ELSEVIER

Artificial Intelligence 92 (1997) 169-227

Artificial Intelligence

Map learning with uninterpreted sensors and effectors [1]

David Pierce [2], Benjamin J. Kuipers *

Department of Computer Sciences, University of Texas at Austin, Taylor Hall, Austin, TX 78712, USA

Received March 1996; revised October 1996

Abstract

This paper presents a set of methods by which a learning agent can learn a sequence of increasingly abstract and powerful interfaces to control a robot whose sensorimotor apparatus and environment are initially unknown. The result of the learning is a rich hierarchical model of the robot's world (its sensorimotor apparatus and environment). The learning methods rely on generic properties of the robot's world such as almost-everywhere smooth effects of motor control signals on sensory features. At the lowest level of the hierarchy, the learning agent analyzes the effects of its motor control signals in order to define a new set of control signals, one for each of the robot's degrees of freedom. It uses a generate-and-test approach to define sensory features that capture important aspects of the environment. It uses linear regression to learn models that characterize context-dependent effects of the control signals on the learned features. It uses these models to define high-level control laws for finding and following paths defined using constraints on the learned features. The agent abstracts these control laws, which interact with the continuous environment, to a finite set of actions that implement discrete state transitions. At this point, the agent has abstracted the robot's continuous world to a finite-state world and can use existing methods to learn its structure. The learning agent's methods are evaluated on several simulated robots with different sensorimotor systems and environments. © 1997 Elsevier Science B.V.

Keywords: Spatial semantic hierarchy; Map learning; Cognitive maps; Feature learning; Abstract interfaces; Action models; Changes of representation

* Corresponding author. E-mail: kuipers@cs.utexas.edu.

[1] This work has taken place in the Qualitative Reasoning Group at the Artificial Intelligence Laboratory, The University of Texas at Austin. Research of the Qualitative Reasoning Group is supported in part by NSF grants IRI-9216584 and IRI-9504138, by NASA grants NCC 2-760 and NAG 2-994, and by the Texas Advanced Research Program under grant no. 003658-242.

[2] E-mail: dmpierce@cs.utexas.edu.

PII S0004-3702(96)00051-3

1. Introduction

Suppose a creature emerges into an unknown environment, with no knowledge of what its sensors are sensing or what its effectors are effecting. How can such a creature learn enough about its sensors and effectors to learn about the nature of its environment? What primitive capabilities are sufficient to support such a learning process?

This problem is idealized to clarify the goals and results of our research. A real robot embodies knowledge designed and programmed in by engineers who select sensors and effectors appropriate to the environment, and implement control laws appropriate to the goals of the robot. A real biological organism embodies knowledge, acquired through evolution, that matches the sensorimotor capabilities of the organism to the demands of the environment. We idealize both of these to the problem faced by an individual learning agent with very little domain-specific knowledge, but with the ability to apply a number of sophisticated, domain-independent learning methods. In addition to its scientific value, this idealized learning agent would be of considerable practical value in allowing a newly-designed robot to learn the properties of its own sensorimotor system. We report here on one learning agent that solves a specific instance of this problem, along with several variations that begin to explore the range of possible solutions to the general problem.

Henceforth, we make a distinction between the learning agent and the robot. The robot is a machine (physical or simulated) that the learning agent must learn how to use. The robot's sensorimotor apparatus is comprised of a set of sensors and effectors. The sensorimotor apparatus is *uninterpreted*, meaning that the agent that is learning how to use the robot has no a priori knowledge of the meaning of the sensors, of the structure of the sensory system, or of the effects of the motor's control signals. From the learning agent's perspective, the sensorimotor apparatus is represented as a *raw sense vector* $\mathbf{s}$ and a *raw motor control vector* $\mathbf{u}$. The former is a vector of real numbers giving the current values of all of the sensors. The latter is a vector of real numbers, called control signals, produced by the learning agent and sent to the robot's motor apparatus. The learning agent's situation is illustrated in Fig. 1.

This paper solves the learning problem by presenting a set of methods that the learning agent can use to learn

(1) a model of the robot's set of sensors,
(2) a model of the robot's motor apparatus, and
(3) a set of behaviors that allow the learning agent to abstract the robot's continuous world to a discrete world of places and paths.

These methods have been demonstrated on a simulated mobile robot with a ring of distance sensors.

These learning methods comprise a body of knowledge that is given to the learning agent a priori. They incorporate a knowledge of basic mathematics, multivariate analysis, and control theory. The learning methods are domain independent in that they are not based on a particular set of sensors or effectors and do not make assumptions about the structure or even the dimensionality of the robot's environment.

In the rest of this paper, we describe a number of learning methods and show how they are used by a learning agent as it develops an understanding of a robot's world

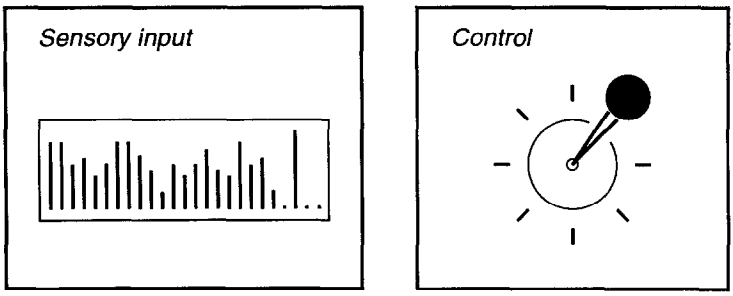


Fig. 1. The learning problem addressed in this paper is illustrated by this interface between a learning agent and a teleoperated robot in an unknown environment. The learning agent's problem is to learn a model of the robot and its environment with no initial knowledge of the meanings of the sensors or the effects of the control signals (except that nothing changes when the control signals are all zero).

by defining a sequence of increasingly powerful *abstract interfaces* to the robot. The learning agent's problem and solution are given below:

Problem.

Given: A robot with an uninterpreted, *almost-everywhere approximately linear* sensorimotor apparatus in a *continuous*, *static* environment.

Learn: Descriptions of the structure of the robot's sensorimotor apparatus and environment and an abstract interface to the robot suitable for *prediction* and *navigation*.

Solution.

Representation: A hierarchical model. At the bottom of the hierarchy are egocentric models of the robot's sensorimotor apparatus. At the top of the hierarchy is a discrete abstraction of the robot's environment defined by a set of discrete views and actions.

Method: A sequence of statistical and generate-and-test methods for learning the objects of the hierarchical model.

An *almost-everywhere approximately linear* sensorimotor apparatus satisfies the following: The derivatives with respect to time of the sensor values can be approximated by linear functions of the motor control vector. A *continuous* world (which includes both the robot and its environment) is one whose state can be represented by a vector $\boldsymbol{x}$ of continuous, real-valued state variables. A *discrete* world, on the other hand, is represented by a finite set of states. The primary example in this paper [3] is a mobile robot in a continuous world with three state variables: two for its position (e.g., longitude and latitude) and one for its orientation (i.e., the direction in which it is facing). A *static* world is one whose state does not change except as the result of a nonzero motor control vector. A static world exhibits no inertia. When the motor controls go to zero, the robot comes to an immediate stop. In a static world, there are no active agents (e.g., pedestrians) besides the robot itself.

[3] Experiments with other robots are described in connection with particular learning methods.

The learning agent's goal is to understand its world, that is, to learn a model of it suitable for prediction and navigation. *Prediction* refers to the ability to predict the effects of the motor control signals. *Navigation* refers to the ability to move efficiently from one place to another. These definitions do not apply perfectly to the learning agent's world: places do not exist a priori—they must be discovered or invented by the learning agent itself. The raw sense vector and the raw motor control vectors are at the wrong level of abstraction for describing the global structure of a world. People do not understand their world in terms of sequences of visual images—they use abstractions from visual scenes to places and objects. In order to understand its continuous world, the learning agent must also use abstractions. Instead of trying to make predictions based on the raw sense vector, it needs to learn high-level features and behaviors. Understanding the world thus requires a hierarchy of features, behaviors, and accompanying descriptions. The hierarchy that the learning agent uses is called the *spatial semantic hierarchy* [16, 18–20].

1.1. The spatial semantic hierarchy

The spatial semantic hierarchy (SSH) is a hierarchical structure for a substantial body of common-sense knowledge, showing how a *cognitive map* can be built on sensorimotor interaction with the world. The cognitive map is the body of knowledge an agent has about the large-scale spatial structure of its environment. ("Large-scale" here means significantly larger than the sensory horizon of the agent, meaning that the map must be constructed by integrating observations over time as the agent travels through its environment.) Since we already have an SSH-based solution for the cognitive mapping problem for a simulated robot with a ring of distance sensors, we focus on learning the sensory features and control strategies necessary to support that solution. The result we obtained was successful, but at the same time revealed some subtle but important changes required to the SSH approach to cognitive mapping.[4]

The spatial semantic hierarchy is comprised of five levels: sensorimotor, control, causal, topological, and metrical. At the *sensorimotor* level, the abstract interface to the robot is defined by the raw sense vector, a set of primitive actions (one for each degree of freedom of the robot, Section 3), and a set of learned features. At the *control* level, action models are learned in order to predict the context-dependent effects of motor control vectors on features. Local state variables are learned and behaviors for homing and path-following are defined (Section 5). The abstract interface to the robot is defined by the set of local state variables, homing behaviors, and path-following behaviors. At the *causal* level, sense vectors are abstracted to a finite set of *views* and behaviors are abstracted to a finite set of *actions* (Section 7). The abstract interface gives the current view and the set of currently applicable actions.

The contribution of this paper is a set of methods for learning these first three levels. This paper's work is complementary to the work done by Kuipers and Byun [18, 19] in which all levels of the descriptive ontology were engineered by hand, and the focus of the learning agent was on learning the structure of the environment. The agent

[4] The most important change is the use of local state variables (Section 4).

selected appropriate control laws from a fixed set to form the control level, which was abstracted to the topological and metrical levels. At the *topological* level, perceptual ambiguities (in which multiple states map to the same view) are resolved and a global representation of the world's structure as a finite-state graph is learned. At the *metrical* level, the topological map is supplemented with distances, directions, and other metrical information.

By showing how to learn the first three levels of the spatial semantic hierarchy, this paper lays the groundwork for building a learning agent that can learn the entire spatial semantic hierarchy using only domain-independent knowledge.

1.2. Overview

Sections 2 through 7 describe a sequence of methods for learning a model of a robot's sensorimotor apparatus and a set of behaviors that allow the learning agent to abstract the robot's continuous world to a discrete world of places and paths. Fig. 30 summarizes the entire set of representations, learning methods, and resulting behaviors, after they have been described in detail in the rest of the paper.

Section 2 describes a method for learning a model of the structure of the robot's sensory apparatus. Section 3 describes a method for learning a model of the structure of the robot's motor apparatus. Section 4 describes a method for learning a set of variables suitable for representing the local state of the robot. Section 5 describes a method for learning a set of robust, repeatable behaviors for navigation through the robot's state space. Section 6 describes a number of experiments (in addition to those described in the previous sections) that demonstrate the generality and some limitations of these learning methods. Finally, Section 7 shows how to define an abstract interface that abstracts from the continuous sensorimotor apparatus to a discrete sensorimotor apparatus.

These learning methods provide a particular solution to the learning problem described in Section 1. This particular solution is an instance of the more general solution outlined below:

(i) Apply a generate-and-test algorithm to produce a set of scalar features.
(ii) Try to learn how to control the generated scalar features. Those that can be controlled are identified as *local state variables*.
(iii) Define *homing behaviors*—behaviors that move a local state variable to a target value.
(iv) Define *path-following behaviors*—behaviors that move the robot while keeping a local state variable at its target value.

The set of learning methods that are presented in this paper does not represent the final word on the problem of learning to use an uninterpreted sensorimotor apparatus. Instead it is one path to the goal. Clearly, there are other ways to instantiate the above sequence of steps. Future work will involve both improving the current set of methods and identifying alternate paths to the solution.

The learning methods and experimental results are interleaved throughout the paper: each section describes a learning method, the representations or objects produced by the method, the source of information used by the method, and one or more demonstrations of the method applied to a simulated robot.

1.3. Contributions

The results of this research are the following:

(i) the demonstration of a learning agent that can solve a nontrivial instance of the learning problem;

(ii) the identification of a plausible though not unique set of primitive capabilities that a robot must have to support such a learning agent;

(iii) the identification of a set of learning methods and intermediate representations that enable the learning agent to go from no domain-specific knowledge to useful cognitive maps of complex environments.

These learning methods are interesting in their own right. First, each one identifies a source of information available through experimentation with an uninterpreted sensorimotor apparatus and, second, each provides a method for exploiting that information to give the learning agent a new way of understanding the robot's sensory input or a new way of interacting with the robot's environment.

The result of this work is an existence proof, demonstrating one path from the beginning to the end of an idealized but important learning problem. We hope that this result can support further work to establish minimal sets of primitives, necessary conditions for success, and the limits of this heterogeneous bootstrapping method for learning.

As intended, the learned set of features and control laws are specific to the robot and the type of environment used for these experiments. The learning method itself also has some degree of dependence on the type of robot and environments used. We used three methods to move towards generality in these results. First, as we needed to add primitive inference capabilities, we required that they be independent of the choice of robot or environment, and that they be plausible to implement using low-level symbolic or neural-net mechanisms.[5] Second, we attempted to minimize, and then make explicit, the assumptions our inference methods make about the nature of the robot or the environment. For example, several feature generators require almost-everywhere temporal and spatial continuity of the sensory inputs.[6] Third, we tested the generality of several key steps in our learning method empirically by applying them to different robot sensorimotor systems and different environments. These results are shown throughout the paper. Naturally, the generality we are able to establish by these means remains limited.

In spite of the limitations of an existence proof, we believe that the approach we have demonstrated is important. First, it shows how a heterogeneous set of learning methods can be used to construct a deep hierarchy of sensory features and control laws. Only a very few previous learning methods such as AM [22] (see also [39]) have constructed similarly deep concept hierarchies. Second, the knowledge con-

[5] We did not always follow this restriction in the implementation itself. For example, we use a fairly sophisticated method called principal component analysis [15] as a feature identification method. However, principal component analysis may be implemented as a neural network [29].

[6] Real sonar sensors may not satisfy this requirement due to specular reflection, a property of sonar sensors that makes them difficult to use, even in systems that are engineered by hand.

tained in this hierarchy shows how a foundational domain of symbolic common-sense knowledge can be grounded in continuous sensorimotor interaction with a continuous world.

2. Learning a model of the sensory apparatus

The learning agent's first step is to learn a model of the robot's sensory apparatus. The output of the learning method used in this step (i.e., the learned model of the sensory apparatus) is a set of groups of related sensors and a description of the physical layout of the sensors. The source of information for this step is the sequence of values produced by the robot's sensors while the agent wanders by choosing motor control vectors randomly. The rest of this section describes the learning method in detail and demonstrates the method on two very different simulated robots.

2.1. A simulated robot

For concreteness, the learning methods are illustrated with a particular robot and environment. The robot's world is simulated as a rectangular room of dimensions 6 meters by 4 meters. The room has a number of walls and obstacles in it. The robot itself is modeled as a point. The robot has 29 sensors. Each sensor's value lies between 0.0 and 1.0. Collectively, these define the raw sense vector s, which is the input from the robot to the learning agent. The first 24 elements of the raw sense vector give the distances to the nearest objects in each of 24 directions. These have a maximum value of 1.0, which they take on when the nearest object is beyond one meter away. The sonars are numbered clockwise from the front. The 21st element is defective and always returns a value of 0.2. The 25th element is a sensor giving the battery's voltage, which decreases slowly from an initial value of 1.0. The 26th through 29th elements comprise a digital compass. The element with value 1 corresponds to the direction (E, N, W, or S) in which the robot is most nearly facing. There is no sensor noise. The robot has a "tank-style" motor apparatus. Its two motor control signals a_0 and a_1 tell how fast to move the right and left treads. Moving the treads together at the same speed produces pure forward or backward motion; moving them in opposition at the same speed produces pure rotation. Moving the treads at different speeds causes the robot to move in a circular arc. The learning agent does not know what any of these sensors or effectors do. The learning agent only knows that that robot's raw sense vector has 29 elements and its raw motor control vector has two elements.

2.2. A language of features

The learning agent develops an understanding of the robot's sensory apparatus by learning new *features*. A *feature*, as defined in this paper, is a function over time whose current value is completely determined by the history of current and past values of the robot's raw sense vector. The type of the feature is determined by the type

of that function's value (thus a vector feature is one whose value at any point in time is a vector.) The types of features used in this paper are the following: *scalar*, *vector*, *group*, *matrix*, *scalar field* (or *image*), *image element*, *focused image*, *vector field*, and *vector field element*. Scalar, vector, and matrix features are based on standard mathematical constructs. The group feature (a type of vector feature) is defined in Section 2.3. The image and image-element features are defined in Section 2.4. The focused-image feature is defined in Section 4.1.1. The vector-field and vector-field-element features are defined in Section 3.2. Examples of features are the raw sense vector (a vector feature) and the elements of the raw sense vector (scalar features). The learning agent produces new features using *feature generators*. A feature generator is a rule that creates a new feature or set of features based on already existing features.

2.3. Discovering related sensory subgroups

A sensory apparatus may contain a structured array of similar sensors. Examples of such arrays are a ring of distance sensors, an array of photoreceptors in a video camera, and an array of touch sensors. The learning agent uses the *group-feature generator* to recognize such arrays of similar sensors. A *group feature* is a vector feature, x, whose elements, x_i, are all related in some way (e.g., all correspond to sensors in an array of similar sensors).

The group-feature generator is based on the following observation. Given a well-engineered array of sensors (e.g., a ring of distance sensors) that measure a property that typically varies continuously with sensor position (e.g., the distance between the robot and nearby objects), the following holds: Sensors that are physically close together in the array "behave similarly". Two sensors are said to behave similarly if

(1) the two sensors' values at each instant in time tend to be similar and

(2) the two sensors' *frequency distributions* are similar.

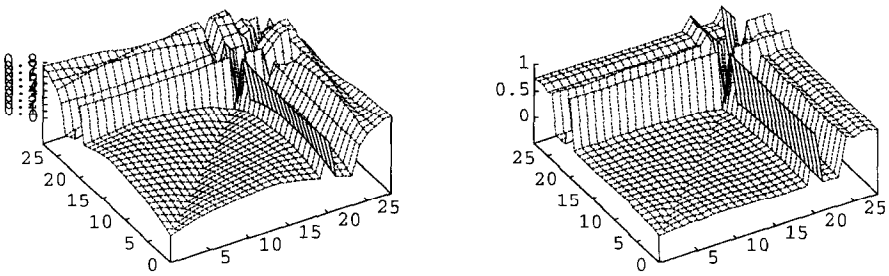
Given a scalar feature x, the frequency distribution (***dist*** x) is an n-element vector that gives, for each of n subintervals in the variable's domain, the percentage of time that the variable assumes a value in that subinterval.

Corresponding to these two criteria are two distance metrics (examples of matrix features) that are used by the group-feature generator.

- The first metric d_1 is based on the principle that in a continuous world, adjacent sensors generally have similar values. The metric is defined, for vector feature x, as a matrix feature:

$$d_{1,ij}(t) = \frac{1}{t+1} \sum_{\tau=0}^{t} |x_i(\tau) - x_j(\tau)|.$$

  Here, $d_{1,ij}(t)$ is the distance between sensors x_i and x_j measured at time t. The variable τ is a time index ranging from 0 to t.
- The second metric d_2 is based on the observation that sensors in a homogeneous array have similar frequency distributions. For example, an array of binary touch sensors can be distinguished from an array of photoreceptors by the fact that the



d_1 d_2

Fig. 2. Two measures of dissimilarity, $d_{1,ij}$ and $d_{2,ij}$, between the ith and jth elements of the raw sensory feature after the robot has wandered for five minutes. The coordinates are indices i and j.

different types of sensors have radically different frequency distributions. Binary touch sensors can assume value 0 or 1 whereas photoreceptors can assume any value from a continuous range. $d_{2,ij}$ is proportional to the sum over the distribution intervals of absolute differences in frequency for elements i and j:

$$d_{2,ij} = \frac{1}{2} \sum_l |(\textbf{\textit{dist}}\, x_i)_l - (\textbf{\textit{dist}}\, x_j)_l|,$$

where l ranges over the subintervals of the frequency distributions. In the implementation, the frequency distributions use 50 subintervals uniformly distributed over the range $[-1, 1]$.

This generator computes these two distance metrics over a period of several minutes while the learning agent moves the robot using the following strategy: choose a random motor control vector; execute it for one second (10 time steps); repeat.[7] The values of the distance metrics, d_1 and d_2, after the example robot has explored for 5 minutes (3000 observations) are given in Fig. 2.

The group-feature generator exploits these distance metrics in two steps:

(1) formation of subgroups of sensors that are similar according to all of the distance metrics, and

(2) taking the transitive closure of the similarity relation to form closed subgroups of related sensors.

Formation of subgroups of similar sensors

The group-feature generator's first step is to use the distance metrics d_k to form subgroups of similar sensors. Elements i and j are similar, written $i \approx j$, if they are similar according to each distance metric d_k:

[7] Our experiments have shown that this strategy is more effective for efficiently exploring a large subset of the robot's state space than choosing motor control vectors randomly at each time step.

$$i \approx j \quad \text{iff} \quad \forall k\colon\ i \approx_k j.$$

The definition of $i \approx_k j$ requires the use of a threshold. One way to define this threshold, that has proven to be more robust than the use of a constant, is this:

$$\varepsilon_{k,i} = 2 \min_j \{d_{k,ij}\}.$$

Each element i has its own threshold based on the minimum distance from i to any of its neighbors. Elements i and j are considered similar if and only if both $d_{k,ij} < \varepsilon_{k,i}$ and $d_{k,ij} < \varepsilon_{k,j}$, that is if j is close to i from i's perspective and vice versa. Combining these constraints gives

$$i \approx_k j \quad \text{if } d_{k,ij} < \min\{\varepsilon_{k,i}, \varepsilon_{k,j}\}.$$

Formation of closed subgroups

The group-feature generator's second step is to take the transitive closure of the similarity relation to produce the *related-to* relation $\sim$. Consider again the ring of distance sensors. Adjacent sensors tend to be very similar according to the distance metric, but sensors on opposite sides of the ring may be dissimilar (according to d_1) since they detect information from distinct and uncorrelated regions of the environment. In spite of this fact, the entire array of distance sensors should be grouped together. This is accomplished by defining the *related-to* relation $\sim$ as the transitive closure of the *similarity* relation $\approx$. Two elements i and j are *related* to each other, written $i \sim j$, if $i \approx j$ or if there exists some other element k such that $i \sim k$ and $k \sim j$:

$$i \sim j \quad \text{iff} \quad i \approx j \vee \exists k\colon\ (i \sim k) \wedge (k \sim j).$$

The related-to relation $\sim$ is clearly reflexive, symmetric, and transitive and is therefore an equivalence relation. Computing the relation $\sim$ for i and j given the relation $\approx$ is straightforward (e.g., [4]). An equivalence class of the relation $\sim$, if not a singleton, is described as a group feature of s.

For the example robot, the raw sensory feature has 29 elements. In order, these are: 24 distance sensors (one of which is defective), a battery-voltage sensor, and a four-element digital compass. The distance metric is computed while the robot wanders randomly for 3000 steps. For each of the elements of the raw sensory feature, the set of similar elements $\{j \mid i \approx j\}$ is computed and shown below:

(0 1 2 22 23) (0 1 2 3 23) (0 1 2 3 4) (1 2 3 4 5) (2 3 4 5 6) (3 4 5 6 7) (4 5 6 7) (5 6 7 8 9) (7 8 9 10) (7 8 9 10 11) (8 9 10 11 12) (9 10 11 12 13) (10 11 12 13 14) (11 12 13 14 15) (12 13 14 15 16) (13 14 15 16 17) (14 15 16 17 18) (15 16 17 18 19) (16 17 18 19) (17 18 19 21) (20) (19 21 22 23) (0 21 22 23) (0 1 21 22 23) (24) (25) (26) (27) (28).

Notice that the distance sensors are grouped together into groups of neighboring sensors. For example, the group (0 1 2 22 23) contains two elements on each side of element 0. The related-to relation $\sim$ is obtained by taking the transitive closure of the similarity relation and is described by the following equivalence classes:

(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)

(20) *defective*

(24) *battery voltage*

(25) *east*

(26) *north*

(27) *west*

(28) *south*

The distance sensors have all been grouped together into a group containing no other sensors.

2.4. A structural model of the sensory apparatus

The grouping of the sensors into subgroups is a first step but it tells nothing about the relative positions of the sensors in the array. This is accomplished by the *image-feature generator*. The image-feature generator is a rule that takes a group feature and associates a position vector with each element of the group feature in order to produce an *image feature* (which represents the structure of the group of sensors). An image feature is a function over time, completely determined by the current and past values of the raw sense vector, whose value at any given time is an *image*. An image is an ordered list of *image elements*. An image element is a scalar with an associated position vector (a vector of n real numbers that represents a position in a continuous, n-dimensional space). An example of the use of an image feature is to represent the pattern of light intensities hitting the photoreceptors in a camera.

The task of the image-feature generator is to find an assignment of positions to elements that captures the structure of an array of sensors as reflected in the distance metric d_1. This means that the distance between the positions of any two elements in the image should be equal to the distance between those elements according to the metric d_1. Expressed mathematically, image feature y should satisfy

$$\|(\textbf{\textit{pos}}\ y_i) - (\textbf{\textit{pos}}\ y_j)\| = d_{1,ij},$$

where $(\textbf{\textit{pos}}\ y_i)$ is the position vector associated with the ith element in the image and $\|(\textbf{\textit{pos}}\ y_i) - (\textbf{\textit{pos}}\ y_j)\|$ is the Euclidean distance between the positions of the ith and jth elements.

Finding a set of positions satisfying the above equation is a constraint-satisfaction problem. If the group feature x has n elements, then the metric d_1 provides $n(n-1)/2$ constraints.[8] Specifying the positions of n points in $n-1$ dimensions requires $n(n-1)/2$ parameters: 0 for the first point, which is placed at the origin; 1 for the second, which is placed somewhere on the x axis; 2 for the third, which is placed somewhere on the x-y plane; etc. Thus, to satisfy the constraints, n position vectors of dimension $n-1$

[8] The metric can be represented as a symmetric matrix with zeros on the diagonal. Such a matrix has $n(n-1)/2$ free parameters.

are required. Solving for the position vectors given the distance constraints can be done using a technique called *metric scaling* [15].[9]

The problem remains that n points of dimension $n-1$ are inconvenient to use, if not meaningless, for large n. In general, sensory arrays are 1-, 2-, or 3-dimensional objects. What is needed is a method for finding the smallest number of dimensions that are needed to satisfy the given constraints without excessive error, where the error is given by the equation

$$E = \frac{1}{2}\sum_{ij}(\|(\textbf{pos}\ y_i) - (\textbf{pos}\ y_j)\| - d_{ij})^2.$$

Metric scaling helps by ordering the dimensions according to their contribution toward minimizing the error term.

Ignoring all but the first dimension (i.e., using only the first element of the position vectors), yields a rough description of the sensory array with large error (unless the array really is a one-dimensional object). Using all $n-1$ dimensions yields a description that has zero error but contains a lot of useless information. Statisticians use a graph called a "scree diagram" (Fig. 3(a)) that shows the amount of variance in the data that is accounted for by each dimension, to subjectively choose the right number of dimensions. The image-feature generator chooses the number of dimensions to be equal to m where m maximizes the expression $\sigma^2(m) - \sigma^2(m+1)$ where $\sigma^2(m)$ is the variance in the data accounted for by the mth dimension. For the example, $m = 2$. The set of two-dimensional positions found by metric scaling for the group of distance sensors is shown in Fig. 3(b).

The set of $(n-1)$-dimensional position vectors optimally describes the structure of a group, but when these positions are projected onto a subspace of lower dimensionality, the resulting description is no longer optimal. Elements that were the right distance apart in $n-1$ dimensions are generally too close together in the two-dimensional projection. To compensate for this, a relaxation algorithm is used to find the best set of positions in a small-dimension space to approximate the given distances in $n-1$ dimensions. The relaxation algorithm is an iterative process. On each iteration, each position vector is adjusted slightly in a direction that reduces the value of the error term E (defined above). The process continues until the error is very small or ceases to decrease appreciably on each iteration.[10]

The relaxation algorithm could be used without metric scaling by simply initializing the vector of positions randomly. Metric scaling provides two benefits. It shows how many dimensions are needed for the image feature, and it provides a starting point for the relaxation algorithm, decreasing the chance that the algorithm finds a local but not global minimum of the error function. The application of the relaxation algorithm to the group of distance sensors is illustrated in Fig. 3(c).

[9] It seems plausible that metric scaling could be implemented using a neural net analogous to that used to implement principal component analysis [29] since in both cases the main computation is the decomposition of an input matrix into a set of eigenvectors.

[10] See [32, p. 65] for a description of the algorithm.

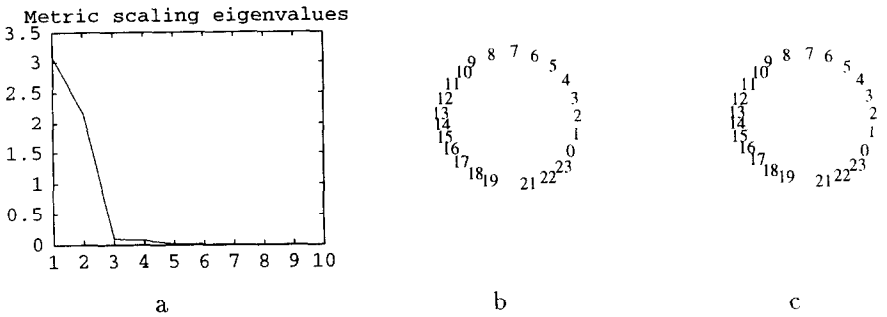


Fig. 3. Learning a structural model of a ring of distance sensors. (a) The scree diagram gives the amount of variance (vertical axis) accounted for by each dimension (horizontal axis) and shows that the first two dimensions account for most of the variance. (b) Metric scaling is used to assign positions to elements of the group of distance sensors. The 22-dimensional position vectors are projected onto the first two dimensions to produce the representation shown above. (c) A relaxation algorithm is used to find a set of two-dimensional positions for the group of distance sensors that best satisfies the constraints $\|\boldsymbol{p}_i - \boldsymbol{p}_j\| = d_{ij}$. (The usefulness of the relaxation algorithm is more obvious in the example of the next section.) Notice the gap corresponding to the defective distance sensor. The element with index 0 corresponds to the robot's forward sensor.

To summarize, the image-feature generator takes a group feature x and produces an image feature y whose position vectors $\boldsymbol{p}_i$ are found using metric scaling and a relaxation algorithm so that they approximately satisfy the constraints

$$\|\boldsymbol{p}_i - \boldsymbol{p}_j\| = k \sum_t |x_i(t) - x_j(t)|$$

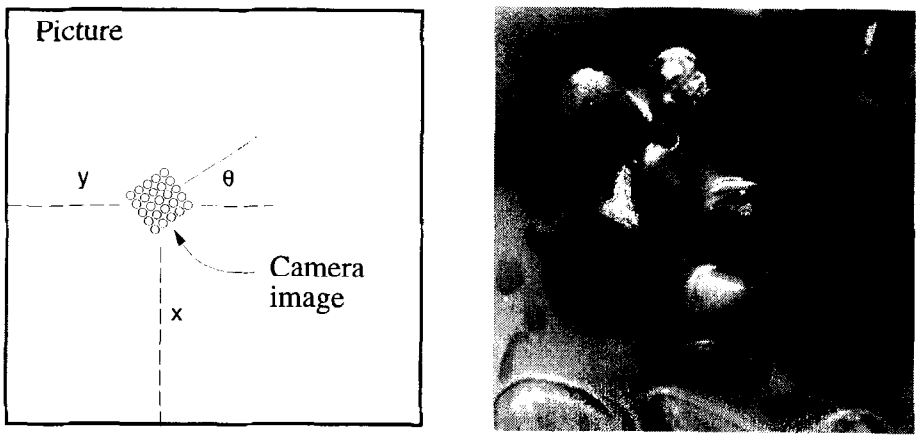
while keeping the dimensionality of the position vectors $\boldsymbol{p}_i$ small. The result of the experiment is a structural description of the robot's ring of distance sensors (Fig. 3(c)) that is used later to analyze the robot's motor apparatus.

2.5. *Learning a sensory model of the roving eye*

The learning methods are further demonstrated using a more fanciful robot called a "roving eye." Its primary sensory array is a retina of photoreceptors.

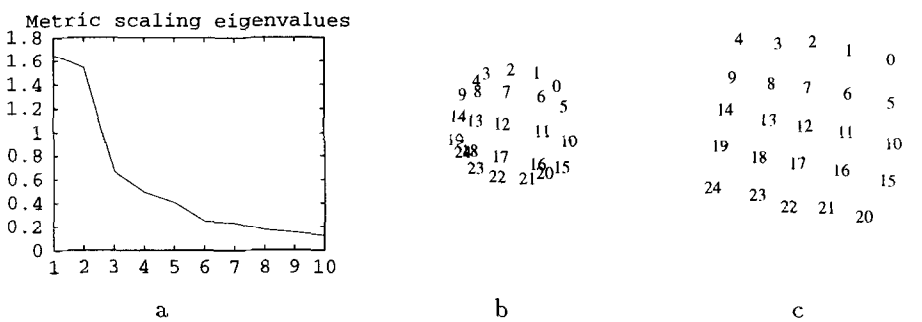
This robot is a simulation of a small camera mounted on the movable platform of an X-Y plotter, pointing down at a square picture 10 centimeters on a side. The camera sees one square centimeter of the picture at a time. The robot has 3 degrees of freedom (translation in two directions and rotation) and its state space is described by three state variables (two for position and one for orientation). The robot's structure is shown in Fig. 4(a). The actual picture used is shown in Fig. 4(b). The sensory system is as before except that the ring of distance sensors has been replaced by a 5 by 5 retinal array looking down on a picture. The motor control vector of this robot has three elements: *rotate*, *slip* (for motion to the left or right), and *slide* (for motion forward or backward).

The results parallel those of the previous experiment. The group-feature generator identified seven equivalence classes: six singletons and one candidate for application



a b

Fig. 4. (a) The robot is a "roving eye" that can see a 1 centimeter wide image that is part of a picture that is 10 centimeters wide. (b) The picture used for the roving-eye experiment is a close-up view of the Oregon coast.



a b c

Fig. 5. (a) The metric-scaling scree diagram for the group of photoreceptors indicates that the sensors are organized in a two-dimensional array. (b) The 2-D projection of the set of positions produced by metric scaling for the group of photoreceptors provides an initial approximation of the grid structure of the array of photoreceptors. (c) The final set of positions are produced using the constraint-satisfaction relaxation algorithm, with the previous set of positions as initial values.

of the image-feature generator. Metric scaling produces the scree diagram of Fig. 5(a) indicating that the sensory array is best modeled as a two-dimensional object. Metric scaling assigns positions to each element of the group feature. Projecting these positions onto the first two dimensions produces the mapping shown in Fig. 5(b). The set of positions produced by metric scaling is improved by the relaxation algorithm so that the distances in the resulting image more closely match the distance metric d_1. The resulting set of positions is shown in Fig. 5(c).

3. Learning a model of the motor apparatus

Using its learned model of the robot's sensory system, the learning agent's second step is to learn a model of the robot's motor apparatus. The result of the learning is a new abstract interface to the robot that identifies the types of motion that the robot's motor apparatus is capable of producing and that tells how to produce each type of motion. The source of information for this step is the sequence of values of a learned *motion feature* (a type of field feature, defined in Section 3.2) as the agent wanders by choosing motor control vectors randomly. In the simulations, if the robot is touching a wall, it is capable of turning but cannot change its position unless it is facing away from the wall.[11]

The image feature makes it possible to define spatial attributes of the sensory input in terms of the locations of sensors in the image. With spatial attributes, it is possible to define spatial as well as temporal derivatives, so motion features can be defined, even without knowledge of the physical structure of the environment. The learning agent uses the new motion feature to analyze its motor apparatus using the following steps:

(i) *Sample the space of motor control vectors.* The robot's infinite space of motor control vectors is discretized into a finite set of representative vectors, $\{\boldsymbol{u}^i\}$.

(ii) *Compute average motion vector fields* (*amvf*'s). The agent repeatedly executes each representative control vector many times in different locations and measures the average value of the resulting motion feature. It is this average value that characterizes the effect of that control vector.

(iii) *Apply principal component analysis* (PCA). The set of computed *amvf*'s is a representation of the effects that the motor apparatus is capable of producing. PCA is used to decompose this set into a basis set of *principal eigenvectors*, a set of representative *amvf*'s from which all *amvf*'s may be produced by linear combination.

(iv) *Identify primitive actions.* Each principal eigenvector is matched against the *amvf*'s produced by the representative control vectors to find a control vector that produces that effect or its opposite. Such a motor control vector, if it exists, is identified as a *primitive action* and can be used to produce motion for one of the robot's degrees of freedom.

(v) *Define a new abstract interface.* For each degree of freedom, a new control signal is defined that allows the agent to specify the amount of motion for that degree of freedom.

The result of the learning is a new abstract interface to the robot comprised of a new set of control signals, one per degree of freedom of the robot. The new interface hides the details of the motor apparatus. For example, whether a mobile robot's motor apparatus uses tank-style treads or a synchro-drive mechanism, the learned interface presents the agent with two control signals: one for rotating and one for advancing. These learned control signals are used to further characterize the robot's motor apparatus using the

[11] The use of a physical robot would require a provision such as an innate obstacle-avoidance behavior to prevent the robot from damaging itself.

static and *dynamic action models* (Sections 4 and 5). Steps 1 through 5 are explained in detail in the rest of this section.

3.1. Sample the space of motor control vectors

The choice of the set of representative motor control vectors must satisfy two criteria: first, they must adequately cover the space of possible *motor control vectors* so that the space of possible *effects* (*amvf*'s) is adequately represented. Second, the distribution of motor control vectors must be dense enough so that, given a desired effect (e.g., an *amvf* that corresponds to one of the robot's degrees of freedom), a motor control vector that produces that effect can be found.

Since we have already made the assumption that the motor apparatus is approximately linear, it suffices to characterize the effects of a uniformly distributed set of unit motor control vectors. (A unit vector has a magnitude of 1 where its magnitude is equal to the square root of the sum of squares of its elements.) For two- and three-dimensional spaces of motor control vectors, respectively, 32 and 100 vectors have been found to be adequate. For the 2-D case, it is easy to find a set of vectors that are uniformly distributed on the unit circle. The ith of n vectors has value $(\cos(2\pi i/n), \sin(2\pi i/n))$. For the 3-D case, a set of vectors uniformly distributed on the unit sphere is found using the relaxation algorithm of Section 2.4. The vectors are constrained to lie on the unit sphere (i.e., to have magnitude 1), and the target distance between any pair of points is much larger than 2. The resulting configuration of vectors is analogous to a collection of electrons on a charged sphere—each vector is as far from its neighbors as possible. These vectors are used as the representative motor control vectors for sampling the continuous space of average motion vector fields. This method generalizes to any dimension.

3.2. Compute average motion vector fields

A *vector field* feature is a function over time, completely determined by the current and past values of the raw sense vector, whose value at any given time is a *vector field*. A vector field is an ordered list of *vector field elements*. A vector field element is a vector with an associated position vector. Given image feature x, (***motion*** x) denotes a vector-field feature (specifically, a *motion* vector-field feature) whose elements measure the amount of motion detected at the corresponding points in the image.

To understand what this feature is measuring, suppose that, corresponding to an object in the robot's environment, there is a property of the image feature (e.g., a local minimum or discontinuity) that changes location from one image element to another on subsequent time steps due to the motion of the robot. A vector from the position of the first image element to the position of the second represents the motion of that object and is an example of a *local motion vector*. A list of local motion vectors, one for each image element, is a *motion vector field*.

The detection of these motion vectors does not require sophisticated object recognition. It simply requires spatial and temporal information, both of which are provided by an image feature. The spatial information is provided by the positions of the elements of the

image; the temporal information is provided by the derivatives of the elements' values with respect to time. A temporal sequence of images, represented as vectors of values and associated positions, can be viewed as an intensity function $E(p,t)$ that maps image positions to values, called intensities, as a function of time. Such a function has both a spatial derivative, $\vec{E}_p$ and a temporal derivative, E_t. The spatial derivative $\vec{E}_p$, also called the *gradient* of E, is a vector in image-position coordinates that gives the direction in which the intensity increases most rapidly.

A large gradient in an image detected by a robot's sensory array corresponds to a detectable property of the environment such as the edge of an object. If the object moves relative to a robot's sensory array (or vice versa), the edges detected in the image will move. This motion results in a change in intensity. A point in the image with a large gradient will, in the presence of motion, also have a large temporal derivative. This is an informal motivation for the *optical flow* constraint equation [11], which defines the optical flow at a point in an image to have magnitude $-E_t/\|\vec{E}_p\|$ and direction $\vec{E}_p$:

$$v = -\frac{E_t}{\|\vec{E}_p\|}\frac{\vec{E}_p}{\|\vec{E}_p\|} = -\frac{E_t\vec{E}_p}{\|\vec{E}_p\|^2}.$$

Here, $\|\vec{E}_p\|$ is the magnitude of the vector $\vec{E}_p$, equal to the square root of the sum of squares of the elements of $\vec{E}_p$. A problem with this formulation is that if the magnitude of $\vec{E}_p$ is small (or zero), then the calculation is prone to error (or is undefined). Since the goal here is to measure average motion over time and since the measurement of the optical flow is more precise at edges or, in general, when the gradient $\vec{E}_p$ is large, we have found it useful to weight the expression using the term $\|\vec{E}_p\|^2$ and measure the value of:

$$v = -E_t\vec{E}_p.$$

In most computer vision applications, images are represented as regularly spaced arrays of pixels (picture elements). With such a representation, it is straightforward to define an approximation for the spatial derivative at a point in the image. The images as defined here, however, do not have such a regular structure so we use a different approach to defining what we call the *sensory flow field*. The sensory flow measured at element i is taken to be a weighted sum of *local motion vectors* v_{ij} in the direction from element i to element j where j ranges over all of the elements close to element i (as defined in Section 2.3). The weight is inversely proportional to the distance between elements i and j. The precise definition of the ***motion*** operator is given below, where $(\boldsymbol{pos}\ x)$ denotes the vector of positions associated with feature x, and $(\boldsymbol{val}\ x)$ denotes the vector of values associated with feature x:

$$\boldsymbol{pos}(\boldsymbol{motion}\ x) \stackrel{\text{def}}{=} \boldsymbol{pos}\ x,$$

$$(\boldsymbol{val}(\boldsymbol{motion}\ x))_i \stackrel{\text{def}}{=} \sum_{j \approx i} v_{ij}/\|p_{ij}\|,$$

$$p_{ij} = (\boldsymbol{pos}\ x_j) - (\boldsymbol{pos}\ x_i),$$

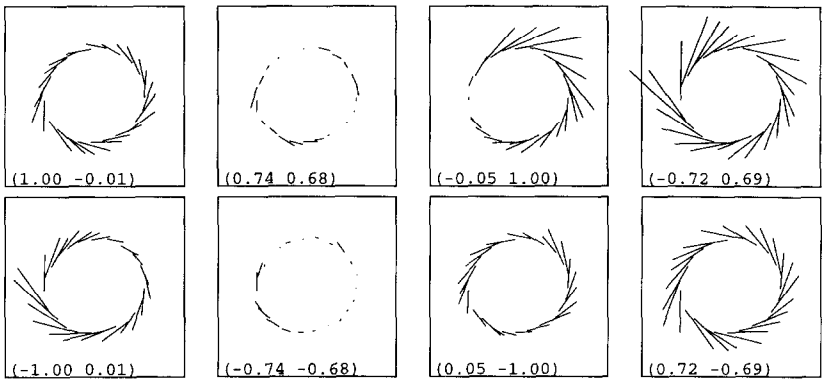


Fig. 6. Examples of average motion vector fields (*amvf*'s) (represented as collections of line segments) and associated motor control vectors (shown in the lower-left corner of each picture). An *amvf* associates an average local motion vector with each position in an image (see Fig. 3). Each line segment represents the position, direction, and magnitude of one of these average local motion vectors.

$$\boldsymbol{v}_{ij} = -E_{t,i}\vec{E}_{\boldsymbol{p},ij},$$

$$E_{t,i} = \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{val}\, x)_i,$$

$$\vec{E}_{\boldsymbol{p},ij} = \frac{((\boldsymbol{val}\, x)_j - (\boldsymbol{val}\, x)_i)}{\|\boldsymbol{p}_{ij}\|}\frac{\boldsymbol{p}_{ij}}{\|\boldsymbol{p}_{ij}\|}.$$

Here, $\|\boldsymbol{p}_{ij}\|$ is the distance in the image between the positions of elements i and j; $E_{t,i}$ is the temporal derivative of the intensity function for element i; and $\vec{E}_{\boldsymbol{p},ij}$ is the component of gradient $\vec{E}_{\boldsymbol{p}}$ at element i in the direction toward element j.

Using the ***motion*** operator, the definition of the *amvf* associated with the ith representative motor control vector $\boldsymbol{u}^i$ is

$$amvf_i = \mu((\boldsymbol{motion}\, x) \mid (\boldsymbol{u} = \boldsymbol{u}^i)),$$

where x is the image feature that has already been learned (Section 2.4), $\boldsymbol{u}$ is the motor control vector used to control the motor apparatus, and μ is an operator that computes the average value of its argument. In this case, the average value is taken over all time steps during which $\boldsymbol{u}^i$ was taken. Examples are shown in Fig. 6. These are obtained after the learning agent has wandered for 20 minutes using the exploration strategy of randomly choosing a representative motor control vector and executing it for one second (ten time steps).

3.3. Apply principal component analysis

The goal of this step is to find a basis set for the space of effects of the motor apparatus, i.e., a set of representative motion vector fields from which all of the motion

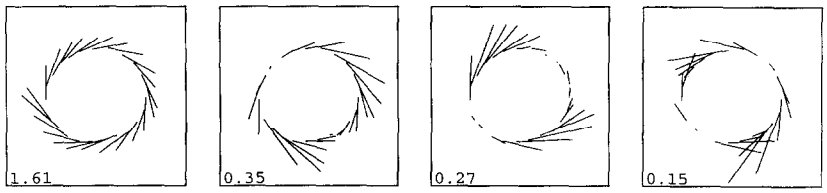


Fig. 7. The first four eigenvectors and the standard deviations of the associated principal components for the space of average motion vector fields. The first corresponds to a pure rotation motion and the second corresponds to a forward translation motion. In these diagrams, the top-left element is associated with the robot's front sensor. The robot's motor apparatus can produce the first two effects directly using the motor control vectors shown.

vector fields may be produced by linear combination. This type of decomposition may be performed using principal component analysis (PCA). (See Mardia et al. [25] for an introduction. Oja [29] discusses how a neural network can function as a Principal Component Analyzer. Ritter et al. [37] show that self-organizing maps [13] can be seen as a generalization of PCA.)

Principal component analysis of a set of values for a variable y produces a set of orthogonal unit vectors $\{v^i\}$, called *eigenvectors*, that may be viewed as a basis set for the variable y. The ith principal component of y is the dot product of y and eigenvector v^i. In practice, y may be approximated as a linear combination of the first few eigenvectors while throwing the remaining ones away.[12] Principal component analysis may be performed using a technique called *singular value decomposition* [35], which identifies the eigenvectors and computes the standard deviation of each principal component. The relative magnitudes of the standard deviations tell how important each eigenvector is for the purposes of approximating the sample values for y. The first four eigenvectors obtained in the experiment are shown in Fig. 7.

3.4. *Identify primitive actions*

In the previous step, principal component analysis was used to determine a basis set of effects for the motor apparatus, namely, the set of eigenvectors. The goal of this step is to discover which motor control vectors can be used to produce those effects. This is accomplished by matching the eigenvectors with the *amvf*'s of all of the representative motor control vectors. The matching involves computing, for each i and j, the angle θ_{ij} between the ith eigenvector and the jth *amvf*. This angle is defined by the equation $\cos\theta_{ij} = v^i \cdot amvf_j$ where the vector fields v^i and $amvf_j$ are treated as simple vectors by flattening their n m-dimensional local motion vectors into a single nm-dimensional vector and ignoring the positions of the local motion vectors. An angle near zero indicates that

[12] The principal components are ordered according to their standard deviations. This means that the first eigenvector accounts for the most variance in the set of observed values for y, and so forth.

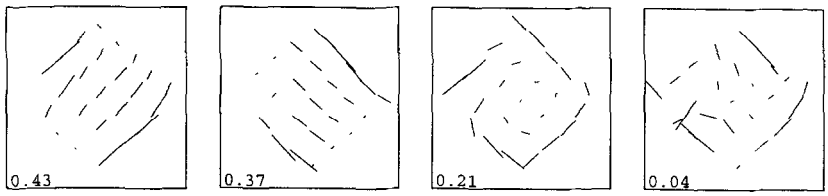


Fig. 8. The first four principal eigenvectors and associated singular values for the roving-eye robot. The first two correspond to pure translation motions and the third corresponds to a pure rotation motion. The robot's motor apparatus can produce the first three effects directly using the motor control vectors shown.

the *amvf* is similar to the eigenvector. An angle near 180 degrees indicates that the *amvf* is similar to the opposite of the eigenvector. If any *amvf*'s match the ith eigenvector to within 45 degrees, then action $\boldsymbol{u}^{i+}$ is defined to be the motor control vector whose *amvf* is most collinear with the ith eigenvector and $\boldsymbol{u}^{i-}$ is defined to be the motor control vector whose *amvf* is most anti-linear.[13] The definitions of control laws (Section 5) assume that the robot's motor apparatus is linear, implying that $\boldsymbol{u}^{i+} = -\boldsymbol{u}^{i-}$. In the case that $\boldsymbol{u}^{i+} \approx -\boldsymbol{u}^{i-}$, they can be approximated by plus and minus $\boldsymbol{u}^i$ respectively, where $\boldsymbol{u}^i \stackrel{\text{def}}{=} \frac{1}{2}(\boldsymbol{u}^{i+} - \boldsymbol{u}^{i-})$. Subsequently, this will be used as the definition of the ith primitive action. The values of $\boldsymbol{u}^i$ are shown in Fig. 7. The analogous results for the roving-eye experiment are shown in Fig. 8.

3.5. Define a new abstract interface

The goal of this step is to define a new interface to the robot that abstracts away the details of the motor apparatus. For each of the robot's degrees of freedom, a new control signal is defined for producing motion along that degree of freedom. Negative values of the control signal move the robot in the opposite direction. For the robot of the example, there are two control signals, one for turning (left and right) and one for advancing (forward and backward). The effect of the control signals is defined by the following equation:

$$\boldsymbol{u} = u_0\boldsymbol{u}^0 + u_1\boldsymbol{u}^1,$$

where u_0 and u_1 (which range from -1 to 1) are the new control signals and $\boldsymbol{u}^0$ and $\boldsymbol{u}^1$ are the primitive actions corresponding to the first two principal eigenvectors.

3.6. Discussion

The learning methods described in this section have also been applied to a simulated synchro-drive robot for which the motor control signals directly specify how fast to

[13] This matching criterion is more restrictive than it appears. In a high-dimensional space such as the space of *amvf*'s, it is highly unlikely that two random vectors will define an angle less than 45 degrees.

turn and advance, respectively. The details of that experiment are given in [31]. The synchro-drive and tank-style robots demonstrate two different motor apparatuses with identical capabilities. The learned abstract interface, since it is grounded in sensory effects rather than motor control signals, is the same for both: it abstracts away the details of the motor apparatus, providing a new set of control signals, one for each of the robot's degrees of freedom.

The learning methods described in this section build on the sensory image structure learned in the previous section. The result is a new abstract interface whose control signals are used in Section 5 to define behaviors for navigation.

4. Local state variables

The result of the agent's learning so far is an abstract interface that includes a model of the robot's sensorimotor apparatus. The model of the *sensory* apparatus is the description of its physical structure represented primarily by the positions of the elements of the learned image feature. The model of the *motor* apparatus is the set of learned primitive actions that tells the agent how many degrees of freedom it has, and how to produce motion in each.

The agent's ultimate goal is to abstract the continuous world of the robot to a *cognitive map* by which the world is viewed as a discrete set of recognizable places with well-defined paths connecting them. The cognitive map gives the learning agent the ability to predict the effects of high-level behaviors and to navigate among a set of recognizable places. Learning the cognitive map requires that the agent learn path-following behaviors for moving the robot through its state space. In order to be useful for prediction, these behaviors must be repeatable in the sense that executing a behavior from a given initial state always moves the robot to the same final state. The following paragraph gives a few examples of such path-following behaviors.

If the learning agent has a feature that gives the distance from the robot to the wall and it knows how to make the robot move while keeping this feature constant, then it can make the robot follow the wall. For a robot with a retina (Section 2.5), a feature as simple as the sum of all of the inputs could be used to define a path-following behavior. Moving while keeping the feature constant would correspond to following a path of constant intensity. A more complex feature based on the retina is a line detector, which could be used as the basis for a line-following behavior. For a robot with a continuous compass giving the robot's heading, a path-following behavior based on the compass's value would move the robot in a constant direction. Finally, consider a robot with an omni-directional photo-sensor responding to a light mounted on the robot and suppose that the robot is in a dark room with white walls. The amount of light detected by the robot's sensor would decrease with distance from the nearest wall. A wall-following behavior could be based on an error signal that was the difference between the light level detected by the sensor and a nominal value (e.g., a value in the middle of the sensor's range of values).

In this section and the next, we describe the following three-step method for learning path-following behaviors:

(1) find a set of features that the learning agent can control, called *local state variables* and use them to define error signals;
(2) learn behaviors for minimizing the error signals; and
(3) learn behaviors that move the robot while keeping the errors near zero.

This section shows how to learn local state variables. Section 5 shows how to use them to define path-following behaviors.

What is required of a local state variable is that it be controllable, i.e., the learning agent must know how its control signals affect it. A feature is controllable if it meets the following definition:

Definition. Let $\hat{\boldsymbol{u}}$ be the vector of control signals u_j. A scalar feature y_i is a *local state variable* if the effect of the control signals on y_i can be approximated locally by

$$\dot{y}_i = \boldsymbol{m}_i \cdot \hat{\boldsymbol{u}} \quad (= \textstyle\sum_j m_{ij} u_j) \tag{1}$$

where $\boldsymbol{m}_i$ is nonzero.

Determining whether a feature is a local state variable while learning the context-dependent value of $\boldsymbol{m}_i$ is the job of the static action model (Section 4.2). The source of information for this step is the set of learned features produced while the learning agent wanders by using its learned primitive actions.

Local state variables are analogous to state variables in the following sense. If x is a state variable, then the constraint $\dot{x} = 0$ reduces the dimensionality of the robot's state space by one. If y is a *local* state variable, then the constraint $\dot{y} = 0$ reduces the dimensionality of the robot's motor control vector space by one.[14] In other words, the constraint reduces the robot's degrees of freedom by one. Since the learning agent does not have access to the robot's state space, it defines local state variables using its knowledge of motor control vector space to which it does have access. They are called *local* state variables because they are not required to be defined everywhere in the robot's state space.

An important feature of local state variables is that they are controllable: feature y_i may be moved to a target value y_i^* using a simple control law. This fact is exploited in the definition of the homing behaviors (Section 5.2). The discovery of local state variables has two components: generating new features (Section 4.1), and testing each feature to see if it satisfies the definition of local state variable (Section 4.2).

4.1. Generating new features

If a sensory system does not directly provide useful features, it may be possible to generate features that are useful. A generate-and-test approach is demonstrated in the following experiment using the tank-style mobile robot in which the agent learns new scalar features that are better candidates for local state variables than are the elements of the raw sense vector.

[14] If $\dot{y} = 0$, then by Eq. (1), $\hat{\mathbf{u}}$ must lie in the subspace perpendicular to vector $\boldsymbol{m}_i$.

4.1.1. A set of feature generators

In this paper, we identify a small set of feature generators that are used to produce new features as candidates for local state variables. Our feature generators are essentially a special case of the functional transformations of [39]. These feature generators are appropriate for a robot with a rich sensorimotor apparatus and are, as we will demonstrate, sufficient for a particular set of environments and sensorimotor apparatuses. We do not claim that this particular set of feature generators is necessary for the robots described in this paper nor that it is sufficient for all robots and environments.

The generated features are based on a set of generic mathematical constructs (e.g., scalars, vectors, matrices, scalar fields, and vector fields) rather than on a human-generated list of salient properties of a robot's environment. The feature generators used for the experiments described in this paper are described below:

- ***splitter*** takes a vector feature of length n and produces n scalar features.
- ***vmin*** and ***vmax*** apply to vector features of length greater than 1. They provide two different ways to reduce a vector feature to a scalar feature.
- ***group*** and ***image*** (described in Section 2) identify useful structure in the sensory apparatus. Group and image features are not scalar features and thus are not able to serve as local state variables, but they do serve as the basis for higher-level features that may turn out to be useful.
- ***lmin*** (local-min) and ***lmax*** (local-max) apply to image features. They produce *focused-image* features. A focused-image feature is a (scalar field, Boolean field) pair where the boolean field is used to mask the scalar field. It can be viewed as an image feature for which each element has an associated weight (0 or 1). The weights focus attention on particular properties of an image, e.g., local minima or maxima.
- ***tracker*** applies to focused-image features and produces *image-element* features (single value-position pairs). From the focused image produced by the ***lmin*** generator, the ***tracker*** generator produces one image-element feature for each local minimum in the image. The tracker implements a form of focus of attention, abstracting away small changes in value and position of an image element in order to produce a feature that tracks an interesting property of the robot's environment such as the minimum distance to a nearby object.
- ***val*** extracts a scalar value feature from an image-element feature.

This set of feature generators has proven successful for the robot with a ring of distance sensors. To handle the "roving eye" robot, we would augment this set with generators for features based on a variety of convolution masks and other two-dimensional image-processing operators. An interesting open problem is to define a general set of feature generators appropriate to learning mobile robots, analogous to the small and general set of functional transformations used by Shen [39] to replicate the performance of AM [22]. We conjecture that a reasonably sized set of feature generators will apply to a broad class of mobile robots and that such a set of feature generators can be discovered by developing solutions for a small subset of that class: initially, each new robot would require one or more new feature generators; eventually the set of feature generators would converge to a generally applicable set.

4.1.2. Generating and testing features

The generate-and-test process of learning potentially useful features executes the following steps in a continuous loop. Initially, there is only one feature, the *raw sensory feature*. This feature is marked as new.

(i) Each generator is applied to each new feature to which it is applicable.

(ii) The features that were new are marked as old, and the features just generated are marked as new.

In generate-and-test approaches to learning, controlling the search through a large space of possibilities is an important concern. Without any constraints, the number of features generated on each iteration of the generate-and-test loop may grow exponentially. There are several ways to constrain a search algorithm. One way is to limit the depth of the search. In the current implementation of the generate-and-test algorithm, it is possible to set a limit on the number of generations of new features that are created. A second way is to limit the breadth of the search. This method is used in genetic algorithms where population size is constrained to a certain number. This method requires a fitness

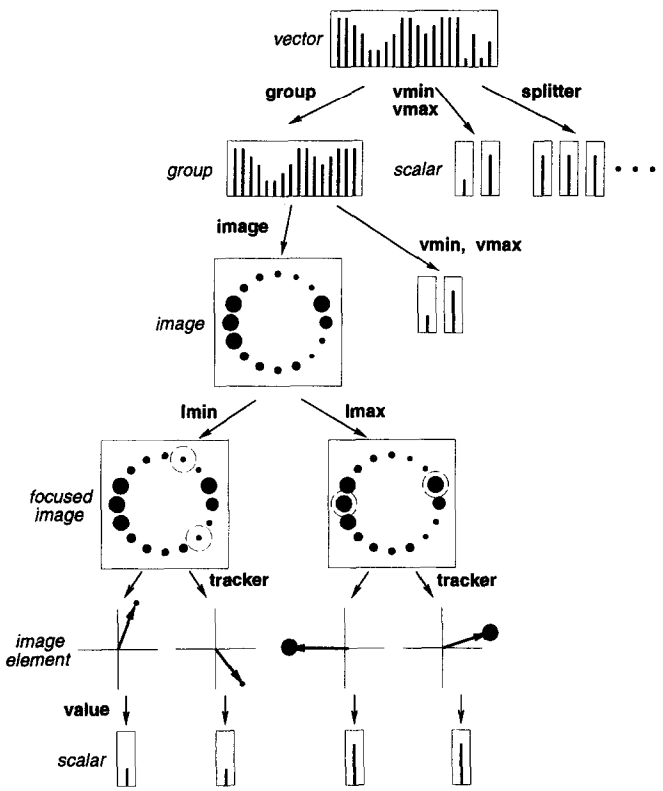


Fig. 9. The complete hierarchy of features and generators in the learning agent's feature-learning process used to produce candidate local state variables. The feature generators are shown in bold face; the feature types are shown in italics.

measure to tell which members of the population are worthy of survival. Such a fitness measure can be defined as a feature tester, though this has not been done here. A third way to constrain a search space is to limit the branching factor. For the feature-learning problem, this is the average number of new features that are generated for each old feature at each step of the generate-and-test process. The branching factor for the feature learning problem is limited in two ways: the total number of generators is kept reasonably small and the number of generators that apply to any given feature is kept small by using strongly typed generators (e.g., the image-feature generator only applies to group features).

4.1.3. An experiment

In the experiments described in this paper, the combinatorial explosion of features has not been an issue. The generators form deep but narrow hierarchies with a tractable set of features. To study this, we devised an experiment in which the agent explores by randomly choosing unit motor control vectors and executing them for one second (10 time steps) each. Fig. 9 shows the complete hierarchy of features and generators for the learning agent's feature-learning process. At the top of the figure is the raw sense vector *s*. We refer to each feature using a name derived from the sequence of feature generators used to produce that feature (where g = group, im = image, tr = tracker). Thus, for example, *s-g-vmin* results from applying the ***vmin*** generator to the feature produced by applying the ***group*** feature generator to the raw sense vector *s*. The features shown in the figure are, from top to bottom and from left to right: *s*, *s-g*, *s-vmin*, *s-vmax*, s_0, s_1, ..., s_{28}, *s-g-im*, *s-g-vmin*, *s-g-vmax*, *s-g-im-lmin*, *s-g-im-lmax*, *s-g-im-lmin-tr*, *s-g-im-lmax-tr*, *s-g-im-lmin-tr-val*, and *s-g-im-lmax-tr-val*. Notice that, depending on the robot's position, there may be multiple *s-g-im-lmin-tr-val* or *s-g-im-lmax-tr-val* features. Each of the generated *scalar* features (the leaves of the tree of generated features) is tested (Section 4.2) to see if it can serve as a local state variable.

4.2. The static action model

The purpose of the static action model (a set of equations of the form given in Eq. (1) of Section 4) is to predict the behavior of each scalar feature. The learning of the static action model for a feature proceeds in three steps. In the first step, the learning agent tries to predict the behavior of the feature without taking into account which primitive action is being used. If it fails, then it tries to predict the behavior of the feature as a function of the action being taken. If this fails for a primitive action, then the agent tries to predict the context-dependent effect of that action on the feature. If a feature is action dependent and is predictable in all contexts, then it can serve as a local state variable.[15] With the information contained in the static action model, it is a simple matter to define homing behaviors for moving the robot so that the local state variable moves toward a target value.

[15] One could use a less constrained definition of local state variable: if a feature is action dependent and predictable in a given context, then it is a local state variable for that context. We have chosen the more constraining definition because it results in more robust control laws.

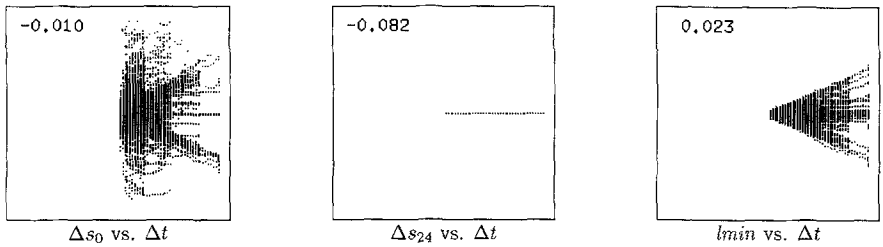


Fig. 10. Plots of Δy_i (vertical axis) versus Δt (horizontal axis), used by the learning agent to try to predict the behavior of feature y_i independently of the motor control vector. Whenever a new motor control vector is used, Δy_i and Δt are reset to 0 (at the center of each plot). From the sets of $(\Delta t, \Delta y_i)$ points, statistics m_i, r_i, and γ_i are computed (see text). The numbers shown are the correlations r_i between Δy_i and Δt. From these statistics the learning agent concludes that features s_0 and *lmin* (short for *s-g-im-lmin-tr-val*) are unpredictable (γ is large and r is small) and that s_{24} is constant ($\gamma < 0.001$).

When trying to predict the effects of actions on features, the learning agent looks for approximately linear relationships between action magnitudes and feature derivatives because the control laws used to define path-following behaviors (Section 5) are based on the assumption that these relationships are approximately linear.

4.2.1. An action-independent model

The first step toward modeling the behavior of a feature y_i is to see if it is possible to predict its behavior independently of the motor control vector being used. The agent explores by repeatedly choosing a primitive action and executing it for one second (ten time steps). It analyzes the behavior of the feature using a device that we call a *correlator*. This produces a set of statistics based on the plot of the feature's value as a function of time (Fig. 10). The coordinate for the horizontal axis is $\Delta t = t - t_0$ where t_0 is the last time the motor control vector changed. The vertical axis gives $\Delta y_i = y_i(t) - y_i(t_0)$.

The statistics are m_i, r_i, and γ_i. The value of m_i is the slope of the line that best fits the set of $(\Delta t, \Delta y_i)$ points. The value of r_i is the correlation between variables Δy_i and Δt. The value of γ_i is the ratio of the standard deviations of Δy_i and Δt. It is a measure of how fast the feature changes as a function of time. A number of properties are defined in terms of these statistics. The feature is *constant* if $\gamma_i < 0.001$. It is *increasing* if $r_i > 0.6$; *decreasing* if $r_i < -0.6$. It is *predictable* if any of these properties holds. Otherwise, it is unpredictable and the learning agent tries to predict the behavior of the feature using an action-dependent model.

For the running example, the features *s-vmin*, *s-vmax*, s_{20} (the broken distance sensor), s_{24} (the battery voltage), and *s-g-vmax* are all diagnosed as *constant* and are thus not suitable for use as local state variables. The rest are candidates for the next step in the learning of the static action model.

4.2.2. An action-dependent model

If the previous step failed to produce a model that predicts the behavior of a feature y_i, then the learning agent uses one correlator for each primitive action to analyze its

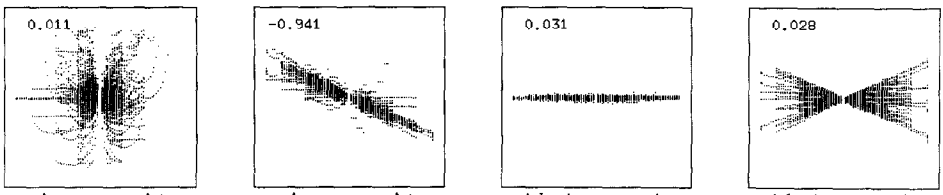


Fig. 11. Plots of Δy_i versus $u_j\Delta t$ for two features and two primitive actions. These are used to see if it is possible to predict the behavior of the feature as a function of the motor control vector. Feature s_0 is unpredictable for action u_0 (r is small and γ is large) but predictable for action u_1 (r is large). Feature *lmin* is constant for action u_0 ($\gamma < 0.001$) but unpredictable for action u_1 (r is small and γ is large).

effect on the feature. In this case, the correlator characterizes the relationship between $u_j\Delta t$ and Δy_i where Δt and Δy_i are defined as before. The agent continues to explore by randomly selecting primitive actions and executing them for a second at a time. It computes the statistics m_{ij} (the slope of the line that best fits the set of $(u_j\Delta t, \Delta y_i)$ points), r_{ij} (the correlation between $u_j\Delta t$ and Δy_i), and γ_{ij} (the ratio of the standard deviations of $u_j\Delta t$ and Δy_i). A feature is labeled *constant* for control signal u_j if $\gamma_{ij} < \gamma_i/4$. The properties *increasing*, *decreasing*, and *predictable* for control signal u_j are defined as before. For each predictable feature-control signal pair, a rule of the form

$$\boxed{\dot{y}_i = m_{ij}u_j}$$

is added to the static action model. If a feature is predictable for all of the primitive actions, then the feature itself is predictable.

For the running example (Fig. 11), all of the distance sensors are found to be unpredictable for primitive action u_0 (rotating). The effect of u_1 (advancing) is to decrease features s_0, s_1, s_2, s_3, and s_{23}; to increase features s_9 through s_{14}. Its effect is unpredictable for features s_4–s_8, s_{15}–s_{19}, s_{21}, and s_{22}. The discrete compass sensors s_{25} through s_{28} are unpredictable for u_0 and constant for u_1. The features *s-g-vmin* and *s-g-im-lmin-tr-val* (a.k.a. *lmin*) are constant for u_0 and unpredictable for u_1. Feature *s-g-im-lmax-tr-val* (a.k.a. *lmax*) is unpredictable for both primitive actions. One might guess that *lmax* would be constant for u_0. In fact, *lmax*, which is only defined when the robot is in a corner, fluctuates too rapidly with small turns to be diagnosed as constant.

4.2.3. A context-dependent model

If u_j has an effect on y_i that is unpredictable, then the learning agent tries to find a partition of sensory space into a discrete set of contexts so that the relationship can be approximated by a linear equation for each context.[16] In general, a *context feature* z_{ij}, for local state variable y_i and control signal u_j, is an integer-valued feature that takes on

[16] This approach is analogous to Drescher's marginal attribution [7].

a finite set of values. This set defines a partition of the robot's state space into a finite set of contexts defined by the predicates $z_{ij} = k$. One way to define a context feature is to first choose a feature x and divide its range of values into a finite set of intervals, $\{I_k\}$, where each interval defines its own context. The context feature is then defined by $z_{ij} = k$ iff $x \in I_k$. Using feature x to define a set of contexts is appropriate if the value of x is a good predictor of the effect of the control signal u_j on the feature y_i. To test the hypothesis that x is a good predictor for the effect of u_j on y_i, a correlator can be used to determine u_j's effect on y_i for each context defined by the predicate $z_{ij} = k$.

Testing each of a large set of features to see if they improve the predictability of a control signal's effect is expensive. Heuristics can be used to guide the search for relevant features to use in defining contexts. For example, it makes sense to first look at features that are closely related to the feature being analyzed, in the sense that they are close together in the tree of features produced by the generate-and-test process.

Currently, only one such heuristic is implemented: if a feature is based on the value of a element of an image, then use the position of that element in the image to define the context. Since there is a discrete set of possible positions for an image-element feature, it is trivial to break the space of possible positions into a discrete set of contexts. For example, in the case of the *lmin* and *lmax* features, there are 23 possible positions and these can be used to break sensory space into a partition of 23 contexts each defined by the predicate $z_{ij} = k$ where z_{ij} is an integer feature whose value is between 0 and 22 and identifies the position associated with the local minimum or maximum.

For each context $z_{ij} = k$, a correlator is used to try to predict the effect of u_j on y_i given that the robot is in that context. The agent continues to explore randomly while computing the statistics m_{ijk}, r_{ijk}, and γ_{ijk}. The properties *constant*, *increasing*, *decreasing*, and *predictable* are defined as before. For each predictable context, a rule of the form

$$\boxed{\dot{y}_i = m_{ijk} u_j, \quad \text{if } z_{ij} = k}$$

is added to the static action model. If m_{ijk} is 0, then the predicate $z_{ij} = k$ defines a "constant context" (which is useful for defining path-following behaviors). If the primitive action's effect on the feature is predictable for every context, then the feature is predictable for that action.

For the running example, the only features with associated context features are *lmin* and *lmax*.

- *lmin* is already predictable (constant) for control signal u_0.
- The effect of u_1 on *lmin* is predictable for every context. Its effect is to decrease *lmin* for contexts 0–5 and 19–22, and to increase it for contexts 7–17. For contexts 6 and 18 (in which the robot's heading is parallel to the wall), *lmin* is constant (see Fig. 12).
- The effect of u_0 on *lmax* is unpredictable for almost every context.
- The effect of u_1 is to decrease *lmax* for contexts 0–5 and 20–22 and to increase it for contexts 8–16. The effect is unpredictable for contexts 6, 7, 17, and 18.

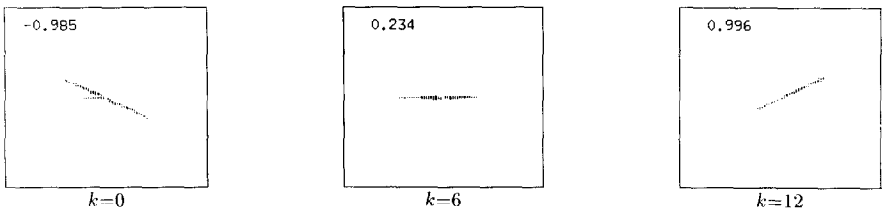


Fig. 12. Example plots of Δy_i versus $u_1 \Delta t$ for the *s-g-im-lmin-tr-val* feature for three different contexts. These are used to see if it is possible to predict the behavior of the feature as a function of the motor control vector and the current context. For action u_1, feature *lmin* is *decreasing*, *constant*, and *increasing* for contexts 0, 6, and 12, respectively.

At this point the only feature that is both predictable and action-dependent (and is thus a local state variable) is *lmin*. Its behavior can be modeled by the equation $\dot{y}_i = m_{i1k} u_1$ where k is the current value of the context feature z_{ij} that represents the location of the local minimum in the image feature. The feature *lmin* was produced by the *tracker* generator. This generator actually produces multiple *lmin* features, one for each local minimum in the input image feature. The number of local state variables depends on the robot's location. There are two local state variables in the neighborhood of a corner, three in the neighborhood of a T-intersection, but just one if only a single wall is within range.

5. Learning control laws

The goal of this step is to learn a suitable set of *homing* and *path-following behaviors* using the results of the preceding sections, specifically, the set of local state variables and the set of primitive actions. Recall that for the robot of the running experiment, the local state variables are the *lmin* features, the only features identified as controllable by the learning agent. The sources of information for this step are the learned static action model (Section 4.2) and dynamic action model (Section 5.3.2).

A *behavior*, as the term is used in this paper, is an object with four components, called *output*, *app*, *done*, and *init*. The *output* component is a function that returns a vector of motor control signals. The *app* component is a scalar function whose value indicates whether the behavior is currently applicable. The value of this feature may be 0 (indicating that the behavior is not applicable) or 1 (indicating that the behavior is applicable) or some number in between (indicating a certainty less than 100% that the behavior is applicable). The *done* signal is a Boolean function that tells when the behavior has finished. The *init* signal is an input signal that tells the behavior to initialize itself (in case it has internal state information that needs to be reset).

Path-following behaviors are learned in three steps:

(1) continuous error signals are defined;

(2) behaviors are learned for minimizing the error signals;

(3) behaviors are learned for moving while keeping the error signals near zero.

5.1. Defining error signals for control laws

The learning agent's approach to exploration, mapping, and navigation uses path-following behaviors in which the robot moves while maintaining an error signal near zero. An example of a path-following behavior based on an error signal involves a person walking down a corridor. The error signal is $e = (y^* - y)$ where y is the distance from the person to the right side of the corridor (left in Britain) and y^* is a constant that depends on the person, his mood, and the number of other people in the corridor. The error signal is used in a control law for moving along the corridor. If the error is positive, the person moves to the left (away from the wall) while walking; if it is negative, he moves to the right. The control law is efficient and repeatable: by using the control law, the person follows an efficient (i.e., straight) path from one end of the corridor to the other, and each time the person follows the path, he ends up in the same place.

In this example, y is a local state variable. The agent's approach to defining path-following behaviors is to first define error signals of the form $e = y^* - y$ for each local state variable y.[17]

5.2. Learning homing behaviors

The purpose of a *homing behavior* is to move an error signal toward zero so that path-following behaviors based on that error signal will become applicable. While it would be possible to use reinforcement-learning methods to learn a homing behavior given an error signal [24,33], most of the relevant learning has already been done. The homing behavior can be defined as an instance of the generic, domain-independent control law in Fig. 13, drawing on the knowledge in the static action model.

For each local state variable y_i and control signal u_j, a homing behavior is defined for reducing the error $e = y_i^* - y_i$. It is applicable in every context $z_{ij} = k$ for which the static action model includes a rule of the form $\dot{y}_i = m_{ijk}u_j$ where m_{ijk} is nonzero. It is done when the error is close to zero. Its output is given by a simple control law. The definition is based on the partition of sensory space used by the static action model to characterize the effects of u_j on y_i. This partition is described by the set of contexts $\{k\}$. The components of the homing behavior (*app*, *output*, and *done*) are defined for each possible context k (Fig. 13). A homing behavior that the agent learns for the mobile robot is illustrated in Fig. 14.

5.3. Learning path-following behaviors

The previous section presented a method for learning homing behaviors that minimize a given error signal. In this section, a method is presented for moving while minimizing

[17] Choosing an optimal target value y^* for a feature y is beyond the scope of this paper. The implemented learning agent chooses a value equal to half the feature's maximum value.

For each context $z_{ij} = k$,

$$\begin{aligned} app(k) &= \max\{0,\ 2|r_{ijk}| - 1\} \\ output(k) &= u_{ijk}\,\mathbf{u}^j \\ done &\equiv \frac{|y_i^* - y_i|}{y_i^*} < 0.1 \end{aligned}$$

where

$$\begin{aligned} u_{ijk} &= \frac{2\zeta\omega}{m_{ijk}}\, e_i + \frac{\omega^2}{m_{ijk}} \int e_i\, dt \\ e_i &= y_i^* - y_i. \end{aligned}$$

Fig. 13. A homing behavior is defined for each local state variable y_i and for each primitive action $\boldsymbol{u}^j$ to achieve the goal $y_i = y_i^*$. The applicability and output are defined as functions of the current context as defined by the context feature z_{ij}. The applicability has a maximum value of 1.0 if the correlation r_{ijk} between u_j and $\dot{y}_i$ has a magnitude of 1.0 and a minimum value of zero if the correlation has a magnitude of 0.5 or less. The output is given by a proportional-integral (PI) control law with parameters $\zeta = 1.0$, $\omega = 0.05$ (see [21]) that minimizes the difference between y_i and y_i^*. The behavior is done when this difference is close to zero. The *init* function resets the value of the integral of the error to zero.

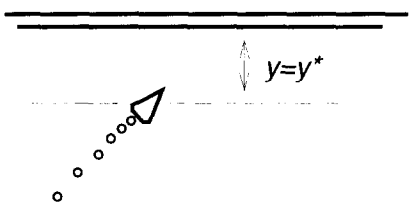


Fig. 14. An example of a homing behavior for the mobile robot with distance sensors and tank-style motor apparatus. The agent's static action model predicts that in this context the second primitive action $\boldsymbol{u}^1$ decreases the value of local state variable y. This information is used in the definition of a homing behavior that is (a) applicable in this context, (b) uses primitive action $\boldsymbol{u}^1$ to move the robot so as to minimize the error $e = y^* - y$, and (c) is done when $y \approx y^*$.

the error signal. The result is a path-following behavior. Learning a path-following behavior involves two steps:

(1) learning how to move in the general direction that keeps the error near zero and
(2) learning the necessary feedback for error correction to avoid straying off the path defined by the minimum of the error signal.

The learning agent uses its static action model to determine which primitive action to use to provide motion along a path. It learns a *dynamic action model* to tell how to use the remaining primitive actions to provide error correction.

5.3.1. Learning open-loop path-following behaviors

The static action model does not give the agent enough information to define *closed-loop* path-following behaviors with error correction, but it does give the agent enough information to define *open-loop* path-following behaviors. [18] An open-loop path-following behavior lacks error correction but is useful for learning the dynamic action model, which is in turn useful for defining path-following behaviors with error correction. Recall that the static action model identifies constant contexts $z_{ij} = k$ in which primitive action $\boldsymbol{u}^j$ has no effect on local state variable y_i.

For each local state variable y_i and primitive action $\boldsymbol{u}^j$, for each constant context $z_{ij} = k$, two open-loop behaviors are defined, one for each direction of motion. The behaviors' outputs are given by

$$\boldsymbol{u} = \boldsymbol{u}^\beta + \sum_{\delta \neq j} u_\delta \boldsymbol{u}^\delta,$$

where $\boldsymbol{u}^\beta = \pm \boldsymbol{u}^j$ and $|u_\delta| \ll 1$. The u_δ components are used in learning the dynamic action model. The purpose of an open-loop path-following behavior is to allow the agent to learn the effects of the *orthogonal* control signals on the feature while motor control vector $\boldsymbol{u}^\beta$ is used. [19] With this knowledge, it is possible to use the other control signals for error correction. The definition of open-loop path-following behaviors is summarized in Fig. 15. A behavior is done when the robot strays too far off the path or when a new behavior becomes applicable indicating that the agent has a choice to make: to continue the current behavior or start a new one.

For the mobile robot of the running example, there is an open-loop path-following behavior based on $\boldsymbol{u}^0$ (for turning) for each local state variable y_i (see Fig. 16(a)). It is applicable whenever $y_i = y_i^*$ since, according to the static action model, turning has no effect on y_i. There is also an open-loop path-following behavior based on $\boldsymbol{u}^1$ (for advancing) for each feature y_i (see Fig. 16(b)). It is applicable when the robot is facing parallel to the object being detected by y_i (that is, when context feature z_{ij} has value 6 or 18). Fig. 20(b) shows a trace of the behavior of the robot that results as the learning agent uses its learned open-loop path-following behaviors to explore the robot's environment.

5.3.2. The dynamic action model

The *static* action model predicts the context-dependent effects of a control signal on the local state variables. The *dynamic* action model predicts the context-dependent effects of control signals on the local state variables while an open-loop path-following behavior is being executed.

The dynamic action model tells, for each open-loop path-following behavior, the effect of each orthogonal action (each primitive action other than the path-following behavior's base action), on the local state variable that is used in the definition of the path-following

[18] In a closed-loop control law, an error signal is used as feedback to determine a motor control vector that minimizes that error.

[19] The primitive actions are orthogonal to each other in the sense that their *amvf*'s are orthogonal to each other (see Section 3.3).

$$
\begin{aligned}
app &\equiv \frac{|y_i^* - y_i|}{y_i^*} < 0.1 \wedge z_{ij} = k \\
output &= \mathbf{u}^\beta + \sum_{\delta \neq j} u_\delta \mathbf{u}^\delta \\
done &\equiv \frac{|y_i^* - y_i|}{y_i^*} > 0.4 \\
&\quad \vee \text{(a new behavior becomes applicable)}
\end{aligned}
$$

Fig. 15. An open-loop path-following behavior is defined for each local state variable y_i, for each primitive action (or opposite) $\boldsymbol{u}^\beta$, and for each constant context $z_{ij} = k$. The predicate $z_{ij} = k$ defines a constant context if it implies that $\boldsymbol{u}^\beta$ maintains y_i constant according to the static action model. The behavior is applicable when the error signal $y_i^* - y_i$ is small. The output has two components: a base motor control vector and a small orthogonal component. During learning of the dynamic action model, the orthogonal component changes every 3 seconds. Only one of the u_δ's is nonzero at a time. The behavior is done when the error signal is too large or a new behavior becomes applicable.

Fig. 16. Two examples of open-loop path-following behaviors. (a) A behavior based on $\boldsymbol{u}^0$ (for turning) and constraint $y_i = y_i^*$ is applicable whenever $y_i = y_i^*$ since $\boldsymbol{u}^0$ never changes the value of y_i. (b) A behavior based on primitive action $\boldsymbol{u}^1$ (advancing) and constraint $y_i = y_i^*$ is applicable whenever $y_i = y_i^*$ and the robot's heading is parallel to the wall on its left (i.e., $z_{ij} = 18$) since in this context $\boldsymbol{u}^1$ keeps the error $e = y_i^* - y_i$ near zero.

behavior's error signal. To learn the dynamic action model, an exploration behavior is used that randomly chooses applicable homing and open-loop path-following behaviors. A behavior runs until it is no longer applicable, or a new path-following behavior becomes applicable. Linear regression is used to learn the relationships between the orthogonal actions u_δ and the features y_i in the context of running the open-loop path-following behavior based on feature y_i, motor control vector $\boldsymbol{u}^\beta = \pm\boldsymbol{u}^j$, and context $z_{ij} = k$. While it is running, linear regressors test the hypotheses $\dot{y}_i = m_{ijk\delta 1} u_\delta$ and $\ddot{y}_i = m_{ijk\delta 2} u_\delta$ by computing the correlations $r_{ijk\delta n}$ between u_δ and $y_i^{(n)}$. If $r_{ijk\delta 1} > r_{ijk\delta 2}$ and $|r_{ijk\delta 1}| > 0.6$, then the rule

$$
\dot{y}_i = m_{ijk\delta 1} u_\delta, \quad \text{if} \quad z_{ij} = k \wedge \boldsymbol{u} = \pm\boldsymbol{u}^j + u_\delta \boldsymbol{u}^\delta
$$

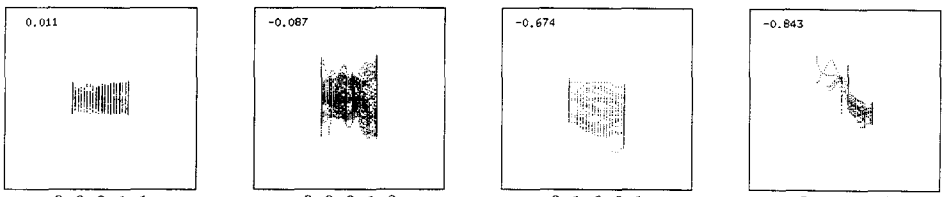


Fig. 17. Plots illustrating the relationships measured by the linear regressors used in learning the dynamic action model. The labels under the plots give the values of i, j, k, δ, and n, where n is the number of the derivative of y_i being tested. The first two plots show the effect of $\boldsymbol{u}^1$ (advancing) on $\dot{y}_0$ and $\ddot{y}_0$ respectively while an open-loop path-following behavior based on $\boldsymbol{u}^0$ is executed. Here y_0 is one of the local state variables (instances of *lmin*) produced by the tracker generator. The second two plots show the effect of $\boldsymbol{u}^0$ (turning) on $\dot{y}_0$ and $\ddot{y}_0$ respectively while an open-loop path-following behavior based on $\boldsymbol{u}^1$ is executed in context $z_{0,1} = 6$. This is the context in which the robot heading parallel to a wall on its right.

is added to the dynamic action model. Otherwise, if $|r_{ijk\delta 2}| > 0.6$, then the rule

$$\boxed{\ddot{y}_i = m_{ijk\delta 2} u_\delta \quad \text{if} \quad z_{ij} = k \wedge \boldsymbol{u} = \pm \boldsymbol{u}^j + u_\delta \boldsymbol{u}^\delta}$$

is added to the dynamic action model. Otherwise, the relationship between u_δ and y_i is either zero or unpredictable.[20]

Suppose that the mobile robot of the running experiment has a wall to its left and that its heading is parallel to the wall (Fig. 16(b)). In this context, primitive action $\boldsymbol{u}^1$ (advancing) maintains the distance to the wall, y_i, constant ($m_{ijk} = 0$). Therefore, the open-loop path-following behavior based on $\boldsymbol{u}^1$ and y_i is applicable. While executing this behavior, the effects of other control signals (i.e., u_0) can be diagnosed. In this example, u_0 affects the second derivative of the feature: $\ddot{y}_i = m_{i1k0,2} u_0$. This is because turning changes the robot's direction of motion relative to the wall and this direction determines how fast the robot moves toward or away from the wall as it advances. Examples of the linear regressors used to learn the dynamic action model for the robot of the running example are illustrated in Fig. 17.[21] According to the dynamic action model, $\boldsymbol{u}^0$ has a predictable effect on y_i while any of the open-loop behaviors based on $\boldsymbol{u}^1$ is executing. For the open-loop path-following behaviors based on $\boldsymbol{u}^0$, the effect of $\boldsymbol{u}^1$ on y_i is unpredictable.

5.3.3. *Learning closed-loop path-following behaviors*

The final step in learning path-following behaviors is to add error correction to the open-loop path-following behaviors in order to define closed-loop path-following

[20] For the dynamic action model, it is necessary to consider both first and second derivatives of the features. Informally, this is because $\boldsymbol{u}^\delta$ may affect the derivative of m_{ij} in the equation $\dot{y}_i = m_{ij} u_j$, that is, $\dot{m}_{ij} = m_{j\delta} u_\delta$. Together, these give $\ddot{y}_i = \dot{m}_{ij} u_j = m_{j\delta} u_\delta u_j = m_{ij\delta 2} u_\delta$, using the product rule and the fact that u_j is constant for a path-following behavior.

[21] The linear regressors operate on filtered versions of y_i and u_j to remove noise that would otherwise hide the relationship between the signals. The signals are filtered using a moving average taken over several seconds.

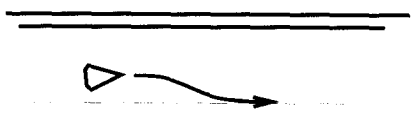


Fig. 18. Defining closed-loop path-following behaviors. The learning agent uses the dynamic action model to add error correction to an open-loop path-following behavior in order to obtain a closed-loop path-following behavior. In this example, a small turning motion is used to keep the robot on the path as it advances.

behaviors. A *closed-loop* behavior is one that receives feedback from the environment in the form of an error signal which it uses to modify its motor control signals so as to minimize the error. Consider again the case where the robot is facing parallel to a wall on its left. In this context, the learning agent knows, because of its static action model, that primitive action $\boldsymbol{u}^1$ leaves feature y_i (the distance to the wall) approximately constant. Moreover, the agent knows, because of its dynamic action model, how control signal u_0 (turning) affects y_i while $\boldsymbol{u}^1$ is being taken. Together, this information is sufficient to define a closed-loop path-following behavior that robustly moves the robot along the wall. If y_i goes below its target value (i.e., if the robot gets too close to the wall), then the agent knows to increase the value of u_0 (i.e., to turn right as shown in Fig. 18). Because of the error correction implemented using control signal u_0, the path-following behavior is robust in the face of noise in the sensorimotor apparatus, small perturbations in the shape of the wall, and even inaccuracies in the action models themselves.

A closed-loop path-following behavior is defined using the generic template in Fig. 19 for each constraint $\boldsymbol{y} = \boldsymbol{y}^*$, for each primitive action or opposite $\boldsymbol{u}^\beta = \pm\boldsymbol{u}^j$, and for each constant context $\boldsymbol{z} = \boldsymbol{k}$. The predicate $\boldsymbol{z} = \boldsymbol{k}$ (where $\boldsymbol{z}$ is a vector of context features z_{ij} and $\boldsymbol{k}$ is a vector of context values k_i) defines a constant context if for each $z_{ij} \in \boldsymbol{z}$ and $k_i \in \boldsymbol{k}$, $z_{ij} = k_i$ defines a constant context for y_i and $\boldsymbol{u}^j$ according to the static action model. The variable $r_{ijk\delta n}$ is the correlation between u_δ and $y_i^{(n)}$ while motor control vector $\boldsymbol{u}^\beta$ is used in context k. The behavior is applicable when all of the elements of $\boldsymbol{y}$ are near their target values (i.e., $\boldsymbol{y} \approx \boldsymbol{y}^*$) and when $\boldsymbol{z} = \boldsymbol{k}$ indicating that the static action model predicts that motor control vector $\boldsymbol{u}^\beta$ keeps the error vector $\boldsymbol{y}^* - \boldsymbol{y}$ near zero. The behavior is done when a new path-following behavior becomes applicable indicating that the agent now has a choice—to continue the current path-following behavior or to choose a new one.

For the example robot, the set of path-following behaviors contains behaviors for turning in place as well as for following walls. For the behavior based on $\boldsymbol{u}^1$ (advancing), the effect of the orthogonal primitive action $\boldsymbol{u}^0$ on the local state variables is predictable and thus it can be used for error correction. For the behaviors based on $\boldsymbol{u}^0$ (turning), no error correction is used since the effect of $\boldsymbol{u}^1$ is unpredictable.[22]

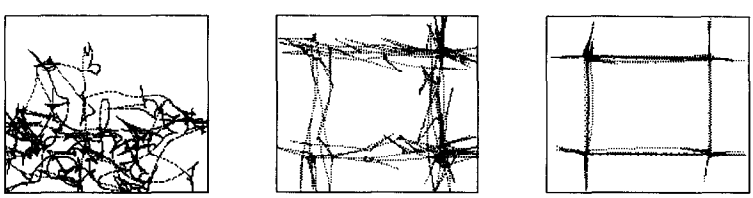
[22] The implemented learning agent learns a context-dependent *static* action model. An extension would be to learn a context-dependent *dynamic* action model for each open-loop path-following behavior. In this way the effect of $\boldsymbol{u}^1$ could become predictable and the action could be used for error correction in a context-dependent control law.

$$
\begin{aligned}
app &\equiv \forall y_i \in \mathbf{y} : \left(\frac{|y_i^* - y_i|}{y_i^*} < 0.1 \right) \wedge \forall z_{ij} \in \mathbf{z} : (z_{ij} = k_i) \\
output &= \mathbf{u}^\beta + \sum_{\delta \neq j} u_\delta \, \mathbf{u}^\delta \\
done &= \exists y_i \in \mathbf{y} : \left(\frac{|y_i^* - y_i|}{y_i^*} > 0.4 \right) \\
&\quad \vee \text{(a new behavior becomes applicable)}
\end{aligned}
$$

where

$$
\begin{aligned}
u_\delta &= \sum_{y_i \in \mathbf{y}} u_{\delta i} \\
u_{\delta i} &= \frac{2\zeta\omega}{m_{ijk\delta 1}} e_i + \frac{\omega^2}{m_{ijk\delta 1}} \int e_i \, dt \qquad \text{if } |r_{ijk\delta 1}| \geq |r_{ijk\delta 2}|, 0.6 \\
u_{\delta i} &= \frac{\omega^2}{m_{ijk\delta 2}} e_i + \frac{2\zeta\omega}{m_{ijk\delta 2}} \dot{e}_i \qquad \text{if } |r_{ijk\delta 2}| > |r_{ijk\delta 1}|, 0.6 \\
u_{\delta i} &= 0, \quad \text{otherwise} \\
e_i &= y_i^* - y_i .
\end{aligned}
$$

Fig. 19. Definition of a closed-loop path-following behavior. Here, $\mathbf{y}$ is a vector of local state variables y_i; $\mathbf{y}^*$ is the corresponding vector of target values; $\mathbf{u}^\beta = \pm \mathbf{u}^j$ is one of the primitive actions or their opposites; $\mathbf{z}$ is a vector of context features z_i, one for each local state variable y_i; and $\mathbf{k}$ is the corresponding vector of context values k_i. The equation $\mathbf{z} = \mathbf{k}$ defines a context in which $\mathbf{u}^\beta$ maintains $\mathbf{y}$ constant according to the static action model. The values of $m_{ijk\delta n}$ and $r_{ijk\delta n}$ are taken from the dynamic action model. Simple PI and PD (proportional-derivative) controllers are used (see [21]) depending on whether the primary effect of $\mathbf{u}^\delta$ is on $\dot{y}_i$ or $\ddot{y}_i$, respectively. Again, $\zeta = 1.0$, $\omega = 0.05$.



(a) (b) (c)

Fig. 20. Exploring a simple world at three levels of competence. (a) The robot wanders randomly while learning a model of its sensorimotor apparatus. (b) The robot explores by randomly choosing applicable homing and open-loop path-following behaviors based on the static action model while learning the dynamic action model. (c) The robot explores by randomly choosing applicable homing and closed-loop path-following behaviors based on the dynamic action model.

Fig. 20 shows the behavior of the robot at three different stages as the agent learns the set of path-following behaviors. Section 6 demonstrates the learning of the set of homing and path-following behaviors for a rectangular environment containing a number of obstacles and a T-shaped environment. In Section 7, the path-following behaviors learned in this section are used as the basis for an exploration and mapping strategy that allows the agent to develop a discrete abstraction of the robot's continuous world.

6. Additional experiments

The previous sections have demonstrated a set of learning methods that a learning agent may use to learn the sensorimotor and control levels of the spatial semantic hierarchy. The purpose of this section is to describe a number of experiments (in addition to those described in the previous sections) that demonstrate the generality and some limitations of the methods for learning the sensorimotor and control levels.

The learning methods are first demonstrated for the mobile robot in a cluttered room. Then, to demonstrate that the learned model of the sensorimotor apparatus applies beyond the particular environment in which the model was learned, the learning agent is transferred to a new, T-shaped environment after its control-level learning has been erased. Here it re-learns the control level and demonstrates a set of learned path-following behaviors. Finally, to demonstrate that the learning of the control level applies beyond the particular environment in which it was learned, the learning agent is transferred to an empty room where it again demonstrates the learned path-following behaviors.

Sections 6.4 and 6.5 describe two experiments in which various of the learning methods failed and explain why they failed. Section 6.4 describes an experiment in which the image-feature generator fails to produce a ring-shaped representation of the structure of the ring of distance sensors. Section 6.5 describes an experiment in which the learning agent fails to discover any local state variables. Section 6.6 summarizes the ways in which the learning methods can fail. Finally, Section 6.7 identifies a number of ways in which the learning methods can be improved.

6.1. A cluttered room

The environment used in this experiment is a rectangular room with dimensions six meters by four meters, containing four rectangular obstacles (Fig. 23). The simulated mobile robot used throughout this section is the same as that described in Section 2.1.

6.1.1. Modeling the sensory apparatus

The first step in modeling the robot's sensory apparatus is to apply the group-feature generator. The learning agent computes distance metrics d_1 and d_2 after wandering for 20 minutes. Their values are qualitatively similar to those shown in Fig. 2. The group-

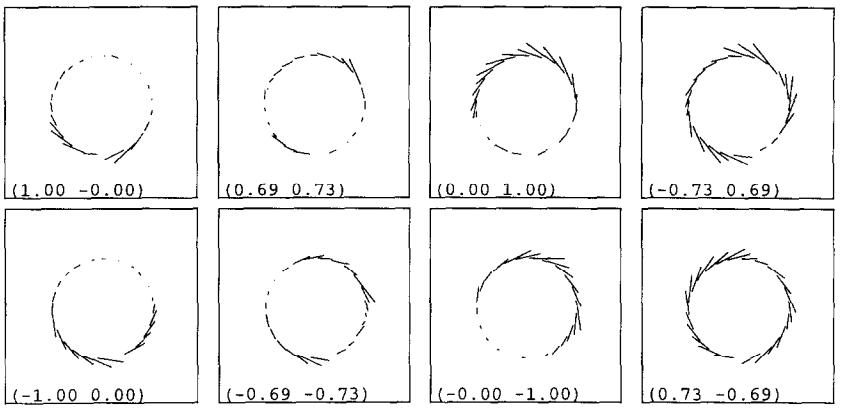


Fig. 21. Example *amvf*'s and associated motor control vectors for the cluttered-room experiment.

feature generator identifies the same groups as those in Section 2.3. The second step in modeling the robot's sensory apparatus is to apply the image-feature generator. The learning agent computes distance metric d_1 for the group of 23 related sensors after wandering for 40 minutes.[23] The outputs of the metric scaling and relaxation algorithm are qualitatively similar to those shown in Fig. 3.

6.1.2. *Modeling the motor apparatus*

The first step in modeling the robot's motor apparatus is to characterize the effects of each of a large set of representative motor control vectors. In this experiment, 100 representative motor control vectors of unit magnitude are chosen. Eight example *amvf*'s and their associated motor control vectors are shown in Fig. 21. These were obtained while the learning agent wandered for 60 minutes, repeatedly choosing a representative motor control vector at random and executing it for one second (ten time steps). The first four eigenvectors produced by principal component analysis from are shown in Fig. 22. The first corresponds to a pure rotation motion and the second corresponds to a pure translation motion. The two motor control vectors identified as primitive actions are shown under the two principal eigenvectors. None of the other eigenvectors match any of the *amvf*'s. Notice that the primitive actions identified here very closely match those shown in Fig. 7. The result of the analysis is that the robot's motor apparatus has two degrees of freedom and that the above primitive actions can be used to produce motion for each degree of freedom.

[23] We use fairly long wandering periods so that the robot adequately explores its state space. For the uncluttered, rectangular room, shorter periods were used because the group- and image-feature generators quickly converged. See Section 6.7 for a discussion of improved feature generators that automatically detect when they have converged.

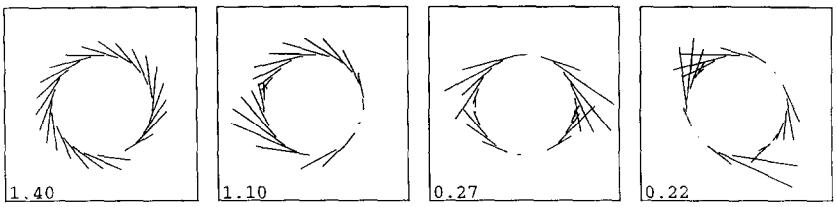


Fig. 22. The first four eigenvectors and the primitive actions for the cluttered-room experiment.

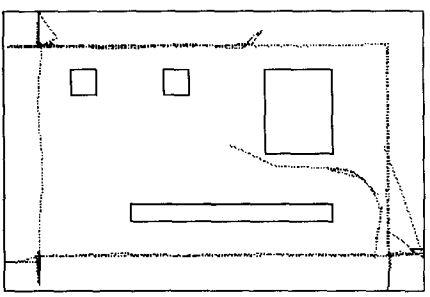


Fig. 23. The path taken by the robot while the learning agent randomly selects learned homing and path-following behaviors. When no other behavior is applicable, it randomly selects primitive actions. The robot is initially in the middle. The learning agent begins by using a homing behavior to move toward the long obstacle and then using a path-following behavior to move along it. The diagonal trajectories in the corners are the results of homing behaviors that move the robot from a wall to a path.

6.1.3. Learning behaviors

As described in Section 4, the learning agent identifies the set of local-minimum features (*s-g-im-lmin-tr-val*) as local state variables (they are the only generated features that are identified as both action dependent and predictable).

The learned static and dynamic action models are qualitatively similar to those learned in Sections 4 and 5. In this experiment, the learning agent again discovers that it can use the first (turning) primitive action for error correction while executing an open-loop path-following behavior based on the second (advancing) primitive action. It uses this information to define closed-loop path-following behaviors. Fig. 23 shows a trace of a random exploration behavior demonstrating the learning agent's learned behaviors.

6.2. Re-learning the behaviors in a T-shaped room

For this experiment, the robot was moved from the cluttered room to a T-shaped environment and the learning agent's control-level learning (i.e., static action model, dynamic action model, and learned behaviors) was erased. Its task was to begin with an intact model of the robot's sensorimotor apparatus and learn an appropriate set of homing

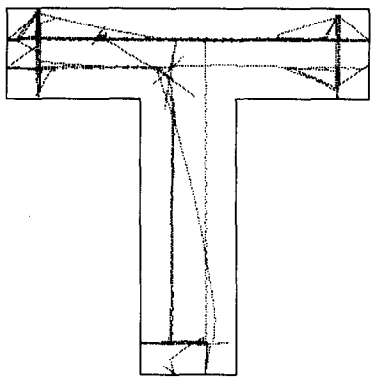


Fig. 24. Re-learning behaviors in a T-shaped room. Path-following behaviors based on the advancing primitive action produce the straight-line trajectories that are parallel to the walls. Path-following behaviors based on the turning primitive action leave the robot in the same place while changing the robot's heading. The homing behaviors based on the advancing action produce most of the rest of the trajectories shown in the picture. A few of the trajectories are produced by a random wandering behavior that is used whenever none of the other behaviors are applicable. (The learning agent selects its behaviors stochastically and occasionally selects a random wandering behavior even when other behaviors are applicable.)

and path-following behaviors. The environment used in this experiment consists of two corridors connected to form a T. The corridor forming the top of the T is 6 meters long and 1.5 meters wide. The shorter corridor is 4.5 meters long and 1.5 meters wide.

The learning agent successfully learns the open-loop and closed-loop path-following behaviors. Fig. 24 shows a trace of a random exploration behavior demonstrating the learned behaviors. This experiment demonstrates that both the set of features and the model of the sensorimotor apparatus that were learned in the first environment are applicable in the second environment.

6.3. Using the behaviors in an empty room

For this experiment, the robot was moved from the T-shaped environment to an empty rectangular room (of dimensions 6 meters by 4 meters). The learning agent's model of the robot's sensorimotor apparatus and its set of learned behaviors were left intact. Fig. 25 shows a random exploration behavior demonstrating that the learned behaviors do not apply only to the environment in which they were learned.

6.4. A long and narrow room

This experiment demonstrates an instance in which the image-feature generator does not produce a ring-shaped representation of the structure of the ring of distance sensors. The environment used in this experiment is a long, narrow, rectangular room. The room is six meters long and one half meter wide. This environment was designed to confuse

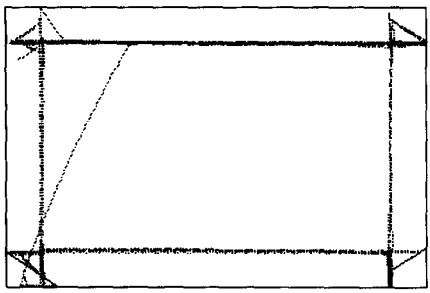


Fig. 25. Using the learned homing and path-following behaviors in an empty room.

the image-feature generator. Since the room is so narrow, the values of distance sensors on opposite sides of the ring are often similar: If a sensor detects the distance to one of the long walls of the room, then the sensor opposite to it detects the distance to the wall on the opposite side of the room. Both sensors produce a small value (less than 0.5). On the other hand, if a sensor returns a large value, then there is a good chance that the sensor opposite to it will also return a large value.

If opposite sensors return similar values, on average, then the image-feature generator will place them close together in the image feature. It is unlikely, in this case, that the image feature will capture the ring structure of the array of distance sensors.

6.4.1. Modeling the sensory apparatus

The result of the group-feature generator is the same as before: The distance sensors are all grouped together. The outputs of the metric scaling and relaxation algorithm are shown in Fig. 26. According to the metric-scaling scree diagram on the left, the structure of the array of sensors is best captured by a four-dimensional representation—there is no arrangement of points in fewer than four dimensions for which the distance between any two points approximates the distance between corresponding sensors as measured by distance metric d_1. The middle figure below shows the projection onto two dimensions of the set of points generated by the metric-scaling algorithm. The figure on the right shows the results of the relaxation algorithm.[24] Notice that sensors that are adjacent in the ring of sensors are close together in the image.[25]

6.4.2. Modeling the motor apparatus

The first four principal eigenvectors for the space of average motion vector fields are shown in Fig. 27. The method actually identifies the turning motor control vector

[24] The metric-scaling algorithm, the relaxation algorithm, and the definition of the image and motion features can all handle images of arbitrary dimension. However, in the current implementation, we have constrained the image feature to be two-dimensional. A goal for future research is to remove this artificial constraint and test the methods on sensory arrays that are genuinely three-dimensional.

[25] Though the results are not shown here, we have also run the relaxation algorithm for this distance metric in three dimensions. In that case the resulting pattern of sensors resembles the pattern of stitching on a baseball.

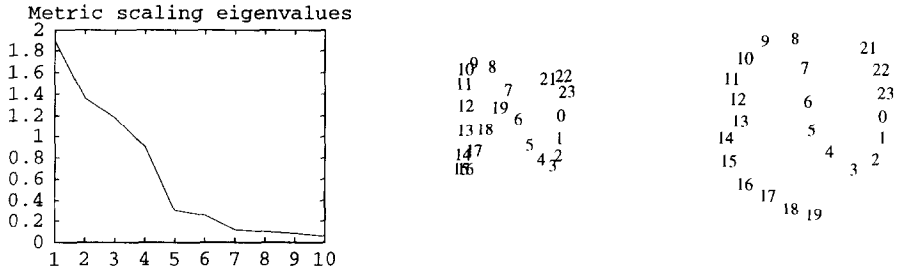
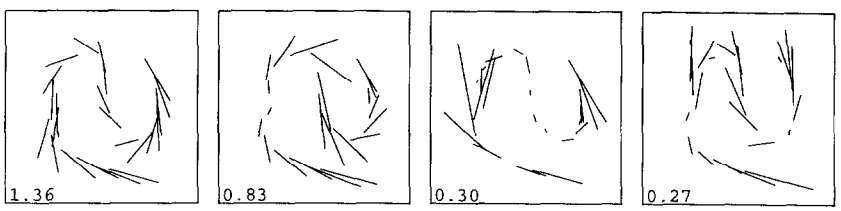


Fig. 26. The outputs of the metric-scaling and relaxation algorithms for the narrow-room experiment.



$\mathbf{u}^0$= (-0.723 0.680) $\mathbf{u}^1$=(-0.532 -0.837) $\mathbf{u}^2$= (-0.877 -0.481)

Fig. 27. The first four eigenvectors and the primitive actions for the narrow-room experiment.

correctly. The second primitive action is primarily an advancing action but has a significant turning component to it. The method erroneously identifies three primitive actions. The second and third primitive actions are both poor approximations of a motor control vector that produces a pure advancing motion.

6.5. *A circular room*

This experiment demonstrates an instance in which the learning agent fails to discover any local state variables. The robot's environment is a circular room three meters in diameter. The results of the learning of the sensorimotor apparatus are summarized by the set of principal eigenvectors and primitive actions shown in Fig. 28. The learning agent identifies two primitive actions corresponding to turning and advancing, but fails to discover any local state variables. The following analysis explains why this happened.

For a feature to be a local state variable, it must be both action-dependent and predictable. For a feature to be predictable, the effects of the primitive actions on the feature must be known for all possible contexts. In the rectangular and T-shaped environments, the local-minimum features (which give distances from the robot to nearby objects or walls) were identified as local state variables. Here is a summary of what was learned by the learning agent (and represented in the static action model) for the robot in the rectangular environment:

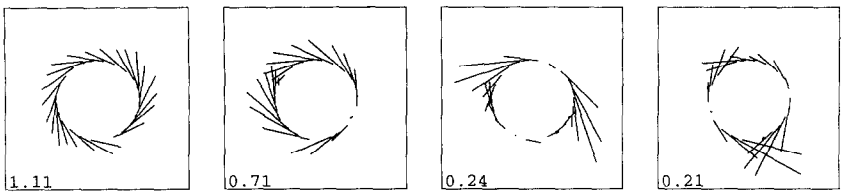


Fig. 28. The first four eigenvectors and the primitive actions for the circular-room experiment.

- The first primitive action (turning) does not affect the local-minimum features. The effects of the primitive action are thus predictable for all contexts.
- The effect of the second primitive action (advancing) is context dependent:
  - When the robot is facing toward a wall, the primitive action reliably decreases the value of the local-minimum feature.
  - When the robot is facing away from a wall, the primitive action reliably increases the value of the local-minimum feature.
  - When the robot is facing parallel to the wall (in either direction), the primitive action leaves the value of the feature constant.

For this experiment (with the circular environment), the learning agent's learned static action model is identical to that described above, but with the following exception: When the robot is facing parallel to the wall, the effect of the second primitive action on the local-minimum feature is unpredictable. Here is an explanation for the difference. When facing parallel to a straight wall, a robot can move for many steps without changing the distance to the wall significantly. This is why it is possible for the linear regression tester that analyzes the effect of the primitive action to conclude that its effect is, to a good approximation, zero in this context. In the circular world, on the other hand, the robot can only advance a few steps without changing the distance to the wall. The only conclusion that the learning agent is able to draw from the linear regression tester is that the effect of advancing is unpredictable in this context.

6.6. Failure modes

Sections 6.4 and 6.5 gave two examples of cases in which the learning agent failed to learn a set of homing and path-following behaviors. This section provides a more exhaustive list of ways in which the learning methods described in this paper can fail. The next section discusses how the learning methods may be improved.

Modeling the sensory apparatus

If there is no structured array of sensors, then the group-feature generator will produce only small or singleton groups and the image-feature generator will not apply. If there is an array of sensors but the sensors do not adequately sample a continuous property of the environment, then the group and image features will fail to produce a representation of the structure of those sensors. For example, if there are only four distance sensors, then the values of adjacent sensors may not be similar enough for the group-feature

generator to group them together. If the environment is large and the learning agent does not adequately explore the environment before applying the group- and image-feature generators, then the measured inter-sensor distance metrics may not accurately reflect the structure of the sensory apparatus.

Representing motion

The motion-feature generator requires an image feature (either learned, as is the case here, or given a priori by the robot's designer). If there is no image feature, then the motion-feature generator will not apply. If the robot's motion is so fast that successive image-feature values are unrelated, then the motion feature will fail to produce meaningful results.

Modeling the motor apparatus

The matching process that identifies primitive actions (i.e., motor control vectors whose *amvf*'s match the principal eigenvectors) can fail to correctly identify a primitive action if the *amvf*'s have not converged (i.e., if the learning agent has not wandered long enough and the values of the *amvf*'s are still fluctuating with time). This is one possible explanation for the failure to identify just two primitive actions in Section 6.4.2.

Generating candidate local state variables

The discovery of local state variables may fail if the language of features and feature generators is not general enough. In such a case, none of the generated scalar features would satisfy the definition of local state variable (as in the experiment described in Section 6.5). On the other hand, if the language of features and generators is too general, the learning agent will quickly become bogged down in a combinatorial explosion of mostly useless features. In this paper, we identified a small set of feature generators that are appropriate for a robot with a rich sensorimotor apparatus and then demonstrated that they are sufficient for a particular set of environments and sensorimotor apparatuses.

Learning action models

The learning agent will fail to correctly learn the static and dynamic action models if it does not explore long enough for the linear-regression calculations to converge. In the case that the learning agent must learn the relationships between a motor control vector and a feature for a large number of contexts, the method requires that the learning agent experiment with the motor control vector in each of those contexts.

Learning path-following behaviors

The learning of path-following behaviors can depend on the set of learned primitive actions. If none of the primitive actions can be used to maintain any of the local state variables constant, then no path-following behaviors will be learned.

In the experiments described in this paper, each learning method builds on the results of the preceding methods, which means that one source of failure for a method is the failure of a preceding method. This observation, if left unqualified, sells the learning methods short. First, the methods are interesting in their own right (for example, the

modeling of the motor apparatus could be applied to a sensory system whose structure was given by the robot's designer rather than being learned). Second, the sequential nature of the learning is partially an artifact of our particular learning problem. For example, the discovery of local state variables does not, in general, depend on the success of the image-feature generator but is instead the result of an independent process of generate and test.

6.7. Future work

Section 6.6 identified a number of ways in which the learning methods can fail. This section provides suggestions for improvements to the learning methods.

Improved feature testers

One way that several of the learning methods can fail is by jumping to a conclusion prematurely. For example, if the group-feature generator uses a distance metric before the distance metric has converged, then the output of the generator may be incorrect. If primitive actions are identified before the *amvf*'s have converged, then the model of the motor apparatus may be incorrect.

In these examples, the distance metrics and the *amvf*'s are examples of *feature testers*—features that are used to characterize other features. A solution to the problem of drawing premature conclusions is to have each feature tester tell when its output is meaningful. It can do this by providing a measure of confidence in addition to its output value. For example, the confidence level for a tester might be close to 1 if the tester's output is stable (changing slowly) and close to 0 if the tester's output is still fluctuating.

For a linear regression tester, the confidence level should be a function of the set of inputs it has received. Consider, for example, how the static action model uses linear regression testers. It uses a separate linear regression tester for each ⟨*feature, primitive action, context*⟩ triple. If the robot is never in a given context, then the confidence level for any linear regression tester based on that context should be zero. The confidence level for a linear regression tester might be defined in terms of the 90% *confidence interval*[26] for the correlation between the input variables. The smaller the confidence interval, the greater the tester's confidence level. Associating confidence levels with features could improve all of the learning steps listed in the previous section by reducing the chance of producing inaccurate or incomplete models.

An improved static action model

The learning agent uses the static action model to define a set of open-loop path-following behaviors—behaviors that move the robot while maintaining a local state variable constant. In the current implementation, open-loop path-following behaviors are based on primitive actions. If a primitive action maintains a local state variable constant, according to the static action model, then it can be used as the "base action" for a path-following behavior. Using only primitive actions as base actions is a limitation of

[26] See, for example, [15, p.415].

the current implementation. The method for learning path-following behaviors would be improved if the static action model could predict the effects of arbitrary motor control vectors, not just the primitive actions. With a more comprehensive static action model, more path-following behaviors could be defined. For example, in the circular room, a path-following behavior could be based on a motor control vector with a large advancing component and a small turning component.

One approach to improving the static action model would be to discretize the space of all motor control vectors into a set of representative motor control vectors and then to learn models of all of these instead of just the primitive actions. Another approach would be to use a neural network [12] to learn to predict the context-dependent effects of arbitrary actions. The network could then serve as the static action model and could be used to find base actions for path-following behaviors.

Reinforcement learning

It may be possible to use reinforcement learning [3, 23, 36, 42, 45] to learn homing and path-following behaviors without the need for the primitive actions or explicit action models. An advantage of such an approach is that it does not presume that a particular model of the sensorimotor apparatus has been learned. A disadvantage is that it is difficult to train more than one behavior at a time [44] whereas it is possible to learn action models for a large number of features simultaneously.

Learning composite primitive actions

Consider a robot that is capable of rotating and advancing and that has a ring of distance sensors that is always oriented in the same direction. The learning methods of Section 2 will succeed in identifying the structure of the ring of sensors. The first three steps of Section 3 will succeed in identifying two basic motions: one for translating in one direction, and one for translating in a perpendicular direction. The fourth step of Section 3, as currently implemented, will fail to identify two corresponding primitive actions since the robot is not capable of directly translating in two directions.

This suggests a topic for future research: to extend the learning of primitive actions to allow for composite actions (action sequences). In the example of the robot with the fixed sensor ring, a primitive action could be composed of a turn to a particular direction followed by an advance. An alternate solution is that of the preceding section: to learn homing and path-following behaviors directly using reinforcement learning without first learning primitive actions. This example illustrates that it is more difficult to learn a model of a sensorimotor apparatus for which an important action has no immediate effect on the sensors.

7. From continuous world to finite-state world

The learning agent has made the transition from raw senses and motor control vectors to local state variables and high-level behaviors (which comprise the control level of the spatial semantic hierarchy). The goal of the next step is to abstract from the continuous sensorimotor apparatus to a discrete sensorimotor apparatus by defining finite sets of

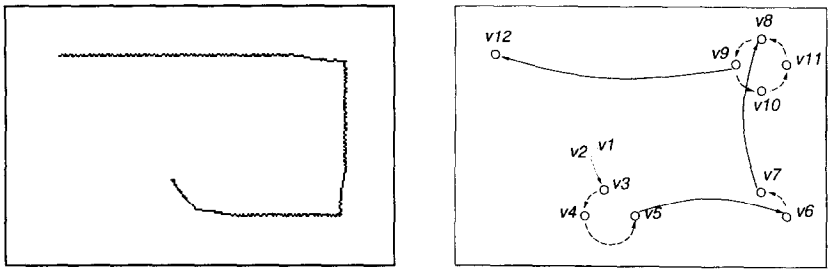


Fig. 29. A demonstration of the discrete abstract interface. We used the abstract interface of the discrete sensorimotor apparatus to select appropriate behaviors to drive it around the room. At each step, the interface provides the view name (e.g., *v1*) that identifies the current state, and a finite set of applicable homing and path-following behaviors. The dotted arrows represent behaviors based on left turn motor control vectors ($u_0 > 0$). The solid arrows represent behaviors based on forward advance motor control vectors ($u_1 > 0$). During this exploration, the robot identifies the 12 unique views shown in the figure on the right.

views and *actions*. The source of information for this step is the set of learned behaviors (including the knowledge of when each is applicable).

For any given state of the robot, there is a finite set of homing and path-following behaviors. These behaviors are the *actions* of the discrete sensorimotor apparatus. Executing one of the actions involves running the corresponding behavior until it terminates. The set of states in which actions terminate is also finite. These states are named via a mapping from sense vectors to symbols called *views*. In our experiments, this mapping is implemented using a matching predicate: two sense vectors are judged to be similar if their Euclidean distance is less than a small constant. If the current sense vector is new then a new view is created and associated with it. If the current sense vector matches one previously seen, it is associated with the same view as the previous sense vector.

This interface abstracts from continuous time to discrete time. While a path-following behavior is executing, the interface is undefined. When the behavior terminates, the interface identifies the current view and lists the current set of applicable behaviors. Fig. 29 demonstrates this interface. Initially ($v1$), no wall is within sensor range and the only available action is the wandering behavior. When the wandering behavior terminates ($v2$), a homing behavior is applicable. Selecting this behavior leads to view $v3$ where two path-following behaviors based on u_0 (turning) are applicable. Selecting the first leads to view $v4$. Selecting it again leads to view $v5$. At this point, two 1-degree-of-freedom path-following behaviors based on u_1 (advancing) are applicable. Choosing the first leads to $v6$. The figure shows the behavior of the robot during a user-guided exploration that leads it to $v12$. The rest of the exploration around the room (not shown) eventually returns the robot to the southeast corner. Using its matching predicate, the learning agent recognizes that it has returned to view $v6$. The robot's experience is represented as a collection of (V_i, A_j, V_k) triples, called *schemas*. This knowledge is the basis for the causal level of the spatial semantic hierarchy.

8. Learning the topology of the environment

In Section 1.1, we described the spatial semantic hierarchy, which is comprised of five levels: sensorimotor, control, causal, topological, and metrical. We have demonstrated a learning agent that has learned the first three levels: The sensorimotor level was learned in Sections 2 and 3. The control level was learned in Sections 4 and 5. The causal level was learned in Section 7. We now describe how the result of the agent's learning could be used to learn the remaining levels of the spatial semantic hierarchy.

The robot's path-following behaviors constrain its motion to a one-dimensional subspace of the robot's complete state space. This 1-D skeleton is the basis for an abstraction of the robot's environment as a graph (a set of nodes and a set of edges connecting the nodes together). The edges correspond to paths—trajectories in the robot's state space produced by path-following behaviors. The nodes correspond to states where paths terminate, that is, states where a new path-following behavior becomes applicable and the agent stops to choose one of the currently applicable paths. The agent's goal is to construct this graph.

In the case where views uniquely identify states, the problem is straightforward. The agent keeps track, for each state it has seen, of all the actions applicable at that state. Each time it takes an action, A_j, that takes it from view V_i to V_k, it adds the edge (V_i, A_j, V_k) to the graph. It continues to explore (intelligently or randomly) until there are no state-action pairs that it has not explored.

In the case that views do not uniquely identify states, a more sophisticated exploration strategy is required. Such strategies are generally based on the following idea: If the current view does not uniquely identify the current state, the agent supplements the current sense vector with the sense vectors of nearby states. With enough information about the surrounding area, the current state can be uniquely identified.

Finally, metrical information can be added to the topological representation by recording the time taken to traverse each path. With this information, navigation including shortest-path planning is possible.

To summarize, the learning agent has made a critical change of representation by abstracting a continuous sensorimotor apparatus to a discrete sensorimotor apparatus with a finite set of sense values and actions. Understanding a continuous world is very difficult. Our learning agent demonstrates a way to reduce the problem of understanding a continuous world to the problem of understanding a discrete world, a problem that has been extensively studied (see Section 9.1).

9. Related work

The work mentioned in this section deals with the general problem of learning a model of an environment. A complete model of an environment is a description that is sufficient for predicting the input/output behavior of the environment, i.e., for predicting the sensory input that will be received from the environment in response to any sequence of actions. In some cases, learning a complete model is impractical, in which case a partial model may be learned.

Methods for learning a model of an environment can be divided into two types: those that deal with finite-state worlds and those that deal with continuous worlds. Examples of the first type are given in Section 9.1. Examples of the second type are given in Section 9.2. Our contribution has been to show how a learning agent can abstract a robot's continuous world to a finite-state world to which finite-state learning methods may be applied.

9.1. Inferring the structure of finite-state worlds

The task of inferring the structure of a finite-state environment is the task of finding a finite-state automaton that accurately captures the input-output behavior of the environment. In the case that the learning agent is passively given examples of the environment's input/output behavior, it has been shown that finding the smallest automaton consistent with the behavior is NP-complete [1,9]. With active learning, in which the agent actively chooses its actions, the problem becomes tractable. Kuipers [17] describes the TOUR model, a method for understanding discrete spatial worlds based on a theory of cognitive maps. Dudek et al. [8] generalize Kuipers and Byun's [18,19] strategy for topological map-learning and provide algorithms for discriminating perceptually identical states. Angluin [2] gives a polynomial-time algorithm using active experimentation and passively received counterexamples. Rivest and Schapire [38] improve on Angluin's algorithm and give a version that does not require the reset operation (returning to the start state after each experiment).

Dean et al. [5] have extended Rivest and Schapire's theory to handle stochastic FSAs. They assume that actions are deterministic but that the output function mapping states to senses is probabilistic. The key to their method is "going in circles" until the uncertainty washes out. Dean, Basye, and Kaelbling [6] give a good review of learning techniques for a variety of stochastic automata. Drescher's schema mechanism [7] employs a statistical learning method called marginal attribution. Schemas emphasize sensory effects of actions rather than state transitions and are ideal for representing partial knowledge in stochastic worlds.

Wei-Min Shen's LIVE system [40] learns the structure of a finite-state environment from experience (and experimentation) within it. His complementary discrimination learning algorithm exploits observed counterexamples to a hypothesized concept definition to refine the boundary between positive and negative examples of the concept. When the environment is only partially observable, LIVE uses locally distinguishing experiments to test the hypothesized properties of unobserved state variables.

A primary focus of the work of Shen and other constructive inductionists [10,28, 39,41] is the learning of new features. At this level of description, our approach and Shen's are similar. However, in terms of the actual methods used and the domains of applicability, the two approaches are very different and are in fact complementary.

We focus on feature-learning methods applicable to robots with continuous-valued sensors and control signals situated in a 2-dimensional approximation of a 3-dimensional world. We provide a language of features and generators especially suitable for robots with structured arrays of sensors. Our emphasis is on learning of sensory features and continuous control laws. Shen, on the other hand, focuses on learning rules (consisting

of conditions, actions, and predictions) that are expressed symbolically. The conditions and predictions are expressed in terms of "percepts", which are high-level, symbolic descriptions (e.g., ON(disk, peg)). The actions may have continuous parameters (e.g., *rotate*(θ)), but each action is atomic rather than continuous.

The two approaches might be combined in the following way: a learning agent first uses our methods to learn to navigate using its continuous sensorimotor apparatus, viewing the world in terms of discrete states and actions. It then uses Shen's methods to learn relationships expressed in terms of these states and actions and to acquire nonspatial knowledge such as the effects of pushing objects, flipping switches, or opening doors.

A more specific example of a potential combination of the two approaches is the use of Shen's complementary discrimination learning in learning context-dependent action models (see Section 4.2.3). We currently use a brute-force method for determining whether a control signal u has a predictable effect on a given feature x. The method involves testing a large set of features to see if any can be used to define a partition of the robot's state space as a set of contexts such that in each context there is a simple, linear relationship between u and x. We expect that methods such as Shen's complementary discrimination learning could be used to generate such partitions more efficiently and more intelligently.

9.2. Inferring the structure of continuous worlds

Applying finite-state automaton learning methods to the real world or a continuous simulation of it requires an abstraction from a continuous environment to a discrete representation. Kuipers and Byun [18, 19] demonstrate an engineered solution to the continuous-to-discrete abstraction problem for the NX robot. NX's *distinctive places* correspond to discrete states and its *local control strategies* correspond to state transitions. These constructs have to be manually redesigned in order to apply to a robot with a different sensorimotor apparatus. Mataric [26, 27] and Kortenkamp & Weymouth [14] have engineered similar solutions on physical robots.

Lin and Hanson [24] use a physical robot, called Ratbot, with 16 sonar sensors and 16 infrared sensors to demonstrate learning of a topological map of locally distinctive places. Their work is inspired by the work of Kuipers and Byun, but they use reinforcement learning[27] to train the local control strategies, rather than engineering them by hand. The target behaviors (e.g., corridor following) are specified by a human teacher. For example, when learning the corridor-following behavior, the robot is rewarded when it moves along the corridor without running into obstacles.

Our approach [30–32, 34] is complementary to that of Lin and Hanson. They specify the desirable behaviors by defining appropriate reward signals and then letting the robot learn on its own how to gain the rewards. Our learning agent, on the other hand, specifies its own target behaviors, eliminating the need for the human teacher. It does this by first learning a set of local state variables and then using them to define a set of error signals. Homing and path-following behaviors are then specified as behaviors that

[27] The reinforcement-learning algorithm is a neural-network version of Q learning [23, 43].

minimize the error signals or move the robot while maintaining them near zero. All of this is accomplished in a domain-independent manner—the robot does not need to be given any knowledge about corridors or corridor-following behaviors.

Once the error signals are defined, there are a number of ways in which the homing and path-following behaviors might be learned. Reinforcement learning is one approach.[28] The approach used in this paper is to learn static and dynamic action models that characterize the effects of actions on the local state variables and then to use these models to directly define the homing and path-following behaviors. This approach does require that the learning agent be given some knowledge of control theory, but the required knowledge consists of domain-independent templates. It would be interesting to combine our approach with that used by Lin and Hanson's Ratbot to produce a learning method that uses neither domain-dependent knowledge nor a knowledge of control theory. The error signals would be defined as for our learning agent and a neural-net version of Q learning would be used to learn the local control strategies based on those error signals. The control laws would be implemented as mappings from sensory inputs to motor control signals. If the sensory inputs include the error signals, their derivatives, and their integrals, then the set of control laws that can be defined in this way includes the PI and PD control laws used by our implemented learning agent.

An important difference between our approach and that of Lin and Hanson is that our approach handles context dependence at the feature level rather than at the behavior level: Our learning agent learns, given the current context, the effects of each primitive action on each feature. Lin and Hanson's learning agent learns, given the current context, which action to take in order to produce a particular behavior. There are two advantages to learning at the feature level. First, what is learned about one feature may be used to define multiple behaviors, e.g., a homing behavior (in which a primitive action is used to increase or decrease the value of the feature) and a path-following behavior (in which a primitive action is used to move while maintaining the feature constant). Second, the learning agent can learn the effects of motor control signals on multiple features simultaneously, whereas it is only possible for Lin and Hanson's learning agent to learn one behavior at a time.

10. Discussion

10.1. What is the value of an existence proof?

As discussed in Section 1.3, the results presented here are an existence proof, demonstrating one path from the beginning to the end of a complex learning problem. Once a single path has been demonstrated, however narrow, future research can broaden the way and find alternate routes.

We have made some progress toward assessing the width and solidity of the path, first by applying the same learning methods to a significantly different robot (Section 2.5), and second by applying our methods to a variety of different environments systematically

[28] In earlier work we explored the use of reinforcement learning to learn homing behaviors [33].

designed to demonstrate both success and failure of the methods (Section 6). This is a step toward determining how much and what type of sensory input a robot must have to learn a meaningful cognitive map of a continuous environment, and how observable and predictable the environment must be for the robot to be able to comprehend it.

The existence proof demonstrates that a hard and interesting learning problem may have a heterogeneous solution, combining the strengths of several focused learning algorithms. While this solution does rely on a number of assumptions about the sensorimotor system and the environment (Section 6.6), we believe that several of those assumptions can be eliminated by future research (Section 6.7). An irreducible minimum set of assumptions would be a significant scientific result.

10.2. Why learn what can be programmed directly?

This paper has shown, among other things, how a learning agent can learn a model of its sensorimotor apparatus. There are several reasons why it is worthwhile to take the trouble to learn what could be directly programmed by a robot's designer.

- *Sensor variation and failure.* Direct programming does not take sensor failure into consideration. For example, if one of a set of distance sensors fails, the learning methods will accommodate the failure with no additional human intervention. These methods will also accommodate random variation in the position or direction of distance sensors. Such variation is inevitable if robots are mass produced.
- *Generality.* Ideally, one learning algorithm applies to many different types of sensorimotor apparatuses and thus can replace the process of designing a particular solution for each sensorimotor apparatus.
- *A deeper understanding of the problem domain.* The design of the learning agent required the identification of sources of information that could be exploited by the learning agent and the development of general-purpose learning algorithms to exploit that information. These sources of information and learning algorithms comprise a deeper understanding of the problem domain. [29]

10.3. What about innate goals?

We have characterized a robot in terms of its set of sensors and effectors, without concern for its innate goals (e.g., survival, curiosity, pain avoidance). The learning methods we have developed function by observing the sensory effects of actions, either during a random walk through the environment (as described in this paper) or during goal-directed behavior.

Reactive behavior in pursuit of innate goals can support the learning methods described here. With a goal such as pain avoidance, for example, a learning agent might quickly learn a reflexive behavior for obstacle avoidance. Such a behavior would help keep the learning agent out of danger as it applies the higher-level learning methods. On the

[29] Of course, designing a learning agent does not *guarantee* a deeper understanding of the problem domain. An opaque method such as neural net or genetic algorithm learning could conceivably learn a model of its sensorimotor apparatus without teaching us anything about perception, behavior, or map building.

other hand, by operating in the background of goal-directed behavior, the learning agent could receive a biased set of experiences and observations of the environment.

Conversely, the learned methods can serve as a foundation for goal-directed learning. When the agent has learned higher-level sensorimotor primitives, it can search for behaviors in an action space of larger granularity, describing the environment at a higher level (see Fig. 20), and making it easier to achieve innate goals such as survival and curiosity.

10.4. How general are the learning methods?

This paper has identified and demonstrated a number of generic methods for modeling and using an uninterpreted sensorimotor apparatus. This section lists several examples where a generic object or method is used that subsumes a more specific object or method but is more general because it makes fewer assumptions.

The learned features are based on a set of generic mathematical constructs (e.g., scalars, vectors, matrices, scalar fields, and vector fields) rather than on a human-generated list of salient properties of a robot's environment. The method for identifying the structure of a sensory apparatus uses generic distance metrics that make no assumptions about what the sensors are sensing (for example, the method does not assume that the sensors measure distances to objects). The method for characterizing the effects of motor control vectors is based on spatial and temporal derivatives, not motion of objects (which would require the identification and tracking of objects). The local state variables learned by the learning agent in the example are defined in terms of the purely generic concept of local minimum, rather than the concept of distance-to-wall, which is only meaningful to the robot's designer. The learned control laws are based on error signals derived from the learned local state variables—the learning agent of the example needed no concept of wall when defining its path-following behaviors. The path-following behaviors are implemented using generic control laws. The parameters used in the control laws are found by analyzing relationships between control signals and local state variables without any understanding of the meanings of the control signals or local state variables. The views of the learning agent's discrete abstract interface are the terminal states of path-following behaviors, as opposed to places meaningful only to the robot's designer.

Related to the concept of generality is the concept of extensibility. The current implementation may be extended by adding new types of features and feature generators. For example, new distance metrics could be used with the group-feature generator to capture new ways of distinguishing different types of sensors; the method for generating and recognizing local state variables could be made more general by adding new feature generators.

10.5. Changes of representation

Each abstract interface that the agent learns provides a new representation to reason with.

- At the sensorimotor level, the group and image-feature generators analyze inter-sensor correlations to produce the image feature, which has substantially more structure than the raw sense vector.
  The learned set of primitive actions provide a new representation of the robot's motor capabilities that is grounded in sensory effects.
- At the control level, behaviors and features are learned that are no longer purely egocentric. Whereas the primitive actions are grounded in sensory effects averaged over time, the homing and path-following behaviors are grounded in the structure of the external environment as reflected by the local state variables.
- At the causal level, the continuous state space is reduced to a finite set of states and trajectories, which can then be represented as the nodes and edges of a graph in the topological map.

11. Summary

This paper has presented a sequence of learning methods sufficient for learning a cognitive map of a robot's continuous world in the absence of domain-dependent knowledge of the robot's sensorimotor apparatus or of the structure of its world. The reader may object that the sequence is tenuous: if any method failed, then the subsequent methods would not even apply. While this is true, we maintain that each of the learning methods is interesting in its own right and is applicable beyond the particular learning problem investigated here. Each learning method identifies a source of information available through experimentation with an uninterpreted sensorimotor apparatus and each provides a method for exploiting that information to give the learning agent a new way of understanding the robot's sensory input or a new way of interacting with the robot's environment.

The learning methods are summarized below and in Fig. 30. Section 2 showed how to use the group and image-feature generators to learn a structural model of a sensory apparatus. They exploit the fact that, in a well-engineered array of sensors sampling an almost-everywhere continuous property of the environment, the layout of the sensors may be reconstructed based on inter-sensor similarities. Section 3 showed how to use this structural knowledge to first define motion detectors and then use them to characterize the capabilities of a motor apparatus using a set of primitive actions, one for each of the robot's degrees of freedom. Section 4 showed how to recognize local state variables—scalar features whose derivatives can be approximated by context-dependent linear functions of the motor control signals. The effects of the primitive actions on the local state variables are captured by the static action model. Section 5 showed how to use the static action model to define homing and open-loop path-following behaviors, how to learn a dynamic action model to predict the effects of the primitive actions on the local state variables while open-loop path-following behaviors execute, and how to use the dynamic action model to define robust, closed-loop path-following behaviors. Finally, Section 7 showed how to use the homing and path-following behaviors to define a discrete abstract interface that allows the learning agent to abstract its continuous world to a finite-state world. By using the finite-state automaton as the target abstraction,

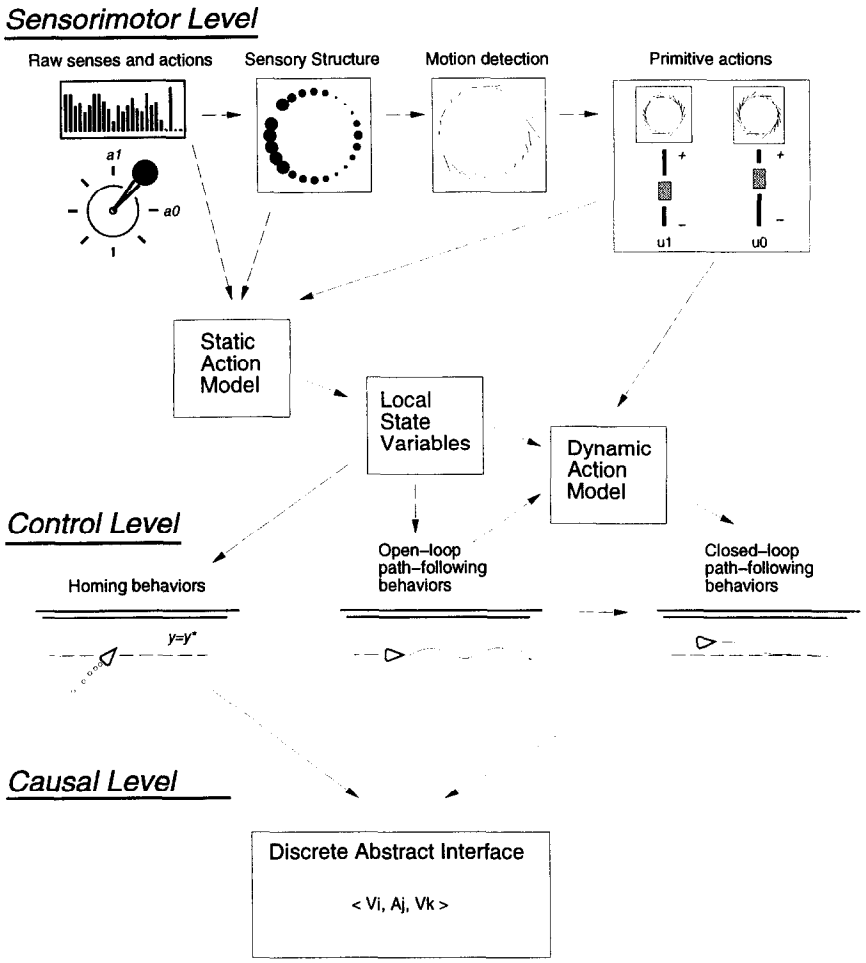


Fig. 30. A graphical summary of the learning methods used in this paper, showing the objects learned at each of the first three levels of the spatial semantic hierarchy.

the learning agent inherits a powerful set of methods for inferring the structure of its world.

In the biological world, the newly hatched organism embodies a great deal more knowledge than our learning agent. However, we hope that an exploration like that reported here will shed light on the structure and learnability of fundamental knowledge about an agent's relationship with its world. If so, it could provide insights into the evolution, development, and learning of spatial knowledge in biological organisms.

The potential for application of these methods to mechanical robots is much more direct. New robots, with new sensors and effectors, are being designed and built all the time. Robots will one day be sent into environments that humans have never directly

experienced (e.g., the deep ocean floor or the surface of another planet). For a newly created robot to be able to orient itself to its sensorimotor system and its environment through autonomous experimentation would be of substantial value. We believe that the methods presented here are a step in that direction.

Appendix A. Computational complexity

This appendix summarizes the complexities of the various learning methods described in this paper. The overall complexity of the sequence of learning methods is potentially exponential in the size of the raw sense vector and the depth of the tree of generated features. In our experience, this level of complexity can be drastically reduced by using an appropriate set of feature generators as is explained in Section A.3.

A.1. Modeling the sensory apparatus

Computing the distance metric d_1 (used by both the group-feature generator and the image-feature generator) is of complexity $O(n^2T)$ where n is the number of elements in the raw sense vector and T is the number of time steps taken before the group-feature generator is applied. Computing the distance metric d_2 (used by the group-feature generator) requires an $O(1)$ computation for each element of the raw sense vector at each time step (to update the frequency distributions) and an $O(n^2)$ computation when the group-feature generator is applied for a total complexity of $O(nT + n^2)$.

Identifying similar subgroups is an $O(n^2)$ computation. Using transitive closure to identify closed subgroups is an $O(n^3)$ computation. The metric-scaling is performed with an iterative algorithm for which each iteration involves an $O(n^3)$ computation. The relaxation algorithm is also iterative with each iteration being $O(n^2)$. Since the dependence of the number of iterations on n is unknown, these are lower limits on the actual complexities. In our experiments, T is much greater than n so the overall complexity of the sensory-modeling step can be approximated by $O(n^2T)$.[30]

A.2. Modeling the motor apparatus

The calculation of the motion feature requires an $O(n^2)$ computation at each time step. The calculation of the *amvf*'s is thus of complexity $O(n^2T)$. The principal component analysis algorithm is of complexity $O(n^3)$. Again, since T is much greater than n, the overall complexity can be approximated by $O(n^2T)$.

A.3. Identifying local state variables

The first step in identifying local state variables is to generate new features. If every subset of the current set of defined features can be used to produce a new set of features, then the complexity of generating and testing features will be at least $O(2^n)$ where

[30] An open problem is to predict the value of T for each learning method that requires an exploration phase.

n is the number of elements of the raw sense vector. In our experience, this potential combinatorial explosion can be avoided by using an appropriate set of feature generators (e.g., generators that collapse many input features into a small set of output features, or that only apply to certain types of features). For example, the group generator does not create arbitrary subsets of the raw sense vector—it creates at most n non-overlapping group features.

The second step in identifying local state variables is to compute the static action model. The complexity of the computation of the action-dependent model is $O(sT)$ where s is the number of singleton features that have been learned and T is the number of time steps over which the model is learned. The complexity of the computation of the action-independent model is $O(sTa)$ where a is the number of primitive actions, The complexity of the computation of the context-dependent model is $O(sTac)$ where c is the average number of contexts associated with each feature.

A.4. Learning control laws

The number of open-loop path-following behaviors is $O(vac)$ where v is the number of learned local state variables, a is the number of primitive actions, and c is the average number of contexts associated with each local state variable. The complexity of the computation of the dynamic action model is $O(vacT(a-1))$. The number of generated closed-loop path-following behaviors (worst-case) is $O(2^v a 2^c)$. In practice, the number of path-following behaviors can be made much less than this upper bound. For example, the terms 2^v and 2^c can be replaced by v^3 and c^3 by only defining path-following behaviors whose error vectors are based on at most three local state variables.

In our experiments, the number of path-following behaviors is kept reasonable by the following facts:

(1) the number of learned local state variables is small and
(2) the number of contexts in which a primitive action maintains a local state variable constant is small relative to the total number of contexts.

In our example, v is at most 3 and c is around 20.

Acknowledgements

The authors would like to thank Rick Froom, Wan Yik Lee, Risto Miikkulainen, Ray Mooney, Lyn Pierce, Mark Ring, Boaz Super, and two anonymous reviewers for their technical, editorial, and moral support.

References

[1] D. Angluin, On the complexity of minimum inference of regular sets, *Inform. and Control* **39** (1978) 337–350.

[2] D. Angluin, Learning regular sets from queries and counterexamples, *Inform. and Comput.* **75** (1987) 87–106.

[3] D. Chapman and L.P. Kaelbling, Learning from delayed reinforcement in a complex domain, Tech. Rept. TR-90-11, Teleos Research, Palo Alto, CA, 1990.

[4] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (MIT Press/McGraw-Hill, Cambridge, MA, 1990).
[5] T.L. Dean, D. Angluin, K. Basye, S. Engelson, L.P. Kaelbling, E. Kokkevis and O. Maron, Inferring finite automata with stochastic output functions and an application to map learning, in: *Proceedings AAAI-92*, San Jose, CA (1992) 208–214.
[6] T.L. Dean, K. Basye and L.P. Kaelbling, Uncertainty in graph-based map learning, in: J.H. Connell and S. Mahadevan, eds., *Robot Learning* (Kluwer Academic Publishers, Boston, MA, 1993) 171–192.
[7] G.L. Drescher, *Made-Up Minds: A Constructivist Approach to Artificial Intelligence* (MIT Press, Cambridge, MA, 1991).
[8] G. Dudek, M. Jenkin, E. Milios and D. Wilkes, Robotic exploration as graph construction, *IEEE Trans. Robotics and Automation* **7** (6) (1991) 859–865.
[9] E.M. Gold, Complexity of automaton identification from given data, *Inform. and Control* **37** (1978) 302–320.
[10] K. Hiraki, Abstraction of sensory-motor features, in: *Proceedings 16th Annual Conference of the Cognitive Science Society* (Lawrence Erlbaum Associates, Hillsdale, NJ, 1994).
[11] B.K.P. Horn, *Robot Vision* (MIT Press, Cambridge, MA, 1986).
[12] M.I. Jordan and D.E. Rumelhart, Forward models: supervised learning with a distal teacher, *Cognitive Sci.* **16** (1992) 307–354.
[13] T. Kohonen, *Self-Organization and Associative Memory* (Springer, Berlin, 2nd ed., 1988).
[14] D. Kortenkamp and T. Weymouth, Topological mapping for mobile robots using a combination of sonar and vision sensing, in: *Proceedings AAAI-94*, Seattle, WA, 1994.
[15] W.J. Krzanowski, *Principles of Multivariate Analysis: A User's Perspective*, Oxford Statistical Science Series (Clarendon Press, Oxford, 1988).
[16] B.J. Kuipers, An ontological hierarchy for spatial knowledge, in: *Proceedings 10th International Workshop on Qualitative Reasoning about Physical Systems*, Fallen Leaf Lake, CA, 1996.
[17] B.J. Kuipers, Modeling spatial knowledge, *Cognitive Sci.* **2** (1978) 129–153.
[18] B.J. Kuipers and Y.-T. Byun, A robust, qualitative method for robot spatial learning, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 774–779.
[19] B.J. Kuipers and Y.-T. Byun, A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations, *J. Robotics and Autonomous Systems* **8** (1991) 47–63.
[20] B.J. Kuipers and T.S. Levitt, Navigation and mapping in large-scale space, *AI Magazine* **9** (2) (1988) 25–43.
[21] B.C. Kuo, *Automatic Control Systems* (Prentice-Hall, Englewood Cliffs, NJ, 4th ed., 1982).
[22] D.B. Lenat, On automated scientific theory formation: A case study using the AM program, in: J.E. Hayes, D. Michie and L.I. Mikulich, eds., *Machine Intelligence* **9** (Halsted Press, New York, 1977) 251–286.
[23] L.-J. Lin, Reinforcement learning for robots using neural networks, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA (1993).
[24] L.-J. Lin and S.J. Hanson, On-line learning for indoor navigation: Preliminary results with RatBot, in: *Proceedings NIPS93 Robot Learning Workshop*, 1993.
[25] K.V. Mardia, J.T. Kent and J.M. Bibby, *Multivariate Analysis* (Academic Press, New York, 1979).
[26] M.J. Mataric, Navigating with a rat brain: A neurobiologically-inspired model for robot spatial representation, in: J.-A. Meyer and S.W. Wilson, eds., *From Animals to Animats: Proceedings 1st International Conference on Simulation of Adaptive Behavior* (MIT Press/Bradford Books, Cambridge, MA, 1991) 169–175.
[27] M.J. Mataric, Integration of representation into goal-driven behavior-based robots, *IEEE Trans. Robotics and Automation* **8** (3) (1992) 304–312.
[28] C.J. Matheus, The need for constructive induction, in: L.A. Birnbaum and G.C. Collins, eds., *Proceedings Eighth International Conference on Machine Learning*, Ithaca, NY (Morgan Kaufmann, San Mateo, CA, 1991) 173–177.
[29] E. Oja, A simplified neuron model as a principal component analyzer, *J. Math. Biology* **15** (1982) 267–273.
[30] D. Pierce, Learning a set of primitive actions with an uninterpreted sensorimotor apparatus, in: L.A. Birnbaum and G.C. Collins, eds., *Proceedings Eighth International Conference on Machine Learning*, Ithaca, NY (Morgan Kaufmann, San Mateo, CA, 1991) 338–342.

[31] D. Pierce, Learning turn and travel actions with an uninterpreted sensorimotor apparatus, in: *Proceedings IEEE International Conference on Robotics and Automation*, Los Alamitos, CA (IEEE Computer Society Press, Silver Spring, MD, 1991) 246–251.

[32] D. Pierce, Map learning with uninterpreted sensors and effectors, Ph.D. Thesis, University of Texas at Austin, 1995; http://ftp.cs.utexas.edu/pub/qsim/papers/Pierce-PhD-95.ps.Z

[33] D. Pierce and B.J. Kuipers, Learning hill-climbing functions as a strategy for generating behaviors in a mobile robot, in: J.-A. Meyer and S.W. Wilson, eds., *From Animals to Animats: Proceedings 1st International Conference on Simulation of Adaptive Behavior*, (MIT Press/Bradford Books, Cambridge, MA, 1991) 327–336; also: Tech. Rept. TR AI91-137, AI Laboratory, University of Texas at Austin.

[34] D. Pierce and B.J. Kuipers, Learning to explore and build maps, in: *Proceedings AAAI-94*, Seattle, WA (AAAI/MIT Press, Cambridge, MA, 1994).

[35] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes in C* (Cambridge University Press, Cambridge, 1988).

[36] M. Ring, Continual learning in reinforcement environments, Ph.D. Thesis, University of Texas at Austin, 1994.

[37] H.J. Ritter, T. Martinez and K.J. Schulten, *Neural Computation and Self-Organizing Maps: An Introduction* (Addison-Wesley, Reading, MA, 1992).

[38] R.L. Rivest and R.E. Schapire, Inference of finite automata using homing sequences. *Inform. and Comput.* **103** 2 (1993) 299–347.

[39] W.-M. Shen, Functional transformations in AI discovery systems, *Artificial Intelligence* **41** (1990) 257–272.

[40] W.-M. Shen, *Autonomous Learning from the Environment* (Freeman, New York, 1994).

[41] W.-M. Shen and H.A. Simon, Rule creation and rule learning through environmental exploration, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 675–680.

[42] R.S. Sutton. Integrated architectures for learning, planning and reacting based on approximating dynamic programming, in: B.W. Porter and R.J. Mooney, eds., *Proceedings 7th International Conference on Machine Learning*, Austin, TX (Morgan Kaufmann, San Mateo, CA, 1990) 216–224.

[43] C.J.C.H. Watkins, Learning from delayed rewards, Ph.D. Thesis, King's College, Cambridge (1989).

[44] S. Whitehead, J. Karlsson and J. Tenenberg, Learning multiple goal behavior via task decomposition and dynamic policy merging, in: J.H. Connell and S. Mahadevan, eds., *Robot Learning* (Kluwer Academic Publishers, Boston, MA, 1993) 45–78.

[45] R.J. Williams, Reinforcement-learning connectionist systems, Tech. Rept. NU-CCS-87-3, College of Computer Science, Northeastern University, Boston, MA (1987).