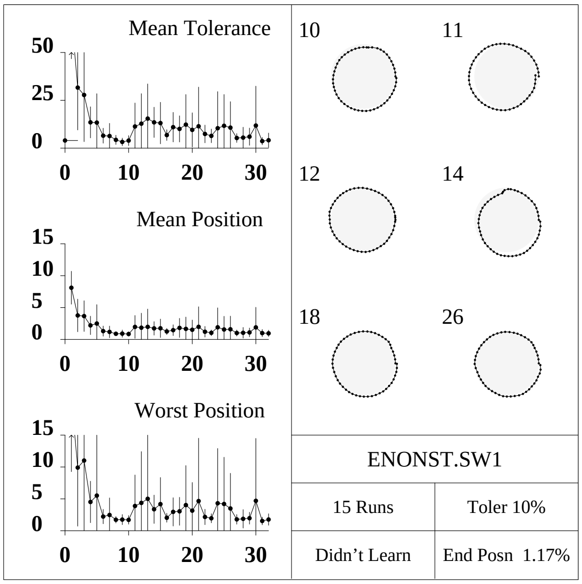


Figure 10.28

A non-stationary environment. The arm dynamics change on trial 10 to produce 1.5 times the torque that they used to. This experiment has a very small search width, meaning very few old inaccurate exemplars are killed.

return to improved performance is, however, not as fast as was the original learning which started from no experience. It is likely that there are two compounded reasons for this.

1. The environmental change is very large, meaning that the new PSTF is, to a large extent, an entirely different function.

2. The learning controller is not warned that a change has occurred, and so must have time to observe this.

From these results it could be argued that the simpler scheme of forgetting everything after a period of increased average prediction error would be preferable. The objection to this is that real environmental changes are likely to (i) be much less severe and (ii) have an affect only in a limited portion of the state space, meaning a loss of valuable data if all information is removed.

10.3.12 Performance of the *k*d-tree

The real-time performance of these learning trials was observed to be good: typically 20 seconds to simulate, learn and display. This gives indirect evidence that nearest neighbour search is performing as desired. To test this more directly the search algorithm was monitored for how many Euclidean distance measurements were taken. These measurements are the dominant cost in nearest neighbour search—we can be sure that the cost of reaching the leaf node in the tree is logarithmic (since the tree is balanced whenever the depth exceeds a small constant times the logarithm of the number of nodes in the tree). Furthermore, reaching the leaf is cheap because Euclidian distance calculations are not required. The details of the nearest neighbour search algorithm are given in Section 6.4.

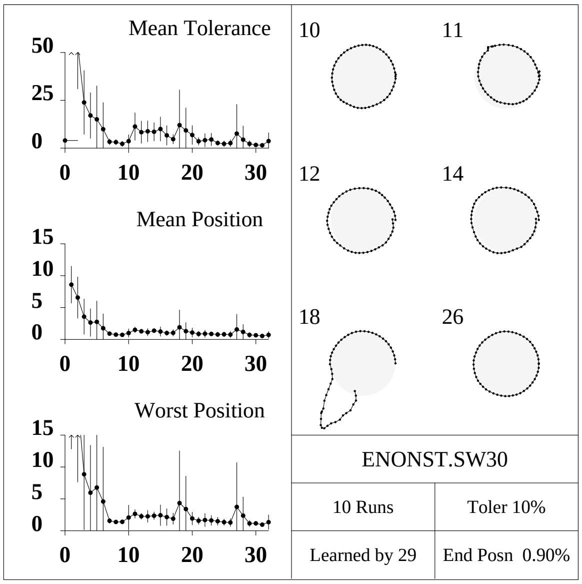


Figure 10.29

A non-stationary environment. The arm dynamics change on trial 10 to produce 1.5 times the torque that they used to. This experiment has a reasonable search width (6%), meaning that some old inaccurate exemplars are killed.

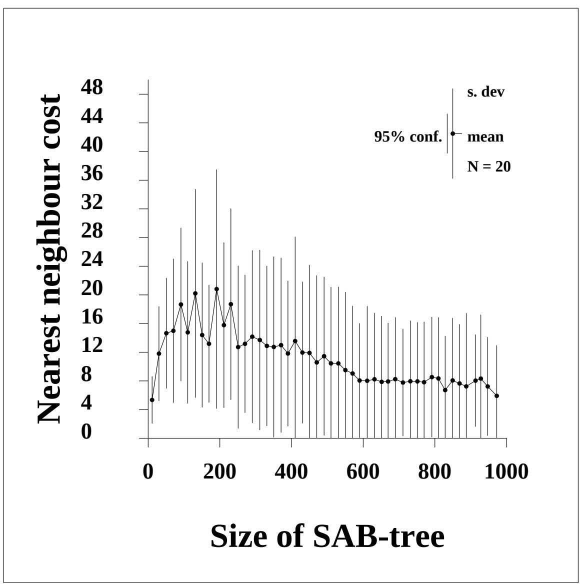


Figure 10.30

The number of *k*d-tree nodes processed. These results use binned averages from 20 independent trials.

The graph of the cost of search against tree size is shown in Figure 10.30. The result is remarkable: although the cost increases initially with tree size, it actually *decreases* as the tree gets larger! The typical number of distance calculations in the final trials is 6.

This is bizarre, but can be explained by the improving behaviour of the arm as learning progresses. The BASIC experiment achieved tolerable behaviour by, on average, the ninth trial. At this point, when partial inversion is used to obtain a candidate action, the requested behaviour and state are very close to the (correct) state-behaviour pair experienced on the preceding, also

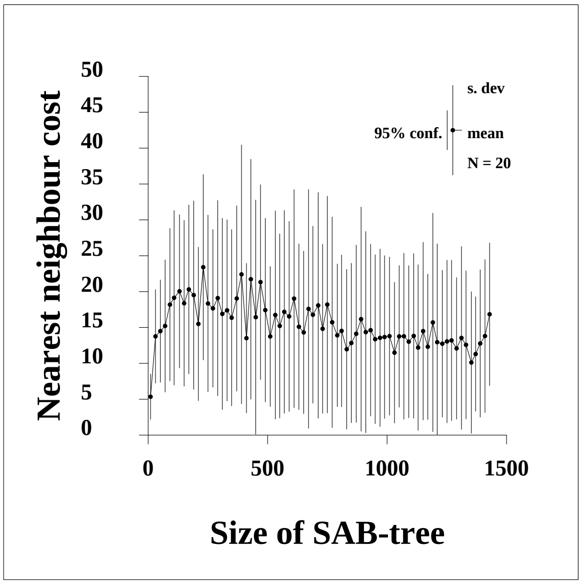


Figure 10.31

The number of kd-tree nodes processed. These results use binned averages from 20 independent trials.

successful trial. Thus there is often a very nearly exact match for the nearest neighbour search to find, which usually means very little search backtracking is required.

To test this theory, we view the corresponding graph (Figure 10.31) for an experiment in which satisfactory performance was never attained. Here we use an experiment with a large amount of simulated noise. In this case it can be seen that the typical number of distance calculations is 15.

The conclusion is that, as hoped, there is empirical evidence that the nearest neighbour computation can be performed in real time because the distribution of exemplars is conducive to nearest neighbour search, and can actually *improve* the speed as the task is learned more accurately.

10.4 Further Arm Experiments

These experiments are concerned with how the arm would perform if it were given a more complex task than tracking a prespecified trajectory. Many robot tasks, such as "pick and place" require that the robot move to a variety of positions, and it is interesting and important to see how performance is affected as the task demands that increasing subsets of the control space are learned.

The experiments also demonstrate the use of ice puck control, described in section 9.2. In these experiments trajectories are generated dynamically and autonomously. Figure 10.32 shows an example of a movement task, including the (stationary) start and goal states. The goal is to make the movement in 25 time steps. Figure 10.33 shows the ice puck trajectory generated to achieve this puck goal. Because puck trajectories are so cheap to compute they are recalculated every control cycle instead of being tracked. Figure 10.34 shows the trajectory recomputed after three very unskilled action choices had taken the hand far from the original trajectory.

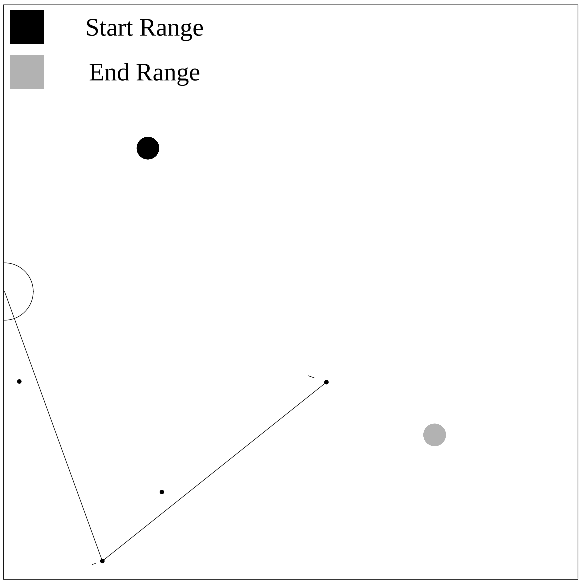


Figure 10.32

The start and end positions for the 1-d explore task. The start state and goal state both have zero velocity.

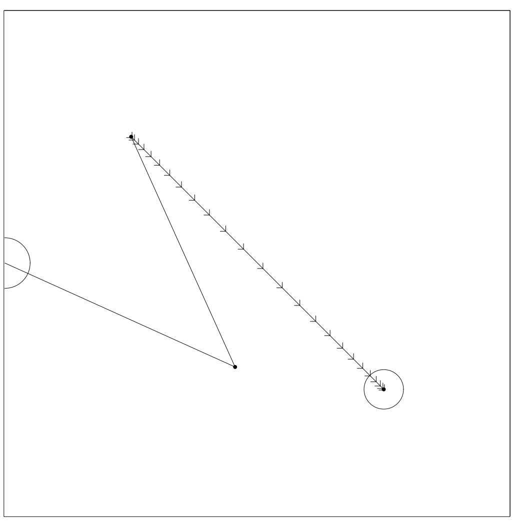


Figure 10.33: The ice puck trajectory to the goal state in 25 time steps, beginning at the exploration start state

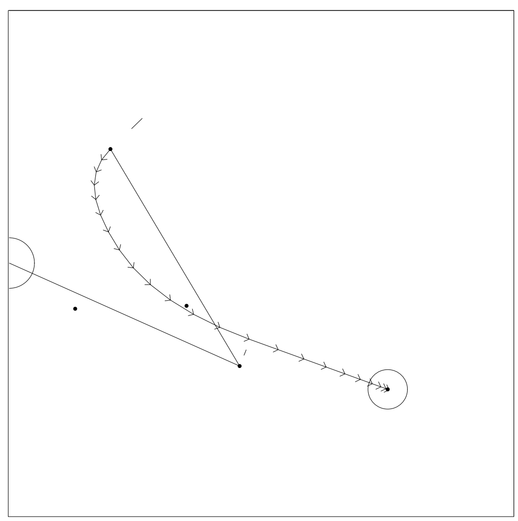


Figure 10.34: The ice puck trajectory to the goal state in 22 time steps, beginning at the state which the arm has arrived in after three, very unskilled, time steps.

Figures 10.35—10.38 demonstrate by means of animations how performance improves upon repetitive attempts at the movement task. It is seen to achieve the puck goal on only the third attempt, and after ten attempts the hand deviates very little from the straight line trajectory computed for the first step. This is indicated in Figure 10.39 which shows the ice puck trajectory generated three steps into a later trial. It matches almost exactly the initial generated trajectory of Figure 10.33.

A more quantitative indication of the performance of the movement task is gained in Figure 10.40 which graphs the mean tolerance error for a number of movement task runs and Figure 10.41 which graphs the final position error. The final position error is a representative statistic of how well the task was performed. The puck goal was to achieve the position in forty time steps. The error after forty time steps was used as the statistic, but the trial was allowed to continue to try getting closer to the puck goal for a further period of time on the early occasions in which the hand did not achieve 3% position error or less within forty steps. The system parameters were all default.

10.4.1 Higher Dimensional Tasks

The previous movement task, once learned, was no harder than the trajectory tracking task, because each subsequent trial would require the same journey through state space. The task requires a one-dimensional strand through state space and so let us call it a *one-dimensional task*. To test how the performance would degrade with increasing dimensionality I devised some two, three, four and five-dimensional tasks. In each case the task generates a random start position and a random end position, each distributed uniformly within certain subspaces of the arm's reaching domain. Varying selections for the start and end distributions produce different dimensionality tasks. They are depicted in Figures 10.42—10.45.

An interesting thing to examine is the distribution of exemplars after the tasks of increasing dimensionality. Figures 10.46—10.49 show a snapshot of the position components of the exemplars in the SAB-tree after twenty random trials. They give an indication of how the dimensionality, and hence the amount that needs to be learned, increases. For example, the trajectories in the two-dimensional diagram all have similar speeds about half way through, which helps nearest neighbour to generalize better than in the three-dimensional tasks, in which the speed of the hand midway through trials varies more greatly.

The tolerance error graphs for these tasks are shown in Figures 10.50—10.53 and the final position error graphs in Figures 10.54—10.57. The graph data in summarized in Table 10.17, in which the "Learned by" statistic in the number of trials until the mean tolerance error was permanently less than 10ms^{-2}, which corresponds to 0.3% of the range of possible tolerance errors.

The table indicates that learning is impaired by tasks of higher dimensionality. However, the nearest neighbour generalization is in this case sufficiently powerful that performance, in each case, does eventually reach an adequate level. This behaviour is not guaranteed in general, and so one

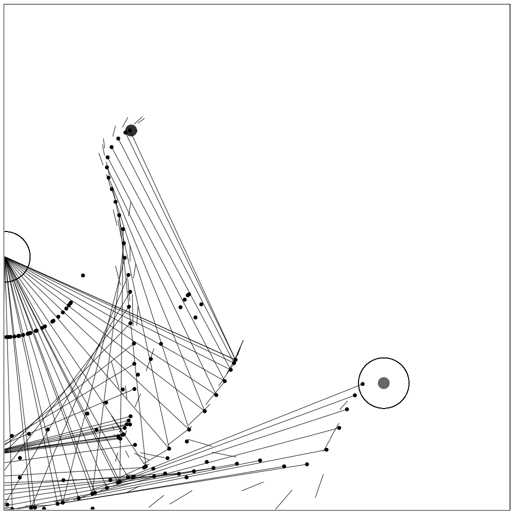


Figure 10.35: Animation of the first explore attempt. The goal time was 40 time steps, but this took much longer to even reach the target's neighbourhood.

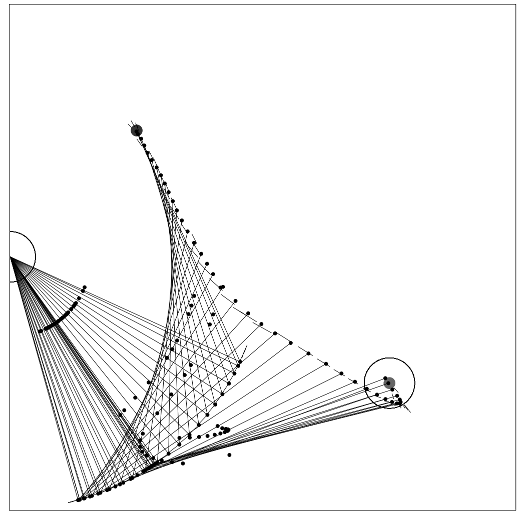


Figure 10.36: Animation of the 3rd explore attempt

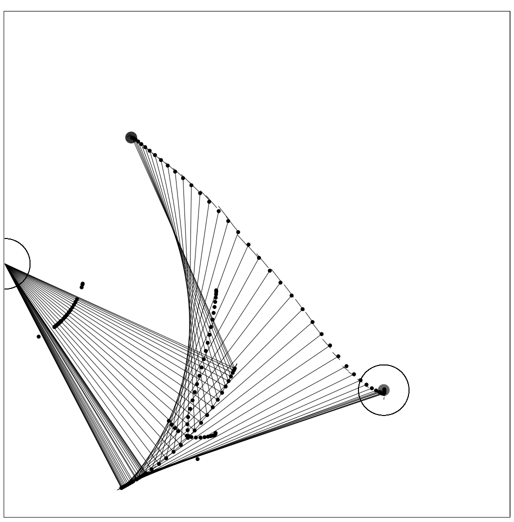


Figure 10.37: Animation of the 6th explore attempt

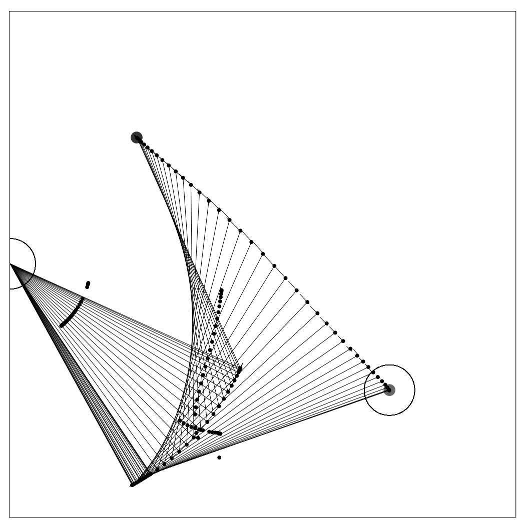


Figure 10.38: Animation of the 15th explore attempt

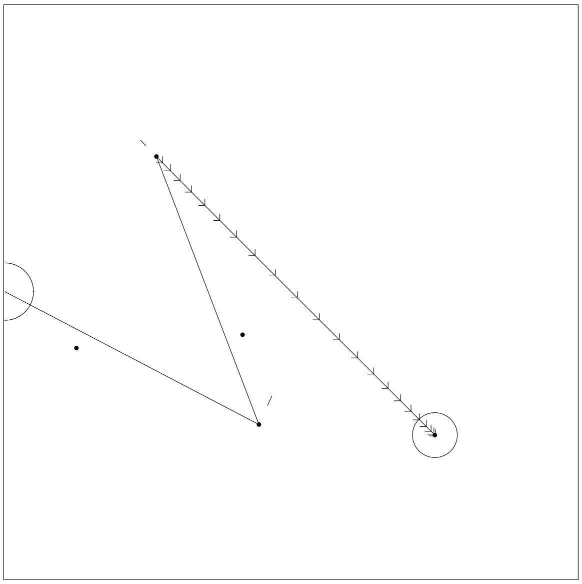


Figure 10.39

The ice puck trajectory to the goal state in 22 time steps, beginning at the state which the arm has arrived in after three well executed time steps. Notice that is nearly identical to the trajectory of Figure 10.33.

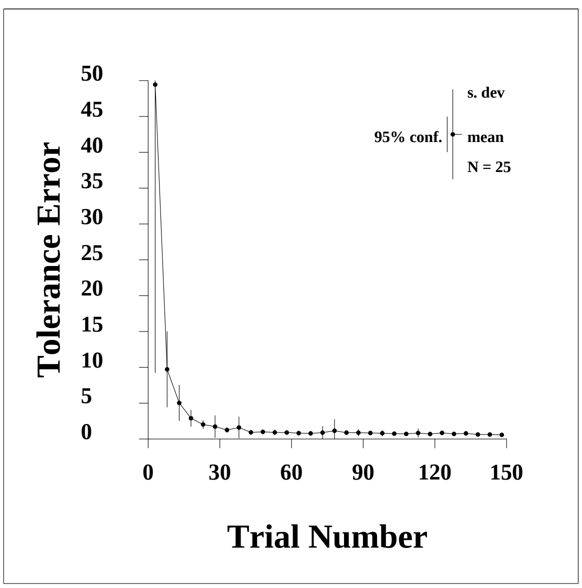


Figure 10.40

1-d exploration. The mean `Tolerance` error graphed against exploration attempt. This is the mean difference between acceleration requested by the Ice Puck controller and that obtained by the *SAB* action chooser. The range of possible accelerations is 300.

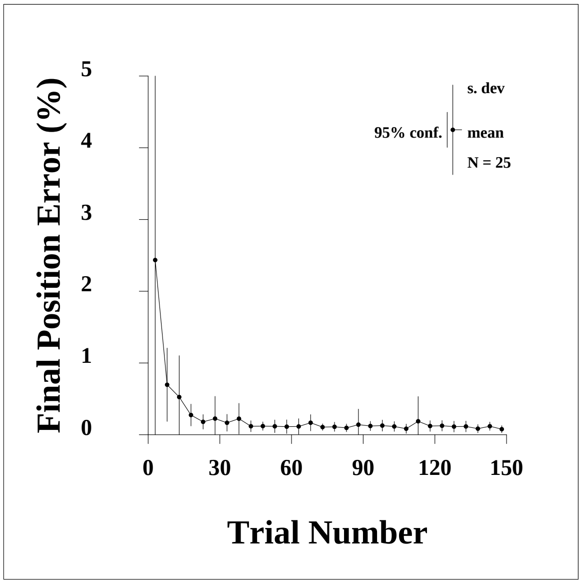


Figure 10.41

1-d exploration. The percentage final position error. This is the distance of the hand from the goal after 40 time steps, expressed as a percentage of the range of possible distances.

Explore Dimension	Learned By	End Position (%)
1	8.3	0.11
2	9.3	0.2
3	12.4	0.36
4	20.9	0.5
5	26.9	0.39
5 (decomp.)	24.8	0.29

Table 10.17: Exploration performance summarized

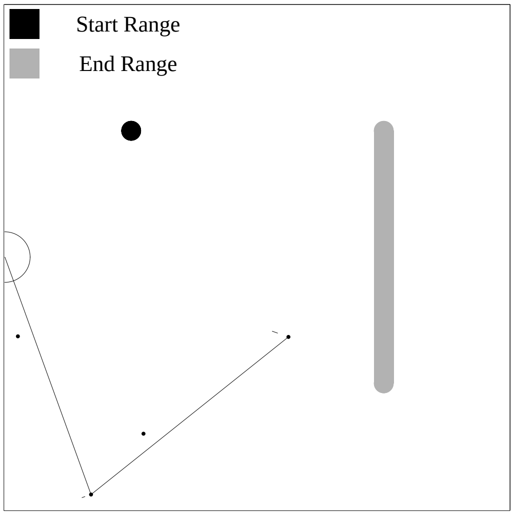


Figure 10.42: **2-d exploration.** The arm always starts in the same state, but the goal is chosen uniformly randomly from a 1-d set of states.

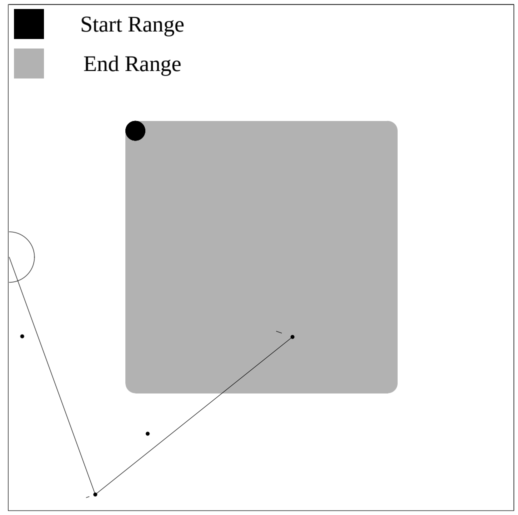


Figure 10.43: **3-d exploration.** The arm always starts in the same state, but the goal is chosen uniformly randomly from a 2-d set of states.

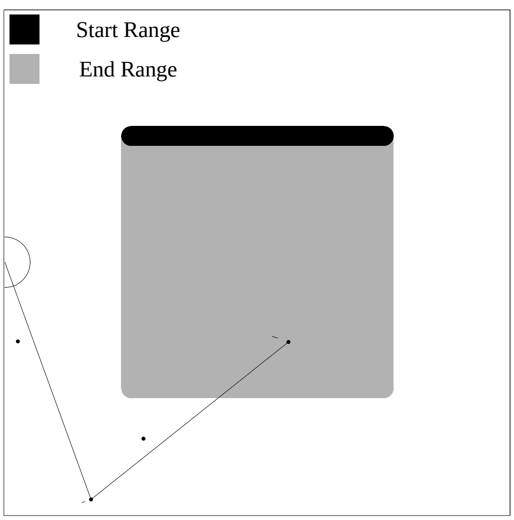


Figure 10.44: **4-d exploration.** The arm starts randomly, from a 1-d set of states. The goal is chosen uniformly randomly from a 2-d set of states.

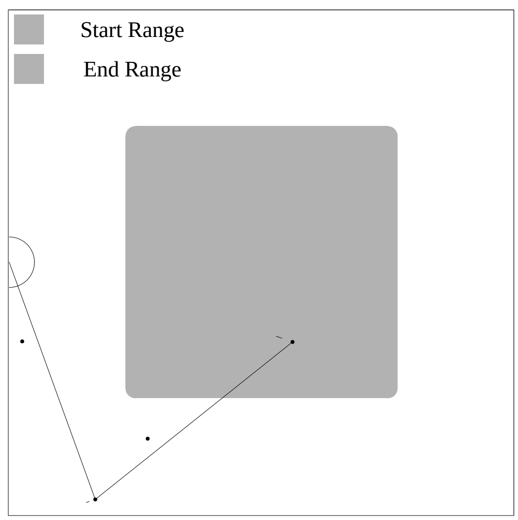


Figure 10.45: **5-d exploration.** The arm starts randomly, from a 2-d set of states. The goal is chosen uniformly randomly from a 2-d set of states.

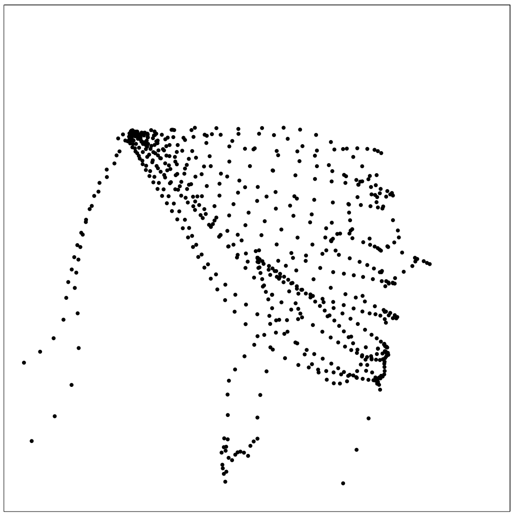


Figure 10.46: **2-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

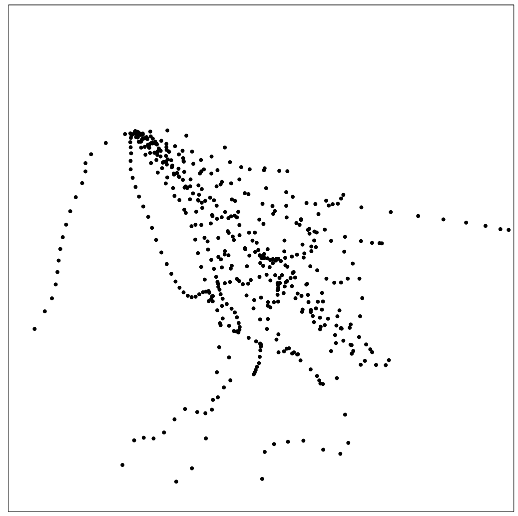


Figure 10.47: **3-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

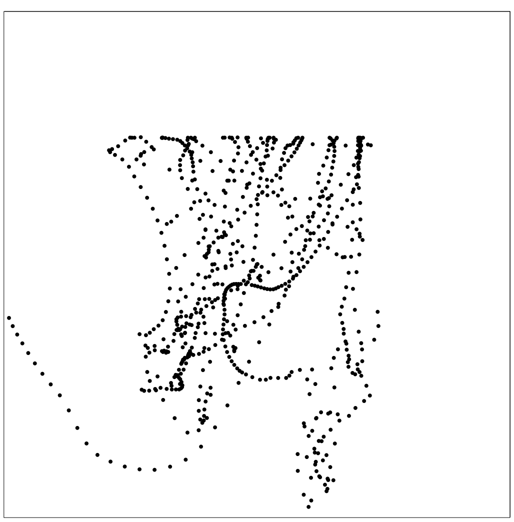


Figure 10.48: **4-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

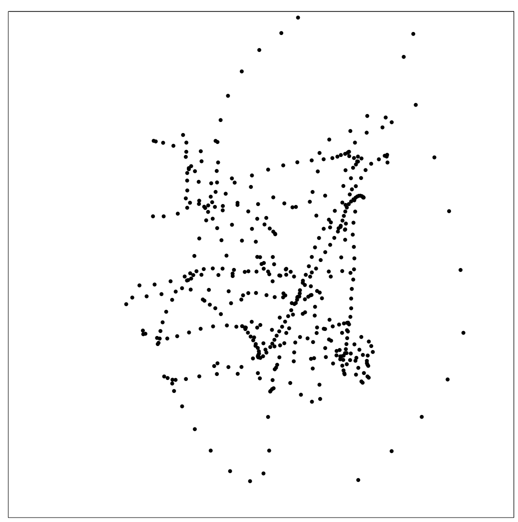


Figure 10.49: **5-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

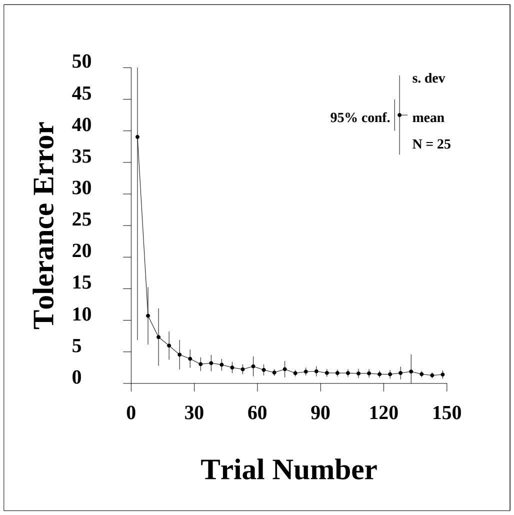


Figure 10.50: **2-d exploration.** The mean `Tolerance` error.

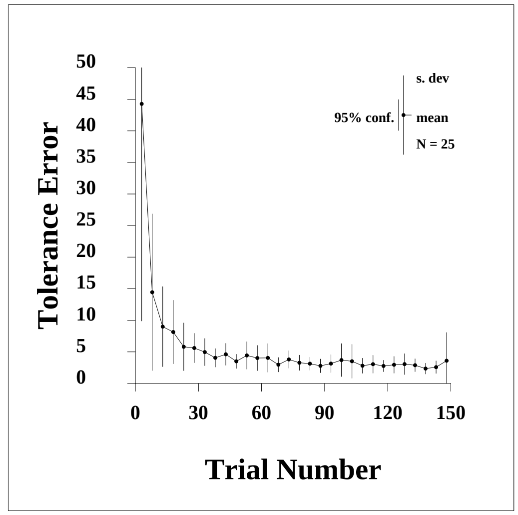


Figure 10.51: **3-d exploration.** The mean `Tolerance` error.

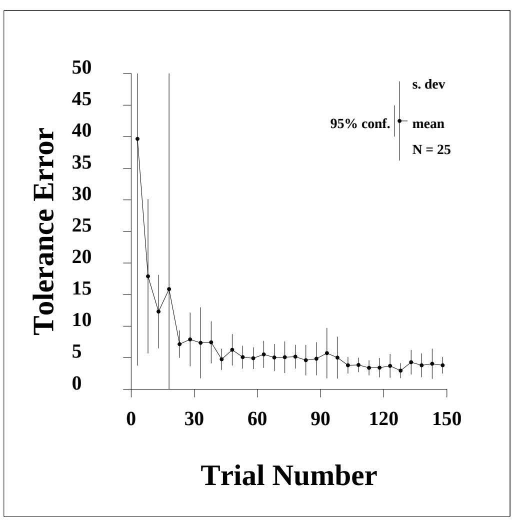


Figure 10.52: **4-d exploration.** The mean `Tolerance` error.

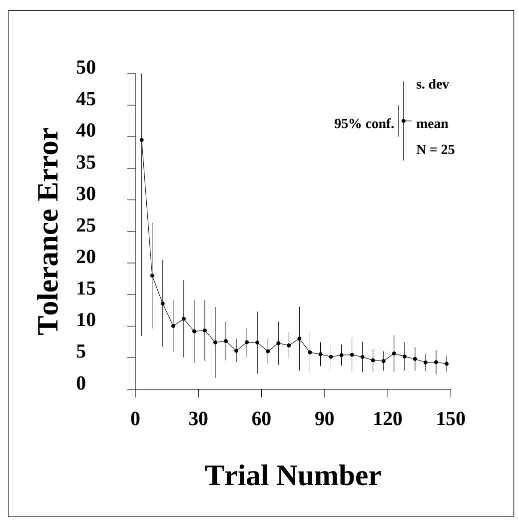


Figure 10.53: **5-d exploration.** The mean `Tolerance` error.

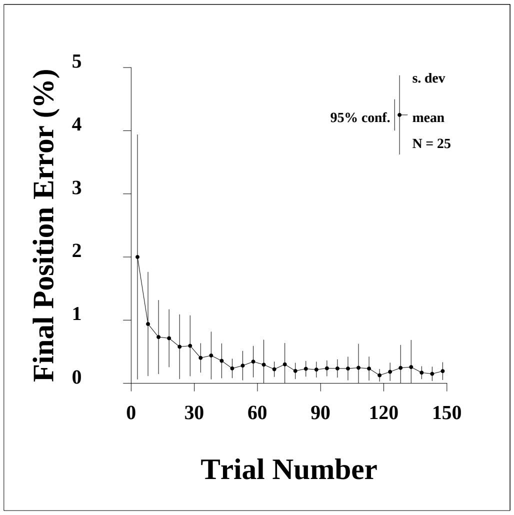


Figure 10.54: **2-d exploration.** The mean final position error, expressed as a percentage.

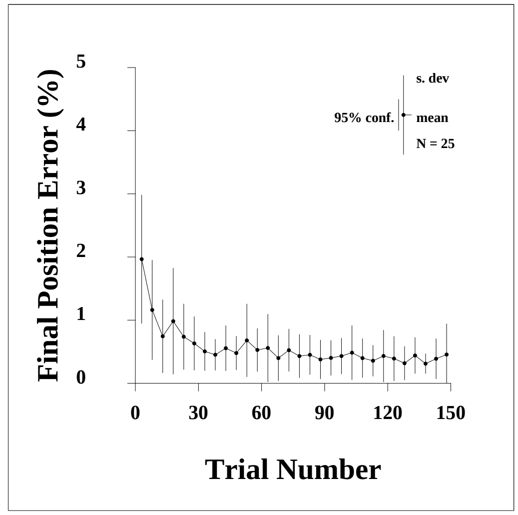


Figure 10.55: **3-d exploration.** The mean final position error, expressed as a percentage.

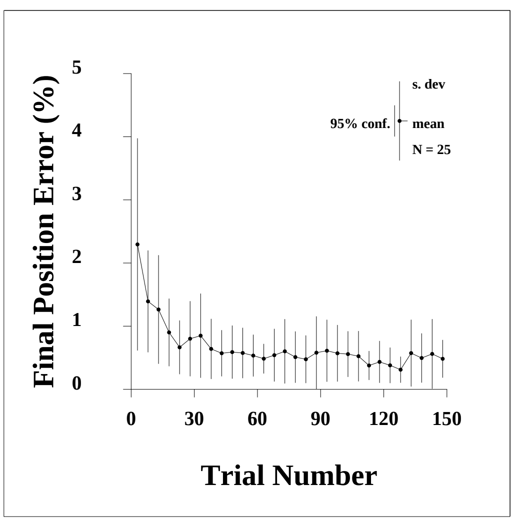


Figure 10.56: **4-d exploration.** The mean final position error, expressed as a percentage.

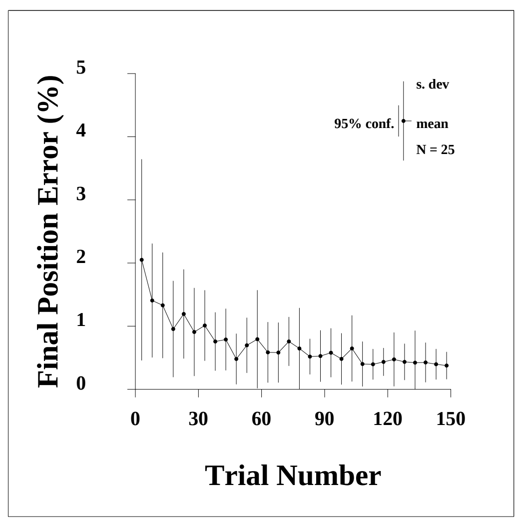


Figure 10.57: **5-d exploration.** The mean final position error, expressed as a percentage.

solution might be, when a high dimensional task is required, to use an abstract controller which breaks it up into low dimensional tasks. This is exemplified by the five-dimensional *decomposed* task. Instead of moving from a random start position to a random end position in forty time steps, the controller uses two tasks:

1. In twenty time steps, take the hand from the random start position to near a stationary state at the centre of the space.

2. Then, in a further twenty time steps move to the random goal position.

The task description, exemplar distribution, tolerance graph and position graph are shown in Figures 10.58—10.61. The summary statistics are included in Table 10.17. It is seen that both learning speed and accuracy improve as a result. It is conjectured that for even higher dimensional tasks in higher dimensional state spaces this kind of dimension reduction at the control level will rather than merely improving performance, be essential for a reasonable learning rate.

10.4.2 Autonomous Time Choice

In the experiments of the previous section the goal was a specified position in forty time steps. Here, we make the task more abstract and simply specify the goal position and leave it up to the world model-based controller to arrive at the goal in a reasonable time, which varies according to how far apart the start and end states are, and where they are situated. The abstract controller here is used as an indication of the kind of simple mechanism which can be used, based on learned world models, to increase the abstraction of the task. It is not to be considered as a serious attempt at a robust optimal controller.

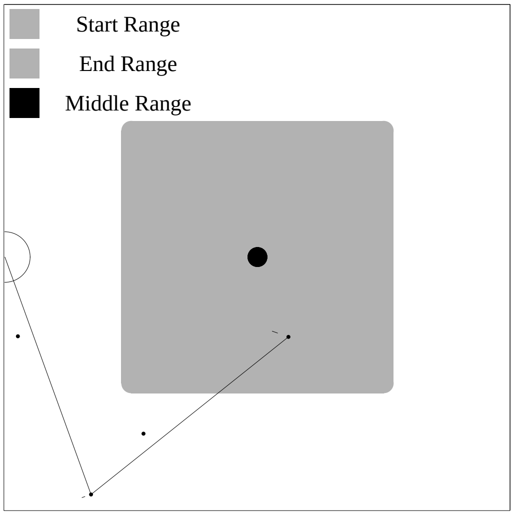
The decision of how many time steps are required is achieved by means of the strength estimation mechanism of Section 8.3. The controller is told the start and end states for the task, and then performs a binary search of possible times to goal. For each possible time to goal it estimates if it has the strength to achieve the time to goal, using an ice puck trajectory. It does this by generating the ice puck trajectory, and then using the strength mechanism to decide if

1. With probability greater than P_{res} (specified by system parameter `presig`, and given value 0.2), twice the required acceleration at the start state can be obtained.

2. With probability greater than P_{res} twice the required acceleration half way through the trajectory can be obtained.

3. With probability greater than P_{res} twice the required deceleration at the end of the trajectory can be obtained.

This is only a heuristic, and is certainly not likely to be near optimal control. Twice the required acceleration is used in the strength estimation to provide a degree of robustness.



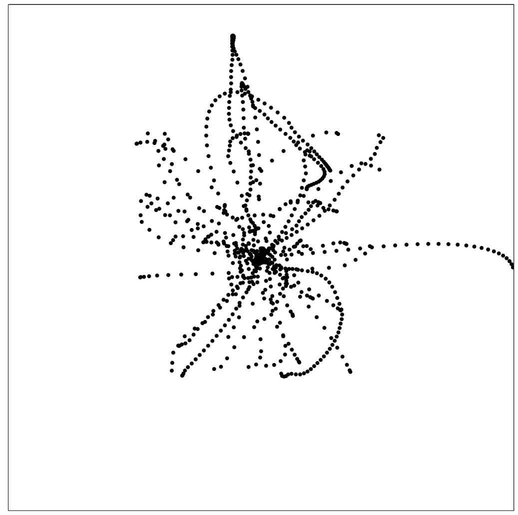
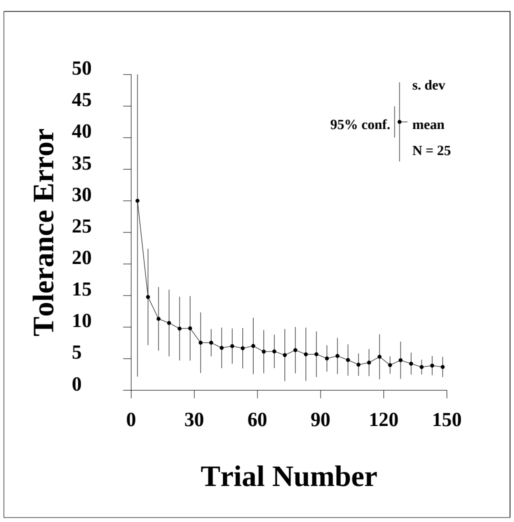


Figure 10.58: **5-d decomposed exploration.** The arm starts randomly, from a 2-d set of states. The goal is chosen uniformly randomly from a 2-d set of states. The controller splits up the problem to reach the stationary middle state half way through path.

Figure 10.59: **5-d decomposed exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.



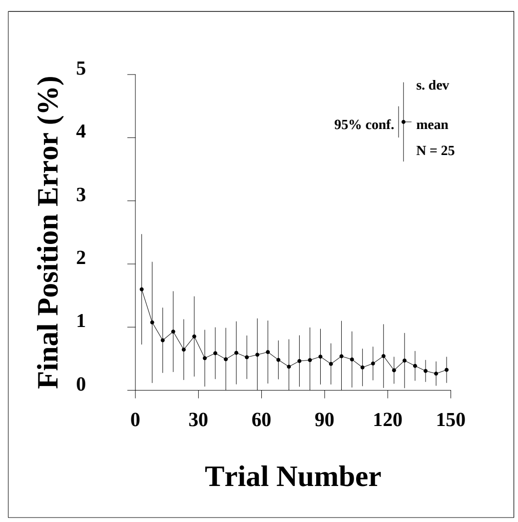


Figure 10.60: **5-d decomposed exploration.** The mean `Tolerance` error.

Figure 10.61: **5-d decomposed exploration.** The mean final position error, expressed as a percentage.

Explore Dimension	Learned By	End Position (%)
1	18.6	0.36
2	31	0.43
3	43.5	0.46
4	87.9	0.44
5	Didn't Learn	0.5
5 (decomp.)	64.1	0.57

Table 10.18: Autonomous time-choice exploration performance summarized

Explore Dimension	Mean time steps to goal
1	24
2	22
3	24
4	29
5	28
5 (decomp.)	36

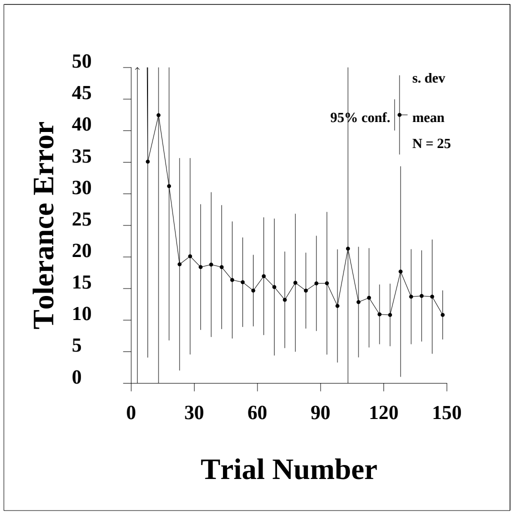
Table 10.19: Autonomous time-choice: mean time per trial

The results for autonomous time choice are shown in Tables 10.18 and 10.19. The times to achieve the goal are all, on average, quicker than forty steps used in the previous task. Learning is slower in all cases because (i) the increased average speed means an increased diversity of the accelerations requested of the *SAB* action chooser and (ii) some extra variation is introduced in the tasks in the initial stages before accurate estimates of strength become available. By the standard of the "Learned by" criterion, the 5-d task does not learn within 150 trials, although the 5-d decomposed task does. The decomposed task is necessarily slower, because the indirect route to the goal is often less efficient.

10.4.3 Disordered Environment

The autonomous-time-choosing 5-d trial was run in a noisy environment. The noise level was sufficient to randomly corrupt the acceleration by typically up to 5ms^{-2} which is 1.5% of the range of accelerations. Whilst subjected to this uncontrolled noise, both tolerance error and final position error are seen to improve with trial number in Figures 10.62 and 10.63.

The next experiment was to confirm a feature of *SAB* learning which was confidently predicted prior to the experiment—that concentration on one particular task cannot cause "over-learning". An autonomous-time-choosing 1-d trial was run for forty trials using the task in Figure 10.32. The task was then changed to an entirely different start and end position for the remaining 110 trials. Figures 10.64 and 10.65 show the results.



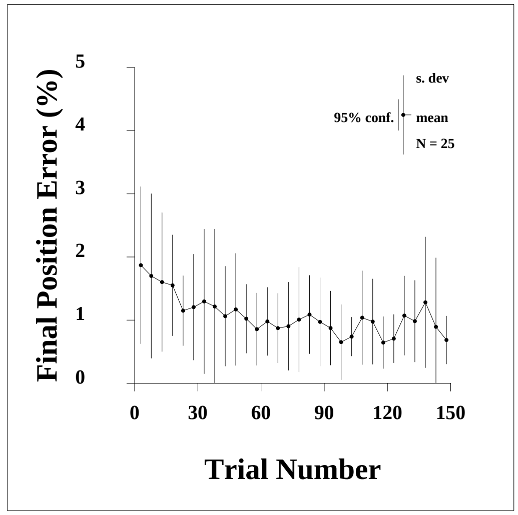
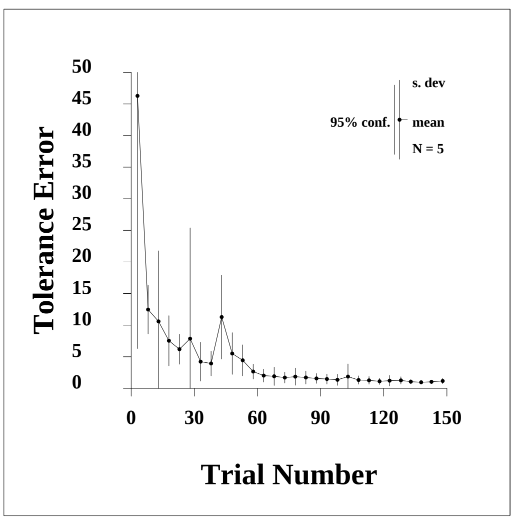


Figure 10.62: 5-d exploration in a noisy environment. The mean `Tolerance` error.

Figure 10.63: 5-d exploration in a noisy environment. The mean final position error, expressed as a percentage.



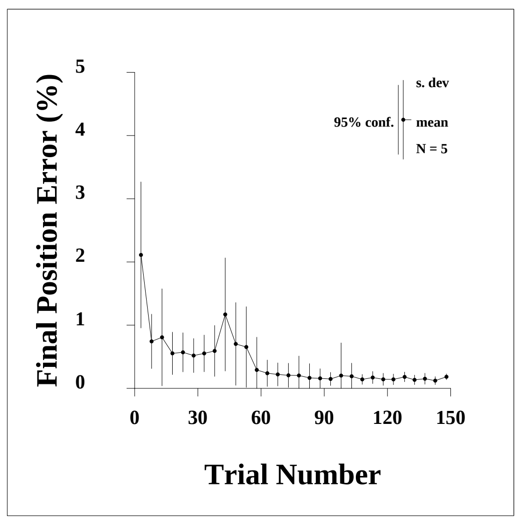


Figure 10.64: 1-d exploration in which the task changes on trial 40. The mean `Tolerance` error.

Figure 10.65: 1-d exploration in which the task changes on trial 40. The mean final position error, expressed as a percentage.

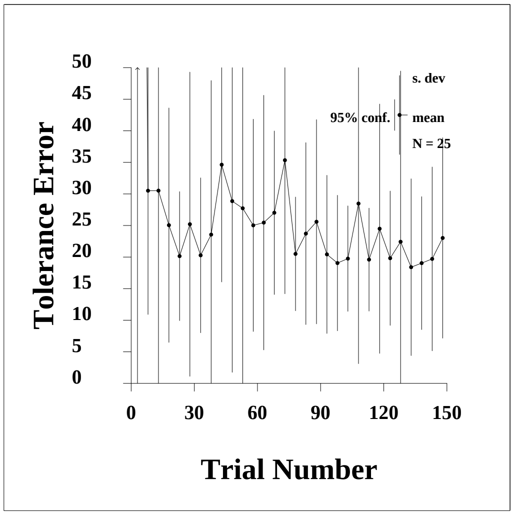


Figure 10.66: 5-d exploration in a non-stationary environment. The mean `Tolerance` error. Dynamics change on trial 40.

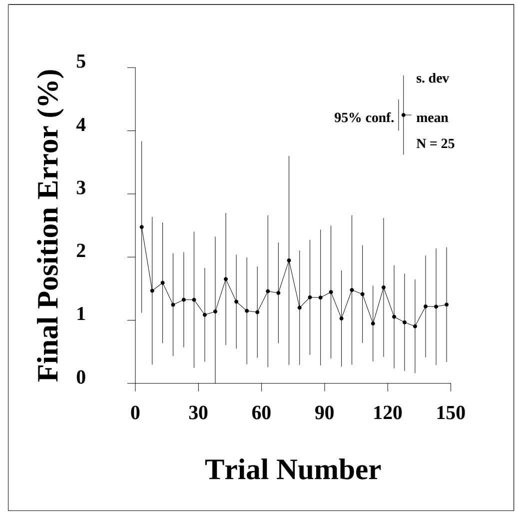


Figure 10.67: 5-d exploration in a non-stationary environment. The mean final position error, expressed as a percentage. Dynamics change on trial 40.

The final experiment was adaptation to a changing environment for the 5-d autonomous-time-choosing task. For this experiment the `Search-width` parameter was tripled. It can be seen in Figures 10.66 and 10.67 that neither improvement nor reduction in performance occurred, except to a small extent for the Tolerance error, which tended to increase slightly for approximately thirty trials after the change. The reason for the lack of large effect is likely to be because the distribution of exemplars was so wide (due to the task's high dimensionality, compounded by variations due to autonomous time choice) that misleading exemplars were not noticeably inaccurate, and were possibly more use than no information at all.

10.5 Juggling a Ball

This simulated experiment is motivated by a similar (but real) experiment by [Aboaf *et al.*, 1989] A visually observed ball is bounced on a two-dimensional surface (see Figure 10.68). The flat surface can be moved upwards at a specified speed and angle. The following information is observed once every bounce:

- The x coordinate of the ball, x_{top}, when the ball reaches the top of the bounce.
- The y coordinate of the ball, y_{top}, when the ball reaches the top of the bounce.
- The horizontal speed of the ball, u_{top}, when the ball reaches the top of the bounce.

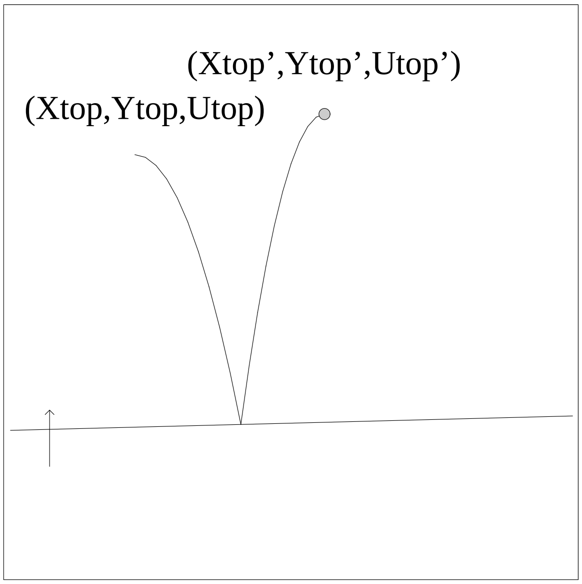


Figure 10.68

A single ball bounce. The surface was moving upwards, and was at a slight angle.

-1.5	$\leq$	x_{top}	$\leq$	1.5
0.5	$\leq$	y_{top}	$\leq$	2
-1	$\leq$	u_{top}	$\leq$	1
0	$\leq$	t_{go}	$\leq$	0.5
-0.5	$\leq$	v_{go}	$\leq$	0.5
-0.2	$\leq$	θ_{go}	$\leq$	0.2

Table 10.20: Ball bouncing: permissible state and action values.

The vertical speed is zero at the top of a bounce. The three values $(x_{\text{top}}, y_{\text{top}}, u_{\text{top}})$ constitute the perceived state of the system.

The ball is controlled by the surface. The surface is always reset to the same height before the ball reaches the top of the bounce. Then, a short period of time, t_{go}, after the ball reaches the top of its trajectory, the surface centre starts to move upwards with constant speed v_{go}. The bat is also placed at angle θ_{go} to the horizontal. Thus the three variables t_{go}, v_{go} and θ_{go} affect the behaviour of the surface and constitute the action used to control the ball. The permissible values of the state and action components are shown in Table 10.20. If any state variable exceeds its range the ball is deemed to have crashed. The following PSTF was learned:

$$\underbrace{(x_{\text{top}}, y_{\text{top}}, u_{\text{top}})}_{\textbf{State}} \times \underbrace{(t_{\text{go}}, v_{\text{go}}, \theta_{\text{go}})}_{\textbf{Action}} \rightarrow \underbrace{(x'_{\text{top}}, y'_{\text{top}}, u'_{\text{top}})}_{\textbf{Behaviour}} \tag{10.3}$$

where $(x'_{\text{top}}, y'_{\text{top}}, u'_{\text{top}})$ is the perceived state of the subsequent bounce.

The abstract controller was based on the notion of an ideal state: $x_{\text{top}} = 0, y_{\text{top}} = 1, u_{\text{top}} = 0$. On each control cycle the controller attempts to apply an action which at least halves the current

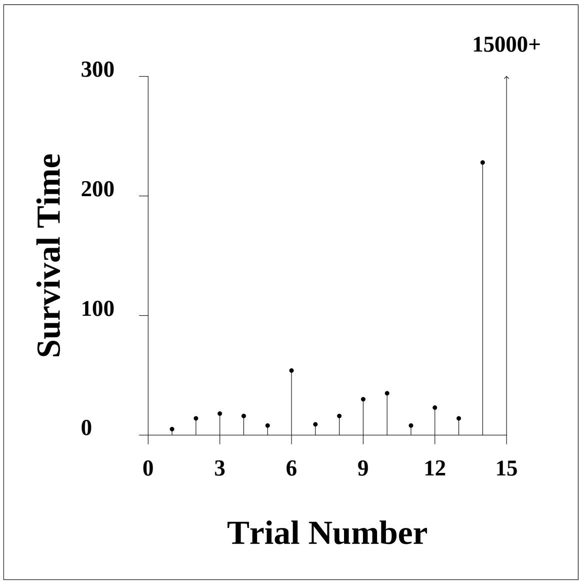


Figure 10.69

The number of bounces before disaster against trial number. Trial 15 was halted, still bouncing at 15000.

error of each state variable. The error of a state variable is its absolute difference from its ideal value. For example, if the current state were $x_{\mathbf{top}} = 1, y_{\mathbf{top}} = 0.8, u_{\mathbf{top}} = -0.2$ then the SAB action chooser would be requested to find an action to produce a next state for which

$$\begin{array}{rcccl} -0.5 & \leq & x_{\mathbf{top}} & \leq & 0.5 \\ 0.9 & \leq & y_{\mathbf{top}} & \leq & 1.1 \\ -0.1 & \leq & u_{\mathbf{top}} & \leq & 0.1 \end{array} \tag{10.4}$$

This abstract controller is more robust than one which aims precisely for the ideal state—if the ball is in danger of crashing it is of primary importance to prevent its state getting worse and only of secondary importance to try to send it immediately to the ideal state. It should be noted that this abstract controller is not trivial in design, and it required some of the human designer's qualitative domain knowledge.

10.5.1 Results

The graph of Figure 10.69 shows the number of bounces against trial number. The simulated world has 2% noise in each variable and all the SAB system parameters were default.

The graph shows that for the first 13 trials the number of bounces before disaster was usually between three and thirty. Trial 14 was successful for over 200 bounces, but there was an eventual serious mistake. Trial 15 was halted after four hours of simulation had produced over 15,000 successful bounces. Figure 10.70 shows the behaviour of the $x_{\mathbf{top}}$ and $y_{\mathbf{top}}$ components of state. They varied greatly until trial 15, when they were kept safely within 5% of the centre of their ranges.

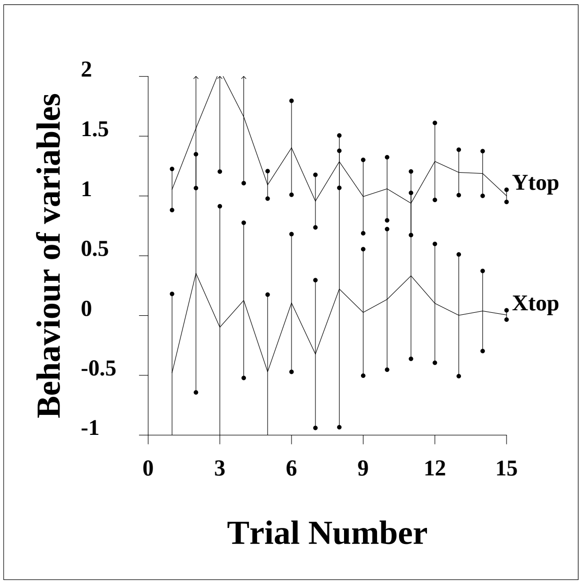


Figure 10.70

The mean and standard deviation of the x_{top} and y_{top} coordinates experienced during the juggle trial.

Further experiments were performed with 5% noise in the perceived state observations. Figures 10.71 and 10.72 show performance respectively without and with the *SAB* smoothing mechanism. In the second case the range width D_{range} was 5% of the width of the *SAB* tree domain. The results provide evidence that smoothing the observations improves performance. It should be noted that despite the inferior results of the unsmoothed experiment, its juggling behaviour was still good, with several trials of more than 1000 bounces. The smoothed experiment was run three other times, on each occasion halted after 10,000 bounces. The successful trial numbers were respectively 16, 20 and 20. A smoothing kernel of twice the width (10% of the domain width) was also tried. It was halted after 5,000 bounces on trial 21. The large smoothing width caused an approximate five-fold decrease in computation speed.

Graphs 10.73 and 10.74 show the performance in a changing environment. The ball's coefficient of elasticity increases during trials (it starts at 0.8 and increases to 1 asymptotically). In Figure 10.73 the initial change is sufficiently quick that by trial 70 the *c.o.e.* is very close to 1, and so no further apparent change occurs. Only after changes have become negligible is there a trial that avoided disaster, indicating that the adaptation mechanism was not able to help *during* the environmental change. In Figure 10.74 the changes are significant until approximately trial 180. There were no disaster-free trials before the experiment was halted on trial 220.

10.6 The Volley Task

The volley task was described in Section 9.5. It is an example of a compound task which has several learning components and is considerably larger and more complex than typical learning control

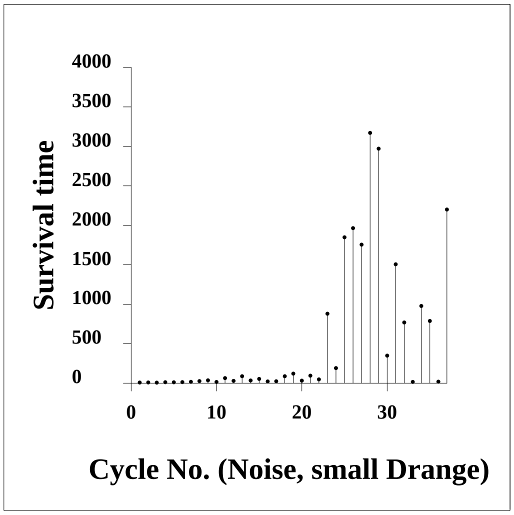


Figure 10.71: The number of bounces before disaster against trial number in a noisy environment: noise 5% and no smoothing.

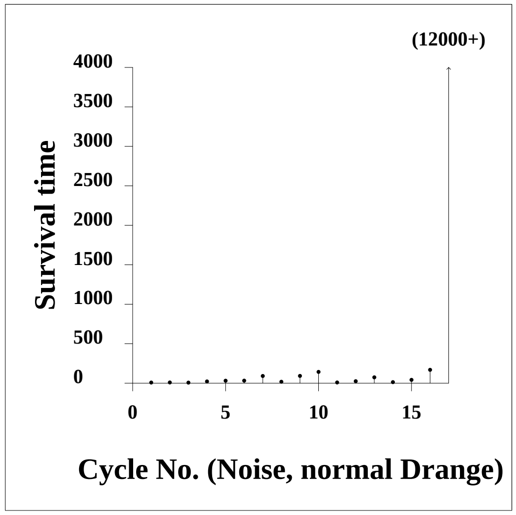


Figure 10.72: The number of bounces before disaster against trial number in a noisy environment: noise 5% and using a smoothing kernel of width 5%.

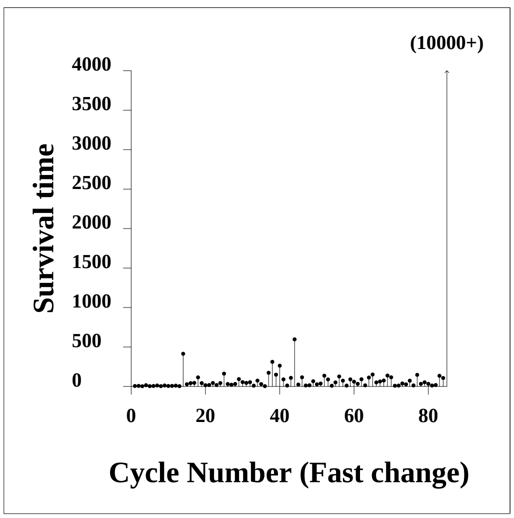


Figure 10.73: The number of bounces before disaster against trial number in a changing environment. Changes have become negligible by trial 70.

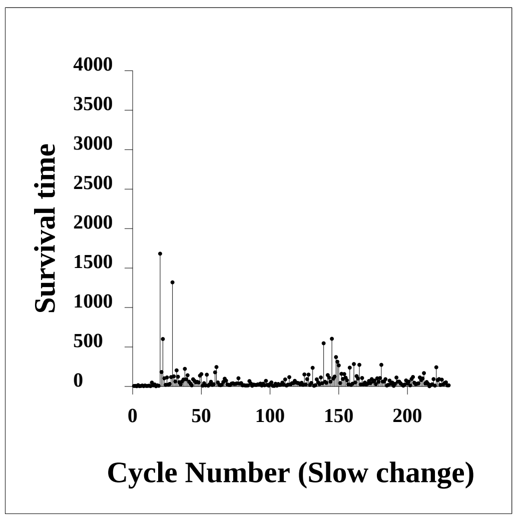


Figure 10.74: The number of bounces before disaster against trial number in a slowly changing environment. Changes are negligible by trial 180.

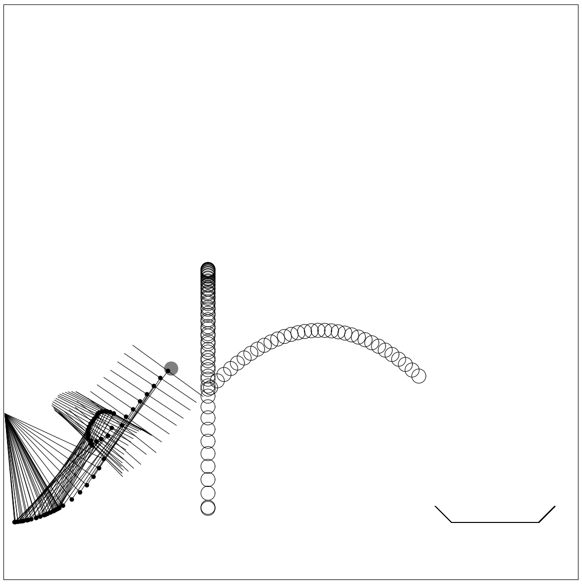


Figure 10.75

An early volley attempt. The ball is fired from the right towards the arm. From the initial state of the ball the controller erroneously predicts it will arrive at the grey circle. The ball is clipped by the bat and flies up vertically before falling to earth.

tasks described in the literature. However, it *is* simulated. There is significant simulated noise to capture the same sort of disorder that a real experiment would have. The task is decomposed into a hierarchy of learning controllers which involve three *SAB*-trees which learn and control the following:

- The perceived state transition function of the arm.
- The dynamics of the bat/ball collision.
- The behaviour of the ball (used for prediction only).

The new *SAB*-trees have default parameters except for the scaling of the components. The components were not difficult to scale—they are all either speeds and positions in the simulated world, and so used the same scaling as that used for the arm.

The simulated ball has noise in its dynamics: enough to vary the arrival position by up to approximately 3% of the length of the arm.

Figures 10.75 and 10.76 show the behaviour of the arm during an early and late trial respectively.

Figure 10.77 graphs the performance of a relatively easy task, where the ball is always fired from the same position and velocity and the bucket always remains in the same place. After five trials a suitable hitting speed is discovered, and a value close to this speed is used for subsequent trials.

Figure 10.78 displays the results of the considerably harder task in which the ball is fired at a random speed for each trial, and the bucket is placed in a random position. It requires approximately twenty trials before the behaviour can be said to be fairly skilled.

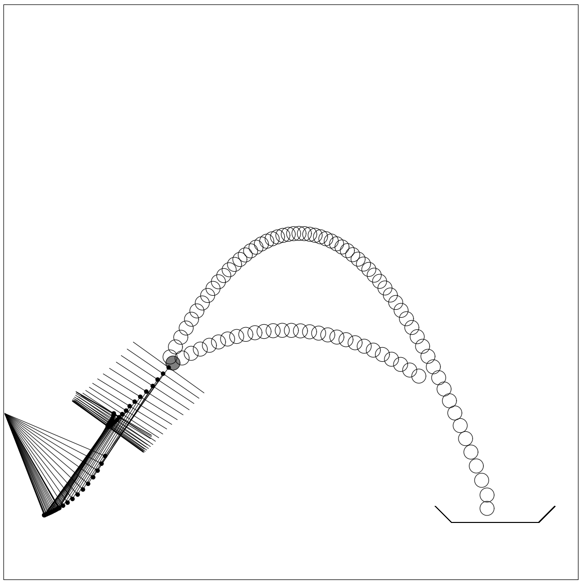


Figure 10.76

A successful volley. As well as modelling its own arm, the controller has correctly predicted where the ball will be when it is in range and found a correct speed with which to hit the ball back to the bucket.

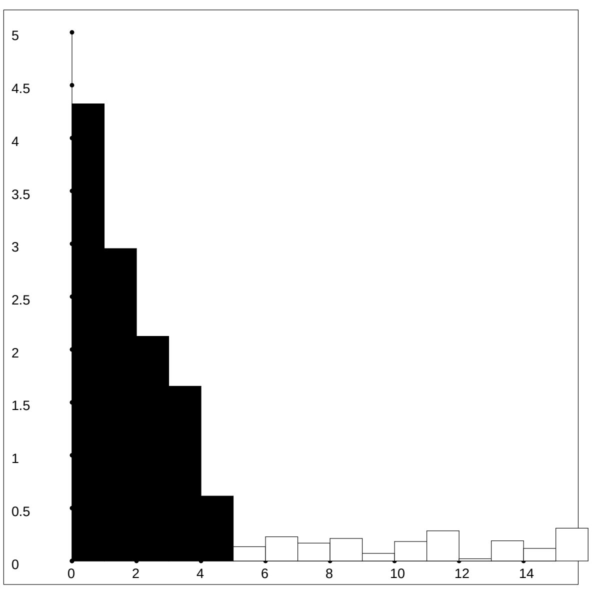


Figure 10.77

A histogram of distance from bucket against trial number. The successful volleys, which landed in the bucket, are shown in white. During these trials the bucket was fixed and the ball always fired with the same speed and direction.

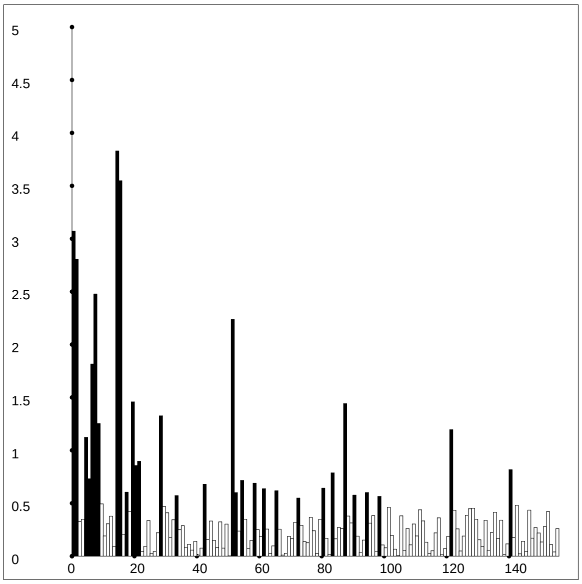


Figure 10.78

A histogram of distance from bucket against trial number. Before each trial the bucket was placed at a random position. The ball was always fired with random speed but constant direction.

Figure 10.79 shows the behaviour when the bucket is placed randomly and the ball is fired with a random speed *and* direction. There is an improvement in behaviour but the probability of failure is still roughly 20% even after over 100 trials. This is because even then there is still a fair probability that the starting state of the ball is sufficiently far from any previous experience that the behaviour predicted by the nearest neighbour is inadequate.

Thus learning behaviour is demonstrated for a complex robotic system. The learning was efficient—approximately 30 seconds per volley attempt initially, rising to 3 minutes per attempt when the SAB-trees were large enough to cause substantial virtual memory swapping on the computer. The final size of the PSTF SAB-tree was approximately 50,000 exemplars.

10.7 Experimental Results: Conclusions

The experimental results have generally confirmed the goals of this investigation. The learning rate was very fast—only a small number of trials of any experiment were required. Indeed, the testing was (by the standards of much previous research for which learning speed was not top priority) somewhat severe in that an experiment was usually classed as unsuccessul if learning had not occurred within a few dozen trials. It was also demonstrated that the rate of learning is not badly affected by increased accuracy requirements. In the "hold still" task of Section 10.2 the time to achieve any given tolerance was seen to increase only logarithmically with the reciprocal of the tolerance.

The learning was also computationally efficient, even with kd-trees of up to 10,000 exemplars. This was particularly noticeable for the hand-eye experiment in which the real time to learn, in-

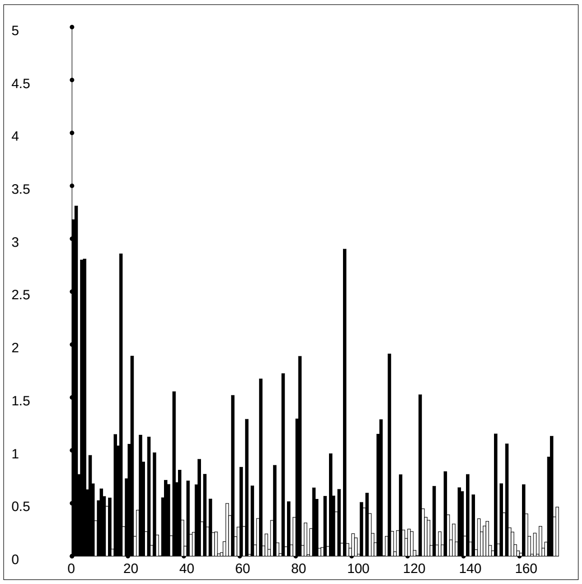


Figure 10.79

A histogram of distance from bucket against trial number. Before each trial the bucket was placed at a random position. The ball was always fired with random speed and random direction.

cluding waiting for arm movements and image processing was only in the order of thirty minutes. Learning did become noticeably slower when *k*d-trees were several tens of thousands of exemplars large. This is conjectured to be because of the extravagant use of memory in the current implementation of the *SAB*-trees, which requires 256 bytes per exemplar, causing a 20,000 node tree to use 4 mageabytes of store which causes substantial virtual memory swapping. Use of very large range search widths also slowed computation. This is because *SAB*-tree update then involved accessing a significant fraction of all exemplars.

The learning was robust. It was demonstrated that all system parameters could be varied considerably without significant detrimental effect on performance. Futhermore, no experiment became "stuck". The smoothing mechanism was demonstrated to help combat noise in the arm experiments and the juggle experiments.

The change adaptation mechanism was also shown to have some beneficial effect for the trajectory tracking and juggling tasks, but for the high dimensional arm task the results were disappointing. A worthwhile area for further study would be to improve the pruning of old inaccurate exemplars by more sophisticated (and probably off-line) statistical analysis.

Finally, a significantly complex task—volleying—was controlled successfully by a hierarchy of *SAB* learners. Learning took place simultaneously at different layers of abstraction, demonstrating how learning world models can be of considerable help even for high level robotic tasks.

Chapter 11

Some Extensions

This chapter reviews some extra topics which were researched during the course of the dissertation, but which are not central to its main ideas. In Section 11.1 Albus' CMAC is discussed in further detail, in particular its relationship to exemplar-based methods. In Section 11.2 the use of Dynamic Programming in conjunction with SAB-learning is examined and extended.

11.1 Albus' CMAC and kd-trees

The Cerebellar Model Articulation Controller (CMAC) is a connectionist model, devised by J. S. Albus in the early 1970s, for learning smooth, continuous, non-linear mappings [Albus, 1975a; Albus, 1975b; Albus, 1981]. It is specifically useful where a high degree of global generalization is positively not wanted, but local generalization to take advantage of the assumed smoothness is welcome. In the next section I will summarize its implementation and behaviour. Although the implementation is connectionist, I then propose an alternative kd-tree implementation with exactly the same behaviour, called "Symbolic CMAC". I conclude by comparing the behaviour of CMAC with that of the nearest neighbour generalization.

11.1.1 CMAC Summary

CMAC implements a mapping from a quantized k_s-dimensional space $\mathbf{S}$ to a real-valued k_p-dimensional space $\mathbf{P}$. When an input vector is presented either for access or update it is pre-processed to obtain a number of indices into a large array of weights. These indices are chosen in a manner (detailed in [Albus, 1975a]) which causes vectors close to each other in $\mathbf{S}$-space to have a large proportion of shared indices. The notion of closeness is defined by a metric $dist : \mathbf{S} \times \mathbf{S} \rightarrow \Re$.

The closer two vectors are, the more indices they share. Conversely, beyond a certain distance they share no common indices. Let us call the set of indices associated with an input vector X the *neighbourhood* of X. The indices address array entries which contain k_p-dimensional weights.

When CMAC is being accessed these weights are all added to produce the predicted vector from $\mathbf{P}$-space. Write $W_N(i)$ as the weight corresponding to the index i at learning iteration N. Let *nhood*(X) denote the set of indices associated with input vector X. The value predicted for input X on the Nth iteration is

$$I_N(X) = \sum_{i \in nhood(X)} W_N(i). \tag{11.1}$$

When CMAC is updated the update increment is distributed evenly among all the indices in the neighbourhood. If on the Nth learning iteration we wish to increment the predicted value at X_N by the value δ_N, the weights are adjusted thus:

$$W_{N+1}(i) = \begin{cases} W_N(i) + \frac{1}{C}\delta_N & \text{if } i \in nhood(X_N) \\ W_N(i) & \text{otherwise.} \end{cases} \tag{11.2}$$

Where $C = \mid nhood(X_N) \mid$. There seems no reason to vary the number of indices in different neighbourhoods, and so generally C is a system constant independent of the neighbourhood. Conventionally the initial value of all weights is zero so $I_N(X) = 0$ for all X.

When the learning data is presented as a set of (X_N, P_N) pairs, the increment on the Nth iteration is obtained by first finding the current predicted value $I_N(X_N)$, then evaluating the error $P_N - I_N(X_N)$ and incrementing CMAC at X_N by this error. An alternative increment, used by [Miller *et al.*, 1987; Miller, 1989], is the error multiplied by a training factor β.

11.1.2 CMAC Discussion

The weight array is multi-dimensional, and so will be very large if k_s is larger than 3 or 4 and the components of the input vectors are quantized to more than 10 or so levels. This problem can be solved for CMAC by hashing the weight indices down to a much smaller array. Thus each cell in the smaller array will correspond to a very large number of cells in the original weight array. The hash collisions have been observed empirically not to cause great inaccuracy. This can be explained by distribution of increments over a neighbourhood: if a proportion of the distributed weights are corrupted, their expected summed error is small compared with the summed values of the valid weights.

CMAC has been used in recent work [Miller *et al.*, 1987] and has performed fairly accurately. This work needed only to learn a one-dimensional "strand" of the inverse dynamics of a robotic manipulator along a repetitive joint space trajectory. The concentration of data around this strand helped minimize hash-collision problems, providing good performance even with surprisingly small hashed-arrays of only 2000 elements.

A further advantage of CMAC is that it can learn when it is not provided with explicit

$$(\text{input}, \text{output}) \tag{11.3}$$

pairs, but simply rough

$$(\text{input}, \text{direction of prediction error}) \tag{11.4}$$

observations. This can be used to learn mappings such as evaluation functions in which an explicit evaluation of a state can rarely be obtained even after observation, but an indication of whether the current value is too optimistic or pessimistic *is* generally available.

CMAC's problems include the need to quantize, the need to keep neighbourhoods small (for efficient update and access), the inaccuracies caused by hash collisions and the difficulty of a formal proof of convergence.

11.1.3 Symbolic CMAC

An explicit geometric representation of the data passed to CMAC can have exactly the same prediction as CMAC itself. I call this new representation "Symbolic CMAC".

The representation is the explicit set of all the (input vector , increment) pairs we have experienced:

$$\mathbf{E} = \{(X_0, \delta_0), (X_1, \delta_1) \ldots (X_{N-1}, \delta_{N-1})\}. \tag{11.5}$$

To predict, the following calculation is used. $J_N(X)$ denotes the prediction at vector X after N observations.

$$J_N(X) = \sum_{(X_i, \delta_i) \in \mathbf{E}} \delta_i \frac{\mid nhood(X_i) \cap nhood(X) \mid}{\mid nhood(X_i) \mid} \tag{11.6}$$

This prediction can be computed naively by initializing the result as 0 and then for i between 0 and $N-1$:

1. Enumerate all the indices in $nhood(X_i)$.
2. Enumerate all the indices in $nhood(X)$.
3. Count the number of duplicate indices.
4. Add to the result: $\delta_i \dfrac{\text{number of duplicates}}{\text{number in } nhood(X_i)}$.

The proof that $J_N = I_N$ is by induction on N.

This rather inefficient implementation can be improved by working out the size of the intersections of neighbourhoods indirectly instead of in this explicit manner. In [Albus, 1975a] the neighbourhoods are designed to

1. Have the property that any two close input vectors will have a large intersection of neighbourhoods, and further vectors will share correspondingly less.
2. Contain a sufficiently small number of indices for acceptable performance.

CMAC's closeness measure is an approximation to Hamming distance. To facilitate computation the symbolic method will use instead the L_∞ metric. That is

$$dist(X,Y) = \max_i \mid X_i - Y_i \mid . \tag{11.7}$$

We can now define the neighbourhood of X as those indices within a fixed distance d of X (this is exactly what the CMAC method of choosing indices does in the one-dimensional case, and what it approximates in higher dimensions). The size of a neighbourhood is $(2d)^{k_s}$.

$$nhood(X) = \{Z \,:\, dist(X,Z) < d\} \tag{11.8}$$

The size of the intersection of two neighbourhoods of X and Y can now be computed. If $dist(X,Y) \geq 2d$ it is zero. Otherwise it is the volume of a k_s-dimensional cuboid. The length of the ith dimension of this cuboid is the distance of overlap in the ith dimension which can be shown to be $2d - \mid X_i - Y_i \mid$. The volume of the cuboid is thus

$$\prod_{j=0}^{k_s-1} (2d - \mid X_j - Y_j \mid) \tag{11.9}$$

Thus, for this metric, the prediction is

$$J_N(X) = \sum_{X_i \,:\, dist(X,X_i) < 2d} \delta_i \; scale(X, X_i) \tag{11.10}$$

where

$$scale(X,Y) = \prod_{j=0}^{k_s-1} (1 - \frac{\mid X_j - Y_j \mid}{2d}). \tag{11.11}$$

Only those vectors lying within distance $2d$ from the input vector are inspected. If the vectors are stored in a balanced kd-tree, the expected cost of obtaining these vectors is $O(k_s(\log N + S))$ where N is the total number of vectors, S is the number of vectors which *do* lie within range and k_s is the dimensionality of **S** space.

The L_∞ metric was used simply because the neighbourhood intersection computation is easy. The L_2 (Euclidian) or L_1 (Hamming) metrics might have been preferred. The L_2 computation is the intersection of two hyperspheres, which is straightforward (and because the intersection size is simply a function of the distance between the two points it can be precomputed in a one-dimensional look up table). The L_1 calculation should also not be computationally expensive, but I have not attempted the unpleasant geometry to calculate the intersection.

11.1.4 Symbolic CMAC: Discussion

In this section I will mention some of the advantages of symbolic CMAC, but the main point of interest is not to suggest an alternative implementation, but the link between the apparently unrelated connectionist and geometric representations.

The **advantages** are:

- It is unnecessary to choose quantization levels for the components of the input vectors. This is an advantage (i) in terms of accuracy and (ii) because it is not necessary to decide in advance which ranges of the vectors need highest resolution.

- The neighbourhoods can be large without a dramatic increase in computational cost: for example they can consist of all points which have all components within 30% of the full range of input vector values. For the original CMAC this would mean an exponential increase in the number of indices in the neighbourhood for each new dimension. To avoid this, CMAC must use a fixed neighbourhood which only approximates the generalization that would be obtained from the Hamming distance metric.

- There are no inaccuracies induced by hashing collisions. This will be most important when the dimensionality is sufficiently high that each cell of CMAC's hash array would correspond to many weights *and* an adequate function approximation is desired over more than a small strand of the space. Without hash collisions the predicted function looks smooth, which may be of use if a gradient estimate is desired.

The **disadvantages** are:

- The memory size is not fixed. It is argued in Section 5.5 that this is acceptable because there will not be time for the memory to get large beyond current physical storage. However, the fixed memory requirement of CMAC, even if large, is more appealing.

- The time for access might be worse. The time to store a new observation in the symbolic CMAC is trivial, but accessing involves a range search. The cost of this search depends critically on the distribution of the data and the size of the neighbourhood. It is certainly bounded above by N, the number of observations, but in general can be expected to be very much cheaper. The author has not undertaken a detailed empirical comparison. The cost of Symbolic CMAC is expected to be similar to that of SAB learning, which in Section 10 is demonstrated as adequate.

- The original CMAC was partially motivated as an attempt to model the behaviour of the human cerebellum. It is clear that Symbolic CMAC is a large step away from the biological model, despite having the same behaviour.

11.1.5 Comparing CMAC with SAB Learning

The intended use of CMAC is the same as that of the nearest neighbour generalization: non-parametric prediction from data. It has been effectively demonstrated, particularly by [Miller *et al.*, 1987]. It is shown here to be equivalent to a geometric representation, with an prediction obtained from basis functions. The basis function is $scale(X, Y)$ used in Equation 11.10, and now it becomes apparent that the choice of local weighting as being the volume of the intersection of two neighbourhoods is only one of a range of possibilities.

The disadvantages over the nearest neighbour are that its behaviour is very sensitive to the size and shape of the neighbourhood and that unlike the nearest neighbour there is no reasonable prediction far from any experience (CMAC will predict 0; nearest neighbour will provide at least a rough estimate). In a high dimensional space in which the data is necessarily sparse this is a serious problem except in familiar subregions. Nearest neighbour also provides an estimate of how reliable its prediction is, as the distance of the nearest neighbour.

Both CMAC and nearest neighbour have the problem of determining the scaling for the variables. The solutions proposed for nearest neighbour based on statistical analysis of the data could be applied equally well to CMAC.

CMAC and Symbolic CMAC are both resistant to a noisy and slowly changing environment. Naive nearest neighbour performs badly with such data, but the modifications used for SAB learning solve these problems. The most important advantage of CMAC and Symbolic CMAC is that they can still learn when trained not on input–output pairs but values indicating the direction of the error of the current prediction.

11.2 Reinforcement Learning using Dynamic Programming

Here we are concerned with a dynamic system which has a continuum of actions and a continuous two-dimensional state space. The task is of the class which needs a non-local control strategy, and so learning the world model is not the only problem.

The main thesis of this work has been that with abstractions provided by world models, other aspects of the controller may be preprogrammed with ease, and need not be learned. In this section, however, I discuss a simple investigation which considers one way of further increasing the controller's autonomy. The method is based on dynamic programming (DP) which is, for example, described in [Burghes and Graham, 1980]. Dynamic programming has recently been popularized as a method to be used in conjunction with reinforcement learning by [Sutton, 1990]. Figure 11.1 shows a simple dynamic system.

A puck is sliding on a bumpy surface. It can thrust left or right with a maximum thrust of one Newton in either direction. However, because of gravity, the actual horizontal acceleration varies. In fact, in the puck position shown, the maximum right thrust is not strong enough to accelerate up the slope. This is made more clear in Figure 11.2. It is a *state space diagram.* The puck's state has two components (its perceived position and velocity) and thus can be represented by a two-component vector. In the figure the horizontal axis corresponds to puck position and the vertical axis corresponds to puck speed. To explain further, the west of the diagram are states in which the puck is to the left, east denotes "puck on right", north denotes "puck moving rightwards" and south denotes "puck moving leftwards". The cross-hairs show the goal state, which is stationary,

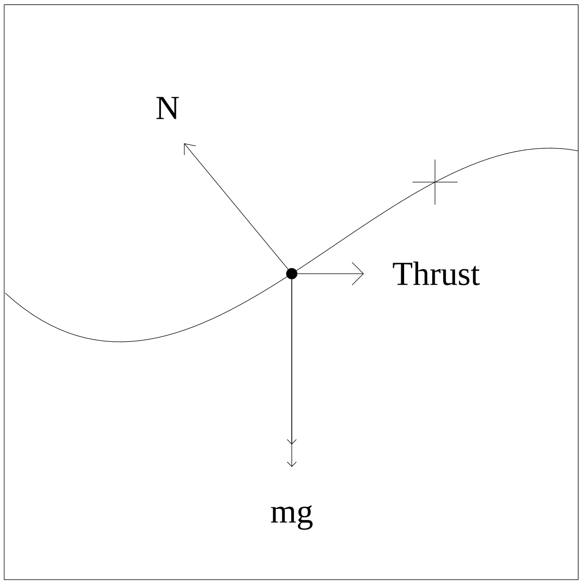


Figure 11.1

A frictionless puck acted on by gravity and a thruster. The target position is shown by the cross hairs.

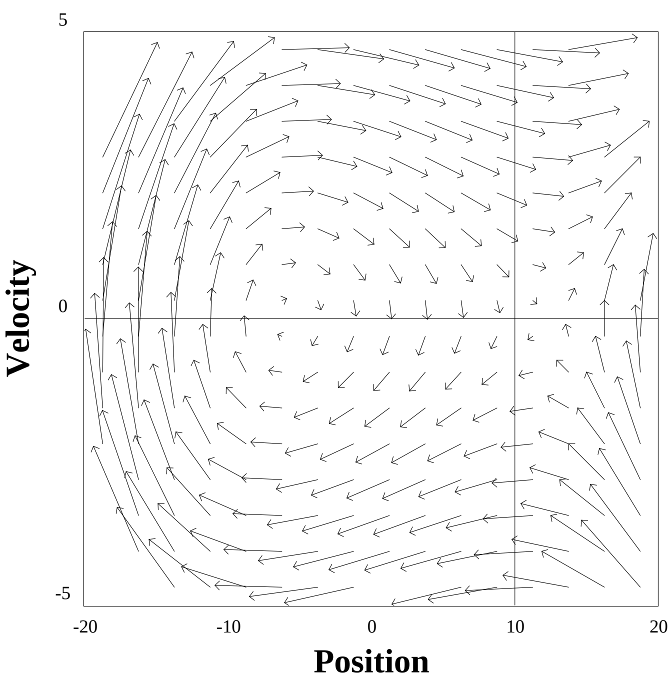


Figure 11.2

The state transition function for a puck which constantly thrusts right.

three quarters of the way up the hill (that is, state $x = 10\text{m}, v = 0\text{ms}^{-1}$, where x denotes position and v denotes velocity).

The arrows show the next state of the puck if it were to thrust rightwards with the maximum legal force of one Newton. Notice that at the centre of state space then even when giving this maximum thrust the steep slope causes the puck to gain negative velocity and slide left. Does this mean that if the puck is stationary at $x = 0$ that it is impossible to get to the goal state?

If the world model is known then the answer can be obtained by searching breadth first through

Algorithm:	Dynamic Programming
Input:	Finite set of states $\mathbf{S} = \{s_1, \ldots, s_n\}$ Finite set of actions $\mathbf{A} = \{a_1, \ldots, a_n\}$ State Transition Function $f : \mathbf{S} \times \mathbf{A} \rightarrow \mathbf{S}$ Set of goal states $\mathbf{U} \subseteq \mathbf{S}$
Output:	A set of triples $\mathbf{O} = \{(s_i, a_i, E(s_i))\}$ where $\mathbf{O} \subseteq \mathbf{S} \times \mathbf{A} \times \textbf{Integers}$
Pre:	$\mathbf{U} \neq \phi$
Post:	The output triples have the following property. $(s_i, a_i, E(s_i)) \in \mathbf{O}$ if and only if $E(s_i)$ is the minimum number of time steps needed to get to any goal state starting from s_i and a_i is an optimal action which can be taken in state s_i to achieve this.
Code:	
1.	$\mathbf{n} := 0$
2.	$\mathbf{O}_0 := \{(s_i, -, 0) : f(s_i, a_i) \in \mathbf{U}$
3.	while $\mathbf{n} = 0$ or $\mathbf{O}_n \neq \mathbf{O}_{n-1}$
3.1	$\quad \mathbf{n} := \mathbf{n} + 1$
3.2	$\quad \mathbf{O}_n := \mathbf{O}_{n-1} \cup$ $\quad \{(s_i, a_i, \mathbf{n}) : (f(s_i, a_i), a_j, E_j) \in \mathbf{O}_{n-1}$ for some $a_j, E_j.\}$
4.	$\mathbf{O} := \mathbf{O_n}$

Table 11.1: The Dynamic Programming Algorithm

all sequences of actions applied starting at the initial state. The more computationally tractable procedure of Dynamic Programming can be used to provide the same information. The dynamic programming algorithm is shown in Table 11.1.

To perform DP with a continuum of state and action variables it is necessary to partition both the state space and the action space into a finite number of possibilities. In this puck example the partitioning is achieved by quantizing the actions to 15 different levels and by quantizing each state variable to 64 levels, producing 4096 discrete states.

Dynamic programming is then performed, using, as the state transition function the analytic world simulation model. The results are portrayed graphically in Figure 11.3. They indicate that it *is* possible to reach the goal state by moving left initially and then accelerating to the right to gain sufficient speed to reach far enough up the hill before gravity brings the puck to a halt.

We are interested in what can be done if there is no predefined analytic world model to consult. Consider the case where the only experience of the world had been gained from the two trajectories shown in Figure 11.4, which consist of a total of 33 experiences (this was obtained by running two puck trials in which, on each control cycle, the puck chose a random thrust in the range -1 to 1

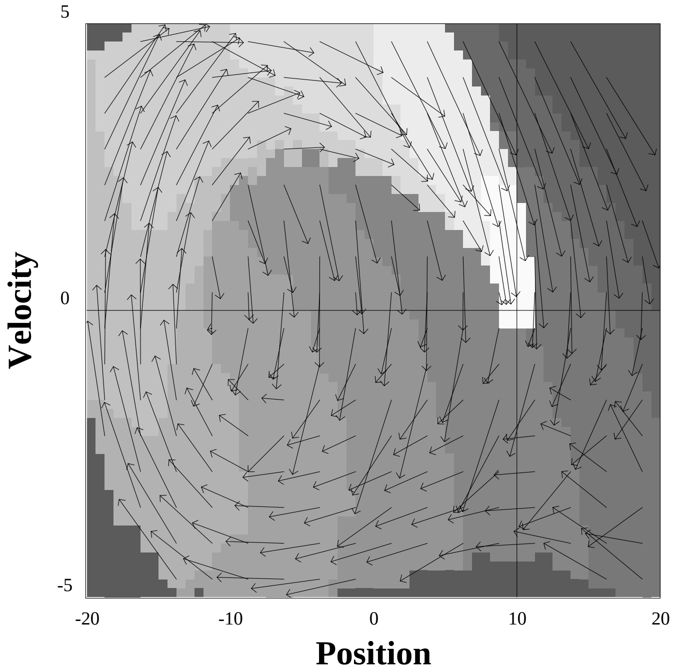


Figure 11.3

The brightness level in this diagram denotes the evaluation that DP gives the states. The arrows show the next states if you apply the actions recommended by DP. Notice that the evaluation function has a major discontinuity.

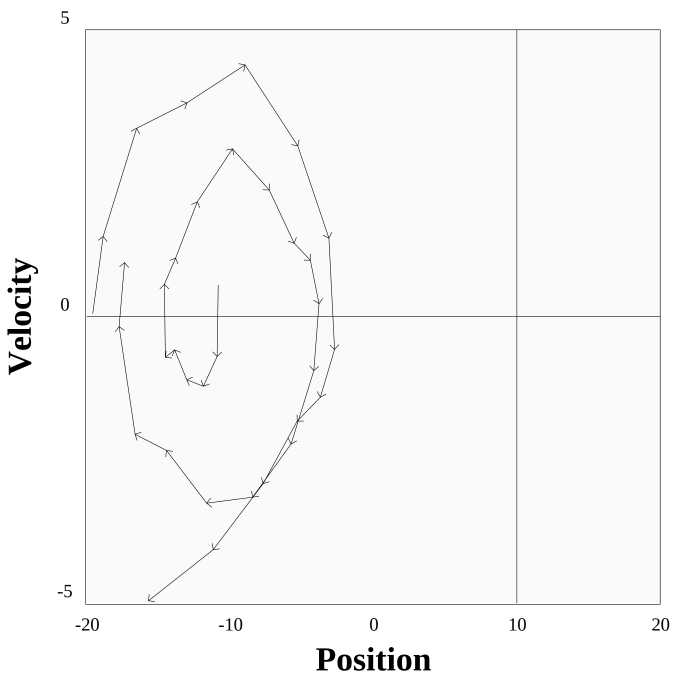


Figure 11.4

Some random experiences in the Puck's state space.

Newton).

A SAB world model model was learned from these experiences:

$$\underbrace{\text{Puck Position and Speed}}_{\textbf{State}} \times \underbrace{\text{Puck Thrust}}_{\textbf{Action}} \rightarrow \underbrace{\text{Puck Acceleration}}_{\textbf{Behaviour}} \tag{11.12}$$

Dynamic Programming was then performed using only this learned world model. The results are shown in Figure 11.5. The strategy obtained looks encouragingly similar to the optimal strategy obtained by DP with the analytic model, although there are errors, particularly in the right hand half of the state space (for example the bottom right hand corner). In fact there is also a region of critical error among the states leading up to the goal. Figure 11.6 shows the results of running using actions recommended by this DP's action map. Unfortunately the misplaced discontinuity leads to a missing of the target. This is not very disheartening because it was based on absolutely no experience near the goal state. Having had this failed attempt more information near the goal has been obtained. When this extra information is added to the SAB-tree, and DP is applied again with the updated world model, a superior controller is obtained (shown in Figure 11.7, though the difference is unfortunately hard to detect visually) which is now sufficient to achieve the goal on the next attempt (see Figure 11.8).

It has thus been shown that the world models obtained from SAB-learning can be used instead of an analytic model to autonomously compute non-local control. However, DP works by quantizing and enumerating the state and action spaces which is exactly what the SAB world model learner has been designed to avoid. Thus it seems a shame that the world model learner can survive high dimensions but the controller which uses it cannot. Even with the fairly trivial Puck example, DP is rather computationally expensive (five minutes processing on a UNIX workstation using the SAB-learned world model).

11.2.1 Variable Resolution Dynamic Programming

Instead of enumerating the state and action spaces, let us follow the same approach that was used for SAB-learning. Here, an experimental method is described which performs computation only within those areas of state and action space which have actually been experienced.

First, let us consider in depth the assumptions that DP makes in order to estimate the minimum time to goal. We shall denote this estimation by E_{ttg}.

1. $E_{\text{ttg}}(s_1) \leq E_{\text{ttg}}(s_2) + 1$ if the world model predicts that for some action it is possible to get from state s_1 to state s_2 in one time step.

2. $E_{\text{ttg}}(\text{Goal State}) = 0$

3. The estimated time to goal (E_{ttg}) for all states in the same quantization cell is the same.

The first and second assumptions are logically valid, and the third assumption is necessary in order to supply DP with a finite number of states. This is not necessarily accurate, but accuracy can be gained by increasing the number of quantization levels, and hence the number of distinguishable states. It is based on the assumption that E_{ttg} is smooth and will thus not vary significantly within each quantization cell.

The smoothness assumption is not necessarily true (for example, there is a discontinuity in Figure 11.3), but away from discontinuities provides an excellent estimate of the optimal action.

To perform the equivalent of DP without needing to look at states other than those that we have experienced, let us make an alternative smoothness assumption:

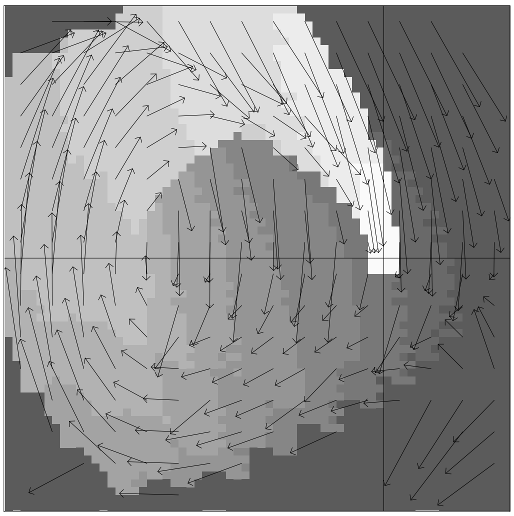


Figure 11.5: Dynamic programming using a world model obtained from *SAB* learning.

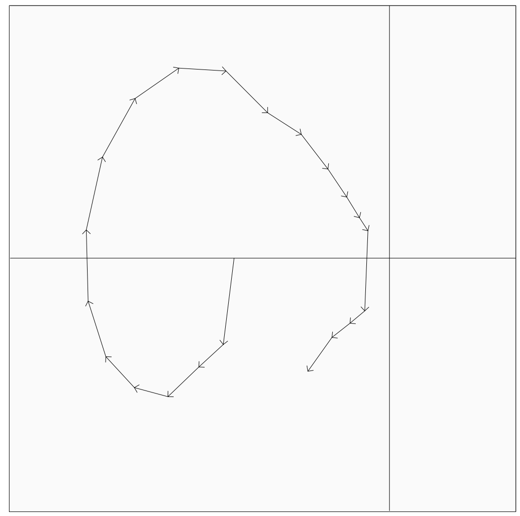


Figure 11.6: Trajectory produced by using the results of Figure 11.5.

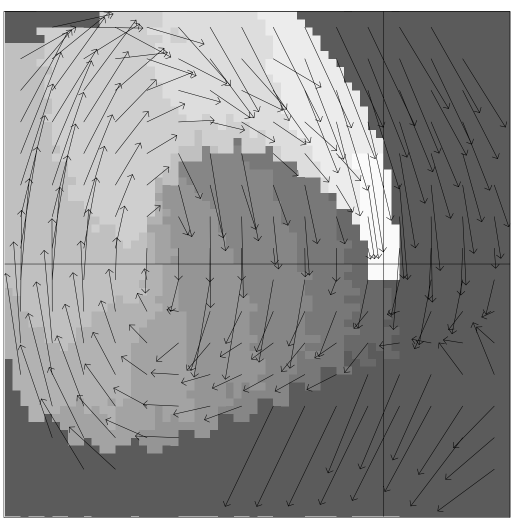


Figure 11.7: Dynamic programming using a learned world model updated with the experience of Figure 11.6.

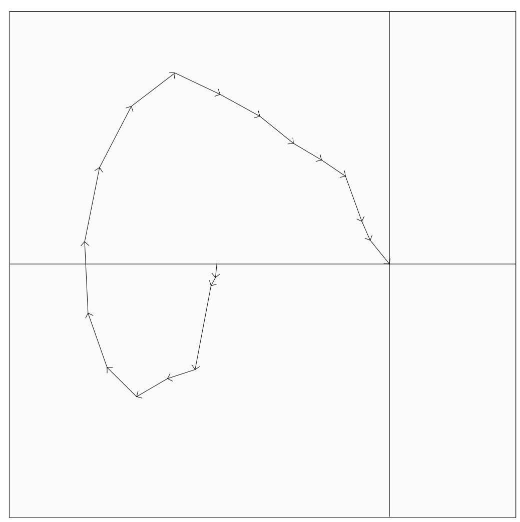


Figure 11.8: Trajectory produced by using the results of Figure 11.7.

3a. E_{ttg} has maximum slope A. For any two states s_1 and s_2

$$\mid E_{\text{ttg}}(s_1) - E_{\text{ttg}}(s_2) \mid \leq A \mid s_1 - s_2 \mid \tag{11.13}$$

The new DP algorithm evaluates states by finding the most pessimistic E_{ttg} which agrees with the world observations and which fits the constraints 1, 2, and 3a above.

Why is the most pessimistic value within the constraints used? The advantage is that it is the most clearly defined interpretation. A moment's reflection shows that the optimistic evaluation within constraints would be useless: it would simply be that $E_{\text{ttg}}(s) = 0$ for all states. There is also a computational advantage in being pessimistic—the E_{ttg} estimate can then be computed for each state that has been experienced with reference only to other states which have been experienced.

The only states which have their E_{ttg}'s recorded are the goal state(s) and all states which have been experienced. The latter are recorded in a *SAB*-tree.

Initially, before any experience, the estimated time of the goal state (s_{goal}) is zero, and by (3a) we know that for any other state s,

$$\mid E_{\text{ttg}}(s) - E_{\text{ttg}}(s_{\text{goal}}) \mid \leq A \mid s - s_{\text{goal}} \mid \tag{11.14}$$

Thus, because we use the most pessimistic estimate, and putting $E_{\text{ttg}}(s_{\text{goal}}) = 0$, we derive

$$E_{\text{ttg}}(s) = A \mid s - s_{\text{goal}} \mid \tag{11.15}$$

This produces Figure 11.9. The apparent quantization is due to the primitive 2-d graph drawing software. There is no quantization for the actual algorithm. Here, A is chosen as 6.4 time steps per metre, which corresponds to gradient 500 in uniformly scaled *SAB*-tree units. The start state ($x = 0\text{m}, v = 0\text{ms}^{-1}$) has $E_{\text{ttg}} = 64$ time steps.

I shall now describe the operation of finding the most pessimistic E_{ttg} that satisfies constraints (1), (2) and (3a). We are given a known goal state s_{goal} and a set of observations

$$\{s_0 \rightarrow s_0{}', \ldots, s_n \rightarrow s_n{}'\} \tag{11.16}$$

The operation is achieved by two algorithms. The first, **Evaluate**, finds the value of any state, given the known values of a set of other states, by means of constraint (3a). The second algorithm computes the E_{ttg} for all experienced states $s_0, s_0{}', \ldots, s_n, s_n{}'$. It uses the **Evaluate** algorithm and constraints (1) and (2). The reader who is interested in the behaviour of these algorithms, but not their implementation, should skip to Section 11.2.4.

11.2.2 The "Evaluate" Algorithm

This takes, as input, the query state s and a set of known evaluations

$$\text{ES} = \{(s_0, E_0), \ldots, (s_k, E_k)\} \tag{11.17}$$

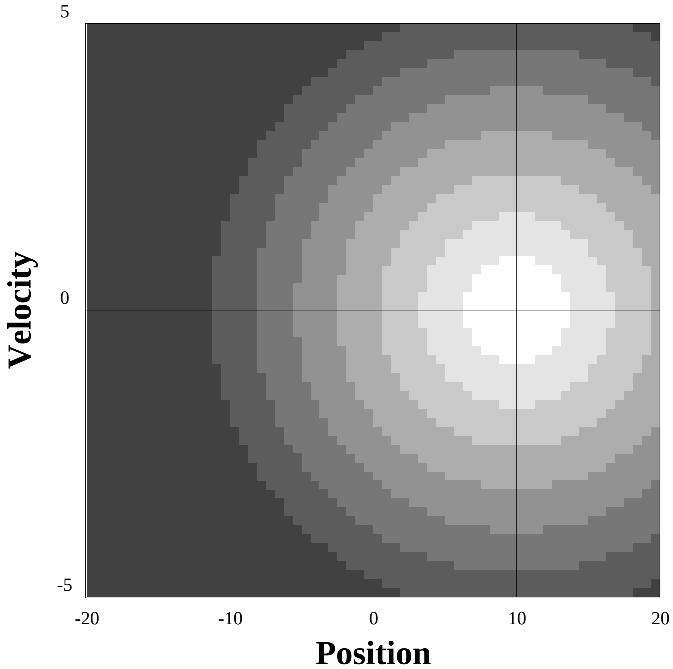


Figure 11.9

E_{ttg}—the pessimistic, slope bounded evaluation function after no experience. The only knowledge is that the target state is zero time steps to success.

Let us call such a set an *evaluation set*.

The simplest algorithm to estimate $E_{\mathrm{ttg}}(s)$ would look at each member of ES in turn. Constraint (3a) tells us that for each member $(s_i, E_i) \in$ ES, $E_{\mathrm{ttg}}(s) \leq E_i + A \mid s - s_i \mid$. In particular we know that $E_{\mathrm{ttg}}(s)$ is no greater than this expression for the value of i which minimizes $E_i + A \mid s - s_i \mid$. To be pessimistic, it must be assumed that $E_{\mathrm{ttg}}(s)$ equals this minimum value.

The algorithm of finding the minimum value by scanning the whole evaluation set would be valid, but slow. A superior algorithm has been developed which searches the evaluation set as a kd-tree in a similar manner to that of a nearest neighbour search. Search states are often reached at which the lowest value of $E_i + A \mid s - s_i \mid$ yet found is sufficiently low that other hyperrectangles in the tree need not be searched. This occurs when the distance d of the hyperrectangle to the query point is sufficiently great, that the lowest possible expression of any point in it (which is $0 + Ad$) is greater than the best yet found, and so search cutoff may occur. With a large value of A almost all the tree is cut away. The value of A in chosen as large anyway, in order to increase the chance the constraint (3a) is valid.

11.2.3 Evaluating all Experienced Points

Here I explain how to fill the set of all observed states with their correct E_{ttg} values, given a set of observed state transitions

$$\{s_0 \rightarrow s_0', \ldots, s_n \rightarrow s_n'\} \tag{11.18}$$

It starts with an estimate of the E_{ttg}'s of all states based solely on the distance to the goal state. It then iteratively reduces this estimate using (1) and (3a) while ensuring the estimate never becomes more optimistic than is necessary to satisfy the constraints.

An evaluation set called ES is constructed.

$$\text{ES} = \{(s_0, E_0), (s_0{}', E_0{}'), \ldots, (s_n, E_n), (s_n{}', E_n{}')\} \tag{11.19}$$

It is initialized by

$$E_i = A \mid s_{\text{goal}} - s_i \mid \qquad E_i{}' = A \mid s - s_i{}' \mid \qquad \text{for every } i. \tag{11.20}$$

Then the following two steps are performed

1. For each i, $E_i := \min(\, E_i\ ,\ 1 + E_i{}'\ )$. This is necessary to ensure constraint (1) is upheld. A state cannot have an evaluation more than one greater than a known successor state.

2. All the E_is and $E_i{}'$s are all recalculated using the **Evaluate** algorithm applied to the evaluation set ES. This is necessary because the previous operation might have made some previously poor states into good states. Other states which are close to these improved states should themselves be given superior evaluations to maintain constraint (3a).

The two operations above are repeated in turn until there are no further changes to ES. Although not proved here, it is believed that, as is the case with the standard Dynamic Programming algorithm, this termination only occurs when all constraints are satisfied, and that the resulting evaluation is the most pessimistic.

11.2.4 Algorithm Behaviour

To illustrate this procedure, let us consider the results of incorporating a small number of transitions. The trajectory and the resulting pessimistic evaluation function are shown in Figure 11.10. Notice that the evaluation has improved for the states early on which subsequently followed an improving trajectory but did not change for the part of the trajectory which became worse.

After further experiences, some random and some using the evaluation function (unsuccessfully), the experiences and evaluation function looked as in Figure 11.11. It can be seen to resemble the evaluation function obtained by normal dynamic programming. This is further confirmed by inspection of Figure 11.12 which shows the action map derived from the evaluation function. However, it can also be seen that unless a particularly lucky random experiment takes place, no trial starting at the state space centre will succeed. Following the arrows in Figure 11.12 it can be seen that it will first travel down and to the left through state space, then up, on the left side until it swings rightwards moving along the top. However it comes down too early, meaning it slows down too soon and doesn't make it up the hill. Thus it gets close, but not close enough. This can be seen to have happened already several times in Figure 11.11. This performance is good, considerably better than with the naive local controller, but not quite good enough.

Rather than wait for a lucky random experiment to occur, I gave the controller a hint by starting the puck in a very good state, which was still scored badly by this evaluation function. This is

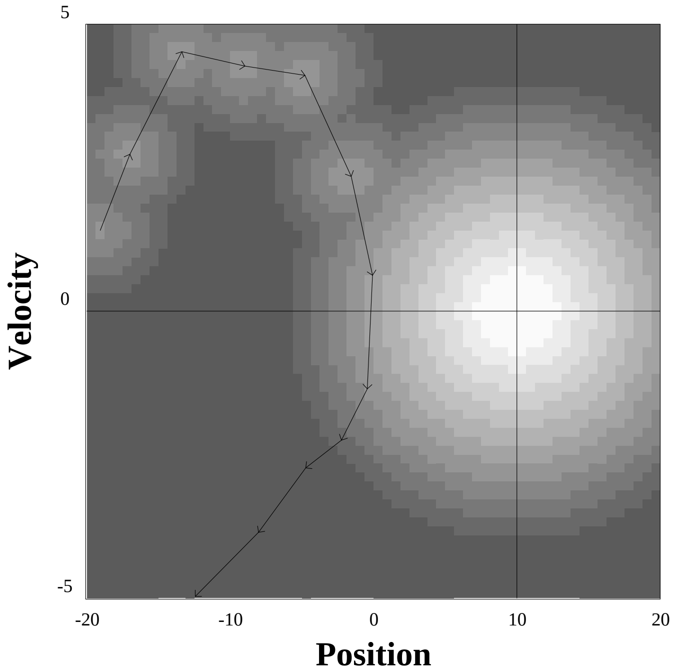


Figure 11.10

Pessimistic slope bounded evaluation after a little experience.

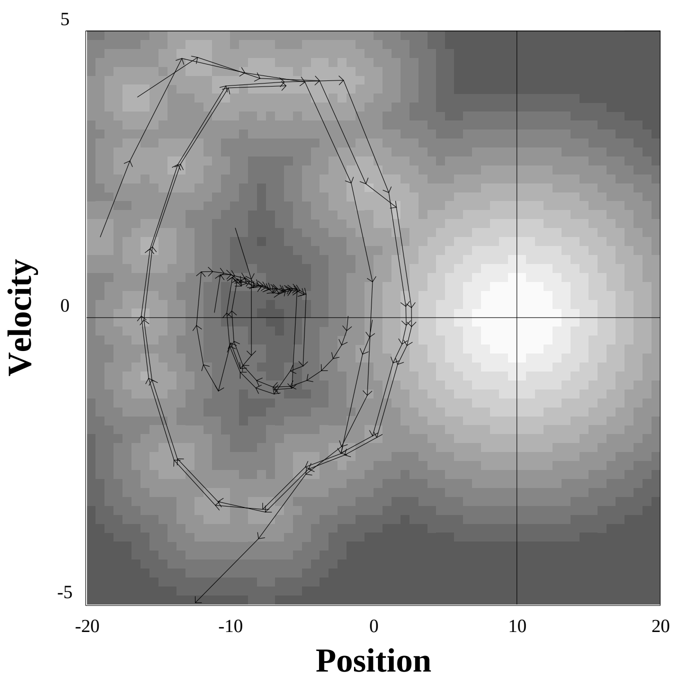


Figure 11.11

$E_{\mathtt{ttg}}$, the pessimistic slope bounded evaluation after further experience.

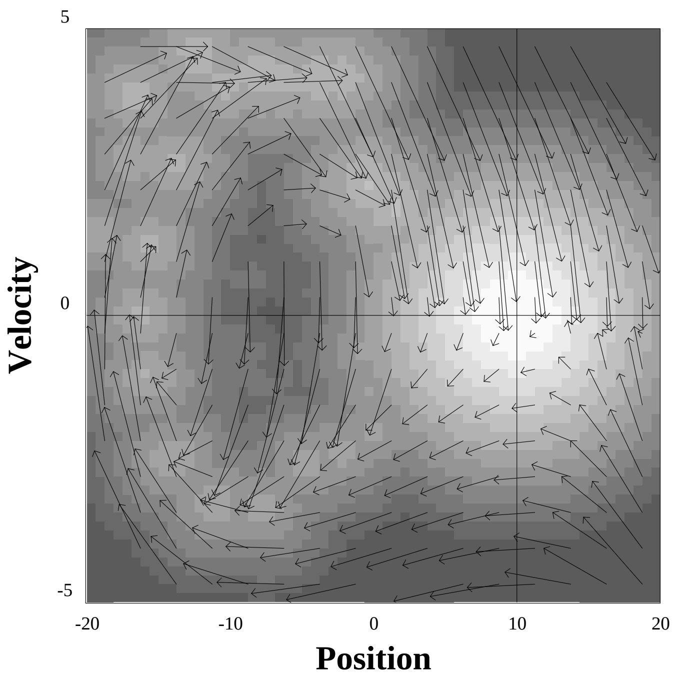


Figure 11.12

The thrusts recommended by the evaluation function of Figure 11.11.

the state of moving rightwards at great speed at $x = 0$. It is good because the hill can be used to decelerate in time to reach the goal. The interesting question was whether the controller would generalize from this extra success.

The trial was carried out, causing the evaluation function shown in Figure 11.13. This new evaluation function is good enough to aid the control of the puck from the centre of state space, as shown in Figure 11.14. This is encouraging because some useful experience in one area of the state space has aided entirely different areas—a very non-local effect.

11.2.5 Discussion

The initial Variable Resolution Dynamic Programming implementation has demonstrated that it is possible to learn to control dynamic systems using non-local strategies. Its advantages are:

- **Variable resolution.** It can adjust to resolutions of interest and is non-parametric. These two important features have been discussed earlier in this dissertation (see Section 5.2).
- **Inexpensive computation.** The cost of updating the evaluation function is low. It happens once at the end of each trial, though there would be nothing to prevent it from occurring incrementally, in parallel with the trials. The typical update cost is $N \log N$ in the number of experiences because the number of dynamic programming iterations is usually only one unless a great "discovery" is made which leads to re-evaluation of non-local parts of the state space. This is a considerable improvement on high dimensional Dynamic Programming.
- **Learning rate.** The rate of learning is high compared with non-DP methods because the one-shot property of SAB learning is retained.

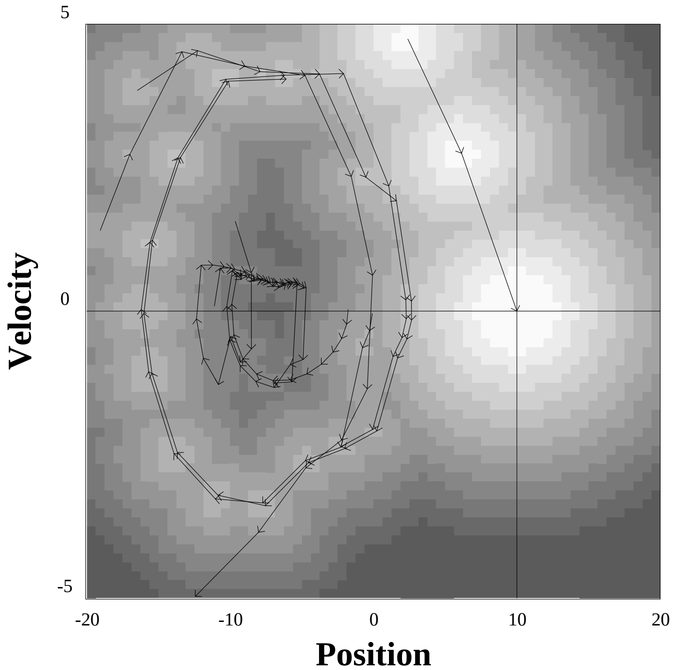


Figure 11.13

E_{ttg}, the pessimistic slope bounded evaluation after further experience, including starting at the state $x = 5, v = 4.5$. This start state lead easily to the goal.

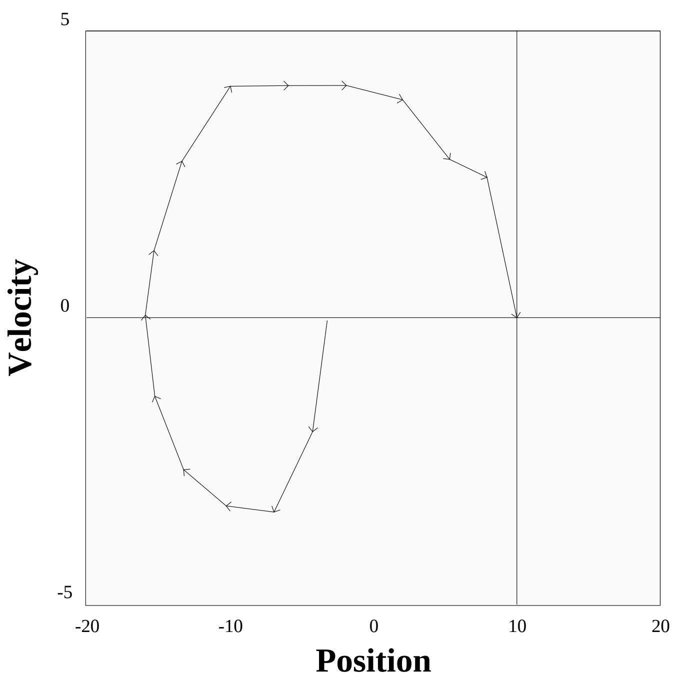


Figure 11.14

The trajectory followed by means of the evaluation function of Figure 11.13.

However, the initial implementation still has some serious problems.

- **Serious discontinuities** seem likely in many evaluation functions. The smoothness assumption (3a) is invalid. Empirical evidence is needed to determine how seriously this undermines performance.
- **Gaining useful experience.** The problem of gaining experience is more difficult than it was for the SAB action chooser because those areas of the state space in which it is important to experiment may themselves be very hard to reach. Furthermore, in a dynamic system such as this, random actions usually result in very quick disaster.
- **Not proven.** The only domain in which it has been attempted is a two-dimensional state space in which quantization of variables is actually adequate. A more complex test is required.

It is not clear that it is important to learn non-local or optimal controllers. The example task here could have been controlled by a slightly more complex local controller which used the tactics

1. Get to a stationary state at the bottom of the valley.
2. Move to $x = -18$m.
3. Accelerate towards the goal.

Such a controller is slightly more work on the part of the system designer, but is not difficult to achieve. It seems likely that to solve the non-local control autonomously it might be more profitable to automate the acquisition of qualitative plans such as that above, rather than to compute them at a low level as we have been doing in this section.

Chapter 12

Conclusion

The conclusion takes the form of a summary of the work of the dissertation, then a description of its important contributions, and finally a discussion of possible extensions.

12.1 Summary

The principal theme of this dissertation has been that realistic robot learning can occur if the things which we choose to learn are models of the world, and that this is particularly useful if the learning is practical, efficient, fast and robust. The supplementary theme has been that learned world models are sufficient to make the rest of robot control considerably easier.

A robot task is controlled by the *SAB* control cycle, reproduced for a final time in the left of Figure 12.1. World models are learned by explicitly storing every experience as a triplet of data: the perceived state in which the experience occurred, the raw action which was then applied, and the resulting perceived behaviour.

By generalizing to new experiences using the closest known experience, it has been explained and demonstrated that learning happens quickly.

Part of the dissertation has dealt with the data structures and algorithms necessary to ensure robust nearest neighbour performance. These have included mechanisms for fast access and update, and robustness to disorder. The dissertation also provides a theoretical analysis of what can be learned, and how quickly, by the nearest neighbour method. The resulting data structure is the *SAB*-tree: a modified, data-sharing, pair of *k*d-trees.

The *SAB* action chooser performs more slowly during early learning, when it knows that it does not know how to achieve requested behaviours. In such cases it can choose the most promising and informative out of a variety of candidate actions, by means of the P_{succ} probability of success heuristic, which is defined in terms of the partially learned forward ($\mathbf{State} \times \mathbf{Action} \rightarrow \mathbf{Behaviour}$) world model. The first candidate action considered by the *SAB* action chooser is always the one

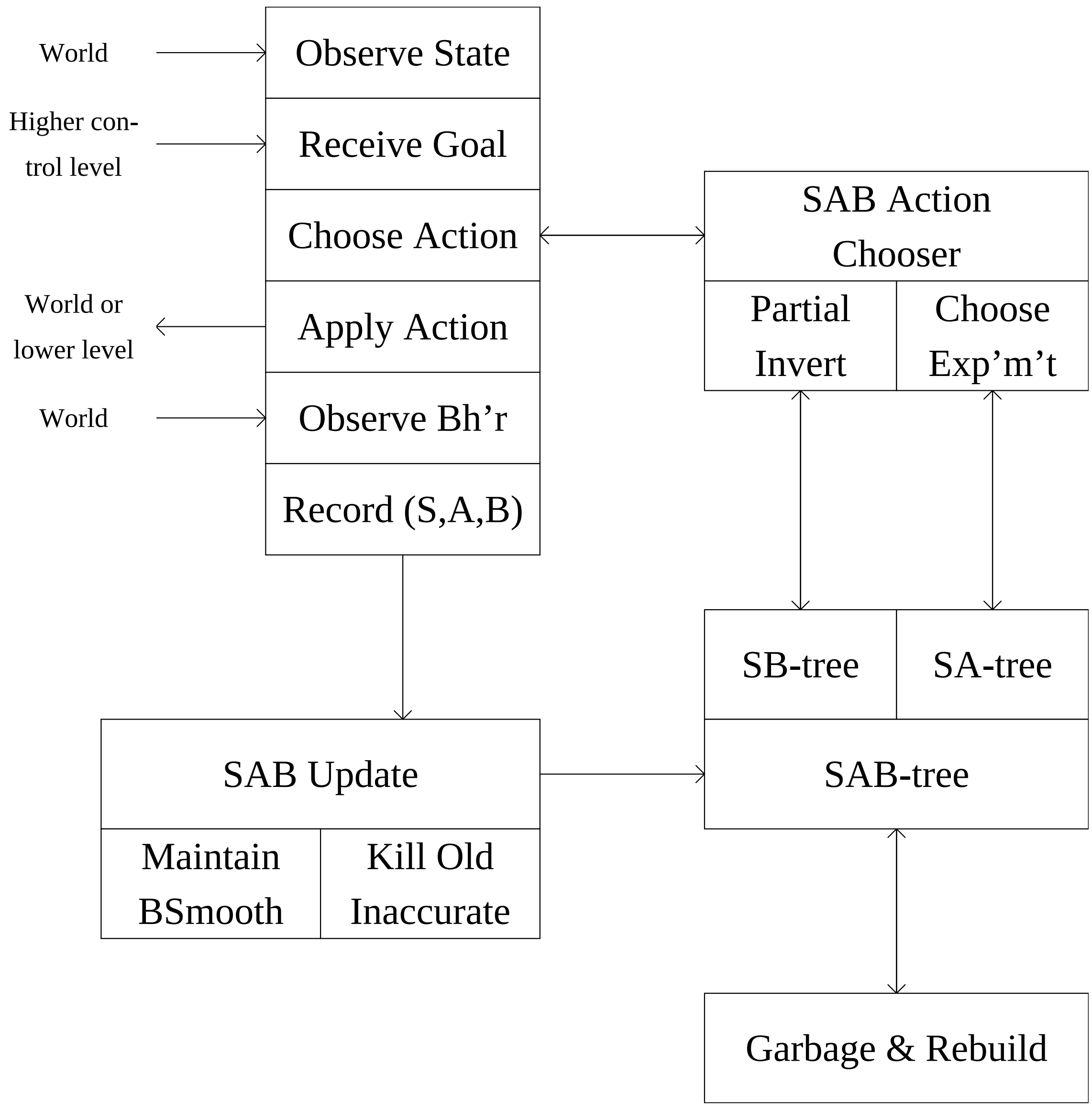


Figure 12.1

The structure of SAB-learning.

recommended by partial inversion. This is obtained by nearest neighbour search of the inverse (**State** $\times$ **Behaviour** $\rightarrow$ **Action**), and possibly faulty, model. This is checked by passing it through the forward model, and if adequate the partial inversion action is used. As more is learned, access becomes faster because the partial inversion estimate is usually known to be suitable, and thus almost no action choosing computation is required. The system becomes increasingly reactive as task performance improves.

An important feature of this work is the attempt to escape the curse of dimensionality. This can only be done in one of two ways.

1. Assume there exists extra structure in the problem and then find the best fit of the data to the structure. This is effective if it is known that any new problem *will* have a solution in the required structure. Otherwise learning can get stuck.

2. Deliberately restrict learning to particular low dimensional subsets of the data which are sufficient to solve the task. This approach was adopted.

The dissertation then describes how

- The part of robot control which is conventionally regarded as the bottom level—the modelling of the robot/world dynamics—can be achieved automatically by *SAB* learning. Furthermore the models can be learned for *any* perceived variables sufficient to determine the state, action and behaviour of the robot.

- Middle abstraction levels such as stability and trajectory tracking can be dealt with in a unified manner—ice puck control. These are tasks in which the behaviour required can be obtained very easily from the current perceived state and current perceived goal.

- Complex tasks, for which local perception-based control strategies do not exist, can be broken up into a hierarchy of simpler *SAB* controllers at varying levels of abstraction. This is demonstrated for the difficult control task of volleying a ball into a bucket with a dynamic robot arm. At this stage the structuring needs to be performed by some expert, presumably human. In order to be able to classify the controller as truly autonomous, it would have to be able to devise the structuring strategy itself. At this stage the lack of full autonomy is a compromise, justified by the observation that a large proportion of the effort in robotic control using conventional methods is not in inventing abstract strategies, but in modelling the world.

12.2 Contributions

12.2.1 Design and Detailed Investigation of the SAB Control Cycle

This work has integrated perception, kinematic and dynamic learning, and posed them as one problem: the relationship between raw perceptions of state, action and behaviour. It was shown that much of the previous model-learning work in the field can be cast in the same format. The *SAB* control cycle was introduced, and its use was demonstrated by a range of detailed empirical experiments.

12.2.2 Using Nearest Neighbour to Learn World Models

The nearest neighbour method had been used for classification for some years. Its use to learn mappings had been suggested, but generally passed over in favour of asymptotically more accurate, though slower and less general, local regression methods. [Clocksin and Moore, 1989] is the first published work known to the author which actually uses the nearest neighbour method to learn a robot world model. Other independent investigations [Mel, 1989; Atkeson and Reinkensmeyer, 1989] have successfully used nearest neighbour within robot learning investigations, though in each case with a different end in sight.

This work has enhanced the *k*d-tree based nearest neighbour method for the purposes of *SAB* learning. In particular it has

- Modified it to simultaneously learn the valid "safe" forward model **State** $\times$ **Action** $\rightarrow$ **Behaviour**, and the useful (but not necessarily valid) inverse model **State** $\times$ **Behaviour** $\rightarrow$ **Action**. The forward model allows the correct model of the world to be learned without the danger of "sticking" or ambiguity dangers which the inverse model can cause. However, when the inverse model is valid it can be consulted for fast performance.

- The simple form of nearest neighbour is in danger of performing poorly in a noisy or non-stationary environment. This work therefore augments the method to cope with these situations in a robust fashion which is also optimized for efficiency. The extra cost is borne entirely at world model update time, which is argued to be less critical than at world model access time because (i) accesses are more common than updates, and (ii) if absolutely necessary the world model could be updated off-line (as the STORE.AFTER experiment demonstrated in Section 10.3).

- The *k*d-tree based nearest neighbour algorithm was tested under a variety of conditions.

12.2.3 SAB Action Chooser

I have implemented a probabilistic mechanism which solves the perform/experiment dilemma by using knowledge about its own knowledge to decide whether experimentation is necessary, and if so what a good experiment would be. I have shown how the process of repeated experimentation can be viewed as an optimization process which will not get stuck in local minima. In the neighbourhood of the optimal action, the error decreases exponentially with subsequent choices (the number of bits of accuracy increases linearly). The *SAB* action chooser also biases the dense, high accuracy, learning to take place only in restricted subregions of the control space, which is essential for learning to take place in high dimensionality.

12.2.4 Integration of *SAB* Control with Compound Controllers

I developed a technique for hierarchical learning to take place at multiple levels of abstraction which, unlike earlier proposals, does not suffer from the danger of being stuck due to the credit or blame assignment problem. This danger is avoided because the system learns, at all levels, only objective observations about the world, rather than action maps.

12.2.5 Incidental Contributions

- Based on empirical tests, I have improved the *k*d-tree building algorithm in order to increase nearest neighbour search speed, at the expense of some tree balance.
- I have provided a survey of recent work in learning robot control.
- I have extended the proofs of PAC-learnability produced by [Aha *et al.*, 1990; Kibler *et al.*, 1988] for nearest neighbour based classification, to the learnability of generally continuous functions.
- I have identified and proved the potentially useful relationship between Albus's CMAC [Albus, 1975a; Albus, 1975b] and a *k*d-tree-based representation—"Symbolic CMAC".
- I have made some initial demonstrations of how world model learning can also help autonomously learn non-local control strategies. This has been by means of dynamic programming, supplemented by a learned *SAB* world model, and by means of an experimental variable resolution version of dynamic programming.

12.3 Future Work and Extensions

The following areas were identified as important issues in Chapter 3, which surveyed the learning control field. Can this work be extended to encompass them?

12.3.1 State Identification

This work has assumed that it was an easy job for the system designer to specify which sensor inputs were sufficient to determine the system state. If this were not the case, then what could be done? One possibility would be to let the robot use the values of all conceivable sensors and previous action signals—and perhaps their time derivatives as well. It is likely that all the information would be sufficient to determine state, but there would be a great deal of redundancy which would slow down all algorithms which were time dependent on state vector length, and also impair the nearest neighbour algorithm. Solutions to this have been proposed:

- In [Vogel, 1989], potential state signals which have no effect on behaviour are detected.
- In [Aha *et al.*, 1990], state components are weighted, and weights of "poorly predicting" components are gradually reduced to zero.
- In [Simons *et al.*, 1982], the most recent sensor readings are initially considered sufficient state representations, but in the presence of apparent non-determinism increasingly older sensor readings and action choices are used to augment the state.

12.3.2 Non-determinism

A new mapping must be learned in place of **State** $\times$ **Action** $\rightarrow$ **Behaviour**. This is

$$\textbf{State} \times \textbf{Action} \times \textbf{Behaviour} \rightarrow [0, 1] \tag{12.1}$$

This is the relationship between the state, the action applied and the probability distribution of the behaviour.

The probability density can be estimated using an exemplar set **E** in a variety of ways [Omohundro, 1987]. The *SAB* action chooser is already designed to choose the action with the highest probability of success, but in its current implementation the uncertainty is only in terms of lack of information. The estimate could be supplemented with an additional term indicating the estimated inherent uncertainty in applying an action.

12.3.3 Induction

The *SAB* learning method could not be called inductive. The representation of the world model is not an attempted explanation of the data, instead it *is* the data.

This non-inductive representation is used precisely to avoid making extra assumptions about the data, which could lead to stuck learning were they incorrect.

Nonetheless, once the data were gathered it would be interesting, and possibly useful for more abstract aspects of control, to try to explain it. There are numerous inductive learning systems, including decision trees and genetic classifier systems, but a particularly suitable method might be Aha's instance-based learning [Aha *et al.*, 1990; Kibler *et al.*, 1988], which obtains a simple explanation in the form of a small number of exemplars under the nearest neighbour generalization.

12.3.4 Further Work on Autonomous Non-local Control

It is intended that variable resolution dynamic programming, as described in Section 11.2, be investigated further. If successful it might help take the automation of system design even further up the hierarchy of abstractions. For example, the acquisition of techniques such as ice puck control, might be automated.

12.3.5 Fast Local Regression

Some of the early work of this investigation used regression based on a selection of local exemplars instead of nearest neighbour prediction. This was abandoned because

- Nearest neighbour gave good performance accuracy for a considerably simpler algorithm.
- Local regression was very hard to implement with adequate real time performance (for reasons described in Section 5.2).
- Local regression was much less noise-tolerant.

However, with increasing computational power it might be easier to use local regression instead of, or in conjunction with, nearest neighbour, particularly to help guide the *SAB* action choosing search.

Further interesting and important extensions include additional empirical tests with real, dynamic robots, and also the intriguing question of how to learn "safely"—in environments where certain sorts of poor performance, even once, would be unacceptably expensive.

12.4 Concluding Remarks

Robotics is a young and promising discipline but it is generally agreed that robots would benefit considerably from more autonomy than that of present industrial machines. It is also well known that models of the behaviour of the robot can considerably simplify robot control. Here we have seen that a robot can learn surprisingly quickly and easily from its environment by simply recording everything that it observes. It is this central idea which has motivated the development of the *SAB* learning system, and it is to this idea that the success of these initial demonstrations must be attributed.

It is hoped that some of the ideas of this dissertation might prove useful in practical systems. It is an exciting thought that the acquisition of world models might be a realistic next step in robot automation.

Appendix A

Format of Graphs

Figure A.1 shows an example of the graph notation used in this dissertation. The graph was obtained from running, in this case, forty experiments (hence the "$N = 40$" on the key). For each x-coordinate on the graph, there were thus forty statistics. The y coordinate corresponding to each x-coordinate is the mean of these forty statistics. In order to represent the spread of the data, the standard deviation of each set of statistics is shown. The length of each bar above the mean point is the standard deviation. Occasionally some standard deviation bars are so large that they do not fit entirely on the graph and are thus clipped.

The standard deviations have a second use. Along with knowledge of the number of statistics, they can indicate the confidence that the mean of the sample is the true mean of the population. For N observations, the probability that the mean lies within $1.96/\sqrt{N}$ sample standard deviations of the mean is approximately 95% [Hoel, 1971]. This is because the sample mean μ_s of a reasonably large set of observations behaves like a normally distributed variable with mean μ_s and variance σ_s^2/N where σ_s^2 is the sample variance. The number of observations for this to be generally accurate has been observed as approximately thirty.

For example, if the mean value is 10, the sample standard deviation 5 and the sample size 40, there is a 95% probability that the true mean of the population is between

$$10 \pm \frac{5}{\sqrt{40}} \tag{A.1}$$

The proportional length of the confidence interval is shown to the left of the key.

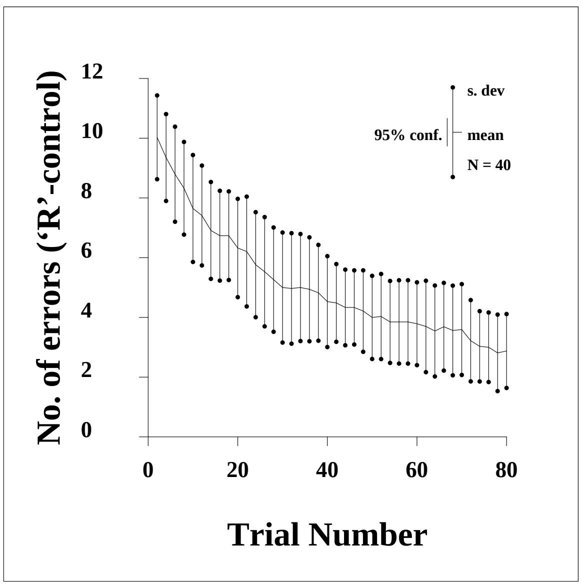


Figure 1.1

A graph typical of this dissertation.

Appendix B

Nearest Neighbour Polynomially Learns Continuous Functions

Looking back at the proof of nearest neighbour learnability given in Section 5.3, the number of exemplars required for the coverage lemma depends directly on ϵ, γ and α where in turn γ depends on β and ϵ depends on β and θ. We know that if we find an n such that (i) $n > N(\epsilon, \gamma, \alpha)$ as defined in Equation 5.12, (ii) $\gamma < \frac{1}{2}\beta$ and (iii) that $\mid I_{\epsilon_1} \mid > 1 - \frac{1}{2}\beta$, then (with probability $1 - \alpha$) we will have achieved tolerance θ.

Here we begin by showing, for the case of a uniformly continuous function, that the dependence of ϵ on θ is linear. This is because a uniformly continuous function has the property (see [Burkhill, 1978]) that

$$\begin{array}{l} \exists G \in \Re : \\ \qquad \forall x, x' \in [0,1]^k \qquad \mid f(x) - f(x') \mid < G \mid x - x' \mid . \end{array} \tag{B.1}$$

Thus if we choose $\epsilon = \theta / G$ we know that for any domain points x,x'

$$\begin{array}{rl} & \mid x - x' \mid < \epsilon \\ \Rightarrow & \mid f(x) - f(x') \mid < G \mid x - x' \mid < G\epsilon = \theta \end{array} \tag{B.2}$$

as required. Using this ϵ, the number of exemplars $N(\epsilon, \gamma, \alpha) = N(\theta/G, \beta/2, \alpha)$ which is

$$n = \left(\frac{2}{\beta} \left\lceil \frac{G\sqrt{k}}{\theta} \right\rceil^k \right) \times \log \left(\frac{1}{\alpha} \left\lceil \frac{G\sqrt{k}}{\theta} \right\rceil^k \right). \tag{B.3}$$

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 $\frac{1}{\alpha}$.

It can also be shown that for bounded-slope functions with discontinuities caused by finite region boundaries, the dependence of ϵ on β and θ is polynomial (proved for a strongly related case in [Aha *et al.*, 1990]). Furthermore, it is conjectured that for asymptotes for which the gradient increases polynomially with the reciprocal of the distance to the discontinuity, that the dependence is again polynomial.

Appendix C

Estimating whether a Directed Behaviour is Attainable

The heuristic described in this appendix was used in the "autonomous time-choice" experiments in Section 10.4. The occasional need for such a heuristic was discussed in Section 8.3. An estimate is required of the probability that a behaviour is attainable at the current state. The task controller can use this probability to decide if it is sufficiently unlikely that it it not worth requesting of the SAB action chooser.

This probability is a heuristic estimate. With the same motivation as has the P_{succ} estimate (Sections 8.1 and 8.2), it is chosen to be robust and computationally cheap. It assumes the goal behaviour has a *direction*. This is equivalent to defining a behaviour origin $\mathbf{o}_B$. The concept of a behaviour having a direction is generally acceptable—for example the behaviour of a dynamic manipulator is an acceleration vector which has a clearly defined origin and direction. Thus, let us define the magnitude and direction of a goal vector by

$$M_{\text{goal}} = \mid \mathbf{b}_{\text{goal}} - \mathbf{o}_B \mid \quad \text{and } \boldsymbol{d}_{\text{goal}} = \frac{\mathbf{b}_{\text{goal}} - \mathbf{o}_B}{M_{\text{goal}}} \tag{C.1}$$

It then estimates the global probability, that there exists any action which can achieve a behaviour whose component, in the requested direction, has at least the requested magnitude.

This global probability is computed using the following approximation:

$$\boxed{\begin{array}{l}\text{There exists an action which}\\ \text{can attain the goal magnitude}\\ \text{in the goal direction}\end{array}} \Leftrightarrow \boxed{\begin{array}{l}\text{There exists an extreme action}\\ \text{which can attain at least the}\\ \text{goal magnitude in the goal di-}\\ \text{rection}\end{array}} \tag{C.2}$$

where an extreme action is an action such that each component is either the minimum or maximum of the permitted values. If the two events of Equation C.2 are considered equivalent then their probabilities are the same. The number of extreme actions is $2^{Dim(\mathbf{Action})}$. For each of these, the probability of achieving a behaviour component at least as large as that requested can be computed using the probability of success heuristic which was used in Section 8.1.

$$\mathbf{Prob}(B.\boldsymbol{d}_{\text{goal}} > M_{\text{goal}} \mid \mathbf{s}_{\text{current}}, a_q, \mathbf{E}) = \operatorname{erf}\left(\frac{(\boldsymbol{d}_{\text{goal}}.\mathbf{b}_{\text{near}}^{\text{smooth}}) - M_{\text{goal}}}{C \mid (\mathbf{s}_{\text{current}}, a_q) - (\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}) \mid}\right) \tag{C.3}$$

where a_q is the qth extreme action and where $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}_{\text{near}}^{\text{smooth}})$ is the nearest neighbour to $(\mathbf{s}_{\text{current}}, a_q)$ in the exemplar set $\mathbf{E}$.

The probability that all of these actions are unable to achieve a large enough component is no more than the probability that the most promising cannot achieve the magnitude (from $\mathbf{Prob}(A \cap B) \leq \mathbf{Prob}(A)$). So if the probability of success of the most promising action is less than some resignation probability, P_{res}, then this estimate recommends that the *SAB* action chooser resign itself to using the best known action.

The computational expense of this scheme is not great for a low dimensional action space ($Dim(\mathbf{Action}) \leq 4$). For a higher dimensional space it might be necessary to perform a hill-climbing search among extreme actions to find an approximation to the most promising extreme action.