Supervised learning and systems with excess degrees of freedom*

Michael I. Jordan
Massachusetts Institute of Technology

May 1988
COINS Technical Report 88-27

Abstract

When distinct outputs of an adaptive system have equivalent effects on the environment, the problem of finding appropriate actions given desired results is ill-posed. For supervised learning algorithms, the ill-posedness of such "inverse learning problems" implies a certain flexibility—during training, there are in general many possible target vectors corresponding to each input vector. To allow supervised learning algorithms to make use of this flexibility, the current paper considers how to specify targets by sets of constraints, rather than as particular vectors. Two classes of constraints are distinguished—*configurational* constraints, which define regions of output space in which an output vector must lie, and *temporal* constraints, which define relationships between outputs produced at different points in time. Learning algorithms minimize a cost function that contains terms for both kinds of constraints. This approach to inverse learning is illustrated by a robotics application in which a network finds trajectories of inverse kinematic solutions for manipulators with excess degrees of freedom.

---

*Preparation of this report was supported by grant AFOSR-87-03 to Andrew Barto from the Air Force Office of Scientific Research, Bolling, AFB. I want to thank Andrew Barto for his support and for helpful commentary on the paper.

Contents

In many domains, it is the case that targets, goals, or other criteria do not uniquely specify the actions which must be produced to meet the criteria. That is, many systems are redundant, and large or even infinite numbers of combinations of values for the system's variables lead to the same results. A classical example of such a system is a manipulator with an excess number of degrees of freedom, and such a system will be used to demonstrate the ideas developed in this paper. However, the problem of excess degrees of freedom is more general than the kinematics of manipulators, and arises in many forms, often under the rubric of "context-sensitivity," or "parallelism."

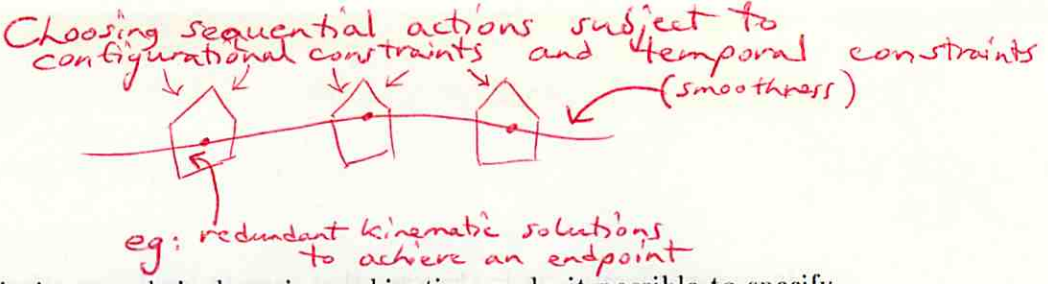
Of the learning problems that a system interacting with an environment must solve, many can be characterized as either "forward" problems or "inverse" problems. A paradigmatic case of a forward problem is concept formation. Given an environment that is characterized by a many-to-one function, the system observes a set of input-output pairs taken from this function, and attempts to choose correct outputs for novel inputs. In an inverse problem, such as goal-reduction, input-output pairs are again assumed to be available, after which it is desired to find the correct *inputs* corresponding to a given set of *outputs*. This latter problem is one-to-many, or ill-posed.

Research on connectionist learning algorithms has made progress on the general problem of converting a set of input-output observations into predictions for novel data (Ackley, Hinton, & Sejnowski, 1985; Hinton, 1987; LeCun, 1985; Parker, 1985; Rumelhart, Hinton, & Williams, 1986; Sutton, 1987). However, these algorithms have tended to emphasize the role of the environment as a source of stimuli, rather than as a recipient of actions; that is, the algorithms have generally been developed in the context of forward problems.[1] Although it is possible to adapt these algorithms to inverse problems by simply reversing the role of the input and output units of the network with respect to the environment, this approach ignores the one-to-many nature of inverse problems.[2]

This paper discusses how to allow a network to choose an action from among a set of possibilities. To this end, I first consider generalizations of the supervised learning paradigm that allow flexibility in the specification of target values for a network. The approach includes the use of don't-care conditions, inequalities and ranges defined on individual output variables, and constraints between the output variables. In the latter case, the constraints may be linear or nonlinear. These

[1] The major exception to this statement is reinforcement learning, see Barto, Sutton, & Anderson, 1983.

[2] This approach is called "direct inverse modeling," and has been studied by Kawato, Fusukawa, & Suzuki (1987), Kuperstein (1987), and Widrow & Stearns (1985), among others. When there are excess degrees of freedom, a network using direct inverse modeling will generally not find an inverse mapping, even though there are infinite number of possible inverse mappings which solve the problem. For example, let $\mathbf{y} = \mathbf{c}(\mathbf{x}_1)$ and $\mathbf{y} = \mathbf{c}(\mathbf{x}_2)$, for a particular vector $\mathbf{y}$, so that $\mathbf{x}_1$ and $\mathbf{x}_2$ are possible solutions when $\mathbf{y}$ is the input vector to the network. After repeated presentation of the stimulus pairs $(\mathbf{y}, \mathbf{x}_1)$ and $(\mathbf{y}, \mathbf{x}_2)$, the network will output a vector $\mathbf{x}$ which minimizes the sum of squared differences $\sum_{i=1}^{2}(\mathbf{x} - \mathbf{x}_i)^2$, but is not in general a solution (i.e., in general, $\mathbf{y} \neq \mathbf{c}(\mathbf{x})$).

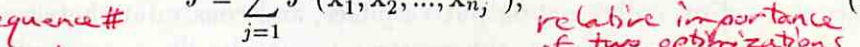


representational devices, used singly or in combination, make it possible to specify regions of the output space from which the network must select a particular output vector.

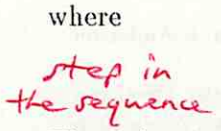
When the constraints on solutions to a problem do not specify a unique solution, it is commonly the case that optimality principles are invoked: The solution is required to optimize some function $k(\mathbf{x})$ defined over the set of vectors that meet the constraints (Wilde & Beighter, 1967). For example, for underconstrained systems of linear equations $\mathbf{Ax} = \mathbf{b}$, there exist an infinite number of solution vectors $\mathbf{x}$. If it is further specified that the solution vector have the minimum norm among all possible solution vectors, then the unique solution is given by $\mathbf{x} = \mathbf{A}^{+}\mathbf{b}$, where $\mathbf{A}^{+}$ is the pseudoinverse of $\mathbf{A}$. Such a result is convenient mathematically; however, the minimum norm principle may or may not have meaning in the context of the problem being addressed. In general, it is desirable that an optimality principle have some grounding in a theoretical understanding of the problem being solved, rather than being used merely to obtain uniqueness.

In the current paper, constraints on solutions are treated by converting them to a function $E(\mathbf{x})$ to be minimized, thus the optimization of a further quantity $k(\mathbf{x})$ simply adds another term to E. I also wish to consider a broader approach to resolving excess degrees of freedom for situations in which actions are chosen not in isolation, but rather in the context of the choices of other actions. That is, I consider inverse problems in which sequences of actions must be generated, where each action is defined by a set of *configurational constraints*, and where it is assumed that the sequential context provides additional implicit *temporal constraints* on the form actions may take. Although temporal constraints can arise in many ways, it is assumed here that they can be captured by a smoothness criterion defined over sequences. Smoothness implies that actions nearby in time are implemented in non-conflicting ways, or relatedly, that actions spread in time and blend with one another, in ways that still respect the configurational constraints. Thus, the optimization problem that is treated here includes both configurational constraints and temporal constraints, and the functional to be optimized takes the general form

$$J = \sum_{j=1}^{m} J^j(\mathbf{x}_1^j, \mathbf{x}_2^j, ..., \mathbf{x}_{n_j}{}^j), \tag{1}$$

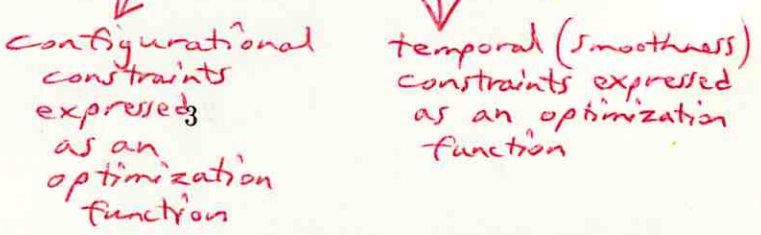


where



$$J^j(\mathbf{x}_1^j, \mathbf{x}_2^j, ..., \mathbf{x}_{n_j}{}^j) = \sum_{i=1}^{n_j} E_i^j(\mathbf{x}_i^j) + \gamma \sum_{i=2}^{n_j} h(\mathbf{x}_i^j, \mathbf{x}_{i-1}^j). \tag{2}$$

The subscripts in these equations index time steps within sequences. The functional J^j is defined on the j^{th} sequence in the set of m sequences being learned. Each such functional is composed of a term that encodes the configurational constraints on individual actions, and a term that enforces smoothness. The parameter γ is a regularization parameter that weights the relative importance of meeting the



sequences are stored in the network architecture (eg trajectories) where they can be recalled via same input, or generalized via new input.

configurational constraints and obtaining smoothness.[3]

As is generally the case with connectionist algorithms, the optimization problem described above is to be solved in the context of a particular network architecture. Thus, the optimization problem includes the storage of the solution sequences in a network from which they can be recalled. This enriches the optimization framework by making it possible to discuss generalization between and within sequences, and the dynamics of the process of generating solution sequences.

Review of supervised learning

Although the ideas developed in this paper can be used in conjunction with any supervised learning algorithm in which an error vector is computed, the focus here is on the backpropagation algorithm as presented by Rumelhart, Hinton, & Williams (1986). Consider a feedforward network of computational units, each of which computes its activation z_i according to:

unit i activation; unit j activation; differentiable; network of units

$$z_i = \phi(\sum_j w_{ij} z_j),$$

where ϕ is a differentiable function and w_{ij} is the weight from unit j to unit i. A pair of non-overlapping subsets of the units are distinguished as input units and output units, and their activations are denoted by the vectors $\mathbf{q}$ and $\mathbf{x}$. The network can be described globally as computing a differentiable, parameterized function g of its input $\mathbf{q}$:

output units' activations; input units' activations; global description

$$\mathbf{x} = g(\mathbf{w}, \mathbf{q}),$$

where the form of g is determined by the activation function ϕ and the network connectivity pattern.

The backpropagation learning rule assumes the existence of a target vector $\mathbf{x}_i^*$ corresponding to each input vector $\mathbf{q}_i$. An error measure E_i is formed, based on the sum of squares of the differences between components of $\mathbf{x}_i^*$ and $\mathbf{x}_i = g(\mathbf{w}, \mathbf{q}_i)$:

error measure; target vector for input $\vec{q}_i$; output $g(\vec{w}, \vec{q}_i)$

$$E_i = \frac{1}{2}(\mathbf{x}_i^* - \mathbf{x}_i)^T(\mathbf{x}_i^* - \mathbf{x}_i).$$

Backpropagation is a gradient descent procedure that changes the parameters $\{w_{ij}\}$ according to the gradient of the error measure E_i, that is, it computes

gradient of E_i with respect to the weights; matrix of partial derivs of $g(\vec{w}, \vec{q})$ evaluated at current $\vec{w}, \vec{q}_i$

$$\nabla_{\mathbf{w}} E_i = -\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}^T (\mathbf{x}_i^* - \mathbf{x}_i),$$

[3] In the remainder of the paper, the smoothness function h is assumed to be a known quadratic form. Thus, the smoothness term can be handled easily by methods from the calculus of variations. However, as will be discussed, the configurational functions E_i are assumed in general to be nonlinear and unknown *a priori*, so that further effort is required to use variational methods.

where $\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}$ is the matrix of partial derivatives of $g(\mathbf{w}, \mathbf{q})$ taken with respect to $\mathbf{w}$, evaluated at $\mathbf{q} = \mathbf{q}_i$ and the current value of $\mathbf{w}$. If it is assumed that there are n input-output pairs, then the steepest descent rule changes the parameter values $\mathbf{w}$ as follows:

After every n input-output pairs, $\vec{q}_i, \vec{x}_i$, the "total" gradient for the sum of n errors is calculated, and the gradient is descended by rate α, via weight adjustments.

$$\Delta \mathbf{w} = -\alpha \nabla_{\mathbf{w}} \sum_{i=1}^{n} E_i \tag{3}$$

This can be expressed in terms of the individual I/O-pairs' gradients.

$$= -\alpha \sum_{i=1}^{n} \nabla_{\mathbf{w}} E_i \tag{4}$$

$$= \alpha \sum_{i=1}^{n} \frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}^T (\mathbf{x}_i^* - \mathbf{x}_i), \tag{5}$$

where α is a learning rate. In the remainder of the paper, the steepest descent learning rule shown in Equation 4 is assumed, and our only concern will be the computations of gradients.

One simple modification of standard backpropagation that will prove useful is to allow a symmetric, positive definite matrix Γ to appear in the error functional:

$$E_i = \frac{1}{2}(\mathbf{x}_i^* - \mathbf{x})^T \Gamma (\mathbf{x}_i^* - \mathbf{x}_i).$$

Modification: symmetric, positive-definite Γ is inserted: serves as a gain (good for noisy environments where error components are correlated)

The gradient of this error term is

Resulting gradient:

$$\nabla_{\mathbf{w}} E_i = -\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}^T \Gamma (\mathbf{x}_i^* - \mathbf{x}_i).$$

The matrix Γ is required to be positive definite so that the error E_i is zero only when $\mathbf{x}_i^* = \mathbf{x}_i$. Such a matrix will be generally useful in a noisy environment when there are correlations between the components of the error vector. Also, the diagonal elements of Γ can be thought of as individual gains associated with the components of the error.

Finally, it is worth noting that the backpropagation algorithm also computes $\nabla_{\mathbf{z}} E_i$, the vector of partial derivatives of the error with respect to the units' activations (the partial derivative $\frac{\partial E_i}{\partial z_j}$ corresponding to the j^{th} unit is used by the algorithm in computing the partials for the incoming weights w_{jk}). The error signals arriving at the input units are therefore given by

$$\nabla_{\mathbf{q}} E_i = \frac{\partial \mathbf{x}_i}{\partial \mathbf{q}}^T (\mathbf{x}_i^* - \mathbf{x}_i). \tag{6}$$

That is, for a fixed set of weights, the backpropagation algorithm is able to transform a vector by multiplying it by the transpose of the Jacobian matrix of the input-output mapping performed by the network. The algorithm computes this product

by implementing a particular factorization of the Jacobian matrix $\frac{\partial \mathbf{x}_i}{\partial \mathbf{q}}$. The structure of the algorithm can be seen clearly by writing this factorization explicitly:

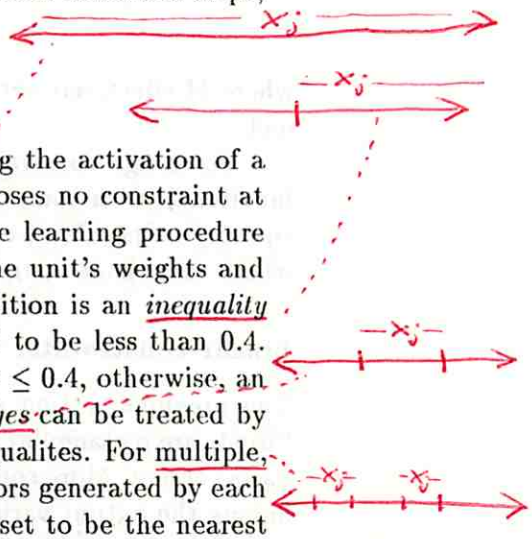
$$\nabla_{\mathbf{q}} E_i = W^{(1)T}\Lambda^{(1)}W^{(2)T}\Lambda^{(2)}...W^{(N)T}\Lambda^{(N)}(\mathbf{x}_i^* - \mathbf{x}_i), \tag{7}$$

where $W^{(k)}$ is the weight matrix linking the $k-1^{th}$ layer of units to the k^{th} layer, and $\Lambda^{(k)}$ is a diagonal matrix whose entries are the derivatives of the activation functions of the units in the k^{th} layer.[4] Note finally that the error vector $\mathbf{x}_i^* - \mathbf{x}_i$ can be replaced by any other vector, which may or may not depend on the output $\mathbf{x}_i$. In the latter case, due to the fact that the Jacobian matrix varies as a function of the input $\mathbf{q}$, it is still necessary for activation to flow forward in the network in order to instantiate a particular Jacobian matrix.[5]

Generalizing supervised learning

Configurational constraints

A broad class of configurational constraints can be obtained by generalizing the process of computing an error vector. The discussion here proceeds in several steps, after which I discuss temporal constraints.



Don't-care conditions, inequalities, and ranges.

This section describes some simple techniques for constraining the activation of a unit to lie in a specified interval. A *don't-care condition* imposes no constraint at all—the interval specified is the entire real axis. Clearly, the learning procedure for a don't-care condition should be to make no change to the unit's weights and to propagate no error from the unit. A more restrictive condition is an *inequality constraint*. For example, the activation x_j might be required to be less than 0.4. In such a case, the learning rule should generate no error if $x_j \leq 0.4$, otherwise, an error $0.4 - x_j$ is generated as if 0.4 were a target value. *Ranges* can be treated by summing the errors produced by the corresponding pair of inequalites. For multiple, disjoint intervals, there are several possibilities, e.g., 1) the errors generated by each pair of inequalities can be summed, or 2) the target value is set to be the nearest interval boundary, for activations not already in some target interval.

These ideas can be easily translated into an error functional form. Let S_1 be the set of indices i such that the i^{th} component of $\mathbf{x}^*$ has a don't-care condition.[6] Furthermore, let S_2 be the set of indices corresponding to the "less-than" inequalities

[4] Here and elsewhere in the paper, expressions for gradients should be read from right to left to correspond to the order in which backpropagation computes the matrix products.

[5] As seen in Equation 7, the only variables in the process by which backpropagation computes the Jacobian transpose are the derivatives of the activation functions ϕ.

[6] In the following, we drop the subscript i denoting the stimulus number, in the interest of clarity. The symbol x_i represents the i^{th} component of $\mathbf{x}$.

and S_3 be the set of indices corresponding to the "greater-than" inequalities. Then the error functional E is given by

$$E = \frac{1}{2}(\mathbf{x}^* - \mathbf{x})^T \Gamma \mathbf{H}(\mathbf{x}^* - \mathbf{x}),$$

where $\mathbf{H}$ is a diagonal matrix defined as a function of $\mathbf{x}$ in the following way. Let

$$h_{ii} = \begin{cases} 0 & \text{if } i \in S_1 \\ 0 & \text{if } i \in S_2 \text{ and } x_i \leq x_i^* \\ 0 & \text{if } i \in S_3 \text{ and } x_i \geq x_i^* \\ 1 & \text{otherwise.} \end{cases}$$

Differentiation of E with respect to $\mathbf{w}$ yields

$$\nabla_{\mathbf{w}} E = -\frac{\partial \mathbf{x}}{\partial \mathbf{w}}^T \Gamma \mathbf{H}(\mathbf{x}^* - \mathbf{x}),$$

where $\mathbf{H}$ effectively zeroes the components of $\mathbf{x}^* - \mathbf{x}$ where constraints are already met.[7]

For range constraints, we simply include a second target vector in the error functional, with components given by the second member of the corresponding inequality pairs. Don't-care conditions are used to fill out the vector for components where there is no range constraint.

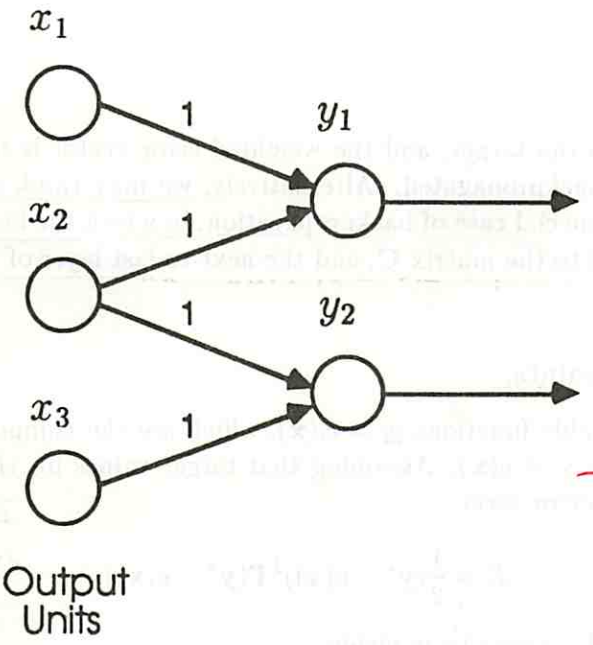
Linear constraints.

The previous section shows how to adapt supervised learning to cases in which targets are rectangular regions in the output space with sides parallel to the standard basis vectors. More complex regions can be represented by linear constraints defined among the output variables.

Consider a network with three output units having activations x_1, x_2, and x_3. Suppose that there are a given pair of constraints of the form $x_1 + x_2 = y_1^*$ and $x_2 + x_3 = y_2^*$, for particular values y_1^* and y_2^*. As shown in Figure 1, we can think of these constraints as being implemented by a pair of linear units[8] adjoined to the output units. These extra units have activations $y_1 = x_1 + x_2$, and $y_2 = x_2 + x_3$,

[7] It should be noted that $\mathbf{H}$ is a discontinuous function of $\mathbf{w}$. However, this doesn't pose difficulties in differentiation because the set on which the derivative is undefined is precisely the set where the corresponding components of $\mathbf{x}$ are equal to $\mathbf{x}^*$. For this set, we therefore have $E = 0$. In the neighborhood of this set, E is quadratic or identically zero; in either case, the left and right derivatives are zero. Thus, E is pointwise continuous and differentiable, even if $\mathbf{H}$ is not.

[8] A linear unit is a unit for which the activation function ϕ is the identity function.



Linear constraints $x_1 + x_2 = y_1^*$ and $x_2 + x_3 = y_2^*$ are implemented by introducing extra output units y_1, y_2 with fixed-weighted inputs from x_1, x_2, x_3. Target values y_1^*, y_2^* are then applied to units y_1, y_2.

Figure 1: Implementing linear constraints.

by virtue of the fixed connections of unity from the output units. Target values are applied to these units rather than directly to the output units. For example, if y_1 is different from y_1^*, then an error $y_1^* - y_1$ is generated. It is natural to apportion this error equally to the units x_1 and x_2, from whence it is backpropagated as usual. All else being equal, this process will work to find a solution to $y_1^* = x_1 + x_2$ in which x_1 and x_2 are equal. However, the constraint $y_2^* = x_2 + x_3$ imposes further conditions on x_2, so that the value actually chosen will be a compromise.

In general, we consider a collection of k constraint functions c_i such that $y_i = c_i(\mathbf{x}) = \mathbf{c}_i^T\mathbf{x}, i = 1, 2, ..., k$ for some set of vectors $\{\mathbf{c}_i\}$. Letting $\mathbf{C}$ denote the matrix whose rows are the vectors $\mathbf{c}_i$, we have $\mathbf{y} = \mathbf{Cx}$. Now suppose that it is known what the value $\mathbf{y}^*$ of the constraint functions should be.[9] Form the error functional

$$E = \frac{1}{2}(\mathbf{y}^* - \mathbf{Cx})^T\mathbf{\Gamma}(\mathbf{y}^* - \mathbf{Cx}).$$

($\mathbf{y}^* - \mathbf{Cx}$: constraint errors)

General form for linear constraint implementation:

This functional can be differentiated with respect to the parameters $\mathbf{w}$, giving

$$\begin{aligned}\nabla_{\mathbf{w}}E &= -(\frac{\partial}{\partial \mathbf{w}}\mathbf{Cx})^T\mathbf{\Gamma}(\mathbf{y}^* - \mathbf{Cx}) \\ &= -\frac{\partial \mathbf{x}}{\partial \mathbf{w}}^T\mathbf{C}^T\mathbf{\Gamma}(\mathbf{y}^* - \mathbf{Cx}).\end{aligned}$$

Constraints are expressed as $C\vec{x} = \vec{y}^*$

The gradient descent learning rule is therefore a simple modification of backpropagation, in which the output vector is multiplied by a constraint matrix $\mathbf{C}$ before

[9] Inequalities are discussed below. Note also that a constraint such as $x_i = x_j$ is rewritten as $x_i - x_j = 0$, a linear constraint with a target value of zero.

being compared to the target, and the weighted error vector is first multiplied by $\mathbf{C}^T$ before being backpropagated. Alternatively, we may think of this rule being implemented as a special case of backpropagation, in which the last layer of weights are fixed according to the matrix $\mathbf{C}$, and the next-to-last layer of units are actually the output units.

Nonlinear constraints.

Consider differentiable functions $y_i = c_i(\mathbf{x})$, which are the components of a vector constraint function $\mathbf{y} = \mathbf{c}(\mathbf{x})$. Assuming that target values for the constraints are given, we form an error term

$$E = \frac{1}{2}(\mathbf{y}^* - \mathbf{c}(\mathbf{x}))^T \Gamma (\mathbf{y}^* - \mathbf{c}(\mathbf{x})).$$

Differentiating with respect to $\mathbf{w}$ yields

$$\begin{aligned} \nabla_{\mathbf{w}} E &= -\frac{\partial}{\partial \mathbf{w}} \mathbf{c}(\mathbf{x})^T \Gamma (\mathbf{y}^* - \mathbf{c}(\mathbf{x})) \\ &= -\frac{\partial \mathbf{x}}{\partial \mathbf{w}}^T \frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T \Gamma (\mathbf{y}^* - \mathbf{c}(\mathbf{x})). \end{aligned} \tag{8}$$

This equation differs from the linear case in that $\mathbf{x}$ is transformed nonlinearly before comparison with the target, and the weighted error vector is first multiplied by the transpose of a matrix that is no longer constant, but depends on the value of $\mathbf{x}$. This matrix, $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$, is the Jacobian matrix of the constraint function $\mathbf{c}(\mathbf{x})$.

There are two types of situations in which Equation 8 can be used. First, it can be used in situations where a priori knowledge of the form of the constraints allows the representation in closed form of the Jacobian matrix. For example, in robotics, the function $\mathbf{c}(\mathbf{x})$ may represent the forward kinematics of the manipulator, in which case the Jacobian matrix is typically easy to derive by differentiation. Given this interpretation of the constraint function, Equation 8 is in fact closely related to a control technique known as "stiffness control," a member of a class of techniques for controlling particular forms of dynamic interactions between a manipulator and its environment (Hogan, 1985; Salisbury, 1980).[10]

A second type of situation is one in which no a priori knowledge of the form of the constraints is given, but a black-box which implements the function $\mathbf{y} = \mathbf{c}(\mathbf{x})$ is available. As suggested by the linear case in Figure 1, it is possible to implement nonlinear constraints by constructing a layered network that is adjoined to the output units of the network being trained, taking $\mathbf{x}$ as input and producing a vector $\mathbf{y}'$ as output. The weights in the adjoined network are set so that the network acts as a "forward model" of the unknown function $\mathbf{c}(\mathbf{x})$. This is done by randomly

[10] We need only give Γ the interpretation of a stiffness matrix to obtain stiffness control, if we neglect the first matrix of the equation, thereby treating the equation as a real-time control rule for $\mathbf{x}$, rather than as a learning rule for $\mathbf{w}$ (actually, both can be done simultaneously).

the space of vectors $\mathbf{x}$, noting the black-box response $\mathbf{y}$ for each random choice of $\mathbf{x}$, and treating $\mathbf{x}$ as an input and $\mathbf{y}$ as a target for the model network.[11] The weights are changed according to the gradient of the error measure

$$E_m = \frac{1}{2}(\mathbf{y} - \mathbf{y}')^T(\mathbf{y} - \mathbf{y}'). \tag{9}$$

Eventually, the network comes to serve as an approximate model $\mathbf{c}'(\mathbf{x})$ of the true constraint function $\mathbf{c}(\mathbf{x})$. As discussed in conjunction with Equation 6, the model network can then be used to multiply a vector $\mathbf{y}^* - \mathbf{c}(\mathbf{x})$ by the matrix $\frac{\partial \mathbf{y}'}{\partial \mathbf{x}}^T$, which is an approximation to the true transpose Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T$. Thus, the term $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T \mathbf{\Gamma}(\mathbf{y}^* - \mathbf{c}(\mathbf{x}))$ in Equation 8 can be approximated by $\frac{\partial \mathbf{y}'}{\partial \mathbf{x}}^T \mathbf{\Gamma}(\mathbf{y}^* - \mathbf{c}(\mathbf{x}))$, by propagating $\mathbf{\Gamma}(\mathbf{y}^* - \mathbf{c}(\mathbf{x}))$ back through the model network.[12] Note that it is not necessary to approximate $\mathbf{c}(\mathbf{x})$ by $\mathbf{c}'(\mathbf{x})$, because we have assumed that the environmental response $\mathbf{c}(\mathbf{x})$ is available. Note finally that the backpropagated vector $\mathbf{y}^* - \mathbf{c}(\mathbf{x})$ does not depend on the output $\mathbf{y}'$ of the model network. However, as discussed previously, $\mathbf{y}'$ must still be computed in order to instantiate the correct Jacobian matrix in the model network.

To summarize, when the form of the constraints are known, then the necessary Jacobian matrix can be computed separately or, in the linear case, as part of the network computation after setting the weights in an adjoined matrix. When the constraints are unknown, then their form can first be learned in an adjoined model network, which then provides the transpose Jacobian through backpropagation.

Linear and nonlinear inequalities.

It is possible to combine the ideas of the previous sections by allowing don't-care conditions and inequalities to refer to the elements of the $\mathbf{y}^*$ vector. Thus, a general form of the error functional for configurational constraints is given by

$$E = \frac{1}{2}(\mathbf{y}^* - \mathbf{c}(\mathbf{x}))^T \mathbf{\Gamma H}(\mathbf{y}^* - \mathbf{c}(\mathbf{x})), \tag{10}$$

where $\mathbf{H}$ is a diagonal matrix given by

$$h_{ii} = \begin{cases} 0 & \text{if } i \in S_1 \\ 0 & \text{if } i \in S_2 \text{ and } c_i(\mathbf{x}) \leq y_i^* \\ 0 & \text{if } i \in S_3 \text{ and } c_i(\mathbf{x}) \geq y_i^* \\ 1 & \text{otherwise,} \end{cases}$$

[11] The standard terminology for such a modeling process is *system identification* (Eykhoff, 1974).

[12] For further discussion on using a model network in this way, including its possible role in a theory of sensorimotor learning, see Rumelhart & Jordan (1988).

Implementing optimality as a constraint:
Express the function to be minimized as a possibly-unattainable low value.
eg: to minimize $c_i(\vec{x}) = \vec{x}^T\vec{x}$:
set up inequality $c_i(\vec{x}) = 0$

then gradually decrease the constraint's gain Γ_i so interaction with other constraints won't produce "false" minimum

where S_1, S_2, and S_3 are sets containing the indices for the don't-care conditions, the "less-than" inequalities, and the "greater-than" inequalities, respectively.[13]

Other optimality criteria on x.

As noted in the introduction, it is straightforward at this point to introduce further optimality criteria given as functions of $\mathbf{x}$. Such functions simply appear among the functions c_i. For example, a minimum norm criterion would appear as $c_i(\mathbf{x}) = \mathbf{x}^T\mathbf{x}$, where the target value for this constraint would be zero. This type of constraint differs from those considered earlier, however, in that the error is not normally assumed to go to zero, and thus a gradient will always be generated. To prevent a local minimum in which this gradient is matched (in the opposite direction) by a gradient from those error components that should go to zero, the system should decrease the gain of the optimality constraint (the i^{th} column of the matrix Γ) to zero over iterations.

Conclusions. Summary:

eg: learn joint angles $\vec{x}$ for various situations $\vec{q}$ such that $\vec{x}$ are described in endpoint coordinates as $\vec{y} = \vec{c}(\vec{x})$

The approach described in this section generalizes the process of computing an error vector for supervised learning algorithms. Rather than forming an error vector $\Delta\mathbf{x}$ by direct comparison of the output $\mathbf{x}$ with some target $\mathbf{x}^*$, the proposal is to compute a term $\Gamma\mathbf{H}(\mathbf{y}^* - \mathbf{c}(\mathbf{x}))$ that measures how far the current output is from meeting a set of constraints, and then to convert this term into an error vector $\Delta\mathbf{x}$ by multiplying by the transpose of the Jacobian matrix of the constraint function. It is worth pointing out that the same algorithm applies whether or not there are excess degrees of freedom (excess degrees of freedom arise from a many-to-one constraint function, or zero diagonal entries in $\mathbf{H}$).

When there are excess degrees of freedom, the configurational constraints associated with a particular input vector specify a region in which the image of the vector must lie. The excess degrees of freedom are resolved in a choice of a particular output vector by factors that influence the path taken toward the region on learning trials for that input. The major such factor is generalization from the constraints associated with other input vectors. That is, for input vectors that are nearby in the input space, the paths taken toward the corresponding regions in the output space are interdependent, via the shared weights of the network. This is an expanded role for generalization, beyond its normal role as an interpolative mechanism.

Excess dofs are resolved by either:
(1) letting the system choose how to fill in the excess dofs by itself
or
(2) imposing optimality criteria (in the form of constraints like $\vec{x}^T\vec{x} = 0$)

Another approach to resolving excess degrees of freedom is to include other classes of explicit constraints in the optimization functional, by expanding the scope of the optimization problem. The temporal constraints discussed in the next section are illustrative of this approach.

[13] Note that this formulation includes cases of don't-care conditions and inequalities defined directly on the output $\mathbf{x}$; certain of the functions c_i may simply pick out particular elements from $\mathbf{x}$.

Sequential imitation problem: (j = sequence #)
Have target sequences $\vec{y}_1^{j*}, \vec{y}_2^{j*}, \ldots, \vec{y}_n^{j*}$ (eg: endpoint path points)
and want to learn corresponding
articulatory sequences $\vec{x}_1^{j*}, \vec{x}_2^{j*}, \ldots, \vec{x}_n^{j*}$ subject to { sequences of: configuration constraints $\vec{y}_i = \vec{c}(\vec{x}_i^j)$; don't care conditions & inequalities H_i^j

Temporal constraints

Terminology.

The following terminology will be adopted for the remainder of the paper.

The outputs $\mathbf{x}$ are members of a vector space referred to as *articulatory space*. Articulatory space is mapped into *target space* by a many-to-one *constraint function* $\mathbf{c}(\mathbf{x})$. The vectors $\mathbf{y}$, $\mathbf{y}'$, and $\mathbf{y}^*$ are all members of target space.

To solve a *sequential imitation* problem, a system must take as input target sequences $\mathbf{y}_1^{j*}, \mathbf{y}_2^{j*}, \ldots, \mathbf{y}_{n_j}^{j*}$, for each j, including specifications as to which vector components are to be treated as inequalities or don't-care conditions, and produce as output articulatory sequences $\mathbf{x}_1^j, \mathbf{x}_2^j, \ldots, \mathbf{x}_{n_j}{}^j$, such that $\mathbf{x}_i^j \in R_i^{j*}$, for all i and for all j, where R_i^{j*} denotes the region in articulatory space where the quadratic form $(\mathbf{y}_i^{j*} - \mathbf{c}(\mathbf{x}_i^j))^T \Gamma \mathbf{H}_i^j (\mathbf{y}_i^{j*} - \mathbf{c}(\mathbf{x}_i^j))$ is minimized.

Temporal constraints (eg: smoothness) arise from the system's dynamical properties

Characterization of temporal constraints.

Temporal constraints appear as functions on a sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$.[14] For example, it might be required that $(\mathbf{x}_i - \mathbf{x}_{i-1})^T(\mathbf{x}_i - \mathbf{x}_{i-1}) < d$, for all i, and some scalar d. One approach to implementing constraints of this form is to treat them as configurational constraints and apply the techniques of previous sections. This can be done by juxtaposing the actions $\mathbf{x}_i$ in an output vector $< \mathbf{x}_1{}^T | \mathbf{x}_2{}^T | \ldots | \mathbf{x}_n{}^T >^T$, which represents the entire sequence, with each action being stored in a spatially distinct buffer indexed by sequential position. However, there are characteristic differences between configurational constraints and temporal constraints that this approach ignores.[15] First, it is natural for there to be a fixed number of components k in the articulatory vector $\mathbf{x}$ over which configurational constraints are defined. However, for temporal constraints and sequences, it is more natural to allow the number of actions n to be arbitrary. Second, configurational constraints can occur between any two components x_i and x_j, whereas temporal constraints tend to involve actions that are proximal in time. As will be seen, this allows a different implementation strategy for temporal constraints. Third, configurational constraints typically arise from fixed structural relations in the effector system and/or environment (this was modeled in previous sections by assuming a fixed form for the constraint function $\mathbf{c}(\mathbf{x})$). Temporal constraints ordinarily do not have such an interpretation; rather, they arise from dynamical properties of systems. Furthermore, treating temporal constraints structurally by storing actions in spatial buffers makes the implementation of configurational constraints more difficult. The (learned) constraint function $\mathbf{c}(\mathbf{x})$ must be copied across buffer locations, and learning at any one position must generalize to other positions. Finally, it is quite possible to have configurational

[14] In what follows, the sequence superscript j is dropped for readability.

[15] The following remarks can be taken as definitional for the terms "configurational" and "temporal."

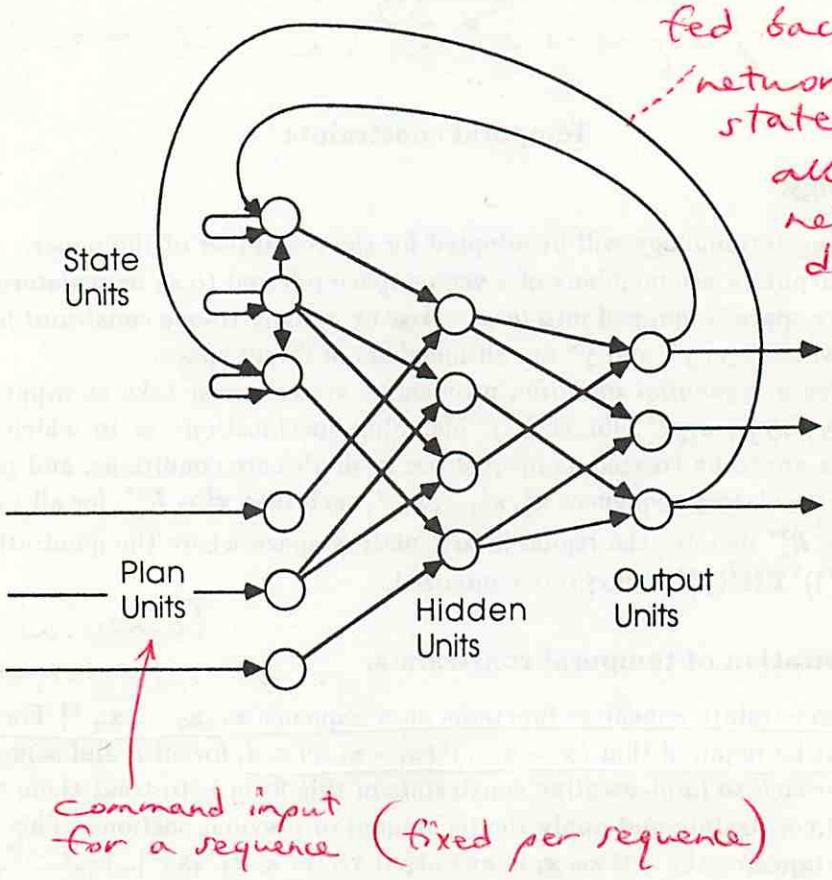


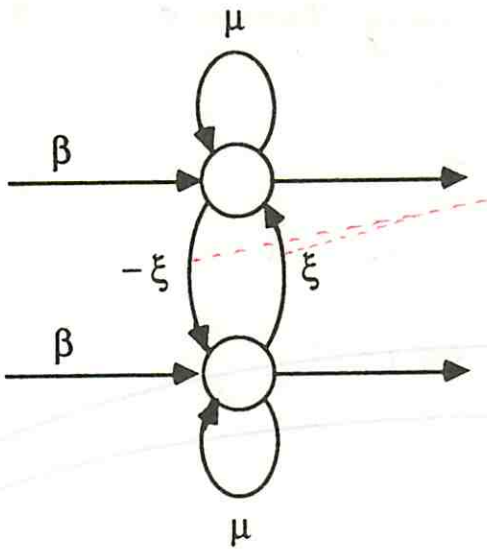
Figure 2: A sequential network.

constraints apply across time without using spatial buffers. This happens when some of the output variables x_i are actually the time derivatives $\dot{v}_i$ of underlying variables v_i. Indeed, it may often be sensible to have networks for both $\mathbf{v}$ and $\dot{\mathbf{v}}$.

Sequential networks.

The preceding discussion suggests that in order to implement configurational constraints with efficiency and generality, it is preferable to have a single output position at which actions appear dynamically in time. Configurational constraints can then be handled with a single model network which is connected to this output position. This architectural constraint also implies that temporal constraints are to be treated by comparing values produced at different moments in time but at the same physical location.

A network with the required architecture is the "sequential network" depicted in Figure 2 (Jordan, 1986a). This network is able to produce sequences of actions as dynamically changing patterns of activation across the output units, due to the recurrent connections arriving at the *state units*. These connections cause the activations of the state units to change, thereby providing a time-varying input to the layered network which learns the action sequences. The total input vector $\mathbf{q}$ also



Cross-connections between the state units cause internal oscillatory behavior, which makes it easier to learn repetitive sequences.

Figure 3: State units and their interconnections.

contains the activation pattern on the *plan units*, which are fed by the external input to the network. The plan stays constant within a sequence, but varies between sequences to allow different sequences to be learned by the same network. The network is a dynamical system in which both the output function and the next-state function change as the weights in the forward path of the network are modified through learning.

A particular interconnection scheme for the state units is shown in Figure 3. (This is the scheme used for the simulations reported later in the paper; all but one of the simulations used networks with only two state units). These state units are linear, with fixed cross-connections of ξ and $-\xi$, and fixed auto-connections of μ. It is easy to show that this linear system is a second-order filter with eigenvalues $\mu \pm i\xi$. It therefore oscillates with a frequency determined by the phase angle $tan^{-1}\frac{\xi}{\mu}$ of the eigenvalues. The state units are also driven by the recurrent connections (with fixed weights of β) from the output units.

In previous study of this type of network (Jordan, 1986a), there were no cross-connections between the state units (the state units constituted a first-order filter). Such a network could also have been used for the simulations reported here; however, these simulations all involve repetitive sequences, which are more easily learned if the state itself has internal oscillatory behavior.

Figure 4 shows the sequential network conjoined with a model network. The articulatory units are the output units of the combined network and encode articulatory space. The target units encode target space and provide input to the state units. It is also possible (and in some cases preferable) for the state units to receive

Sequential network plus facility for learning forward model $\vec{y} = c(\vec{x})$

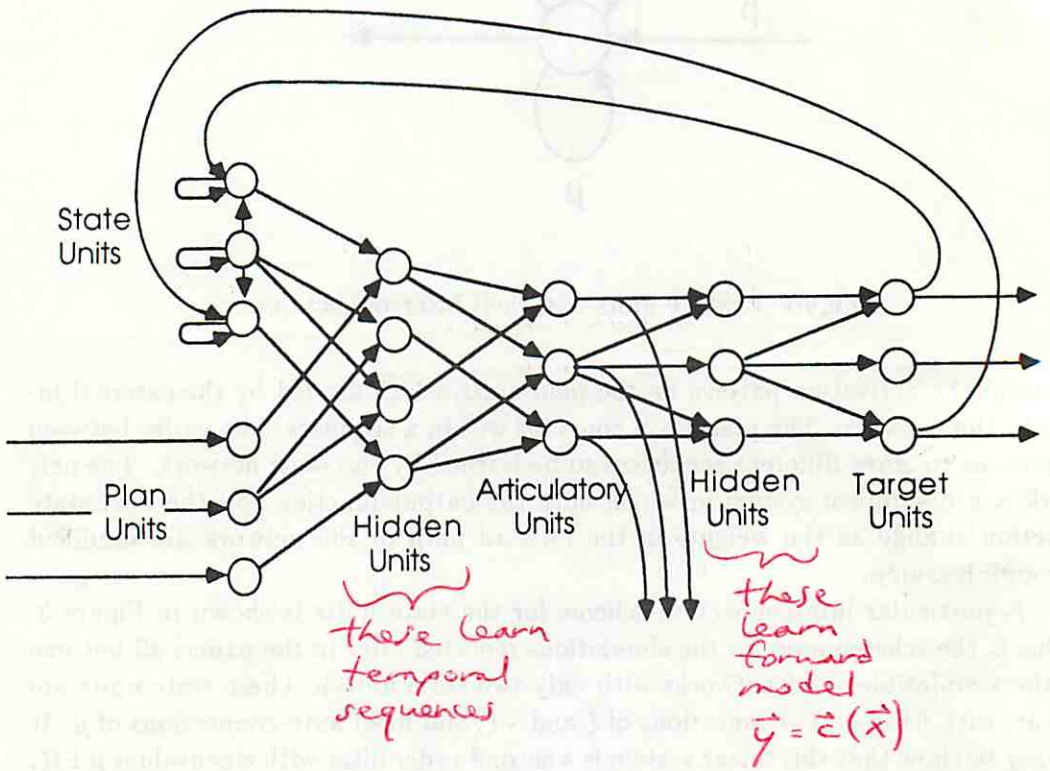


Figure 4: The sequential network conjoined with a model network.

input from the articulatory units as well.

A smoothness constraint.

Suppose that the configurational constraints on a sequential network having three articulatory units are of the following form at times i and $i+1$:

$$\left\{ \begin{array}{c} x_1 + x_2 = 1.6 \\ x_3 = * \end{array} \right\}, \left\{ \begin{array}{c} x_1 = * \\ x_2 + x_3 = 0.4 \end{array} \right\}.$$

configurational constraint sequence over articulatory units x_1, x_2, x_3

We also require $0 \leq x_k \leq 1$, for all k. Of the infinite number of solution sequences, the following solution is "best," if it is desired to perform the sequence smoothly and quickly:[16]

$$\left[\begin{array}{c} 1.0 \\ 0.6 \\ 0.0 \end{array} \right], \left[\begin{array}{c} 1.0 \\ 0.4 \\ 0.0 \end{array} \right].$$

"Best" solution sequence, in terms of smoothness. — activations change as little as possible but as much as necessary

For this solution, x_2 is required to change by only 0.2 units between time steps. The choices of $x_2 = 0.6$ and $x_2 = 0.4$ force $x_1 = 1.0$ at time step i, and $x_3 = 0.0$ at $i+1$. The don't-care condition on x_3 at i allows the value 0.0 to be anticipated, so that no change need occur for x_3 between time steps. Similarly, the don't-care condition on x_1 at $i+1$ allows the value 1.0 to spread forward in time.

One approach to obtaining solutions of this kind is to enforce a smoothness constraint, such that the values of the components of $\mathbf{x}_i$ change as little as possible over time. A simple smoothness constraint is achieved by minimizing a functional of the form

$$I = \frac{1}{2} \sum_{i=2}^{n} (\mathbf{x}_i - \mathbf{x}_{i-1})^T (\mathbf{x}_i - \mathbf{x}_{i-1}),$$

Smoothness function: (to minimize) $\frac{1}{2}$ squared distance between successive $\vec{x}$'s

where each term is one-half the squared Euclidean distance between successive articulatory vectors. Differentiating these terms with respect to the weights $\mathbf{w}$ yields

Gradient with resp to weights:

$$\begin{aligned} \nabla_{\mathbf{w}} I &= \sum_{i=2}^{n} \left(\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}} - \frac{\partial \mathbf{x}_{i-1}}{\partial \mathbf{w}} \right)^T (\mathbf{x}_i - \mathbf{x}_{i-1}) \\ &= -\sum_{i=2}^{n} \left\{ \frac{\partial \mathbf{x}_{i-1}}{\partial \mathbf{w}}^T (\mathbf{x}_i - \mathbf{x}_{i-1}) + \frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}^T (\mathbf{x}_{i-1} - \mathbf{x}_i) \right\}, \end{aligned} \tag{11}$$

where $\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}$ depends both on the current input vector $\mathbf{q}_i$ and the current value of $\mathbf{w}$. Since $\mathbf{w}$ does not change until an entire pass through the set of sequences has

[16] It is assumed that it takes longer to move an articulator further.

weights are learned in the context of configuration constraint sequences $\vec{y}_i^{\,i} = \vec{c}_i^{\,i}(\vec{x}_i^{\,i})$ $(i = 1 \ldots n)$

been made, the matrices $\frac{\partial \mathbf{x}_i}{\partial \mathbf{w}}$ and $\frac{\partial \mathbf{x}_{i-1}}{\partial \mathbf{w}}$ are evaluated at a common value of $\mathbf{w}$, and a simple implementation of Equation 11 is therefore possible. We assume that each unit in the sequential network stores the value of its activation at the previous time step. The second term in the equation is computed simply by backpropagating the error vector $\mathbf{x}_{i-1} - \mathbf{x}_i$. To compute the first term in the equation, each unit temporarily restores the previous value of its activation, and the error vector $\mathbf{x}_i - \mathbf{x}_{i-1}$ is backpropagated.[17]

The smoothness constraint is similar to the temporal difference methods studied by Sutton (1987). In fact, Equation 11 is essentially the same as Sutton's TD(0) algorithm, but proceeding backward and forward in time simultaneously. Also, in Sutton's algorithm, there is an ultimate value to be predicted forward in time; this grounds the algorithm. In the current case, there is no ultimate value to be predicted in either direction. The algorithm is grounded by the configurational constraints, which prevent the network from minimizing I trivially by choosing $\mathbf{x}_1 = \mathbf{x}_2 = \ldots = \mathbf{x}_n$.

Other temporal constraints.

By adapting the techniques discussed previously for configurational constraints, it is possible to minimize functionals of the form

One way of combining configuration & temporal constraints

$$I = \frac{1}{2}\sum_{i=2}^{n}(\mathbf{d}(\mathbf{x}_i) - \mathbf{d}(\mathbf{x}_{i-1}))^T \mathbf{\Lambda}\mathbf{G}(\mathbf{d}(\mathbf{x}_i) - \mathbf{d}(\mathbf{x}_{i-1})),$$

Configuration constraints are expressed as $\vec{y} = \vec{d}(\vec{x})$ and temporal smoothness is enforced in the target (eg: endpoint) domain.

where the function $\mathbf{d}$ allows successive vectors to be compared in a domain other than the articulatory domain, and the matrices $\mathbf{\Lambda}$ and $\mathbf{G}$ allow don't-care conditions, inequalities, and gain parameters.

Another class of temporal constraints allows comparisons between vectors at three or more time steps. Although this generalization is conceptually straightforward, the implementation becomes rather more complex. A better approach to using higher-order temporal constraints may be to make first-order comparisons in networks whose activations have an interpretation as the velocities or accelerations of some underlying system.

Combining configurational and temporal constraints.

The smoothness constraint is clearly of interest only when it is combined with configurational constraints that specify different regions of articulatory space at different times. We therefore consider the functional J of Equation 1 which is the sum of functionals J^j that combine configurational and temporal constraints. For the j^{th}

[17] It is actually the derivative of ϕ that must be restored, but for a logistic function, the derivative is a function of the activation: $\phi' = z(1 - z)$.

Combining configuration & temporal constraints expressed in the articulatory domain:

(via config constraints $\vec{y} = \vec{c}(\vec{x})$)

For learning articulation sequence $\vec{x}_1^j, \vec{x}_2^j, \ldots, \vec{x}_n^j$, expressed as target sequence $\vec{y}_1^{j*}, \vec{y}_2^{j*}, \ldots, \vec{y}_n^{j*}$

For learning smoothness of transitions from $\vec{x}_i^j$ to $\vec{x}_{i+1}^j$

(states relative importance)

sequence, J^j is given by

$$J^j = \frac{1}{2}\{\sum_{i=1}^{n_j}(\mathbf{y}_i^{j*}-\mathbf{c}(\mathbf{x}_i^j))^T\mathbf{\Gamma}\mathbf{H}_i^j(\mathbf{y}_i^{j*}-\mathbf{c}(\mathbf{x}_i^j))+\lambda\sum_{i=2}^{n_j}(\mathbf{x}_i^j-\mathbf{x}_{i-1}^j)^T\mathbf{\Lambda}\mathbf{G}_i^j(\mathbf{x}_i^j-\mathbf{x}_{i-1}^j)\}. \quad (12)$$

Upon differentiating with respect to w, it is found that terms from the first and second summation combine, yielding

Resulting gradient:

$$\begin{aligned}\nabla_{\mathbf{w}}J^j = & - \frac{\partial \mathbf{x}_1^j}{\partial \mathbf{w}}^T \frac{\partial \mathbf{y}_1^j}{\partial \mathbf{x}_1^j}^T \mathbf{\Gamma}\mathbf{H}_1^j(\mathbf{y}_1^{j*} - \mathbf{c}(\mathbf{x}_1^j)) \\ & - \sum_{i=2}^{n_j} \frac{\partial \mathbf{x}_i^j}{\partial \mathbf{w}}^T (\frac{\partial \mathbf{y}_i^j}{\partial \mathbf{x}_i^j}^T \mathbf{\Gamma}\mathbf{H}_i^j(\mathbf{y}_i^{j*} - \mathbf{c}(\mathbf{x}_i^j)) + \lambda\mathbf{\Lambda}\mathbf{G}_i^j(\mathbf{x}_{i-1}^j - \mathbf{x}_i^j)) \\ & - \lambda\sum_{i=2}^{n_j} \frac{\partial \mathbf{x}_{i-1}^j}{\partial \mathbf{w}}^T \mathbf{\Lambda}\mathbf{G}_i^j(\mathbf{x}_i^j - \mathbf{x}_{i-1}^j). \quad (13)\end{aligned}$$

For $i = 1$, only configurational constraints are computed. For $i > 1$, terms in the first summation are computed by a single backpropagation pass, in which $\mathbf{\Gamma}\mathbf{H}_i^j(\mathbf{y}_i^{j*} - \mathbf{c}(\mathbf{x}_i^j))$ is propagated through the model network, yielding a transformed error to which $\lambda\mathbf{\Lambda}\mathbf{G}_i^j(\mathbf{x}_{i-1}^j - \mathbf{x}_i^j)$ is added at the articulatory units, with the resulting sum being propagated into the sequential network. Terms in the second summation are computed within the sequential network alone, using restored activation values as discussed in the previous section.

The smoothness term should not normally reach zero, thus the parameter λ must go to zero over iterations, so that a positive smoothness term does not prevent the configurational constraints from being achieved. In the simulations reported in the next section, this was done by defining λ to be proportional to the total error in the configurational constraint term.

So smoothness term doesn't "balance out" configuration constraints term for a total of 0-gradient.

Geometrically, the optimization problem posed in Equation 12 is depicted in Figure 5. The regions R_i^{j*} are defined implicitly as minima of the configurational constraint terms. These regions may overlap if there are don't-care conditions or inequalities. The system must find a trajectory that passes through the regions in the correct order. There are many such trajectories; the smoothness constraint defines particular trajectories with minimal variation. To find such trajectories, we have proposed using gradient descent based on Equation 13. It is worth noting that this approach modifies candidate trajectories indirectly, by changing the parameters of an underlying dynamical system (the sequential network). That is, it is more accurate to say that the learning rule modifies the vector field of a dynamical system, rather than modifying particular stored trajectories. As suggested by Figure 5, this has implications for the region of articulatory space in the neighborhood of a learned trajectory, due to the continuity of the vector field.

Gradient descent in J is susceptible to problems with local minima as are all gradient descent procedures. With respect to the sequential imitation problem, i.e.,

Geometrical representation:

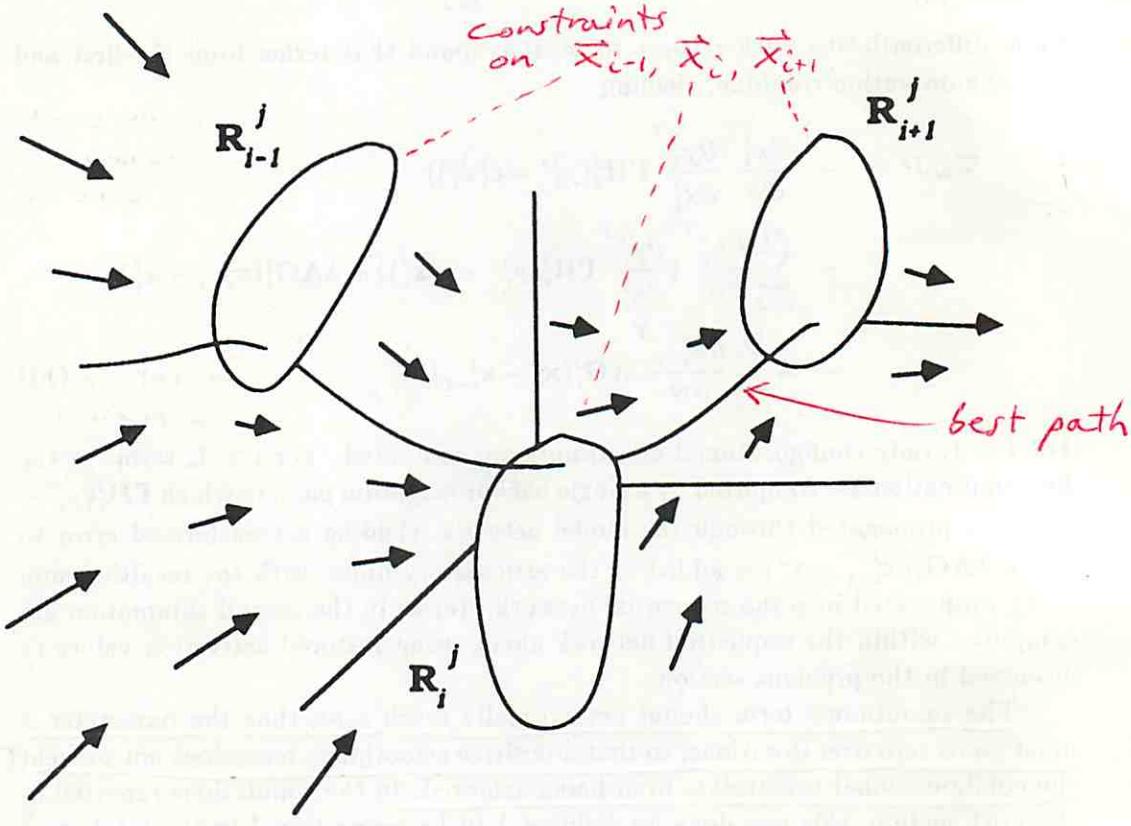


Figure 5: A geometrical representation of the sequential imitation problem. The vector field defines the dynamics of the sequential network that generates candidate trajectories.

the problem of finding sequences that meet the configurational constraints, local minima have been found to be less of a problem in practice than is the case with standard backpropagation. The smoothness constraint links the steps taken for each $\mathbf{x}_i$ with respect to the error in the configurational constraints; thus, for the system to be in a local minimum, it must simultaneously be in local minima for each of these searches conceived separately. Also, excess degrees of freedom in the mapping $\mathbf{c}(\mathbf{x})$ decrease the chances of reaching local minima. This is because excess degrees of freedom increase the number of rows in the transpose Jacobian matrix, thereby generally decreasing the size of its nullspace. On the other hand, excess degrees of freedom greatly increase the size of the solution set, so that there are a large number of local minima with respect to the smoothness constraint. In simulations with the planar manipulators to be discussed in the next section, it has often been found that solutions in distant parts of articulatory space have similar smoothness values. Whether or not this is a problem depends on the application. Further constraints can be added if certain canonical solutions are to be preferred. For example, a term $\gamma \sum_{i=1}^{n_J} (\mathbf{x}_i^j - \mathbf{x}^{(0)})^T (\mathbf{x}_i^j - \mathbf{x}^{(0)})$ can be added to J where $\mathbf{x}^{(0)}$ is a canonical "low-energy" position. When large initial values of λ and γ are used, the system will be biased to finding solutions such that all articulatory vectors are close to $\mathbf{x}^{(0)}$.

Simulations

In this section, I present the results of a series of simulation experiments which demonstrate the computational viability of the approach and serve to illustrate several of its properties.

Architectures, environments, tasks, and procedures

The domain chosen for the simulation experiments is that of planar manipulators, the task being to make specified sequences of "button presses" in the plane. As shown in Figure 6, the first manipulator has two translational degrees of freedom (its base can be moved up and down, and left and right), and four revolute joints, for a total of six degrees of freedom. Articulatory space is therefore a six-dimensional space of translational and joint angle displacements. Given that the manipulator has a single endpoint, target space is two-dimensional Cartesian space. There are four excess degrees of freedom. Thus, as shown in Figure 7, there are a wide variety of postures corresponding to each Cartesian position. The second manipulator shown in Figure 6 also has two translational degrees of freedom and four revolute joints, yielding a six-dimensional articulatory space. Target space is four-dimensional, given the arrangement of the joints into arms with two endpoints. However, button positions are two-dimensional. Thus, for this manipulator, there are excess degrees of freedom in target space as well as in articulatory space. The simplest approach to resolving the excess degrees of freedom in target space is to use don't-care conditions, so that the targets appear as hyperplanes in target space.

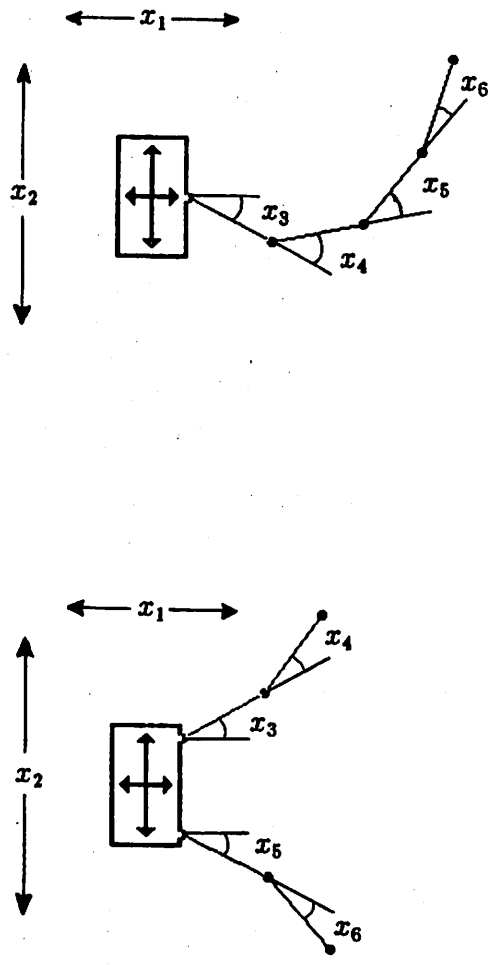


Figure 6: Two six degree-of-freedom planar manipulators.

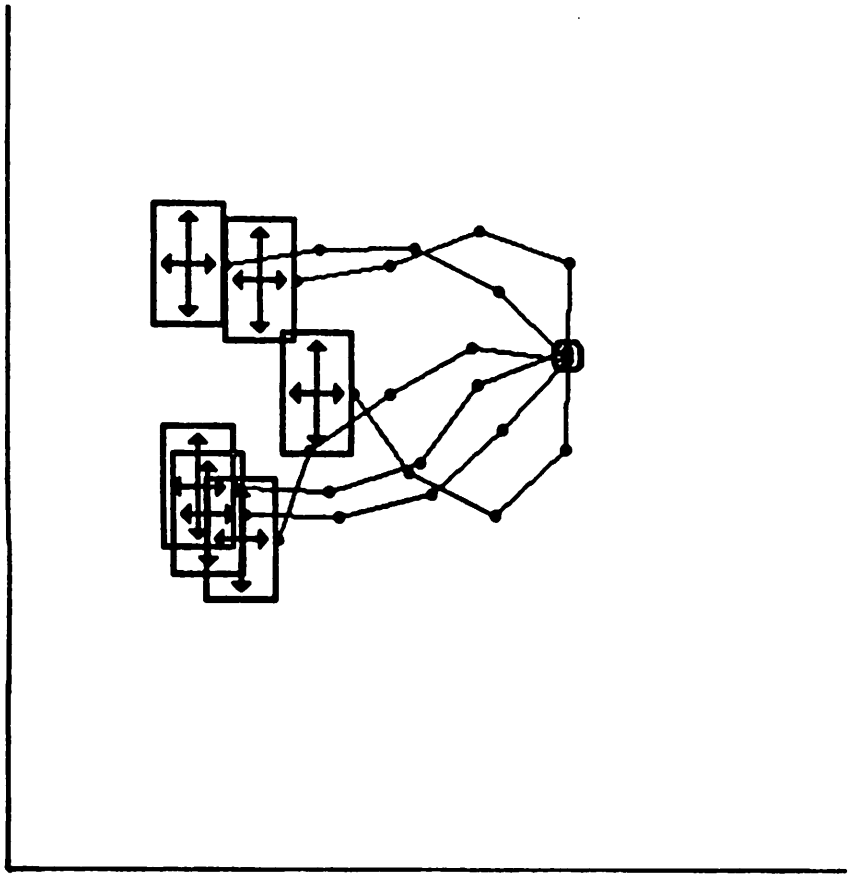


Figure 7: Various inverse kinematic solutions.

As will be demonstrated, this effectively requires one of the endpoints to be at the button position, while leaving the other endpoint unconstrained.

For both manipulators, the constraint function $\mathbf{c}(\mathbf{x})$ mapping between articulatory and target space is the forward kinematics of the manipulator, and the Jacobian matrix of $\mathbf{c}(\mathbf{x})$ is the "Jacobian" of the manipulator (Paul, 1981). The task to be solved is the inverse kinematics problem: Given target points in Cartesian space, find corresponding values for the articulatory variables. The system must find sequences of inverse kinematic solutions along a trajectory, and do so in such a way that the solutions for points nearby in time differ minimally. To do this, the system must take advantage of the excess degrees of freedom of the manipulator.

The networks used to control these manipulators are of the general form shown in Figure 4. The networks each have six articulatory units, one for each of the manipulators' articulatory degrees of freedom. Each of these units computes its activation by passing the weighted sum of its input through a logistic function with asymptotes at -1 and 1:

$$x_i = \frac{1 - e^{-s_i}}{1 + e^{-s_i}},$$

where

$$s_i = \sum_{j=1}^{n} w_{ij} x_j.$$

The activations of the first two articulatory units are interpreted as translational displacements. The remaining four articulatory units represent the sines of the angles of the four joints. Since these activations are real numbers between -1 and 1, the angles are restricted to displacements between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$ radians.

The networks have six plan units, and four hidden units in both the model subnetwork and the sequential subnetwork. The hidden units are logistic units, with asymptotes of 0 and 1. For the first manipulator, there are two target units, with activations ranging between -1 and 1, where each unit represents one dimension of the Cartesian workspace.[18] There are also two state units, connected as shown in Figure 3, and receiving fixed gain feedback from the target units. For the second manipulator, an extra pair of state units and target units were added, with a homologous interconnection structure.

For a network to be able to learn sequences, it must first learn something about the *forward* kinematics through a system identification phase. This was done by repeatedly choosing random activation patterns $\mathbf{x}$ for the articulatory units (chosen from a uniform distribution), and learning the mapping from the articulatory units to the target units by comparing the outputs of the target units $\mathbf{y}'$ to the actual

[18] There are many other possible network representations of articulatory space and target space than the relatively direct variable encodings used here. For example, values in both spaces can be coarse-coded (Hinton, McClelland, & Rumelhart, 1986; Kuperstein, 1987), yielding increased generalization, decreased learning time, and better resistance to noise and damage. For the purpose of demonstrating the algorithm, however, the current representational scheme is sufficient.

Cartesian position(s) $\mathbf{y}$ of the manipulator endpoint(s) and using the backpropagation learning rule (with the error functional E_m given in Equation 9) to change the weights in the network to reduce the discrepancy. After approximately 2000 trials for each network, the average discrepancy was judged to be sufficiently small, and the exploration phase was ended. From this point on, the weights in the path from the articulatory units to the target units were fixed, and the remaining weights in the network were learned as the network attempted to perform sequences.

For the sequence learning experiments, a learning trial involved the following procedure. First, the plan vector for the sequence being learned was clamped on the plan units. Other than in the experiment on parametrized plans (see below), plan vectors were simply particular random vectors, one for each sequence. Next, the state units were initialized with the pattern $< 0, 1 >$ (or $< 0, 1, 0, 1 >$ in the case of the second manipulator). This started the oscillatory behavior on these units. On each time step, activation passed forward to the articulatory units, determining a posture for the manipulator. Activation also passed forward to the target units, thereby instantiating a particular Jacobian transpose in the model subnetwork. Incremental changes to the weights were then computed by backpropagation in accordance with Equation 13. Units then saved their current activations in a temporary location, to allow the smoothness constraint to be implemented at the next time step. Finally, the target values were fed back through the recurrent links to the state units.[19] This procedure was followed on each successive time step, with the target position changing to follow the target sequence.

Results — Manipulator I

Sequence learning.

In the first simulation, the network learned the cyclic sequence shown in Figure 8. The manipulator was required to touch the buttons in counterclockwise order. Figure 9 shows the results on particular trials during the learning process. In this figure, each panel displays an overlay of the position of the manipulator at four successive time steps during a particular learning trial. As can be seen in the last panel, by trial 77 the system has found a solution for the sequential imitation problem. Note that the solution that the system has found is a good one; in spite of the infinite number of postures that are possible for each target position, the postures chosen at each time step are related to the postures chosen at nearby time steps. Thus, the values for the articulatory degrees of freedom need not change greatly between time steps and the trajectory appears to be reasonably coordinated.

[19] Note that external target values, rather than the outputs of the target units, were fed back through the recurrent connections during learning. This allowed learning to proceed faster than would otherwise be the case. Of course, when the network performance was tested, no external target values were available, and actual target unit output values were fed back. When an external target value was a don't-care condition, actual output values were fed back in either case.

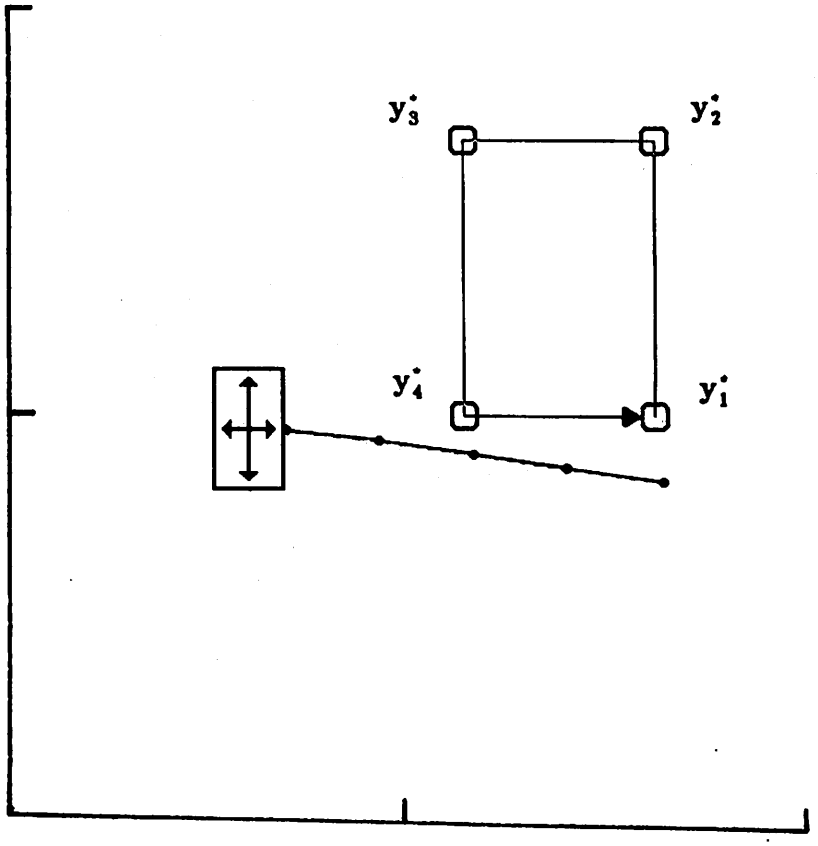


Figure 8: A sequence to be learned.

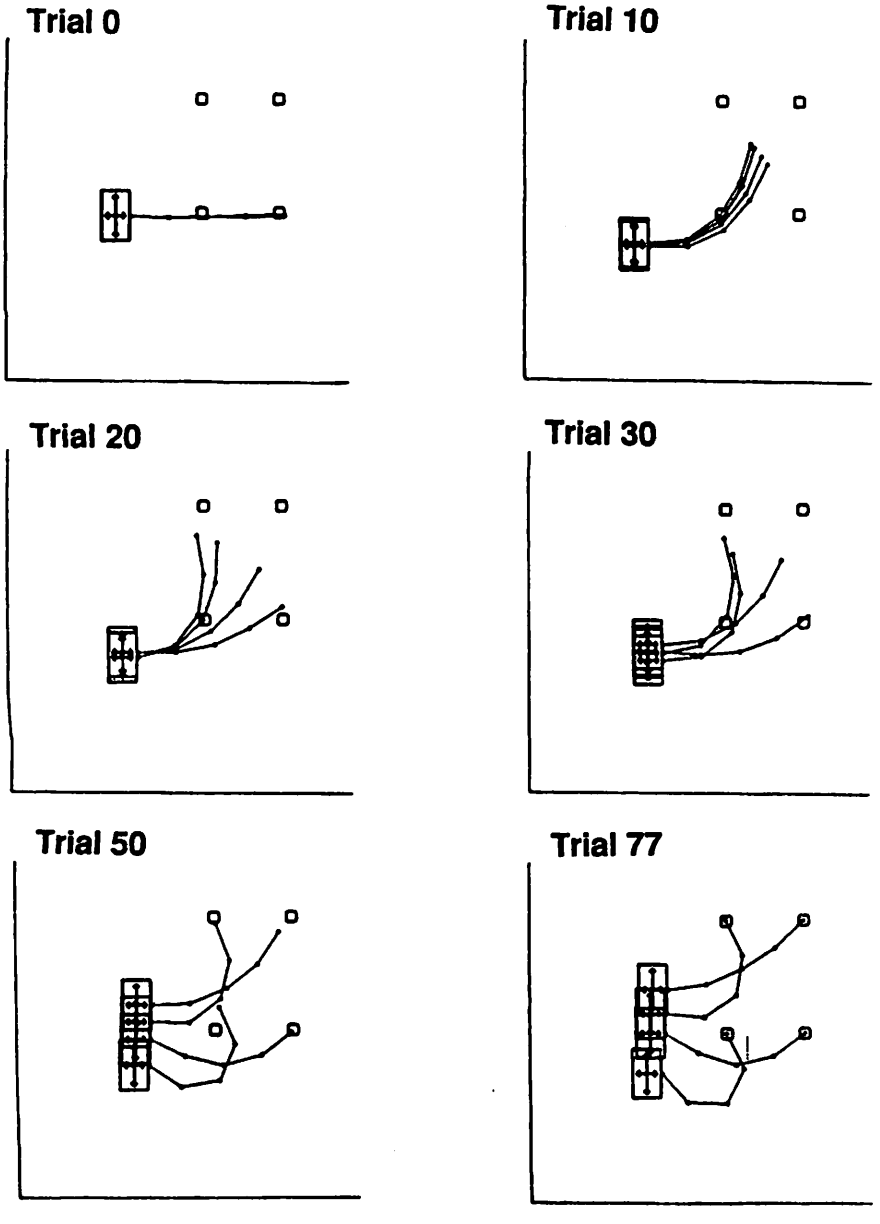


Figure 9: Epochs in the learning of the sequence.

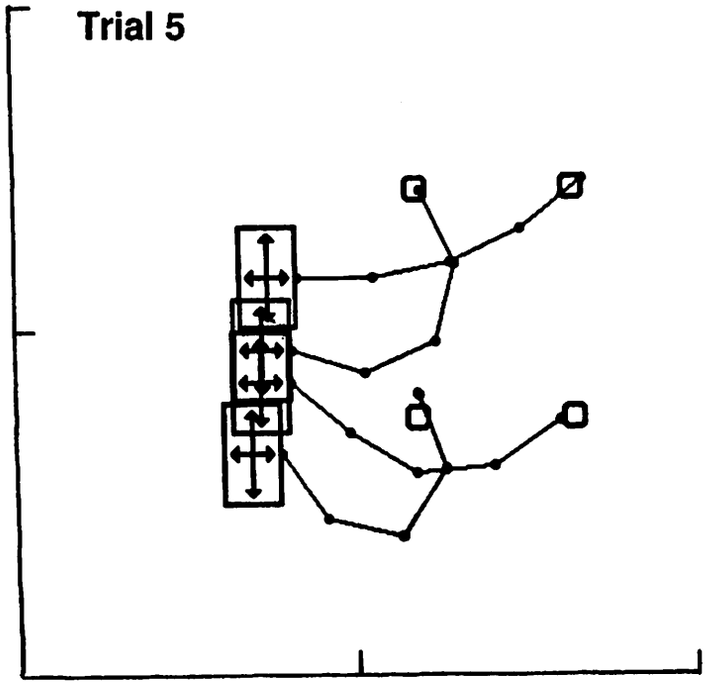


Figure 10: Learning a shifted sequence.

During the learning process, the network stores the solution it is developing, and after learning can perform the learned sequence if it is given the corresponding plan vector as input. The same network can learn many different sequences.

Generalization and Interference.

In a further simulation, the network learned a second sequence that was a shifted version of the previously learned sequence. That is, each button position was shifted downwards with respect to its position in the original sequence. It was expected that the previous learning should generalize, so that learning time for the second sequence should decrease. As shown in Figure 10, this was in fact the case. After only 5 learning trials on the second sequence, the system has nearly found a solution.

Generalization can hinder learning as well, when previous learning starts the system with weights that are far from a solution for a particular sequence. Also, solutions that are suboptimal with respect to the smoothness constraint are likely to be found if they are too close to the initial weights. In general, the ability of the system to find good solutions for each of several different sequences depends on the relationships between the sequences. The presentation scheme is also important—trials should be interspersed, rather than blocked. Finally, note that in general, the patterns of interference or generalization that are obtained are not absolute, but depend on the particular representation used for the articulatory degrees of freedom

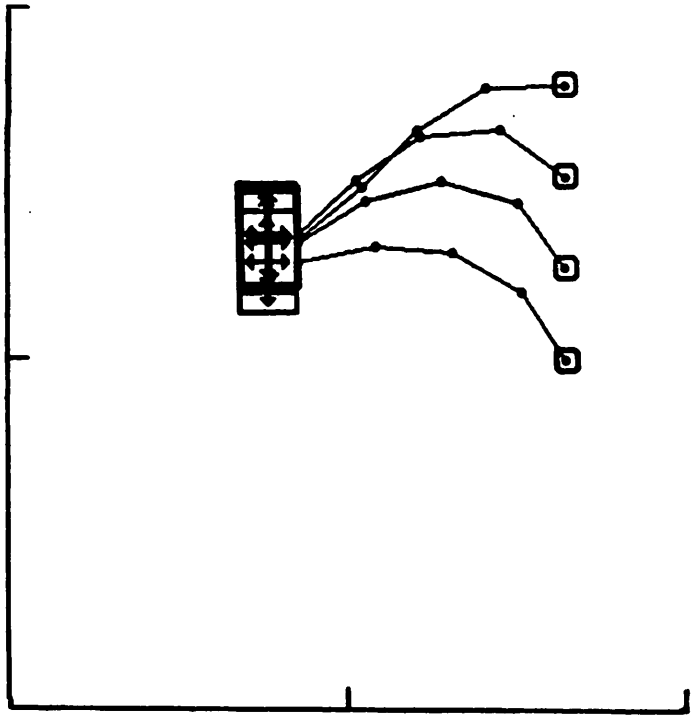


Figure 11: Learning a linear sequence.

and the target degrees of freedom. Particular symmetries in these representations lead to particular kinds of generalizations.

Using excess degrees of freedom.

Figures 11 and 12 show the results of the network learning a pair of sequences which are straight lines in Cartesian space. In these figures, the button at the medial position is at the same position in Cartesian space. However, because of the smoothness constraint, the inverse kinematic solutions found by the network are different in the two figures. This is an example of *coarticulation*: The forms chosen for an action depend on the temporal context in which the action is embedded, even though the target point is the same. The system is able to take advantage of its excess degrees of freedom to blend actions, while still respecting nonlinear constraints that require the endpoints to be at particular positions at successive time steps.

The system appears to have access to a wide variety of possible inverse kinematic solutions and can find different solutions that are appropriate for different temporal contexts. This is demonstrated by the postures shown earlier in Figure 7; they are in fact solutions found by the network for the same target point embedded in different

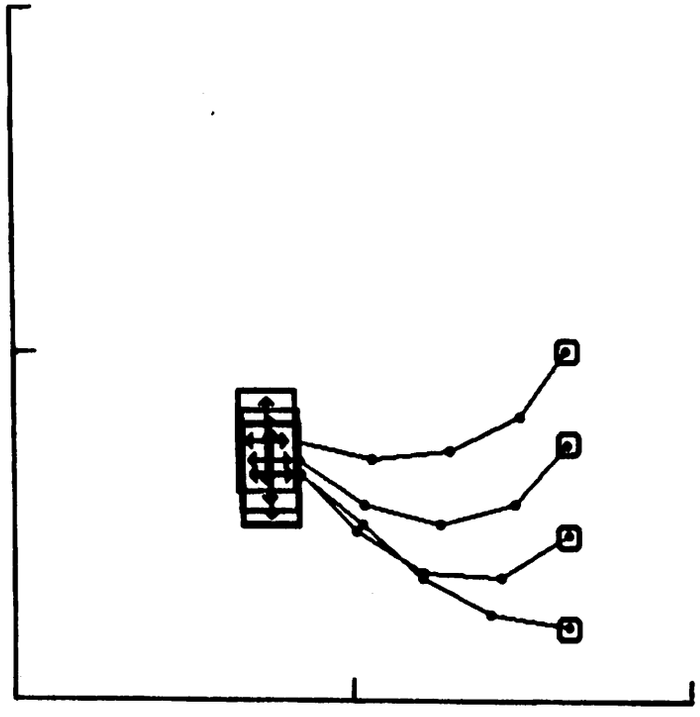


Figure 12: Learning another linear sequence which shares a target with the previous sequence.

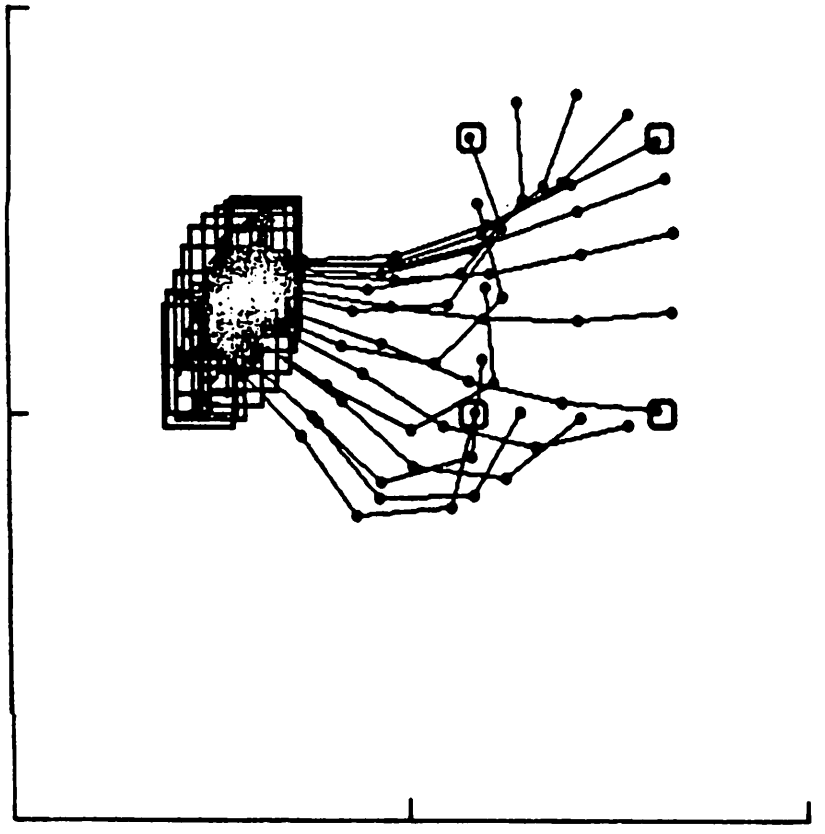


Figure 13: Performance when the state trajectory is reparameterized.

sequences.

Interpolation.

An effect of the continuity of the mappings in the network is that it is possible to scale the speed at which the system performs a sequence by reparameterizing the state trajectories. This can be done by changing the weights ξ and μ between the state units so that the phase angle $tan^{-1}\frac{\xi}{\mu}$ becomes smaller or larger. To demonstrate this scaling property, a network first learned the cyclic sequence of Figure 8. Then the weights μ and ξ were modulated so as to change the phase angle by one-fourth. Also, the state units were updated four times more frequently than the remainder of the network. The resulting trajectory is shown in Figure 13, where it can be seen that the manipulator automatically interpolates between the learned positions.

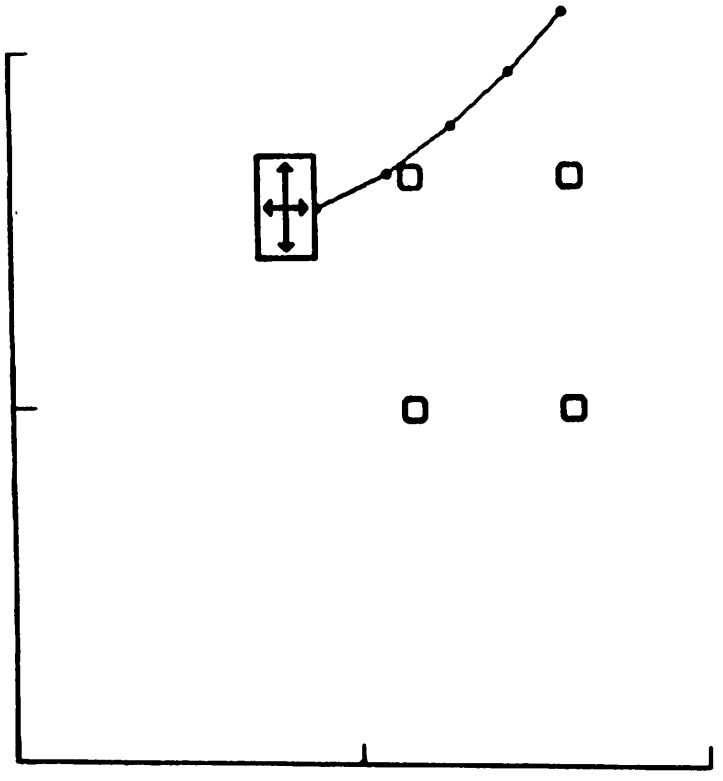


Figure 14: Starting the manipulator in an unknown position.

Nonlinear dynamics.

A network with feedback connections from the target units to the state units can exhibit interesting nonlinear dynamical phenomena (Jordan, 1986b). After learning the sequence shown in Figure 8, the network was initialized with random values for the state units. When activation from these units was passed through to the articulatory units, the manipulator was seen to be in the initial position shown in Figure 14, a position which the manipulator had never visited. However, due to the dynamics of the network, the manipulator endpoint tends to be drawn in to the learned trajectory, showing that the learned trajectory is an attractor in target space. This is demonstrated in Figure 15. Starting the system in a variety of other positions, including inside the learned trajectory, led to similar results. In a further experiment, the network learned to perform the same sequence in a clockwise order, in the context of a different plan. When the system was started in random positions, the manipulator endpoint was drawn in to one of the two learned cycles, depending on which plan was present on the plan units. The learned trajectories are limit cycles that partition target space into two basins of attraction.

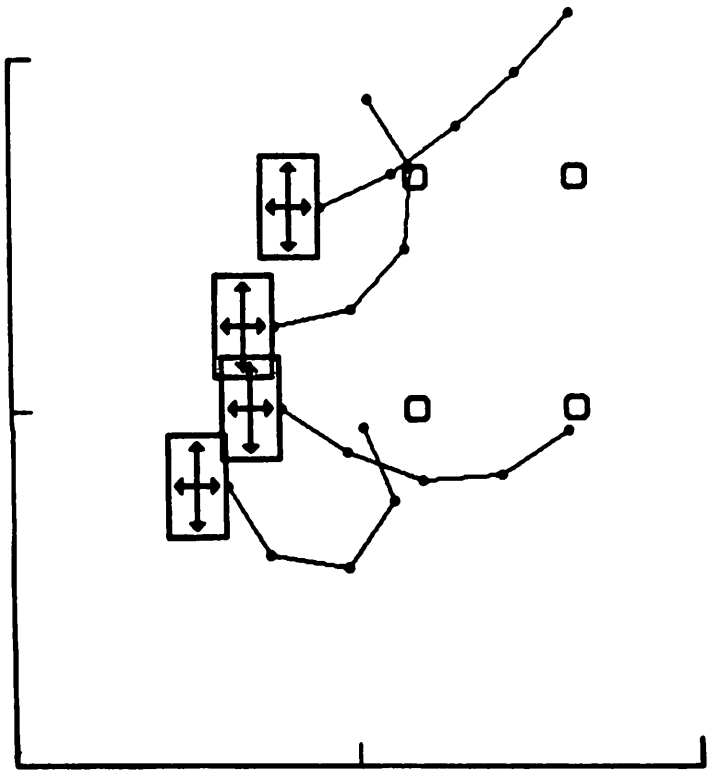


Figure 15: Attractor dynamics.

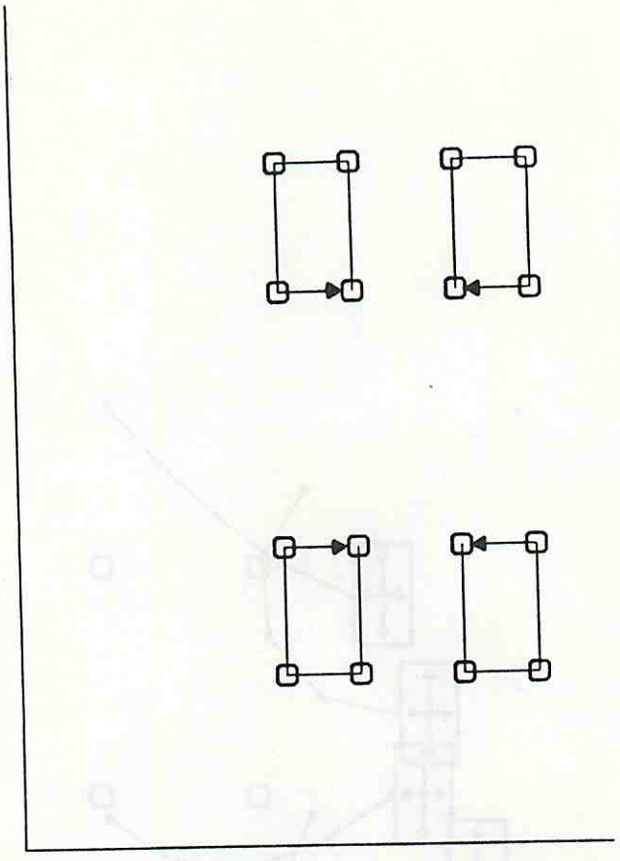


Figure 16: Four sequences to be learned by a network with modifiable plans.

Learning parametrized plans.

The section on generalization and interference showed that a sequential network is sensitive to relationships between the sequences that are being learned. The question arises as to how the network could incorporate such relationships in a useful way into the process of performing sequences. One possibility lies in the choice of the plan representations; indeed, Jordan (1986a) showed that experimenter-devised plan vectors that encoded particular aspects of the sequence being learned led to faster learning than random plan vectors. Of course, the more interesting possibility exists that that network itself could form appropriate plan representations based on experience with the sequences.

To investigate this possibility, a network which could modify its plan representations learned the four sequences shown in Figure 16. To insure that the external input provided no information about how to form the plan representations, the technique of using standard basis input vectors was borrowed from Hinton (1986). The network was constructed with a set of four *local* plan units, one and only one of which was turned on for each of the sequences shown in Figure 16. These units were connected to a set of four *distributed* plan units, which were then connected in the

normal way to the remainder of the sequential network.[20] The weights between the local and distributed plan units were initially set to small random values, and were learned as error signals were propagated back to the distributed plan units.[21]

The results after 650 learning trials are shown in Figure 17. The figure shows the *final* performance of the network for each sequence, and also displays the activations of the distributed plan units. The first such unit clearly encodes the vertical dimension of the workspace. This is useful given the translational articulatory degree of freedom in that direction. The second unit appears to take on three values that encode the average direction and amount of curvature of the arm. For the top-left and bottom-left sequences, this curvature is large, in opposite directions, whereas for the rightmost pair of sequences, little curvature is required. The third unit appears to encode the fact that pairs of sequences are diagonal neighbors. This may be due to the similar vertical excursions needed to execute corresponding sequences, but it may also encode the direction of movement—the top-left and bottom-right sequences are counterclockwise, whereas the other two sequences are clockwise. Indeed, investigation of the connections from this unit to the hidden units suggest that the unit serves to gate particular hidden unit clusters that vary in opposing directions across time. Finally, the fourth distributed plan unit appears to redundantly encode the vertical dimension of the workspace. In general, these units developed representations that appear to be useful parameterizations of the sequences to be performed. Although further work is needed, such an ability is encouraging with respect to the generative potential of these networks.

Inaccuracies in the model.

A further simulation considered the question of inaccuracies in the network's model of the forward kinematics. This issue is of relevance for several reasons. First, it may be difficult in more complicated situations to obtain a complete forward model, and questions arise concerning the efficacy of a model which is only qualitatively good. Second, it may be desirable to explore the forward kinematics and learn sequences at the same time, and thus the sequence learning will have to make do with a partial model. Finally, the robustness of the learning system as a whole depends on its *robust*ness to damage to the model or change in the physical system being modeled.

To address these issues, an experiment was conducted in which random noise was added to the weights in the model subnetwork after the model was learned. The noise was rather large: The power in the noise was one-half the power in the original weights. After the weights were distorted in this manner, the network attempted to learn the cyclic sequence shown in Figure 8, with the distorted weights held fixed. The results are shown in Figure 18. As can be seen, it was still possible for the system to find an exact solution to the problem, even with an distorted model. The

[20] The distributed plan units were logistic units with asymptotes at plus one and minus one.

[21] For this simulation, the network also used feedback from the articulatory units to the state units. There were six additional state units, each of which was a first-order linear filter.

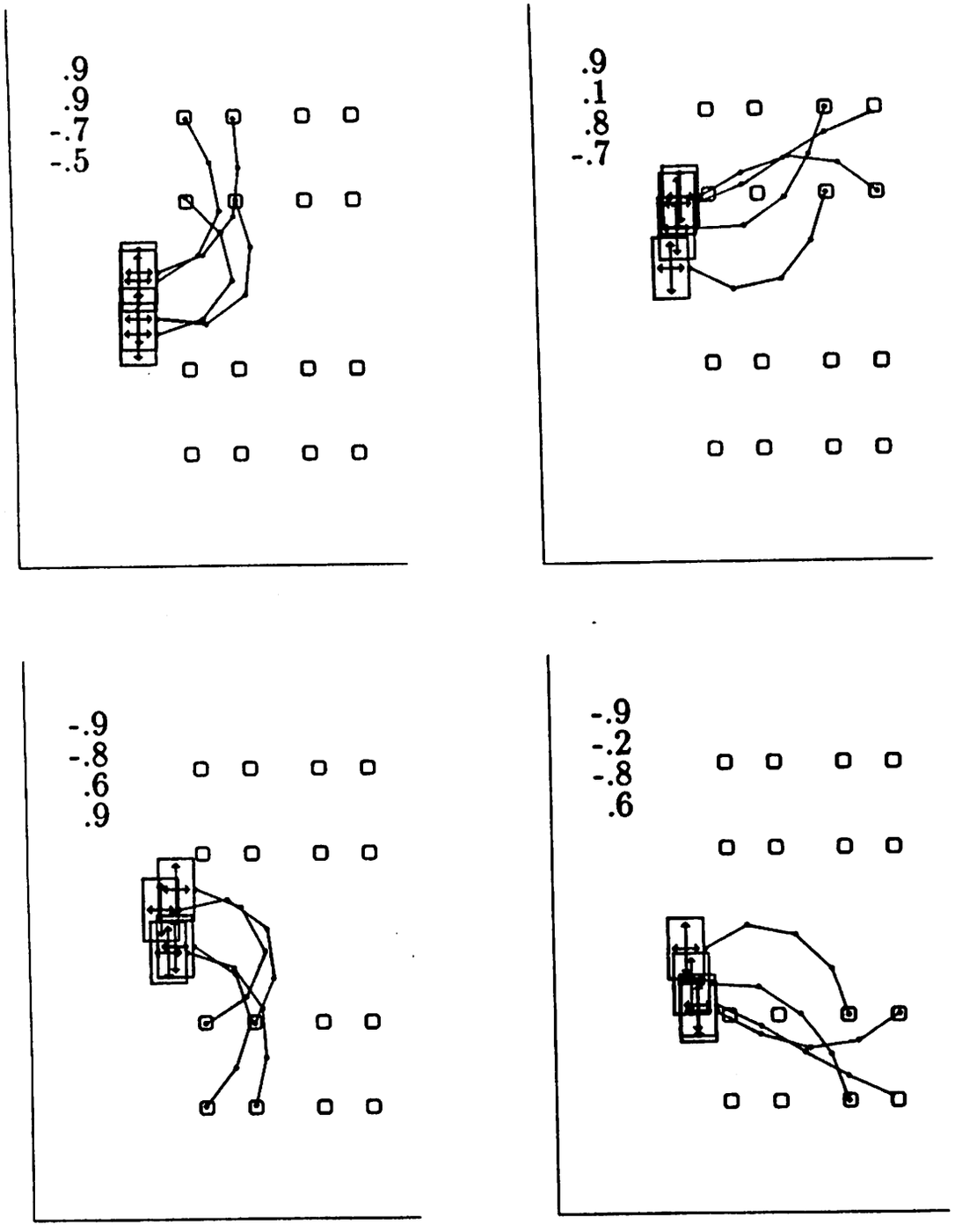


Figure 17: Activation patterns of the distributed plan units for each sequence and corresponding performance of the network.

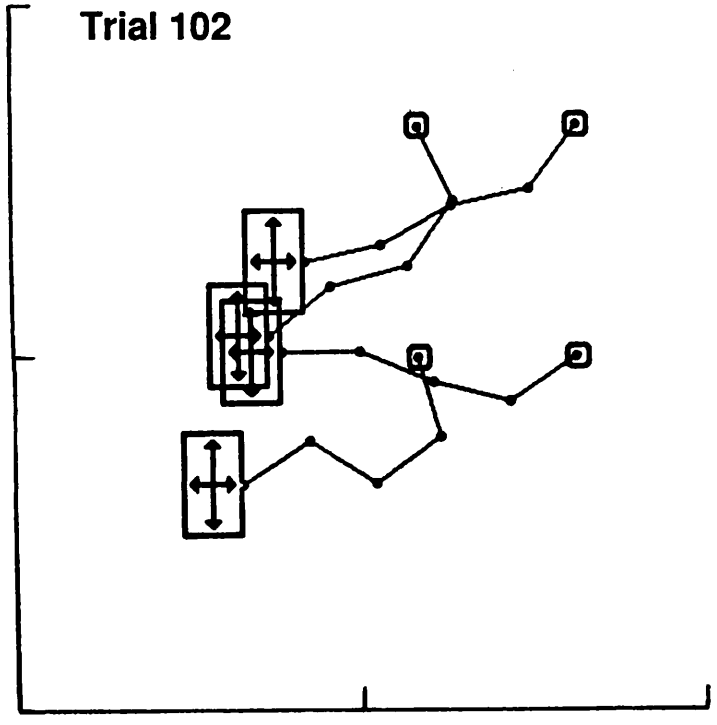


Figure 18: Learning with an inaccurate model.

distortion had two effects on the solution: first, it took longer to learn the sequence than before, and second, the solution was fairly awkward, that is, the trajectory in articulatory space was less smooth than before. Both of the effects are due to the fact that the system has an incorrect version of the Jacobian transpose; thus, the directions that the system moves in articulatory space is no longer down the true gradient. However, because the error vectors $\mathbf{y}_i^* - \mathbf{y}_i$ can still be computed correctly, sequences for which the articulatory vectors $\mathbf{x}_i$ are in the correct regions R_i^* are still stable points of the algorithm. These stable points can be reached if the biased steps in articulatory space have a positive inner product with the true gradient.

Results — Manipulator II

The task that the system must learn is again a sequential positioning task. What is different is that only one endpoint is required to touch a button at any given moment; thus, target values need to be specified for only two of the four target units at each time step. The other two target units have don't-care conditions.

The choice of which pair of target units are given target values and which are given don't-care conditions can be made in several ways. One approach is to assume that a endpoint-to-button assignment is set up in advance by the teacher. Another approach, used in the simulation reported here, is to allow the choice to be made by the system itself. At each time step, the current endpoint positions are compared to the target position, and target values are assigned to the endpoint closest to the target. With this algorithm, the endpoint-to-button assignment can change during the course of learning, depending on the directions in which endpoints are pulled.

The simulation results are shown in Figure 19. The system was able to find the obvious endpoint-to-button assignments for this task. Furthermore, the system coarticulates: as the assigned endpoint touches a button, the other endpoint tends to anticipate or perseverate other actions in the sequence. This coarticulation can be observed in the articulatory variables as in previous simulations, or directly in the target variables. It is worth mentioning that this behavior is reminiscent of that shown by skilled typists (Gentner, Grudin, & Conway, 1980), and is similar to the behavior of a simulation model of typing proposed by Rumelhart and Norman (1982).

At the target level, the effect of don't-care conditions on target units is that they allow values to spread in time. At the articulatory level, however, the effect can be complex, depending on the particular couplings between articulatory variables and target variables. In general, don't-care conditions on target units decrease the number of constraints on the articulatory variables and give the articulatory variables new directions in which to vary. Such is the case, for example, with the translational degrees of freedom in the above task. In some cases, the number of constraints may decrease to zero, in which case a don't-care condition is effectively imposed directly on the articulatory variable.

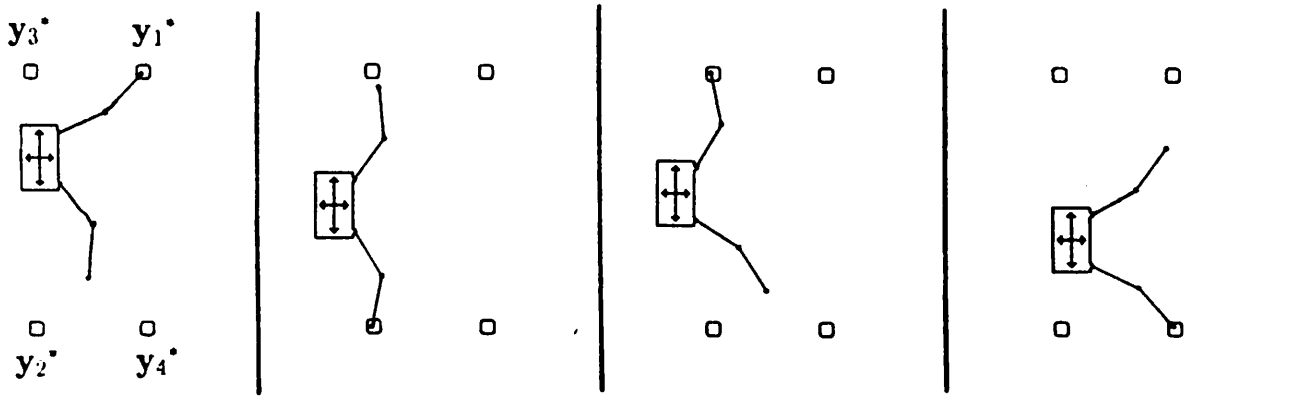


Figure 19: Learning a sequence with the second manipulator.

Conclusions

This paper has discussed methods whereby a general class of constraints can be imposed on the forms of the actions a system must learn. This approach extends the applicability of the supervised learning paradigm to inverse learning problems, in which there are in general many possible target vectors corresponding to each input vector. By using configurational constraints to specify target regions, such indeterminacy need not be resolved *a priori* in the definition of the training set. Rather, the resolution of the indeterminacy becomes part of the learning algorithm. Such resolution may occur in several ways. First, implicit constraints on the solutions found by the learning algorithm are provided by the network architecture and the topology of the set of input vectors seen during training. Ordinarily, these constraints are said to provide for generalization from the training data to novel inputs; they can play a related role in the choice of appropriate output vectors for indeterminate training inputs. Second, if generalization is unsatisfactory, then additional explicit configurational constraints can be added to the quantity being optimized. Finally, the form of an output vector can be made to be sensitive to a larger context than that defining the configurational constraints. This can be done by enlarging the scope of the optimization problem to include other classes of constraints, such as the temporal constraints discussed here.

References

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science, 9*, 147-169.

Barto, A. G., Sutton, R. S. & Anderson, C. W. (1983). Neuronlike elements that solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics, 13*, 834-846.

Eykhoff, P. (1974) *System identification.* New York: Wiley.

Gentner, D. R., Grudin, J., & Conway, E. (1981). *Finger movements in transcription typing* (Tech. Rep. 8001). La Jolla, CA: University of California, Center for Human Information Processing.

Hinton, G. E. (1986). A network which learns distributed representations of concepts. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Erlbaum.

Hinton, G. E. (1987). *Connectionist learning procedures.* (Tech. Rep. CMU-CS-87-115). Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Hogan, N. (1985). Impedance control: An approach to manipulation: Part I–Theory. *ASME Journal of Dynamic Systems, Measurement, and Control, 107*, 1-24.

Jordan, M. I. (1986a). *Serial order: A parallel, distributed processing approach* (Tech. Rep. ICS-8604). La Jolla, CA: University of California, Institute for Cognitive Science.

Jordan, M. I. (1986b). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Erlbaum.

Kawato, M., Fusukawa, K., & Suzuki, R. (1987). A hierarchical neural-network model for control and learning of voluntary movement. *Biological Cybernetics, 57*, 169-185.

Kuperstein, M. (1987). Adaptive visual-motor coordination in multijoint robots using parallel architecture. *IEEE Conference on Robotics and Automation*, pp. 1595-1602.

LeCun, Y. (1985). A learning scheme for asymmetric threshold networks. *Proceedings of Cognitiva 85.* Paris, France.

Parker, D. (1985). *Learning-logic.* (Tech. Rep. TR-47). Cambridge, MA: MIT Sloan School of Management.

Paul, R. P. C. (1982) *Robot manipulators.* Cambridge: MIT Press.

Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Volume 1.* Cambridge, MA: MIT Press.

Rumelhart, D. E. & Jordan, M. I. (1988). *Supervised learning with a distal teacher.* Paper in preparation.

Rumelhart, D. E. & Norman, D. A. (1982). Simulating a skilled typist: A study of skilled cognitive-motor performance. *Cognitive Science, 6,* 1-36.

Salisbury, J. K. (1980). Active stiffness control of a manipulator in Cartesian coordinates. *Proceedings of the 19th IEEE Conference on Decision and Control,* 95-100.

Sutton, R. S. (1987). *Learning to predict by the methods of temporal differences.* (Tech. Rep. TR87-509.1). Waltham, MA: GTE Laboratories.

Widrow, B. & Stearns, S. D. (1985). *Adaptive signal processing.* Englewood Cliffs, NJ: Prentice-Hall.