

# A Distributed Robotic Control System Based on a Temporal Self-Organizing Neural Network

Guilherme A. Barreto, *Student Member, IEEE*, Aluizio F. R. Araújo, Christof Dücker, and Helge Ritter

**Abstract**—A distributed robot control system is proposed based on a temporal self-organizing neural network, called competitive and temporal Hebbian (CTH) network. The CTH network can learn and recall complex trajectories by means of two sets of synaptic weights, namely, competitive feedforward weights that encode the individual states of the trajectory and Hebbian lateral weights that encode the temporal order of trajectory states. Complex trajectories contain repeated or shared states which are responsible for ambiguities that occur during trajectory reproduction. Temporal context information are used to resolve such uncertainties. Furthermore, the CTH network saves memory space by maintaining only a single copy of each repeated/shared state of a trajectory and a redundancy mechanism improves the robustness of the network against noise and faults. The distributed control scheme is evaluated in point-to-point trajectory control tasks using a PUMA 560 robot. The performance of the control system is discussed and compared with other unsupervised and supervised neural network approaches. We also discuss the issues of stability and convergence of feedforward and lateral learning schemes.

**Index Terms**—Distributed control, neural networks, robotics, self-organization, stability analysis, temporal sequences.

## I. INTRODUCTION

CONTROL of movements in both biological and artificial systems demands the availability of several *sensorimotor transformations* that convert sensory signals into motor commands that drive a set of muscles or robotic actuators. Such transformations are highly nonlinear and it is very difficult to express them in a closed analytical form. Artificial neural networks (ANNs) can be used to learn one or more sensorimotor transformations required to perform a given robotic task without precise knowledge about the parameters of the robot [1], [2]. Among the neural learning paradigms, self-organized (or unsupervised) learning schemes have appealing properties that may justify their use in robotics.

- i) They do not need external supervision.
- ii) Training is usually very fast, which is important for real-time applications.
- iii) Information is represented in a localized fashion, facilitating the interpretation of the results.

Manuscript received August 24, 2001; revised July 30, 2002. This work was supported by FAPESP (a Brazilian research agency) through Ph.D. scholarship 98/12699-7 and Grant 00/12517-8.

G. A. Barreto and A. F. R. Araújo are with the Department of Electrical Engineering, University of São Paulo (USP), São Carlos, Brazil (e-mail: gbarreto@sel.eesc.sc.usp.br; aluizioa@sel.eesc.sc.usp.br).

C. Dücker and H. Ritter are with the Neuroinformatics Group, Faculty of Technology, University of Bielefeld, Bielefeld, Germany (e-mail: chrisd@techfak.uni-bielefeld.de; helge@techfak.uni-bielefeld.de).

Digital Object Identifier 10.1109/TSMCC.2002.806067

Such properties can reduce considerably the computational load involved in robot programming, which is an item responsible for up to one-third of the total cost of implementation of an industrial robotic system [3].

Many self-organizing neural networks (SONNs) have been proposed in order to learn, for example, inverse kinematics and dynamics [4]–[12]. This is accomplished by allowing the robot arm to execute random movements within its workspace and measuring the sensory consequences. This action-perception cycle, known as *Piaget's circular reaction* [13], forms a closed-loop control system that allows accurate motion to be learned by the network. What is learned is an associative mapping between the randomly-generated motor actions and their corresponding sensory effects. The learned mapping is then used for control, i.e., every time the robot experiences a given sensory pattern, it should provide the appropriate motor signal.

In the previous approach, successive robot movements are supposed to be uncorrelated with each other, i.e., the order in which the motion occurs is not important. However, an inherent property of robotic tasks is that they have a well-defined sequential nature in the sense that a given robot arm should assume specific configurations (states) successively in time along a pre-defined path. This temporal characteristic is not incorporated into the self-organizing learning procedures cited above, which implies that only static sensorimotor transformations, such as inverse kinematics, can be learned by the network. In these cases, the temporal order of the robotic task at hand is set in advance by the neural network designer.

An alternative is to use temporal neural networks which can directly cope with sequential aspects of the robotic task. For a SONN to handle *temporal* data, it must be given memory about past states of the task being modeled. Currently, four techniques have been used for this purpose [14]. The first one adds *short-term memory* (STM) mechanisms, such as *tapped delay lines* and/or *leaky integrators*, to the input of the SONN [15]. The second technique includes STM mechanisms internally to the network in the activation and/or learning rules [16], [17]. The third technique uses several temporal SONN's hierarchically, trying to capture spatiotemporal aspects of the input sequence through successive refinements [18]. The fourth approach uses a feedback loop to insert temporal information into the SONN [19], [20].

Such temporal SONNs should learn to associate consecutive states of a trajectory and store these state transitions for posterior reproduction. Usually, for the purpose of recall, the network receives the current state of the robot and responds with the next one, until the trajectory is completely retrieved. This approach, known as the *associative chaining hypothesis* [21] has

been widely used by unsupervised as well as supervised neural network models to model temporal characteristics of sensorimotor control [22]–[31].

The main motivation for the present work is to emphasize the feasibility of applying temporal SONNs to real-time, distributed control of robotic manipulators. The contribution of the paper is two-fold: i) It is the first one to report an implementation of a controller for a robotic manipulator based on a temporal SONN, and ii) the control task is designed to be performed in real-time and in a distributed fashion. The design of the composite system (neural network based controller + distributed communication tool) is facilitated by the separation of the neural network design and the robot control task into two modules which are linked through a distributed communication tool. The performance of the unsupervised neural learning algorithm combined with a distributed communication tool is evaluated based on its ability to learn and reproduce complex trajectories accurately and without ambiguities. The resulting distributed neurocontrol system is simple, very fast, and robust to noise and faults.

The remainder of the paper is organized as follows. In Section II, the neural network is presented and its learning and recall procedures are discussed in details. In Section III, the robot control platform and its main components are introduced. A distributed communication tool is also presented and its use in the robotic task of interest is discussed. In Section IV, several tests with the whole system (robot + neural network based controller + distributed communication tool) are carried out. In Section V, some features of the proposed distributed, real-time robotic control system are compared with other self-organizing as well as supervised approaches. The paper is concluded in Section VI.

## II. NEURAL NETWORK MODEL

It is assumed that the trajectories we are interested are finite sequences of discrete points. However, most signals in nature are analog and need to be discretized before simulation on digital computers. This is done by sampling at regular intervals and adopting a system in which time proceeds by intervals of  $\Delta t$ . We will use the symbol  $t$  to represent a particular point in time, where  $t \in \{0, \Delta t, 2\Delta t, 3\Delta t, \dots\}$ . In this formulation,  $\Delta t$  can be considered to be the unit of measure for the quantity  $t$ , and thus, it is reasonable to omit the units and express  $t$  simply as a member of the set of integer numbers  $t \in \{0, 1, 2, 3, \dots\}$ .

Then, one can define a *trajectory*  $\mathbf{S}$  as a finite set of state vectors  $\mathbf{s}(t) = [s_1(t), s_2(t), \dots, s_n(t)]^T$ ,  $\mathbf{s}(t) \in S \subset \mathbb{R}^m$ , grouped according to their order of occurrence in time, that is,  $\mathbf{S} = \{\mathbf{s}(t), \mathbf{s}(t-1), \dots, \mathbf{s}(t-N+1)\}$  where  $N$  is the length of the sequence. Trajectories can be classified as *simple* and *complex*. In complex ones, an individual state can occur more than once or it can be shared with other trajectories. We refer to either repeated or shared elements of a trajectory as *recurrent* states. Simple trajectories are those without recurrent items.

Complex trajectories are responsible for ambiguities that occur during the recall process, which are resolved through additional contextual information. It is worth emphasizing that the term context is used very generally here to mean any secondary or additional source of information, derived from a different sensory modality, a different input channel within the same modality, or the temporal history of the input.

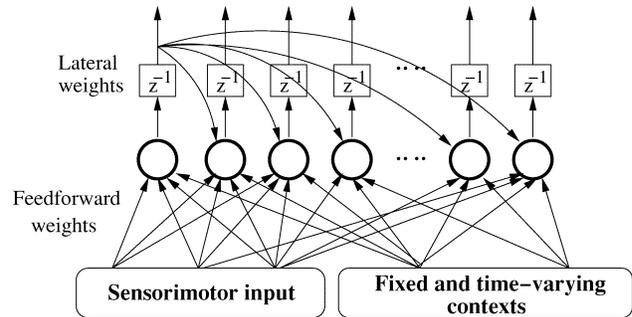


Fig. 1. Architecture of the temporal neural network. For simplicity, only some lateral weights are shown.

### A. Network Architecture

The architecture of the neural algorithm, called *Competitive and Temporal Hebbian* (CTH) network, is shown in Fig. 1. This two-layer network is inspired by Grossberg’s Outstar Avalanche network [32] and Amari’s temporal associative memory model [33].

The CTH network consists of a broadcasting input layer and a competitive output layer, which carries out the processing. The model has feedforward and lateral weights that play distinct roles in its dynamics. This architecture differs from those of standard SONNs by possessing context units at the input and delay lines at the output. The delay lines, however, are needed only for training in order to learn unidirectional temporal transitions. The goal of the CTH network is to provide a spatiotemporal sequence of robot arm configurations (states) between a starting and an ending position. The movement is executed by an industrial manipulator comprising a set of joints individually driven by actuators.

The network input vector comprises a sensorimotor vector  $\mathbf{s}(t) \in \mathbb{R}^m$ , a fixed context vector  $\mathbf{C}_F \in \mathbb{R}^m$ , and a time-varying context vector  $\mathbf{C}_T(L, t) \in \mathbb{R}^{L \cdot m}$ . The sensory input vector  $\mathbf{s}(t)$  at time step  $t$  is defined as

$$\mathbf{s}(t) = [\mathbf{r}(t), \boldsymbol{\theta}(t)]^T \quad (1)$$

where  $\mathbf{r}(t) \in \mathbb{R}^3$  is the Cartesian position of the end-effector at time step  $t$  and  $\boldsymbol{\theta}(t) = \{\theta_i\}$ ,  $i = 1, \dots, dof$ , where *dof* stands for degrees-of-freedom, is a particular set of joint angles that produces  $\mathbf{r}(t)$ . Each  $\mathbf{s}(t)$  defines the state of the robot arm at a given instant of time. In this sense, one can say that the CTH network is used to associate a sequence of Cartesian position  $\mathbf{r}(t)$  of the end-effector with a sequence of joint angles  $\boldsymbol{\theta}(t)$  needed to solve the robotic task at hand.

The fixed context is time-invariant and usually set to a particular state of the temporal sequence, the initial or final one being the usual option. It is kept unchanged until the end of the current sequence has been reached. This type of context acts as a kind of *global sequence identifier*. Time-varying context is an STM mechanism implemented as *tapped delay lines* [14]. Thus

$$\mathbf{C}_T(L, t) = [\mathbf{s}(t-1), \dots, \mathbf{s}(t-L)]^T$$

where  $\mathbf{C}_T(t, L) \in \mathbb{R}^{L \cdot m}$ ,  $L$  is an integer called *the memory depth*. A suitable length of the time window is usually determined by trial-and-error (see Section V).

### B. Updating Feedforward Weights

The feedforward weight vector  $\mathbf{w}_j$ ,  $j = 1, \dots, M$  connects the input units to the output neuron  $j$ , which is defined as

$$\mathbf{w}_j(t) = [\mathbf{w}_j^s(t), \mathbf{w}_j^F(t), \mathbf{w}_j^T(t)]^T$$

so that  $\mathbf{w}_j(t) \in \mathfrak{R}^{(L+2) \cdot m}$ . At each time step, the current state vector  $\mathbf{s}(t)$  is compared with each feedforward weight vector in terms of Euclidean distance as follows:

$$D_j^s(t) = (\mathbf{s}(t) - \mathbf{w}_j^s(t))^T \mathbf{P} (\mathbf{s}(t) - \mathbf{w}_j^s(t)) \quad (2)$$

where  $\mathbf{P} \in \mathfrak{R}^{m \times m}$  is a diagonal matrix, called *projection matrix*, whose elements are set to 0 or 1 [34]. The matrix  $\mathbf{P}$  is used to select the appropriate input variable to be used to search for the winner. For example, if the matrix  $\mathbf{P}$  is chosen as

$$\mathbf{P} = \left( \begin{array}{c|ccc} \mathbf{I}_r & \dots & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & \dots & 0 \end{array} \right)$$

where  $\mathbf{I}_r$  is an identity matrix with  $\dim(\mathbf{I}_r) = \dim(\mathbf{r}(t)) \times \dim(\mathbf{r}(t))$ , and only the vector  $\mathbf{r}(t)$  influences the search for the winner. No matter what values  $\boldsymbol{\theta}(t)$  assumes, it will not contribute to the computation of  $D_j^s(t)$ . A fixed context distance  $D_j^F(t) = \|\mathbf{C}_F(t) - \mathbf{w}_j^F(t)\|$  and a time-varying context distance  $D_j^T(t) = \|\mathbf{C}_T(t, L) - \mathbf{w}_j^T(t)\|$  are also computed. While  $D_j^s(t)$  is used to find the winners of the current competition,  $D_j^F(t)$  and  $D_j^T(t)$  are used to solve ambiguities during trajectory reproduction.

For accuracy in reproduction, every trajectory state should be memorized for posterior recall in the correct temporal order. That is, all  $N$  states of a trajectory should be stored and recalled in the correct order. Standard competitive networks, however, tend to cluster the input states by similarity and may split the trajectory in discontinuous segments, causing a jerky movement of the robot arm. To overcome such a situation, the network ‘‘penalizes’’ each winning neuron by excluding it from subsequent competitions and, hence, avoiding that it stores more than one state of the trajectory being learned. This ‘‘exclusion’’ mechanism is implemented through a function  $R_j(t)$ , called *the responsibility function*, that indicates if a neuron  $j$  is already responsible for the storage of a trajectory state. If  $R_j(t) > 0$ , neuron  $j$  is excluded from subsequent competitions. If  $R_j(t) = 0$ , neuron  $j$  is allowed to compete.

Furthermore, to save memory space, every time a recurrent item occurs it should be encoded by the same neuron that stored it the first time, otherwise many copies of this item will exist in the network. However, according to the exclusion mechanism implemented by the responsibility function, each neuron can win a competition only once. Thus, for each occurrence of an item in the sequence, a different neuron is chosen to store it. A possible way of eliminating such a problem is through the definition of a constant  $0 < \varepsilon \leq 1$ , called *similarity radius*. This constant establishes a neighborhood around each  $\mathbf{s}(t)$  inside which the weight vector  $\mathbf{w}_j^s(t)$  can be considered sufficient similar to  $\mathbf{s}(t)$ . In other words, the exclusion mechanism avoids that different states are learned by the same neuron, while the

similarity radius avoids that the same state be learned by different neurons.

Thus, if neuron  $j$  has never won a competition (i.e.,  $R_j(t) = 0$ ) or if the pattern stored in its weight vector is within the neighborhood of the current input (i.e.,  $D_j^s(t) \leq \varepsilon$ ), then neuron  $j$  will be evaluated, for the purpose of competition, by  $D_j^s(t)$  only. Otherwise, this distance is weighted by the responsibility function, excluding neuron  $j$  from subsequent competitions. This behavior can be formalized in terms of a function  $f_j(t)$ , called *choice function*, as follows:

$$f_j(t) = \begin{cases} D_j^s(t), & \text{if } D_j^s(t) \leq \varepsilon \text{ or } R_j(t) = 0 \\ R_j(t) \cdot D_j^s(t), & \text{otherwise.} \end{cases} \quad (3)$$

During training, we set  $\varepsilon \ll 1$  in order to assign a *different* neuron to each *different* (nonrecurrent) trajectory state. If a repeated item occurs, it will be mapped to the neuron that stored its first occurrence. For the purpose of reproduction, the similarity radius should assume higher values (i.e.,  $\varepsilon \approx 1$ ) to avoid incorrect evaluation of (3) resulting from measurement noise in the sensory vector.

The output neurons are then ranked as follows:

$$f_{\mu_1}(t) < f_{\mu_2}(t) < \dots < f_{\mu_M}(t) \quad (4)$$

where  $M$  is the number of output neurons, and  $\mu_i(t)$ ,  $i = 1, \dots, M$  is the index of the  $i$ th neuron closest to  $\mathbf{s}(t)$ . We consider  $K$  neurons,  $\boldsymbol{\mu}(t) = [\mu_1(t), \mu_2(t), \dots, \mu_K(t)]^T$ ,  $K \leq M$ , as the winners of the current competition which represents the current state vector  $\mathbf{s}(t)$  and its context.

The activation values decay linearly from a maximum value  $a_{\max} \in \mathfrak{R}$  for  $\mu_1(t)$ , to a minimum  $a_{\min} \in \mathfrak{R}$  for  $\mu_K(t)$ , as follows:

$$a_{\mu_i} = \begin{cases} a_{\max} - \left( \frac{a_{\max} - a_{\min}}{\max(1, K-1)} \right) (i-1), & \text{for } i \leq K \\ 0, & \text{for } i > K \end{cases} \quad (5)$$

where  $a_{\max}$  and  $a_{\min}$  are user-defined. For  $t = 0$ , the activations are set to  $a_j(0) = 0$ , for all  $j$ . The responsibility function  $R_j(t)$  is then updated as follows:

$$R_j(t+1) = R_j(t) + \beta a_j(t) \quad (6)$$

where the constant  $\beta \gg 1$  is called *the exclusion parameter*. For  $t = 0$ , we set  $R_j(0) = 0$ , for all  $j$ . Finally, the weight vectors  $\mathbf{w}_j(t)$  are adjusted:

$$\mathbf{w}_j^s(t+1) = \mathbf{w}_j^s(t) + \eta a_j(t) [\mathbf{s}(t) - \mathbf{w}_j^s(t)] \quad (7)$$

$$\mathbf{w}_j^F(t+1) = \mathbf{w}_j^F(t) + \eta a_j(t) [\mathbf{C}_F(t) - \mathbf{w}_j^F(t)] \quad (8)$$

$$\mathbf{w}_j^T(t+1) = \mathbf{w}_j^T(t) + \eta a_j(t) [\mathbf{C}_T(t-1) - \mathbf{w}_j^T(t)] \quad (9)$$

where  $0 < \eta \leq 1$  is the learning rate. For  $t = 0$ ,  $\mathbf{w}_j(0)$  is initialized with random numbers between 0 and 1. Similar versions of  $\mathbf{s}(t)$  may exist at the weight vectors of neurons  $\mu_2, \dots, \mu_K$  winners because  $a_{\mu_i} < 1$ ,  $i = 2, \dots, K$ . The degree of similarity of these copies with  $\mathbf{s}(t)$  is determined by the position of the neuron in the ranking shown in (4), which is reflected in the activation pattern in (5). The reason for the existence of several copies of a trajectory state to provide the CTH network with both tolerance to faults and noise.

### C. Updating Lateral Weights

A set of lateral weights  $\mathbf{m}_j(t) = [m_{j1}, m_{j2}, \dots, m_{jM}]^T$ ,  $\mathbf{m}_j(t) \in \mathbb{R}^M$  encodes the temporal order of the trajectory states using a simple Hebbian learning rule to associate the winner of the previous competition with the winner of the current competition:

$$\Delta m_{jr}(t) = \begin{cases} 0, & \text{if } m_{jr}(t) \neq 0 \\ \lambda a_j(t) a_r(t-1), & \text{otherwise} \end{cases} \quad (10)$$

where  $0 < \lambda \leq 1$  is a gain parameter. Through (10), the network “looks” backward one time step to establish a causal link corresponding to the temporal transition between two consecutive trajectory states  $\mathbf{s}(t-1) \rightarrow \mathbf{s}(t)$ <sup>1</sup>. This transition is encoded in the lateral weight [connecting the neurons associated with the activation pair  $[a_{\mu_i}(t-1), a_{\mu_i}(t)]$ ,  $i \leq K$ . Successive application of (10) leads to the encoding of the temporal order of the trajectory. It is worth noting that (10) learns the temporal order of the trajectory states, but does not memorize the duration between states. Initially,  $m_{jr}(0) = 0$  for all  $j, r$ , indicating absence of temporal associations at the beginning of the training.

A lateral connection  $m_{jr}(t)$  is updated only once, avoiding the formation of *biased transitions*. An incremental adjustment of a lateral weight associated with a recurrent state would eventually force this connection to assume high values. A strong lateral connection would *bias* the recall process by favoring the transition it encodes, even if the context information suggests the use of another transition.

### D. Stability of Feedforward and Lateral Learning

*Feedforward Learning:* Equation (7) can be rewritten as

$$\mathbf{w}_j^s(t+1) = (1 - \eta a_j(t)) \mathbf{w}_j^s(t) + \eta a_j(t) \mathbf{s}(t).$$

Thus, the weight vector  $\mathbf{w}_j^s$  can be modified such that its previous value is retained by a factor of  $(1 - \eta a_j(t))$ , whereas the current trajectory state  $\mathbf{s}(t)$  affects the weight by a factor of  $\eta a_j(t)$ . If we set  $\eta = 1$ , we get  $\mathbf{w}_{\mu_1}^s(t+1) = \mathbf{s}(t)$  for the winning neuron  $\mu_1(t)$ , where we also used the fact that  $a_{\mu_1}(t) = 1$  as defined by (5).

Thus, the entire trajectory is stored, state-by-state, in a single pass of its states and their corresponding contextual information. In other words, there is no need to present a trajectory to the network more than once. This strategy is referred to as *one-shot learning* (OSL) and constitutes an important feature of the CTH network. By using the OSL strategy, convergence time is substantially reduced compared to that of supervised networks. OSL also guarantees high accuracy during trajectory recall.

In many situations, however, the OSL strategy may lead to problems of memory storage and retrieval [35]. This is particularly true for trajectories having many states (high sampling rate). For these cases, the trajectory states have to be clustered first. This can be easily done through the CTH network if we set  $0 < \eta < 1$ . In addition, we have to set  $\beta = \lambda = 0$  to avoid exclusion of neurons and temporal order learning during the clustering stage. Once the clustering stage is over, we use  $\eta = 1$ ,  $\beta \gg 1$  and  $0 < \lambda \leq 1$ , thus proceeding with the OSL and lateral order learning schemes as defined originally for the

<sup>1</sup>This is the reason we use  $\mathbf{C}_T(t-1)$ , rather than  $\mathbf{C}_T(t)$  in (9).

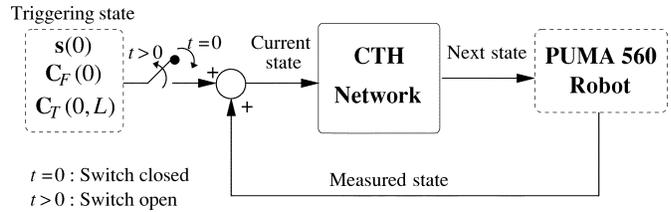


Fig. 2. Closed-loop neurocontrol scheme for autonomous trajectory reproduction.

CTH network. Detailed discussion on clustering properties of competitive networks and theoretical analysis of their convergence process can be found in [36].

*Lateral Learning:* The stability of temporal order learning and recall through time-dependent Hebbian learning rules, such as that in (10), has been studied in-depth by [33] and [37]. In particular, Herz [37] has achieved an important result: the recall process of certain delay networks is governed by a Lyapunov (or “energy”) function. The corresponding insight is that the time evolution during recall (i.e., the sequence of retrieved state transitions) can thus be understood as a downhill march in an abstract spatiotemporal energy landscape.

### E. Recall of a Stored Trajectory

Once a trajectory is learned, it can be retrieved either from its initial or any intermediate state. The trajectory recall process is a closed-loop control scheme (see Fig. 2) which comprises five steps:

- 1) recall initiation;
- 2) computation of neuronal activations;
- 3) computation of neuronal outputs;
- 4) delivery of control signals to the robot;
- 5) determination of feedback sensory signals.

For recall purpose, the parameter  $K$  is always set to 1.

1) *Recall Initiation:* To initiate reproduction, any trajectory state can be presented to the network by the robot operator ( $t = 0$  in Fig. 2). The fixed context is usually set to the target Cartesian position of the end-effector and the initial values of the temporal context are set equal to the triggering state. For  $t > 0$ , the network dynamics will then evolves autonomously to recall the part of the stored trajectory that follows the triggering state.

2) *Computation of Activation:* For each input state, the activation  $a_{\mu_1}$  of the winning neuron,  $\mu_1$ , should be computed according to (5). The feedforward weight vector  $\mathbf{w}_{\mu_1}^s$  of the winner is the closest to the current input state.

3) *Computation of Output:* The winner—the only output neuron with  $a_j(t) > 0$ —will then trigger the neuron whose weight vector stored the successor of the current input state. This is possible because of the state transition learned during the training phase and encoded by a lateral weight connecting these neurons. Thus, the output equation is defined as follows:

$$y_j(t) = y^F(t) \cdot y^T(t) \cdot G \left( \sum_{r=1}^m m_{jr}(t) a_r(t) \right) \quad (11)$$

where  $y^F(t) = 1 - D_j^F(t) / \sum_{r=1}^m D_r^F(t)$  and  $y^T(t) = 1 - D_j^T(t) / \sum_{r=1}^m D_r^T(t)$ . The function  $G$  is chosen so that  $G(u) \geq$

0 and  $dG(u)/du > 0$ . For  $t = 0$ , the output values are set to  $y_j(0) = 0$ , for all  $j$ .

It is worth noting that for a simple trajectory, the third factor on the right-hand side of (11) alone will correctly indicate the neuron that stored the next state of the trajectory. For a complex one, additional disambiguating information is required, since the third factor will produce the same value of  $y_j(t)$  for all candidates for the next state. The ambiguity is resolved by the first and second factors on the right-hand side of (11); the candidate neuron with the highest values for the first and second factors or, equivalently, the one with the lowest values for  $D_j^F(t)$  and  $D_j^T(t)$  is considered the correct one to be chosen.

4) *Delivery of Control Signals*: The control signal to be delivered to the robot is computed from the weight vector of the neuron with highest value of  $y_j(t)$  and the matrix  $\bar{\mathbf{P}} = (\mathbf{I} - \mathbf{P})$  as follows:

$$\mathbf{u}_{ctrl}(t) = \bar{\mathbf{P}}\mathbf{w}_{j^*}^s(t) = \bar{\mathbf{P}} \begin{pmatrix} \mathbf{r}^{next} \\ \boldsymbol{\theta}^{next} \end{pmatrix} = \begin{pmatrix} 0 \\ \boldsymbol{\theta}^{next} \end{pmatrix} \quad (12)$$

where  $j^* = \arg \max_j [y_j(t)]$ . Note that  $\mathbf{u}_{ctrl}(t)$  outputs only the joint angles associated with  $\mathbf{r}^{next}$ .

5) *Determination of Feedback Signals*: When the robot arm attains its new position, a new sensory vector  $\mathbf{s}$  is formed with current sensor readings and presented to the CTH network. Thus, sensory signals provide feedback information about the current state of the arm after the execution of a given motor action. The steps 2–5 continue until the end of the stored trajectory.

It is worth noting that, during the recall process, if the network receives as input a state belonging to the stored trajectory, it searches for another stored state, such that the latter forms a state transition together with the input state. This is equivalent to saying that the network “looks” forward one time step, in order to output the stored pattern that succeeds in time the current one. In summary, during learning the network has a *past-oriented* behavior, whilst during recall the network has a *future-oriented* (one-step ahead) behavior.

### III. ROBOTIC PLATFORM

In the current implementation, we used the PUMA 560 robot, a manipulator with 6 degrees of freedom connected to the Unimation Controller (Mark III) which itself contains several controllers. The separate servo controllers, one for each joint, are driven by a main controller LSI-11/73 CPU. Large parts of the original PUMA controller software were replaced and a VME-based SUN Sparcsystem 4/370 workstation was employed to *directly* control the robot in real-time via a high speed communication link. In addition, we developed a simple user-interface based on a distributed communication tool that allows the robot to be controlled remotely from any personal computer attached to a local area network. Details are given next.

#### A. Hardware and Software for Low-Level Robot Control

Neural-based control algorithms for a robot require the capability to quickly process and respond to high bandwidth sensory input coming from, for example, a video camera. The design of a distributed control system has to overcome the limitations of

current commercially available robot controllers, such as lack of computational power, lack of expandability or compatibility to other systems, and not much transparency of their programming languages or operating systems.

In industrial applications the robot is programmed in an interpreted robot language VAL II. The VAL II software was replaced, since the main controller LSI-11 is not capable of handling high bandwidth sensory input itself. Furthermore, VAL II does not support flexible control by an auxiliary computer. To achieve a tight real-time control by the Unix workstation it was installed the software package *Robot Control C Library and Real-time Control Interface (RCCL/RCI)* [38]. This package allows the user to issue robot motion requests from a high level control program (“planning task” which is written and executed as an ANSI C program) to the trajectory level (“control task”) via shared memory communication. The control task is executed periodically at a high priority (kernel mode) and is responsible for reading feedback data, generating intermediate joint set-points and carrying out a “watchdog” function. During each control cycle (typically 20 ms), a command package is sent to the robot controller via the parallel port. The receiving main controller LSI-11 CPU is reprogrammed to dispatch commands to the joint servos, collect feedback data from them and perform elementary safety checks. At power-up time the reprogramming software is downloaded and started by the host computer through a serial line, emulating the controller console. The software then resides at the controller and can be addressed through the parallel port.

This robot control scheme was proposed by Walter and Schulten [8]. The present work introduces some new ideas and improvements. Specifically, a user-friendly interface was developed to facilitate robot motion requests and reading of feedback data. Instead of programming the planning and control levels using the C-language functions of the RCCL/RCI package (which demands high degree of expertise), the user includes in his/her C code for the neural network simple function calls, which commands the robot in a transparent way. Moreover, the robot control (motion requests) can be performed remotely, i.e., the neural network can control the robot from any personal computer connected to the same local area network.

#### B. Tool for Distributed Processing

Usually, in the fields of artificial intelligence, pattern recognition and robotics, different modules designed to execute a specific task must be integrated. Each of these modules has its own data structure and analyzes a certain type of pattern. An efficient communication among different modules is crucial to the correct functioning of the distributed control system as a whole. For this purpose we use a new communication framework called *Distributed Applications Communication System (DACS)*, developed by Fink *et al.* [39]. The DACS communication tool was designed to integrate heterogeneous pattern analysis systems that handles different types of data structures. Its functioning is based on the *client-server* architecture: a central computer (the server) manages the resources of the network system, supplying other machines (the clients) with the routes for the requested resource. The DACS tool provides a simple set of functions and libraries for applications to communicate with remote modules.

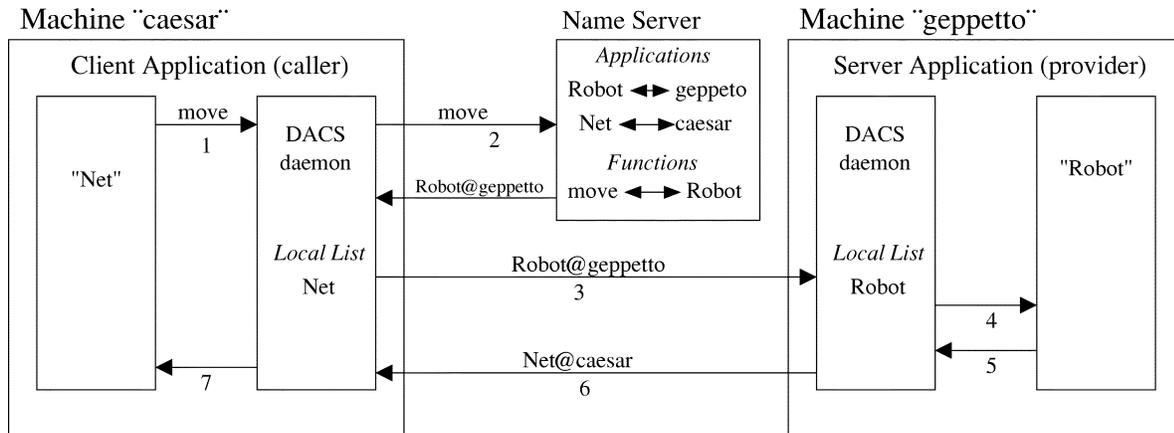


Fig. 3. Interactions during a synchronous function call.

The client computers as well as the server run locally, in background, a program called DACS-daemon, which is responsible for encoding/decoding the data and for correct addressing of the requests. Each module (application) has to register with the system under a unique name that is immediately passed to a name server and enables other modules to address it. The DACS library and daemons are realized as several parallel tasks using *threads*<sup>2</sup> [40]. To avoid that faulty communication links block the whole system or cause deadlocks, separate threads are set up for each module and each connection to another module. Local connections are created when an application registers with the daemon to last until the application unregisters. Network connections are set up dynamically depending on the messages to be sent. Each message has to pass a central routing thread to determine the appropriate connection to be used. This introduces a potential bottleneck, however, only simple operations have to be carried out by this thread.

To access a particular module, a client performs a *remote procedure call* (RPC): a message requesting the use of that module. In the case of acceptance, a bidirectional synchronous RPC is established. This type of RPC blocks the requesting process (client) until a feedback signal from the server arrives, while an asynchronous one does not. To allow transparent communication between modules, C language-dependent data structures used to implement the CTH network are transformed by DACS to a typed and flexibly structure called *Network Data Representation* (NDR) within the application-attached communication front end. This transformation avoid problems with data type inconsistency arising from unidentical data interface definition between service requester and service provider. The data transformations can either be achieved using available routines to generate primitive NDR-objects directly or by automatically generating conversion functions from data type definitions.

In this paper, the client application is the CTH network that delivers, by means of a synchronous RPC, control signals (joint angles) requesting the robot to move to a certain position. The server processes such an RPC, encodes it to the RCCL/RCI syntax, and sends the request to the RCCL/RCI application, which sends the control signals to the robot. After the motion is

completed, the new position of the robot is read by the sensors and the way back is performed. That is, it goes from the robot to RCCL/RCI application, from RCCL/RCI to the server, which translates the feedback signal to the neural network syntax, and from the server to the neural network via DACS. The entire process is transparent for the neural network user; the only thing he/she has to know is the DACS syntax to implement the RPC.

Details about the interaction between the CTH network (client) and the RCCL/RCI package (server) during a synchronous RPC through DACS is shown in Fig. 3. The application called *robot* provides a function called *move*. In the machine *caesar*, the application *net* resides, only knowing that there is a service provided by the function *move* available anywhere in the system. The applications and the functions are already registered in the name server at this stage. Seven steps are performed by DACS to carry out a synchronous remote procedure call to function *move* from function *net*:

- 1) The *net* application sends a message to the local DACS-daemon, addressing it to *move*. This message is tagged as type "function" which affects the resolving of the address.
- 2) Using an usual RPC to the name server, the DACS-daemon determines where the function *move* is located. The result of the name server request is the full address of the application that registered the function *move*—*robot@geppetto* in this case.
- 3) The DACS-daemon on *caesar* sends the message to the DACS-daemon on *geppetto*.
- 4) The application *robot* is found in the table of local applications, so the message is delivered to the application providing the function *move*. The DACS library decodes the address and the arguments and calls the appropriate function. The result is encoded and addressed to the sender of the message—here *net@caesar*.
- 5) The result is delivered to the local DACS-daemon on the machine *geppetto*.
- 6) This daemon delivers the message directly to the daemon on machine *caesar*. Note that at this point no call to the name server is necessary since the address is already known.
- 7) Finally, the daemon delivers the message locally to the *net* application where the library decodes the result and returns it to the caller.

<sup>2</sup>In programming, a part of a program that can execute independently of other parts.

IV. TESTS WITH THE DISTRIBUTED CONTROL SYSTEM

The CTH network presented in this paper was previously evaluated in robotic tasks through simulations [30]. Simulations have the same advantages of *off-line* robot programming and constitute an important step toward the complete evaluation of a robot control system. This is particularly true when the implementation in a real robot is costly, dangerous to humans, or can damage the robot hardware if the user is not an expert. However, simulation environments are only an approximation of reality. Thus, the ultimate goal of a neurocontrol algorithm developed under simulated conditions is to be implemented as a controller of a real robot. The PUMA 560 robot used in the tests belongs to the Laboratory of Robotics of the Neuroinformatics Group at the University of Bielefeld, Bielefeld, Germany.

Simple and complex trajectories were generated by moving the robot arm through specific pathways within its workspace, sampling at regular intervals and recording the corresponding joint angles and Cartesian positions of the end-effector. These trajectories were then used to train the network. It is worth noting that the network training procedure could be carried out *on the fly*, i.e., while the trajectories are being generated. It is also possible for the robot operator him/herself to specify a sequence of angular positions that obeys the range of values imposed to the joint angles, without actually moving the robot and then to compute the corresponding Cartesian positions of the end-effector through the forward kinematic function available in the RCCL/RCI package.

The ranges of values for the joint angles are the following (in degrees):

- $\theta_1 \in [-120, 45]$ ;
- $\theta_2 \in [-140, -90]$ ;
- $\theta_3 \in [-5, 90]$ ;
- $\theta_4 \in [-90, 90]$ ;
- $\theta_5 \in [-80, 0]$ ;
- $\theta_6 \in [30, 150]$ .

The fixed context is always set to the final Cartesian position of the end-effector for a given trajectory. The temporal context has depth  $L = 2$ , i.e.,  $C_T(L, t) = \{s(t-1), s(t-2)\}$  and its initial value is  $C_T(L, 0) = \{s(0), s(0)\}$ , where  $s(0)$  is the triggering state. The values of the other parameters are the following:  $\varepsilon = 10^{-6}$  (training),  $\varepsilon = 1$  (recall),  $a_{\max} = 1$ ,  $a_{\min} = 0.98$ ,  $\beta = 100$ ,  $\eta = 1$ , and  $\lambda = 0.8$ .

Two trajectories, a simple and a complex one were presented to the CTH network only once (since  $\eta = 1$ ), one after the other. After a trajectory is learned it can be retrieved, in a state-by-state basis, through the scheme depicted in Fig. 2. Every angular position achieved by the robot is measured through optical encoders mounted at each joint and then sent to the network input to continue the recall process. This process is repeated until the end of the trajectory has been reached. A polynomial is fitted to the sequence of retrieved joint angles in order to smooth the robot movements.

The measurement of angular positions by the optical encoders is inherently inaccurate, depending basically on the sensor calibration and the end-effector velocity. The faster is the end-effector, the higher is the inaccuracy due to dynamic coupling between arm segments. For all the tests performed in this paper, the end-effector velocity was set to 0.5 m/s,

TABLE I  
STORED AND MEASURED (IN ITALIC) VALUES OF THE JOINT ANGLES FOR A SIMPLE TRAJECTORY. MSE VALUES FOR THE JOINT ANGLES ARE GIVEN IN THE LAST LINE

$l$	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	$\theta_6$
1	-90.019 <i>-89.993</i>	-120.389 <i>-119.723</i>	4.008 <i>4.665</i>	0.0150 <i>0.1042</i>	-63.585 <i>-62.677</i>	89.889 <i>90.024</i>
2	-78.953 <i>-79.312</i>	-118.327 <i>-118.411</i>	0.787 <i>0.905</i>	0.0071 <i>0.1042</i>	-62.124 <i>-62.488</i>	100.958 <i>100.551</i>
3	-69.901 <i>-70.274</i>	-118.557 <i>-118.536</i>	1.115 <i>1.119</i>	0.0009 <i>0.1042</i>	-62.550 <i>-62.443</i>	110.012 <i>109.575</i>
4	-60.777 <i>-61.103</i>	-120.814 <i>-120.744</i>	4.679 <i>4.551</i>	-0.0057 <i>0.1042</i>	-63.829 <i>-63.740</i>	119.140 <i>118.785</i>
5	-53.822 <i>-54.065</i>	-121.313 <i>-124.216</i>	10.317 <i>10.128</i>	-0.0101 <i>0.1042</i>	-65.968 <i>-65.852</i>	126.096 <i>125.804</i>
6	-18.706 <i>-48.954</i>	-128.386 <i>-128.186</i>	17.031 <i>16.691</i>	-0.0135 <i>0.1042</i>	-68.611 <i>-68.460</i>	131.213 <i>130.922</i>
7	-13.967 <i>-44.149</i>	-133.817 <i>-133.591</i>	26.300 <i>25.928</i>	-0.0162 <i>0.1042</i>	-72.150 <i>-72.279</i>	135.952 <i>135.724</i>
	0.0751	0.0801	0.1092	0.0116	0.1305	0.1033

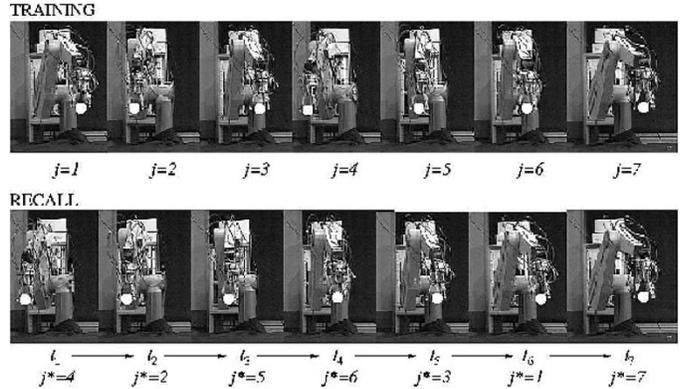


Fig. 4. Learning (upper row) and recall (lower row) of the states a trajectory. White circles mark the positions of the end-effector.

which is a value sufficient to move the arm fast enough and to produce accurate measures of the joint angles. The errors between stored and measured joint angles are computed by  $MSE_i = (1/T) \sum_{t=1}^T [\theta_i^m(t) - \theta_i^s(t)]^2$ , where  $\theta_i^m$  and  $\theta_i^s$  are the measured and stored values of the  $i$ th joint angle, respectively.

In the first test, it is shown which neuron stored a particular state of a trajectory and its order in time during trajectory reproduction. For this test, it was used a simple trajectory with  $N = 7$  states, approximately drawing a straight line. For simplicity, since there is no recurrent state, a network of only  $M = N = 7$  neurons is trained with  $K = 1$ . The upper row of Fig. 4 shows each neuron  $j$  of the network, numbered from left to right and the trajectory state each one stored. The lower row of Fig. 4 shows the sequence of active neurons during a correct trajectory reproduction. Table I shows the numerical values of the joint angles stored in the CTH network for the trajectory in Fig. 4 and the values measured by the optical encoders which were used as feedback signals during trajectory reproduction.

For the next test, we set  $K = 2$  and increased the number of neurons to  $M = 30$ . Fig. 5(a) shows a sequence of arm positions approximately describing an eight-figure trajectory, together with the index of the neuron that stored the position and the corresponding instant of occurrence of that position. This trajectory has seven states with an intermediate state occurring

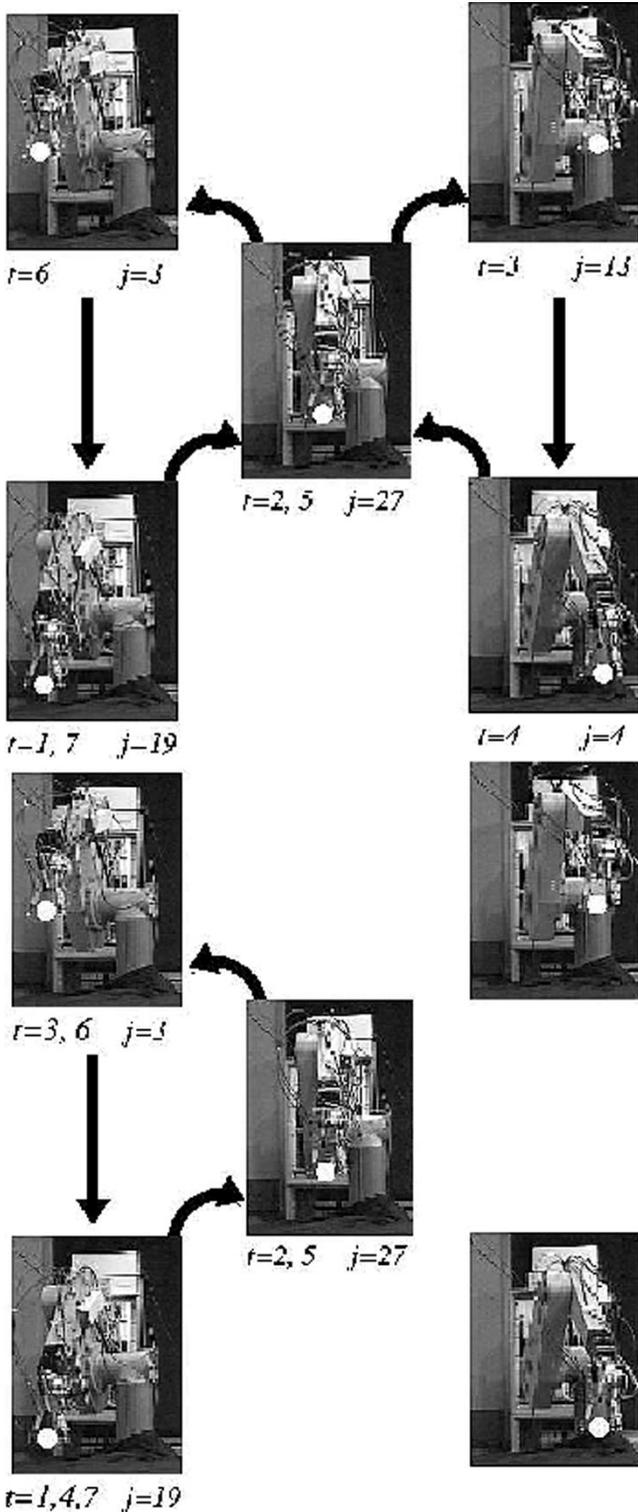


Fig. 5. Eight-figure trajectory performed by the PUMA 560 robot. The arrows represent state transitions.

twice ( $t = 2$  and  $t = 5$ ) but in different temporal context. Eight-figure trajectories, due to the existence of recurrent states, have been widely used as benchmark in temporal sequence recall to demonstrate the role of temporal context information. Table II shows the stored and measured values of the joint angles for the eight-figure trajectory.

TABLE II  
STORED AND MEASURED (IN ITALIC) VALUES OF THE JOINT ANGLES FOR AN EIGHT-FIGURE CLOSED TRAJECTORY. MSE VALUES FOR THE JOINT ANGLES ARE GIVEN IN THE LAST LINE

$l$	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	$\theta_6$
1	-90.000 <i>-89.478</i>	-135.000 <i>-134.822</i>	0.000 <i>0.2011</i>	0.000 <i>0.0189</i>	-15.000 <i>-45.414</i>	90.000 <i>90.512</i>
2	-70.583 <i>-70.958</i>	-122.661 <i>-122.543</i>	-1.3379 <i>-1.1261</i>	0.0082 <i>0.0189</i>	-55.997 <i>-56.335</i>	109.368 <i>108.980</i>
3	-53.631 <i>-53.990</i>	-136.899 <i>-136.600</i>	6.1990 <i>6.3412</i>	0.0082 <i>0.0189</i>	-19.597 <i>-49.748</i>	126.317 <i>125.939</i>
4	-53.631 <i>-53.617</i>	-121.623 <i>-122.159</i>	13.588 <i>13.359</i>	0.0066 <i>0.0189</i>	-71.962 <i>-70.971</i>	126.320 <i>126.260</i>
5	-70.583 <i>-70.199</i>	-122.661 <i>-122.643</i>	-1.3379 <i>-0.9854</i>	0.0082 <i>0.0189</i>	-55.997 <i>-56.361</i>	109.368 <i>109.726</i>
6	-90.007 <i>-90.019</i>	-116.281 <i>-116.804</i>	8.1602 <i>8.2314</i>	0.0070 <i>0.0189</i>	-72.181 <i>-71.221</i>	89.917 <i>89.8762</i>
7	-90.000 <i>-89.467</i>	-135.000 <i>-134.684</i>	0.000 <i>-0.0201</i>	0.000 <i>0.0095</i>	-15.000 <i>-45.319</i>	90.000 <i>90.504</i>
	0.1388	0.1138	0.0185	0.0002	0.3495	0.1352

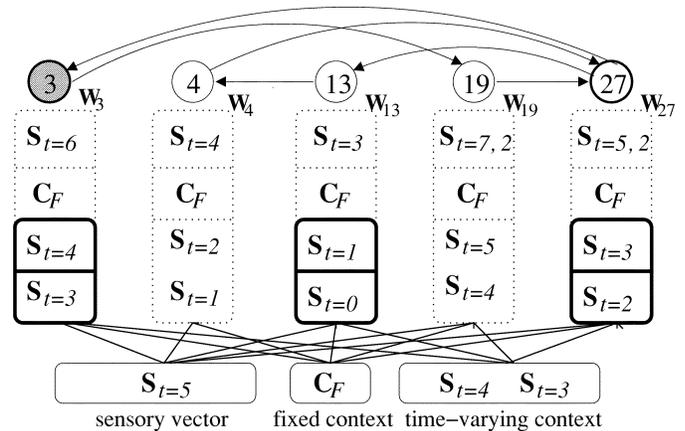


Fig. 6. Resolving ambiguities during recall of complex trajectory.

Fig. 6 illustrates how (11) treats ambiguities. No matter what branch of the figure-eight the robot arm is currently executing, when it arrives at the crossing point ( $t = 2$  and  $t = 5$ ), it has to decide between one of two possible directions to follow. This ambiguity is resolved by the time-varying context. The states  $s_{t=2}$  and  $s_{t=5}$  are equal, but they occur in different temporal contexts. During training both states were stored by neuron  $j = 27$  (Fig. 6). During recall, when the robot arm reaches the state  $s_{t=5}$ , the network has to decide which is the next state (control signal) to be sent to the robot. Following the lateral connections in Fig. 6, the candidates for the next trajectory state are stored in the weight vectors of neurons  $j = 3$  and  $j = 13$ . Since the repeated states belong to the same trajectory, the fixed context is the same for the two neurons. Only, the time-varying context contains the information (past states) that can extinguish the ambiguity:  $C_T(5, L) = \{s_{t=4}, s_{t=3}\}$ . This information matches exactly with that in the lower portion of the weight vector  $w_3$ , i.e.,  $C_T(5, L) \equiv w_3^T$ . Hence,  $D_3^T(t) < D_{13}^T(t)$ , implying that  $y_3 > y_{13}$ . Thus, neuron  $j = 3$  is chosen and the control signal is extracted from  $w_3^T$  as shown in (12).

Table III shows typical results for reproduction of the eight-figure trajectory obtained for different situations. The second line shows the first winners, i.e., those labeled as  $\mu_1(t)$  in (4), for

TABLE III  
 WINNING NEURONS DURING TRAINING AND RECALL. (RECALL-1) THE TEMPORAL CONTEXT IS TURNED OFF. (RECALL-2) NEURONAL FAULTS ARE SIMULATED. (RECALL-3) NOISE IS ADDED TO THE INPUT

	t=1	t=2	t=3	t=4	t=5	t=6	t=7
<b>Training</b>	19	27	13	4	27	3	19
<b>Recall-1</b>	19	27	3	19	27	3	19
<b>Recall-2</b>	5	22	28	5	22	1	5
<b>Recall-3</b>	19	22	13	4	27	1	19

the seven time steps of the stored trajectory during training. The third line (recall-1) shows those neurons activated during the reproduction phase disregarding the temporal context information. It can be noted that errors occurred since the active neurons during training are different from those active during recall. The consequence of an error is that the robot arm cannot go through the two sides of the eight-figure trajectory, being trapped in one side only, as shown in Fig. 5(b)

The next two tests illustrate the fault and noise tolerance of the CTH network when trained with  $K > 1$ . The fourth line (recall-2) shows the neurons activated after simulating the collapse of all  $\mu_1$  neurons. Since  $K = 2$ , the responsibility for the reproduction of the trajectory is assumed by the second winners, i.e., those neurons labeled as  $\mu_2$  during training. Since  $a_{\mu_2} = a_{\min} < a_{\mu_1} = a_{\max}$ , the trajectory is reproduced with an error ( $MSE_{\mu_2} = 0.13589$ ) that is slightly higher than that obtained by summing up the last line of Table II and averaging over the 6 joints ( $MSE_{\mu_1} = \sum_{i=1}^6 MSE_i/6 = 0.131$ ). The fifth line (recall-3) shows the neurons activated when simulated Gaussian white noise (zero mean, variance=0.1) is added to the network input. The additional noise forces the network to select sometimes the second winners rather than the first winners as shown in Table III for  $t = 2$  and  $t = 6$ . This occurs because the second winners are now closer to the *noisy* input states than the first winners, in an Euclidean distance sense. Depending on the magnitude of the noise, incorrect evaluation of the neurons can occur, impairing the trajectory reproduction, however such a situation was not observed in the experiments. This robustness property of the CTH network is equivalent to that of the SOM model [36]. The main difference is that the neurons in the CTH network do not need to be arranged in a fixed lattice with pre-defined neighborhood relationships.

## V. DISCUSSION

The CTH network was applied to the learning of robot trajectories, a common problem in industrial scenarios. Usually, a trajectory is “taught” to the robot by the so-called *walk-through* method, in which an operator guides the robot end-effector through the sequence of desired positions [41]. These positions and their temporal order are then stored in the controller memory (*look-up table*) for posterior recall by the operator him/herself. This method is time-consuming and costly because the robot is out-of-production during the trajectory learning process. Furthermore, as the trajectory being learned becomes more and more complex and/or several trajectories

have to be learned, the robot operator may experience difficulties in setting the temporal order of the states of the trajectories and resolving all potential ambiguities due to recurrent trajectory states. Thus, any trajectory learning algorithm that reduces the time the robot takes off-line is welcome. The CTH network described in this paper was designed with this goal in mind since the learning process is very fast, can be carried out in an online fashion and the resolution of ambiguities requires minimal human supervision. In addition, the CTH network exhibits some tolerance to neuronal failure and to noise, properties that are not present in the conventional (nonadaptive) look-up table methods [42]. These two aspects are particular important if the proposed method is to be implemented in hardware.

It is important to note that the CHT network acts basically as a *trajectory planner* or *high-level trajectory controller*, setting the reference values for the low-level joint controllers at each time step. In this paper, the joint controllers are conventional PID controllers, but other control methods, such as fuzzy or neural, could be equally used. Thus, the CHT network can be applied independently of the type of low-level controllers. The feedback pathway shown in Fig. 2 allows the CTH network to work autonomously, monitoring the trajectory reproduction process in a step-by-step basis. This is important for safety purposes since a trajectory only continues to be reproduced if the feedback pathway exists.

Thus, if any problem occurs during the execution of the required motion by the robot, such collision with an obstacle or the joints reach limit values, the feedback pathway can be interrupted and the reproduction is automatically stopped. The conventional walk-through method does not possess the feedback pathway. In this case, all the trajectory states are sent to a memory buffer and executed in batch-mode. If any problem occurs, one has to wait for the execution of the whole trajectory in order to take a decision or to turn-off the robot power. From the exposed, we can say that the neurocontrol scheme shown in Fig. 2 is a kind of *self-monitored trajectory reproduction* approach.

*Generalized Versus Specialized Inverse Modeling and Control:* It can be said that we have proposed a neurocontrol system which implements a *specialized inverse modeling and control scheme* [1] in the sense that the CTH network is used to train the network to operate in specific (localized) regions of the robot workspace defined by the learned trajectories. Other real-time implementations of self-organizing architectures implement a *generalized* inverse modeling and control scheme since they try to learn sensorimotor mappings globally [4], [8]–[10].

The success of the generalized method depends on the ability of the neural network to generalize correctly to respond to inputs it has not been trained on. Thus, the training samples will have to cover the input space of the plant and hence this procedure is not very efficient since the network will have to learn the responses of the plant over a wide range than what may be actually necessary. As pointed out by Prabhu and Garg [1] the specialized scheme produces more accurate positioning results and is suitable for on-line learning as we have shown in this paper.

Furthermore, none of the generalized real-time learning schemes cited above take into account temporal aspects of the robotic task. As far as we know, our approach is the first

real-time implementation of a specialized inverse modeling and control system that automatically takes into account sequential aspects of the robotic task. Several supervised approaches have been proposed and simulated [22], [23], [28], but they are not suitable for online learning since they require a very long training process.

*Comparison With Other Temporal SONNs for Robotics:* We have noted that few temporal SONNs have been applied to control of manipulators [25], [29], [31]. The network by Althöfer and Bugmann [25] cannot handle trajectories with recurrent states, whereas the model by Barreto and Araújo [29] can only deal with trajectories with shared states. None of them can learn and recall trajectories that contain both repeated and shared states.

Another network by Barreto and Araújo [31] is able to handle recurrent trajectory states but uses memory space inefficiently since every time a recurrent state occur it is stored by a neuron different from the one that stored this state previously. The similarity radius mechanism allows the CTH network to save memory space by maintaining only a single copy of each recurrent state of a trajectory. For example, to store the seven states of the eight-figure trajectory, the network by Barreto and Araújo [31] requires seven neurons, whereas the CTH network requires only five neurons, as shown in Fig. 6, since the recurrent states are stored only once. The larger the number of occurrences of a given state of the trajectory, the larger the savings in memory use.

*Selection of CTH Network Parameters:* The CTH network has a relatively high number of nine parameters, but they are fairly easy to select. If one always set  $a_{\max} = \eta = \lambda = 1$ , the other parameters can be chosen as follows:

- 1) A value  $\beta = 100$  or  $1000$  suffices to exclude neurons from competition.
- 2) The simulations in [30] suggest  $K = 1$  or  $2$ .
- 3) For these values of  $K$ , the value  $a_{\min} = 0.98$  results in acceptable accuracy in the case of faults.
- 4) The length  $L$  of the local context is chosen on a trial-and-error basis, but it can be made adaptive (see [21]). The hint is to start with  $L = 1$  and increase this value if the network is unable to resolve possible ambiguities.
- 5) The number of output neurons  $M$  must guarantee the storage of  $n$  original trajectories and the associated  $K - 1$  redundant sequences. If  $n$  is known beforehand,  $M = K \cdot \sum_{l=1}^n N^l$ , where  $N^l$  is the number of components of the sequence  $l$ .

If  $n$  is unknown,  $M$  should be given initially a reasonably high value. If this number is eventually found to be insufficient, a constructive technique should be used to add new neurons to the network [43].

*Why Not to Use Other Distributed Communication Tools?:* There are several tools available that provide useful communication primitives for distributed processing [44]. Nevertheless, some inconveniences in each of the most popular ones motivated the implementation of the DACS tool which is more flexible and general than the other tools. For example, the **ONC RPC** [45] tool allows synchronous calls in a typical client-server concept with well structured data types that have

to be known at compile time. The necessary knowledge of the destination machine when getting the client handle makes it hard to distribute several modules in a dynamical way [45]

Another possibility is to use the **PVM** [46], which is a well-known standard system used to parallelize complex algorithms. However, the PVM is a totally distributed system without a centralized instance of a service to keep track of the system configuration which results in an enhanced overhead during reconfiguration. Furthermore, the facilities to exchange data between heterogeneous architectures are very basic. Several other tools have been investigated by Fink *et al.*, [39], but due to reasons similar to the ones mentioned above, they were not selected to be used in the distributed system proposed in this paper.

## VI. CONCLUSIONS

In this paper, we demonstrated the feasibility of using a temporal self-organizing neural network to real-time control an industrial robot. The implementation of the control scheme is facilitated by the separation of the network design and the robot control task into two distinct modules which are linked through a distributed communication tool. The resulting distributed neurocontrol system is simple, fast and robust to noise and faults.

We believe that the proposed distributed control system is flexibly enough to be further improved and applied to standard industrial robotic manufacturing systems, assembly lines, spray painting, selective soldering, point soldering, etc.

## REFERENCES

- [1] S. M. Prabhu and D. P. Garg, "Artificial neural network based robot control: An overview," *J. Intell. Robot. Syst.*, vol. 15, pp. 333–365, 1996.
- [2] S. N. Balakrishnan and R. D. Weil, "Neurocontrol: A literature survey," *Math. Comput. Modeling*, vol. 23, no. 1–2, pp. 101–117, 1996.
- [3] J. Heikkonen and P. Koikkalainen, "Self-organization and autonomous robots," in *Neural Systems for Robotics*, O. Omidvar and P. van der Smagt, Eds. New York: Academic, 1997, pp. 297–337.
- [4] M. Kuperstein and J. Rubinstein, "Implementation of an adaptive neural controller for sensory-motor coordination," *IEEE Contr. Syst. Mag.*, vol. 9, no. 3, pp. 25–30, 1989.
- [5] T. M. Martinez, H. J. Ritter, and K. J. Schulten, "Three-dimensional neural net for learning visuomotor coordination of a robot arm," *IEEE Trans. Neural Networks*, vol. 1, pp. 131–136, Jan. 1990.
- [6] M. Kuperstein, "Infant neural controller for adaptive sensory-motor coordination," *Neural Networks*, vol. 4, pp. 131–145, 1991.
- [7] H. Ritter, T. Martinez, and K. Schulten, *Neural Computation and Self-Organizing Maps: An Introduction*, Addison-Wesley, Ed. Reading, MA, 1992.
- [8] J. A. Walter and K. J. Schulten, "Implementation of self-organizing networks for visuo-motor control of an industrial robot," *IEEE Trans. Neural Networks*, vol. 4, pp. 86–95, Jan. 1993.
- [9] T. Hesselroth, K. Sarkar, P. van der Smagt, and K. Schulten, "Neural network control of a pneumatic robot arm," *IEEE Trans. Syst., Man, Cybern.*, vol. 24, pp. 28–38, Jan. 1994.
- [10] M. Jones and D. Vernon, "Using neural networks to learn hand-eye co-ordination," *Neural Comput. Applicat.*, vol. 2, pp. 2–12, 1994.
- [11] P. Morasso and V. Sanguineti, "Self-organizing body schema for motor planning," *J. Motor Behavior*, vol. 27, no. 1, pp. 52–66, 1995.
- [12] J. L. Buessler, R. Kara, P. Wira, H. Kihl, and J. P. Urban, "Multiple self-organizing maps to facilitate the learning of visuo-motor correlations," *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, vol. III, pp. 470–475, 1999.
- [13] J. Piaget, *The Origin of Intelligence in Children* Paris, France, 1963.
- [14] G. A. Barreto and A. F. R. Araújo, "Time in self-organizing maps: An overview of models," *Int. J. Comput. Res.*, vol. 10, no. 2, pp. 139–179, 2001.
- [15] J. Kangas, "Phoneme recognition using time-dependent versions of self-organizing maps," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, pp. 101–104, 1991.

- [16] G. J. Chappell and J. G. Taylor, "The temporal Kohonen map," *Neural Networks*, vol. 6, no. 3, pp. 441–445, 1993.
- [17] T. Koskela, M. Varsta, J. Heikkonen, and K. Kaski, "Time series prediction using recurrent SOM with local linear models," *Int. J. Knowledge-Based Intell. Eng. Syst.*, vol. 2, no. 1, pp. 60–68, 1998.
- [18] O. Carpinteiro, "A hierarchical self-organizing map model for sequence recognition," *Neural Processing Lett.*, vol. 9, no. 3, pp. 209–220, 1999.
- [19] T. Voegtlin, "Context quantization and contextual self-organizing maps," in *Proc. IEEE-INNS-ENNS Int. Joint Conf. Neural Networks*, vol. 6, Como, Italy, 2000, pp. 20–25.
- [20] K. Horio and T. Yamakawa, "Feedback self-organizing map and its application to spatio-temporal pattern classification," *Int. Comput. Intell. Applicat.*, vol. 1, no. 1, pp. 1–18, 2001.
- [21] D.-L. Wang and B. Yuwono, "Anticipation-based temporal pattern generation," *IEEE Trans. Syst., Man Cybern.*, vol. 25, no. 4, pp. 615–628, 1995.
- [22] M. I. Jordan, "Attractor dynamics and parallelism in a connectionist sequential machine," in *Proc. 8th Annu. Conf. Cognitive Sci. Soc.*, Amherst, MA, 1986, pp. 531–546.
- [23] L. Massone and E. Bizzi, "A neural network model for limb trajectory formation," *Biolog. Cybern.*, vol. 61, pp. 417–425, 1989.
- [24] M. I. Jordan and D. E. Rumelhart, "Forward models: Supervised learning with a distal teacher," *Cogn. Sci.*, vol. 16, pp. 307–354, 1992.
- [25] K. Althöfer and G. Bugmann, "Planning and learning goal-directed sequences of robot arm movements," in *Proc. Int. Conf. Artificial Neural Networks*, Paris, France, 1995, pp. 449–454.
- [26] G. Bugmann, K. L. Koay, N. Barlow, M. Phillips, and D. Rodney, "Stable encoding of robot trajectories using normalized radial basis functions: Application to an autonomous wheelchair," in *Proc. Int. Symp. Robotics*, Birmingham, U.K., 1998, pp. 232–235.
- [27] A. F. R. Araújo and M. Vieira, "Associative memory used for trajectory generation and inverse kinematics problem," in *Proc. IEEE Int. Joint Conf. Neural Networks*, Anchorage, AK, 1998, pp. 2052–2057.
- [28] A. F. R. Araújo and H. D'Arbo, "Partially recurrent neural network to perform trajectory planning, inverse kinematics and inverse dynamics," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, San Diego, CA, 1998, pp. 126–133.
- [29] G. A. Barreto and A. F. R. Araújo, "Unsupervised learning and recall of temporal sequences: An application to robotics," *Int. J. Neural Syst.*, vol. 9, no. 3, pp. 235–242, 1999.
- [30] A. F. R. Araújo and G. A. Barreto, "Context in temporal sequence processing: A self-organizing approach and its application to robotics," *IEEE Trans. Neural Networks*, vol. 13, pp. 45–57, Jan. 2002.
- [31] ———, "A self-organizing context-based approach to tracking of multiple robot trajectories," *Appl. Intell.*, vol. 17, no. 1, pp. 101–119, 2002.
- [32] S. Grossberg, "Some networks that can learn, remember and reproduce any number of complicated space-time patterns, I," *J. Math. Mechan.*, vol. 19, pp. 53–91, 1969.
- [33] S. Amari, "Learning patterns and pattern sequences by self-organizing nets of threshold elements," *IEEE Trans. Comput.*, vol. C-21, pp. 1197–1206, Nov. 1972.
- [34] J. Walter and H. Ritter, "Rapid learning with parametrized self-organizing maps," *Neurocomput.*, vol. 12, pp. 131–153, 1996.
- [35] W. K. Estes, *Classification and Cognition*. New York: Oxford Univ. Press, 1994.
- [36] T. Kohonen, *Self-Organizing Maps*, 2nd ed. Berlin-Heidelberg, Germany: Springer-Verlag, 1997.
- [37] A. V. M. Herz, "Spatiotemporal association in neural networks," in *The Handbook of Brain Theory and Neural Networks*, M. A. Arbib, Ed. Cambridge, MA: MIT Press, 1995, pp. 902–905.
- [38] J. Lloyd, M. Parker, and R. McClain, "Extending the RCCL programming environment to multiple robots and processors," in *Proc. IEEE Conf. Robotics Automat.*, Philadelphia, PA, 1988, pp. 465–469.
- [39] G. A. Fink, N. Jungclauss, H. Ritter, and G. Sagerer, "A communication framework for heterogeneous distributed pattern analysis," in *Proc. IEEE Int. Conf. Algorithms Applicat. Parallel Process.*, 1995, pp. 881–890.
- [40] G. Coulouris and J. Dollimore, *Distributed Systems—Concepts and Design*, 2nd ed. Reading, MA: Addison-Wesley, 1994.
- [41] K. Fu, R. Gonzalez, and C. Lee, *Robotics: Control, Sensing, Vision and Intelligence*. New York: McGraw-Hill, 1987.
- [42] M. H. Raibert and B. K. P. Horn, "Manipulator control using the configuration space method," *Indust. Robot*, vol. 5, pp. 69–73, 1978.
- [43] B. Fritzke, "Growing cell structures—A self-organizing network for unsupervised and supervised learning," *Neural Networks*, vol. 7, no. 9, pp. 1441–1460, 1994.

- [44] D. Y. Cheng, "A survey of parallel programming languages and tools," NASA Ames Res. Cent., Moffett Field, CA, 1993.
- [45] *Network Programming Guide*. Mountain View, CA: Sun Microsystems, 1990.
- [46] V. S. Sunderam, "A framework for parallel distributed computing," *Concurrency: Practice Experience*, vol. 2, no. 4, pp. 315–339, 1990.



**Guilherme A. Barreto** (S'02) was born in Fortaleza, Ceará, Brazil, in 1973. He received the B.S. degree in electrical engineering from Federal University of Ceará in 1995 and the M.S. degree in electrical engineering from the University of São Paulo, São Paulo, Brazil, in 1998 for the dissertation entitled "Unsupervised neural networks for temporal sequence processing." He currently he is pursuing the D.Phil. degree in electrical engineering at the same university. The theme of his Ph.D. research is self-organizing neural networks for nonlinear systems

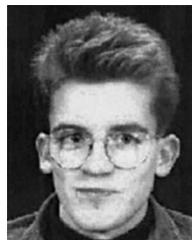
modeling and control.

His research interests are in the areas of unsupervised neural networks and applications in robotics, time series modeling and prediction, and dynamic systems theory.



**Aluizio F. R. Araújo** was born in Recife, Pernambuco, Brazil, in 1957. He received the B.S. degree in electrical engineering from Federal University of Pernambuco, Recife, in 1980, the M.S. degree in electrical engineering from the State University of Campinas, Campinas, Brazil, in 1988, and the D.Phil. degree in computer science and artificial intelligence from the University of Sussex, Sussex, U.K., in 1994.

He worked in São Francisco Hidroelectrical Company for five years, and in 1998, he became an assistant professor at University of São Paulo, where, in 1994, he was promoted to Adjunct Professor. His research interests are in the areas of neural networks, machine learning, robotics, dynamic systems theory, and cognitive science.



**Christof Dücker** received the diploma in computer science from the University of Bielefeld, Bielefeld, Germany, in 1995. He is currently pursuing the Ph.D. degree in computer science from the University of Bielefeld, working with the Neuroinformatics Group (AG Neuroinformatik) and the Sonderforschungsbereich 360, Project D4 "Multisensorbased Exploration and Assembly."

His field of research is the definition and implementation of robot manipulator control within the SFB 360 context.



**Helge Ritter** studied physics and mathematics at the Universities of Bayreuth, Heidelberg and Munich, Germany, and received the Ph.D. degree in physics from the Technical University of Munich in 1988.

Since 1985, he has been engaged in research in the field of neural networks. In 1989, he moved, as a guest scientist, to the Laboratory of Computer and Information Science, Helsinki University of Technology, Helsinki, Finland. Subsequently, he was assistant research professor at the then newly established Beckman Institute for Advanced Science and Technology and the Department of Physics, University of Illinois at Urbana-Champaign. Since 1990, he has been a professor with the Department of Information Science, University of Bielefeld. His main interests are principles of neural computation, in particular, self-organizing and learning systems and their application to machine vision, robot control, data analysis, and interactive man-machine interfaces.

Dr. Ritter was awarded the SEL Alcatel Research Prize in 1999 and the Leibniz Prize of the German Research Foundation DFG in 2001.