Learning Convergence in the Cerebellar Model Articulation Controller

Yiu-fai Wong and Athanasios Sideris

Abstract—This paper presents a new way to look at the learning algorithm in the cerebellar model articulation controller (CMAC), proposed by Albus [1], [2]. We obtain a proof that the CMAC learning always converges with arbitrary accuracy on any set of training data. We also propose an alternative way to implement CMAC based on the insights obtained in the process. We test the new implementation scheme with a computer simulation for learning the inverse dynamics of a two-link robot arm.

I. INTRODUCTION

The cerebellar model articulation controller (CMAC) was proposed by Albus [1], [2]. It has been traditional to regard it as a look-up table. Actually CMAC is capable of very fast learning and shares certain features with interpolation and approximation. However this model has often been overlooked by the neural network research community. But the ideas behind CMAC can be found in Kanerva associative memory [3] and certain learning schemes using neurons with local receptive fields [4], [5]. Recently, there has been growing interest in CMAC. Several researchers have applied this model of neural network to solve various problems. For example, Moody [5] has used a modified form of CMAC for predicting chaotic time series. Hormel [6] has applied Kohonen-type algorithms to adapt the storage mechanisms of CMAC to match the input distribution. Carter and others [7] have investigated the fault tolerance of CMAC networks. Kraft and Campagna [8] have compared a CMAC-based controller with two traditional adaptive controllers. Miller has had extensive experience in using CMAC for real-time control of robots with vision [9], [10].

In this paper we will present a brief review of CMAC. We then show the nature of learning in the system and prove the key result that the CMAC learning algorithm always converges.

II. MECHANISMS OF CMAC

A schematic sketch of CMAC is shown in Fig. 1. We will now describe briefly the building blocks of CMAC.

For simplicity, let us start with CMAC for one-dimensional mapping, say $R \rightarrow R$. Suppose the input is bounded between $[a, b]$ and the interval is discretized into a fixed number of levels in natural order indexed by integers. Each input $x$ can then be identified by its discretization level, $x_d$, an integer. There is a mapping between the discretized input space and a set $S$ of association cells which is also indexed by the same set of integers defined by the discretization levels. For each input $x$, we first compute its discretization level, $x_d$. Then $A^*$ consecutive association cells are excited starting at $x_d$. (This will be true if, instead of describing which association cells are excited by a particular input, we can also describe which inputs excite a particular association cell. That is, we can use receptive fields of the association cells to describe this mapping.) We then sum the weighted outputs of these cells to get the CMAC output. Note that this map induces a metric on $S$ such that nearby inputs excite some common association cells.

If the discretization is very fine, there will be too many associations cells so that it is physically impossible to implement them (this is especially true for multidimensional inputs). Albus solved this problem by hash coding the set of association cells into a smaller, manageable memory, $A_p$, with $A_p$ memory locations. Each location is associated with a weight. Therefore, the $A^*$ association cells are mapped to $A^*$ addresses in $A_p$ by hash coding. We then sum the weights from these addresses to produce the output. Because of the hashing procedure, in the following, we will sometimes identify association cells with addresses and cell outputs with weights.

Collision occurs when two different association cells are mapped to the same address. The famous “birthday” problem [11] reminds us that collisions are bound to occur. For example, suppose we want to learn a function of one variable on some bounded interval which is discretized such that 300 input–output pairs are selected for training. Suppose further that each input $x$ excites $A^* = 31$ cells, and the average overlap between the association cells that the neighboring inputs excite is 27. This is because we train CMAC at some sampled points along the graph and hope or assume that
CMAC will generalize to the untrained ones. Then 1227 = 31 + 4 × 299 distinct cells will be excited. To calculate the probability of no collision if we have a memory of $10^6$ addresses, we consider the problem of putting $r$ balls into $n$ cells. The probability that each cell has at most one ball is

$$
\frac{(n)_r}{n^r} = \left(1 - \frac{1}{n}\right) \cdots \left(1 - \frac{r - 1}{n}\right).
$$

Therefore, for the example above, the probability of no collisions is only 0.0005.

Having described the mapping from input to output, we next describe the learning algorithm for CMAC. Given the $i$th training sample, the desired output is $d_i$ and the network output is $g_i = \sum w_j$, where $w_j$ is the weight stored in memory location $j$ and $j$ indexes the cells which are excited. The error signal is $e_i = d_i - g_i$. The CMAC scheme calls for evenly distributing the error among the weights. That is, for each address excited by the input,

$$
\delta w_j = \delta_i / A^*.
$$

Note that this is one-shot error correction. This is the basic skeleton of CMAC. The reader is referred to [1] for details and the case of multidimensional inputs.

III. UNDERSTANDING CMAC

Although CMAC has a simple learning rule, it is capable of fast learning. This has been reported by Albus [1] and Moody [5]. It is also confirmed in the example shown in Section V, where CMAC tries to learn the inverse dynamics of a two-link robot arm along a trajectory. The training scheme is shown in Fig. 2.

As pointed out by Albus, CMAC also has a proper generalization property. This can be intuitively understood from the overlapping association cells which nearby inputs excite, providing some degree of interpolation. For a new input $x'$ which is close to some learned inputs $x_1, \ldots, x_k$, the association cells that $x'$ excites will have some overlap with the association cells these learned inputs excite. Therefore, a natural interpolation occurs. The more the overlap, i.e., the larger $A^*$, the better the generalization. But there is a price to pay for larger $A^*$. Specifically, the computational complexity will be higher and a larger $A_p$ would be required so that hashing would not pose serious problems for learning convergence, as we will see later on.

One natural question to ask is, Is CMAC capable of learning any mapping? Miller [9] showed that one can identify the CMAC learning rule with the LMS rule [12]. This ensures the convergence of CMAC learning to some local minima because the LMS rule does not guarantee global convergence. This is not too satisfactory because simulation results seem to suggest that CMAC learns very well. In fact, no one has previously done a rigorous analysis of CMAC. However, for precise analysis of CMAC, we found it useful to assume that we have a big enough memory for the association cells so that there is no need for hashing. If we do that, we can actually prove that CMAC is capable of learning any mapping. However, we will get back to the hash coding to address the observation that the problem associated with collision in hash coding does not seem to hinder the accuracy of the learning process. Here is the statement of our main result and its proof.

**Theorem 1:** Given a set of training samples composed of input–output pairs from $\mathbb{R}^m \rightarrow \mathbb{R}^m$, CMAC always learns the training set with arbitrary accuracy if the input space is discretized such that no two training input samples excite the same set of association cells.

**Proof:** For better exposition of CMAC, we will assume $n = m = 1$ until near the end of the proof since the proof for the case of $\mathbb{R} \rightarrow \mathbb{R}$ easily extends to the more general case stated in the theorem.

The $i$th training sample has its input excite $A^*$ association cells. Let $x_{ij}$ be its $j$th address, $j = 1, \ldots, A^*$. (Here we identify address with association cell.) Let $g(x_{ij})$ be the weight at that address. Then the output for the $i$th sample is $g_i = \sum g(x_{ij})$. If $d_i$ is the desired output, the error is $e_i = d_i - g_i$. According to the CMAC learning rule, these $A^*$ weights are changed by an amount $\delta_i / A^*$, as shown in (2).

Generally there are two ways to update the weights. One can update after each epoch. One can also update after each presentation. First, we look at the latter case. But the conclusion is the same for both cases, as will be seen later. After the $k$th presentation, how is the $k$th update going to affect the outputs for the other training samples? Because of the $k$th update, the new output $g_i$ for the $i$th training sample is $\sum g(x_{ij}) + c_{ik} \delta_k / A^*$, where $c_{ik}$ is the number of association cells both sample $i$ and $k$ address, as shown in Fig. 1.

But let us look at this procedure in another way. We have $n$ training samples and each of them addresses $A^*$ weights. When we correct for the $k$th training sample at the $l$th iteration, the associated weights are changed by the same amount, namely $\delta_k / A^*$ where $\delta_k$ is the output error for the $k$th training sample at the $l$th iteration. Let us consider the accumulated output error for the $k$th training sample, $E_k = \sum \delta_k$. The total change in the individual weights associated with $k$th training sample caused by updates on itself is $\Delta_k = E_k / A^*$. Let us call this $\Delta_k$ the accumulated weight error for the $k$th input sample. The contribution to the CMAC output of the $i$th training sample will then be $c_{ik} \Delta_k$. When the learning converges, i.e., $\delta_k$ goes to zero, $\Delta_k$ will converge to a constant. If we can recover the weights $w_i$ from the accumulated weight errors $\Delta_i$, we see that, instead of treating

---

1This problem was picked for simulation because of interest in applying CMAC to neural control.
the weights as variables to be learned, we can also treat the accumulated weight errors $\Delta_i$ as variables to be learned.

This is the key step in obtaining the new understanding. Indeed, from (2) we can obtain a procedure to recover the weights. For each training sample $i$, compute the set of association cells that its input excites and add $\Delta_i$ to each of these association cells. At the end of this procedure, the correct weights have been recovered. Therefore, the convergence of the CMAC learning procedure is equivalent to the convergence of the $\Delta_i$'s. When the $\Delta_i$'s converge, the error goes to zero. The learning is then complete.

It is quite obvious that for each training sample $i$, if the initial weights are set to zero (see Section IV for the case of nonzero initial weights), the CMAC output $g_i$ is

$$g_i = \sum_k c_{ik} \Delta_k.$$  \hspace{1cm} (3)

The goal of the learning rule is to make $g_i = d_i$. If we change $g_i$ to $d_i$, the above expression is a linear system:

$$C \Delta = D,$$  \hspace{1cm} (4)

where the $ij$th element of $C$ is $c_{ij}$ and the $i$th element of $D$ is $d_i$.

A key observation now is that the CMAC updating rule is nothing more than a Gauss–Seidel iterative scheme [13] to solve the linear system, when the learning rule corrects the output error at each presentation. If, however, we instead update only after each epoch, that is, we save the changes in the weights for each training sample and update only after all the training samples have been presented, then the learning scheme is instead equivalent to Jacobi iteration [13]. We remark that the Gauss–Seidel iteration is easy to implement both in software and in hardware.

It is also clear in using the scheme proposed by Albus that, once the training set and the required resolution, i.e., the discretization level, are chosen, the matrix $C$ is determined and symmetric; all elements are nonnegative, too. We ask whether the Gauss–Seidel method, when applied to this matrix $C$, will converge to the desired solution. In other words, will CMAC learning converge?

If we can prove that $C$ is positive definite, then we are sure that Gauss–Seidel method will converge [13]. In the following arguments we will show that an inherent property of $C$ makes it a positive definite matrix. Hence convergence is guaranteed.

So, what is this $C$ matrix? As pointed out above, $c_{ij}$ is the number of association cells both sample $i$ and $j$ address. We represent the addresses each input sample excites by a vector of characteristic functions $\Theta_i(t)$ where

$$\Theta_i(t) = \begin{cases} 1 & \text{if } t\text{th address is excited by input sample } x_i \\ 0 & \text{otherwise}. \end{cases}$$

Let us call this vector the *indicator*.

It follows then that $c_{ij}$ is just the correlation between the indicators for the $i$th and $j$th training samples:

$$c_{ij} = \Theta_i \star \Theta_j = \sum_t \Theta_i(t) \Theta_j(t).$$  \hspace{1cm} (5)

Hence given any vector $y \neq 0$,

$$y^T C y = \sum_{ij} y_i c_{ij} y_j$$

$$= \sum_{ij} y_i \Theta_i(t) \Theta_j(t) y_j$$

$$= \sum_{ij} y_i \Theta_i(t) y_j \Theta_j(t)$$

$$= \sum_t \left( \sum_i y_i \Theta_i(t) \right)^2$$

$$\geq 0$$

which is just the “energy” of the indicators weighted by the vector $y$. Hence $C$ is positive semidefinite.

Lastly, it is easy to see that, by the construction inherent in CMAC, weighted indicators cannot vanish. For note that each indicator $\Theta_i(t)$ has a minimum $\min t_i$ for which $\Theta_i(t_{\min}) = 1$, with $t_{\min}$ the discretization level of $x_i$. Among all these training samples, there is only one sample which has a $t_{\min}$ which is the minimum among all the $t_{\min}$'s, namely the sample with the minimum discretization level. We have called this sample $j$. It follows that $\Theta_j(t)$ cannot be expressed as a linear combination of the other indicators, i.e., $y_j = 0$. Applying this argument recursively, we see that $y = 0$. This says that the sum of squares in (6) cannot vanish for any nonzero vector $y$. Hence $C$ is strictly positive definite. This completes the proof that CMAC always converges for one-dimensional inputs.

For multidimensional inputs, i.e., $n$ arbitrary, we have found that the proof for positive semidefiniteness still holds. Since we do not describe CMAC with a multidimensional input in this paper (the reader can obtain the details from Albus’ paper [1]), we simply point out that the mapping between the input space and the set of association cells $S$ induces an $n$-dimensional Euclidean structure on $S$. Thus we can index the association cells in lexicographical order. The indicator for each input is formed by concatenating the indicators for each coordinate. The proof for strict positive definiteness easily carries over. To get multidimensional outputs, we simply construct $m$ set of associations cells or memories. This is the plan for the proof for the mapping $\mathbb{R}^n \rightarrow \mathbb{R}^m$.

Now we can explain the fast convergence and proper generalization based on the correlation matrix $C$. Let $\Delta^*$ be the solution of $C \Delta = D$ and $\gamma(t) = \Delta(t) - \Delta^*$. Write $C$ as $C = L + R$, where $L$ is the lower diagonal part of $C$ and $R$ is the upper off-diagonal part of $C$. From matrix theory [13], Gauss–Seidel iteration gives

$$\gamma(t) = (-L^{-1}R)^t \gamma(0).$$  \hspace{1cm} (7)

If we further expand $\gamma(t)$ in terms of the eigenvectors $u_j$ of $(-L^{-1}R)$, say $\gamma(t) = \sum_j \rho_j u_j$, we obtain

$$\gamma(t) = \sum_{j=1}^N \rho_j^t \rho_j u_j$$  \hspace{1cm} (8)
where \( \lambda_j \) are eigenvalues of \((-L^{-1}R)\), with \( |\lambda_i| \leq |\lambda_j| \) for \( j > i \). From [13], the positive definiteness of matrix \( C \) guarantees that \( \lambda_j < 1 \) for all \( j \). Thus, initially the training error decreases very fast owing to the first few terms that are associated with the small eigenvalues. But the overall convergence is asymptotically proportional to \( \lambda_1 \), as \( l \) becomes large. Fig. 5 shows the mean square error versus iterations for the example in Section V. Indeed, a slower decrease in mean square error sets in after an initial steep decrease. Therefore, the theory is that we have convergence of the training error exponential in the number of iterations even though it can be very slow if the spectrum of \( L^{-1}R \) is close to 1.

How about generalization? In one dimension, the matrix elements decrease linearly from the diagonal element and they are nonnegative. Hence every row of \( C \) can be interpreted as an interpolation filter. Larger \( A^* \) gives more interpolation. But the price we pay is slightly more computational complexity. At worst, the marginal positive definiteness of the correlation matrix \( C \), when subject to hash coding, might lead to singularity, as explained below.

**Hash Coding**

Finally, we have to cope with hash coding since in higher dimensions, the number of association cells gets prohibitively large. If there is no collision, everything is fine. If collision does occur, it corresponds to perturbations to the matrix \( c_{ij} \).

Let us pause for a minute to think about the operations here. First, it is very unlikely that collision occurs when mapping the \( A^* \) associations cells any particular input excites to the smaller memory \( A_p \), as noted from (1). One possible strategy is to divide the \( A_p \) into \( A^* \) submemories and hash the \( k \)th component of the \( A^* \) addresses for all inputs into the \( k \)th submemory, where \( k = 1, \ldots, A^* \). So we can assume that this type of collision does not happen. What actually happens is that different association cells get mapped to the same address. Now, if \( \Theta(t) \) denotes the indicator for input \( x_i \) in the smaller memory \( A_p \), it is composed of \( A^* \) pulses of length 1 scattered around. Since the hash coding is certainly a deterministic algorithm, the same association cells are mapped to the same addresses. So the new correlation matrix \( C' \) is still symmetric, nonnegative, and positive semidefinite, with \( c'_{ij} \geq c_{ij} \).

To better illustrate the nature of this perturbation, we can consider the problem of putting \( s \) balls into \( n \) cells where \( r \) cells are already occupied. The probability of a ball hitting one of these \( r \) balls is \( p = 1/r \). Let \( q = 1 - p \). Then the probability for \( k \) collisions is \((\binom{n}{k})q^k(1-q)^{n-k-1}\). A simple numerical example will show that the probability of more than one collision is exceedingly small. For instance, if \( n = 10^6 \), \( r = 31 \), and \( s = 31 \), \( \text{Prob}(0 \text{ collision}) = 0.999904 \), \( \text{Prob}(1 \text{ collision}) = 0.000096 \), and \( \text{Prob}(2 \text{ collisions}) = 4 \times 10^{-7} \). For neighboring inputs, the number of balls \( s \) is small and the probability of having to add 1 to \( c_{ij} \) is extremely small; for inputs far apart, \( s = A^* \), this probability is higher but still very small.

Therefore, we see that the perturbation can be regarded as a random but symmetric matrix with integer elements \( m_{ij} \), where the \( m_{ij} \)'s are random variables with distribution \( P(m_{ij} = k) = \binom{n}{k}q^k(1-q)^{n-k} \). Here \( s \) is a number between 0 and \( A^* \), depending on how close the input samples \( x_i \) and \( x_j \) are to each other. Larger \( A^* \) means more perturbation to the original matrix. Consequently, we see that if we assume that the hashing function is random enough, in the hashing part only two parameters matter: \( A_p \) and \( A^* \). We can always choose \( A_p \) and \( A^* \) such that the probability in (1) approaches 1. In that case, most \( m_{ij} \)'s are 0; rarely will \( m_{ij} \) be greater than 1. To a first-order approximation, that is, if we assume that no collision occurs when mapping the \( A^* \) cells any particular input excites to \( A_p \), then the perturbation is of the nature we described above.

What really causes trouble is that the new \( \Theta(t) \)'s can become linearly dependent, in which case it is possible to choose \( y \) such that (6) vanishes. The matrix \( C' \) is no longer positive definite, which means that the Gauss–Seidel iterative scheme could not converge. The linear dependence of the indicators \( \Theta(t) \) is an example of generalized "ghosts," as noted by the authors in [14], which can create problems when each memory is encoded by a subpopulation of neurons. The analysis of the random matrix, we believe, will be closely related to the capacity of CMAC with hash coding. We have not actually conducted an analysis to address this question. But based on the simulation results with different \( A_p \), CMAC is very resistant to collisions.

**IV. ROBUSTNESS AGAINST NOISE IN LEARNING**

Lastly, we address the question of robustness of CMAC to noise added to the weights in the learning process. To better understand it, we have to resort to rigorous mathematical analysis.

Let us form the matrix \( A = [\Theta_1(t)\Theta_2(t)\cdots\Theta_N(t)]^T \), where \( N \) is the number of training samples, \( \Theta_i(t) \) is the indicator for input sample \( x_i \) as defined in Section III, and the superscript \( ^T \) denotes the transpose operation. We see that the matrix \( C \) defined in Section III can be written as \( C = AA^T \). Let \( W \) be a vector denoting the output of the association cells, or the weights in the smaller memory \( A_p \). Thus, the goal of CMAC learning is to find a set of weights such that

\[
AW = D
\]

where \( D \) is the vector of the desired outputs.

The linear system \( C \Delta = D \) we derived in Section III is just \( AA^T \Delta = D \). After \( \Delta \) has converged, the procedure to restore the weights \( W \) corresponds to the matrix operation \( A^T \Delta \). Thus, putting all the equations together, the weights are given by \( W = A^T(AA^T)^{-1}D \). Note that \( (AA^T)^{-1} \) is the pseudoinverse \( A^+ \) of \( A \) [15]. We should also note that the linear system in (9) is underspecified. That is, it has more unknowns (weights) than equations (one equation for each training sample). Therefore, there are an infinite number of solutions. Any vector of weights which lies in the null space \( \ker(A) \) of \( A \) will produce zero output. Therefore, we expect that a different set of initial weights may lead to a different set of weights \( W \).

With this observation, we write the initial weights as \( W_0 = W_0^0 + W_0^1 \), where \( W_0^0 \in \ker(A) \) and \( W_0^1 \in \ker(A)^\perp \) is the subspace orthogonal to \( \ker(A) \) in the input space. Such
decomposition is uniquely defined [15]. Let $G_0 = AW_0$ denote
the vector of initial outputs $g_0$ for the input samples. Then after
the kth presentation and weight correction, the CMAC output
for the rth input sample is $c_{ik} \Delta_k + g_0$. Therefore, we must
replace each element $d_i$ of $D$ by $d_i - g_0$ to arrive at the sys-
tem $C \Delta = D$, where $D = D - G_0$. After the learning con-
verges, the weights are given by

$$W = A^+ (D - G_0) + W_0$$
$$= A^+ D + (W_0 - A^+ AW_0)$$
$$= A^+ D + W_0 - W_0^1$$
$$= A^+ D + W_0^0.$$

We see that the final weights are $W = A^+ D + W_0^0$. The norm
of the weights is $\|W\| = \|A^+ D\|^2 + \|W_0\|^2$. Therefore,
to get a minimum-normed-weight vector, we should initialize
the weights to zero.

Based on the above results, we see that CMAC is very
robust to noise in the weights. This robustness against noise is
a tremendous advantage for hardware implementation, where
noise will always be present. Suppose in evaluating the
error signal $\delta_i = d_i - g_i$, a noise term $\eta_i$ is added. Then
after a certain number of iterations, the error correction signal
$\delta_i$ is basically due only to the noise term, which is then
added to the weights. Such perturbation can be broken into
two components. The component which lies in $\ker(A)^\perp$ will
change $D$ a little, while the one which lies in the null space of $A$ has no effect on the CMAC outputs for inputs that belong
to the training set. If the noise is random and not biased,
then convergence will be achieved, but only in the sense that
the weights will asymptotically fluctuate around their no-noise
values; the fluctuations will not go to zero.

One method to force the weights to converge is to let the learning rate $\eta$ decrease with the iteration number after a certain number of iterations that had $\eta = 1/A^*$. Suppse we demand that $\lim_{m \to \infty} \sum_{k=1}^m \eta_k = +\infty$ and
$\lim_{m \to \infty} \sum_{k=1}^m \eta_k^2 < \infty$. Then the weights converge to the correct values in mean square [16]. That is, if we
write the weight vector after $k$ iterations as $W_k = W_0 + W_k^1$, where $W_0 \in \ker(A)$ and $W_k^1 \in \ker(A)^\perp$,
$\lim_{k \to \infty} E \left[ \|W_k - A^+ D\|^2 \right] = 0$.

In the above analysis, $W_k^0$, the component which lies in $\ker(A)$, will increase the norm of the weights. There is thus
no guarantee that $W_k^1$ will not affect (and to what extent it may affect) the CMAC outputs when presented with inputs which
do not lie in the training set but are still close to some of the
training samples. A detailed analysis of this question will give
a better understanding of the nature of generalization in CMAC. It would also be interesting to know if small weights in CMAC might lead to a better generalization property. In back-propagation, for example, it is argued that small weights
lead to better generalization [17].

V. A NEW IMPLEMENTATION OF CMAC

We can recapitulate the results obtained in previous section
as follows:

1) CMAC learning is essentially solving a linear system
with known matrix algorithm which converges; this
explains why
- CMAC learning is highly accurate;
- CMAC converges exponentially fast.

2) CMAC is doing some kind of interpolation for inputs it
has not been trained on; this gives it a certain generalization
ability.

3) CMAC learning is very robust to the noise added in the
learning process.

Having said that, we are finally ready to propose a new
implementation for CMAC based on the insights obtained in
the proof:

1) given a set $S$ of training samples;
2) discretize the input space with fine enough resolution so
that no two different inputs occupy the same discretization
level;
3) pick an appropriate memory size $A_p$ and $A^*$;
4) for each sample $i$, determine the amount of overlap in
the addresses with sample $k$;
5) repeat step 3 for all training samples. This establishes
the matrix $C = c_{ik}$;
6) use Gauss–Seidel or better matrix algorithm to solve the
linear system $C \Delta = D$, where $\Delta$ is the accumulated
weight error;
7) to recover the weights, use the procedure outlined
in Section III, which is summarized as the operation
$A^* \Delta + W_0$.

We remark that it is relatively easy to compute the $c_{ij}$.
The number of operations is at most $O(n)$, where $n$ is the
number of training samples. The matrix $C$ is of dimension
$n$ and is very sparse. It is not hard to see how to compute
the output for any given input. This is just given by (3) in
Section III. We should also point out that the original
implementation proposed by Albus has advantages of simplicity
and real-time processing while our proposed scheme is more
suitable for off-line learning.

VI. A SIMULATION OF THE NEW SCHEME

To illustrate the proof derived above, we carried out a
computer simulation of the new scheme and compared the
result with the original scheme. The task we set out to learn
is to determine the inverse dynamics of a two-link robot arm
along a given trajectory in the joint space. The trajectory was
discretized into 300 points, which is shown in Fig. 3. The
torque required to control the arm so that it moves along the
trajectory is computed by the dynamics equation found in [18]
and is shown in Fig. 4. This is how the training is obtained.
Next we train the CMAC with the original implementation
and the new implementation using the same training set and
system parameters. The only difference is in the choice of
$A_p$. For the original implementation, the hashing memory is
chosen to be 50021, which is large enough to allow only slight
collisions. In the new implementation, we assume an infinite
memory; i.e., we calculate the $c_{ij}$ in the space of association
cells before hashing.
propose a general learning scheme using population coding, a more general form of coding the input–output relationship via the aggregated activity of a subpopulation of neurons.

**ACKNOWLEDGMENT**

The authors thank J. Hopfield for suggesting a simulation with the new scheme. They would also like to thank K. Orita for providing the programs for the robotic simulations. Thanks are also due to E. C. Posner for his numerous insightful comments on the manuscript.

**REFERENCES**


