Chapter 3

The CMAC neural network

3.1 Introduction

This chapter describes the operation of the CMAC neural network. It is largely a tutorial, although the CMAC's performance will be analyzed in detail and some new results will be presented. CMAC is an acronym for Cerebellar Model Articulation Controller[1]. The CMAC was first described by Albus in 1975 [3, 2] as a simple model of the cortex of the cerebellum (see Chapter 2). Since then it has been in and out of fashion, extended in many different ways, and used in a wide range of different applications. Despite its biological relevance, the main reason for using the CMAC is that it operates very fast, which makes it suitable for real-time adaptive control.

The operation of the CMAC will be described in two ways: as a neural network and as a lookup table. Then the properties of the CMAC will be explored. Comparisons will be drawn between the CMAC and the multi-layer perceptron (MLP) neural network, which is described in Appendix C. Other information about CMAC operation can be found in [85] and [123].

3.2 The CMAC

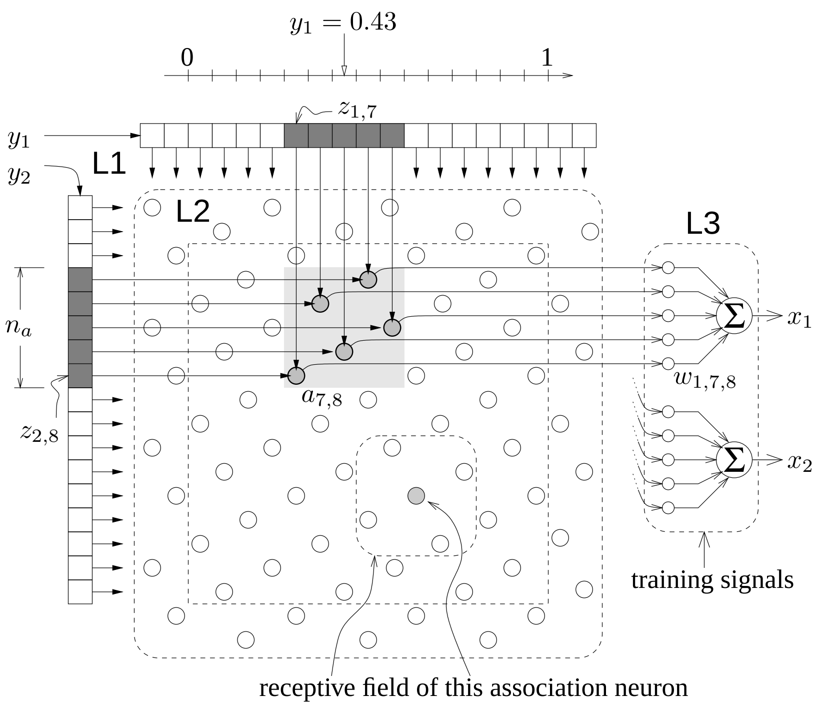
3.2.1 The CMAC as a neural network

The basic operation of a two-input two-output CMAC network is illustrated in figure 3.1a. It has three layers, labeled L1, L2, L3 in the figure. The inputs are the values y_1 and y_2. Layer 1 contains an array of "feature detecting" neurons z_{ij} for each input y_i. Each of these outputs one for inputs in a limited range, otherwise they output zero (figure 3.1b). For any input y_i a fixed number of neurons (n_a) in each layer 1 array will be activated ($n_a = 5$ in the example). The layer 1 neurons effectively quantize the inputs.

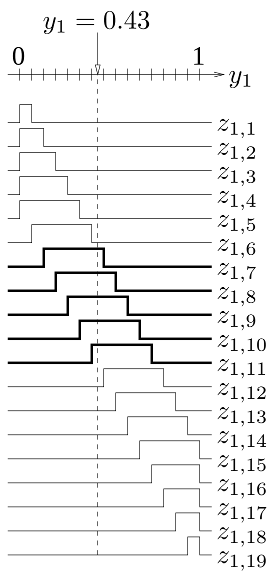
Layer 2 contains n_v association neurons a_{ij} which are connected to one neuron from each layer 1 input array (z_{1i}, z_{2j}). Each layer 2 neuron outputs 1.0 when all its inputs are nonzero, otherwise it outputs zero—these neurons compute the logical AND of their inputs. They are arranged so exactly n_a are activated by any input (5 in the example).

Layer 3 contains the n_x output neurons, each of which computes a weighted sum of all layer 2

[1]CMAC is pronounced "see-mac".



(a)



(b)

Figure 3.1: ***(a)*** *An example two-input two-output CMAC, in neural network form* ($n_y = 2$, $n_a = 5$, $n_v = 72$, $n_x = 2$). ***(b)*** *Responses of the feature detecting neurons for input 1.*

outputs, i.e.:

$$x_i = \sum_{jk} w_{ijk}\, a_{jk} \tag{3.1}$$

The parameters w_{ijk} are the weights which parameterize the CMAC mapping (w_{ijk} connects a_{jk} to output i). There are n_x weights for every layer 2 association neuron, which makes $n_v n_x$ weights in total.

Only a fraction of all the possible association neurons are used. They are distributed in a pattern which conserves weight parameters without degrading the local generalization properties too much (this will be explained later). Each layer 2 neuron has a receptive field that is $n_a \times n_a$ units in size, i.e. this is the size of the input space region that activates the neuron.

The CMAC was intended by Albus to be a simple model of the cerebellum. Its three layers correspond to sensory feature-detecting neurons, granule cells and Purkinje cells respectively, the last two cell types being dominant in the cortex of the cerebellum. This similarity is rather superficial (as many biological properties of the cerebellum are not modeled) therefore it is not the main reason for using the CMAC in a biologically based control approach. The CMAC's implementation speed is a far more useful property.

It is apparent that, conceptually at least, the CMAC deserves to be called a neural network. However, actual software implementations are much more convenient if the CMAC is viewed as a lookup table.

3.2.2 The CMAC as a lookup table

Figure 3.2 shows an alternative table based CMAC which is exactly equivalent to the above neural network version. The inputs y_1 and y_2 are first scaled and quantized to integers q_1 and q_2. In this example:

$$q_1 = \lfloor 15\, y_1 \rfloor \tag{3.2}$$
$$q_2 = \lfloor 15\, y_2 \rfloor \tag{3.3}$$

as there are 15 input quantization steps between 0 and 1 (note $\lfloor\cdot\rfloor$ is the floor function). The indexes (q_1, q_2) are used to look up weights in n_a two-dimensional lookup tables (recall that $n_a = 5$ is the number of association neurons activated for any input). Each lookup table is called an "association unit" (AU)—they are labeled AU1...AU5 in figure 3.2. The AU tables store one weight value in each cell. Each AU has cells which are n_a times larger than the input quantization cell size, and are also displaced along each axis by some constant. The displacement for AU i along input y_j is d_j^i (the example values for d_j^i are given in figure 3.2). If p_j^i is the table index for AU i along axis y_j then, in the example:

$$p_j^i = \left\lfloor \frac{q_j + d_j^i}{n_a} \right\rfloor = \left\lfloor \frac{q_j + d_j^i}{5} \right\rfloor \tag{3.4}$$

The CMAC mapping algorithm (based on the table form) is shown in table 3.1. Software CMAC implementations never use the neural network form. The CMACHASH function used by the CMACQUANTIZEANDASSOCIATE procedure takes the AU number and the table indexes and generates an index into a weight array. Implementations of CMACHASH, and the way the weights are arranged in memory, will be described later.

The CMAC is faster than an "equivalent" MLP network because only n_a neurons (i.e. n_a table entries) need to be considered to compute each output, whereas the MLP must perform calculations for all of its neurons.

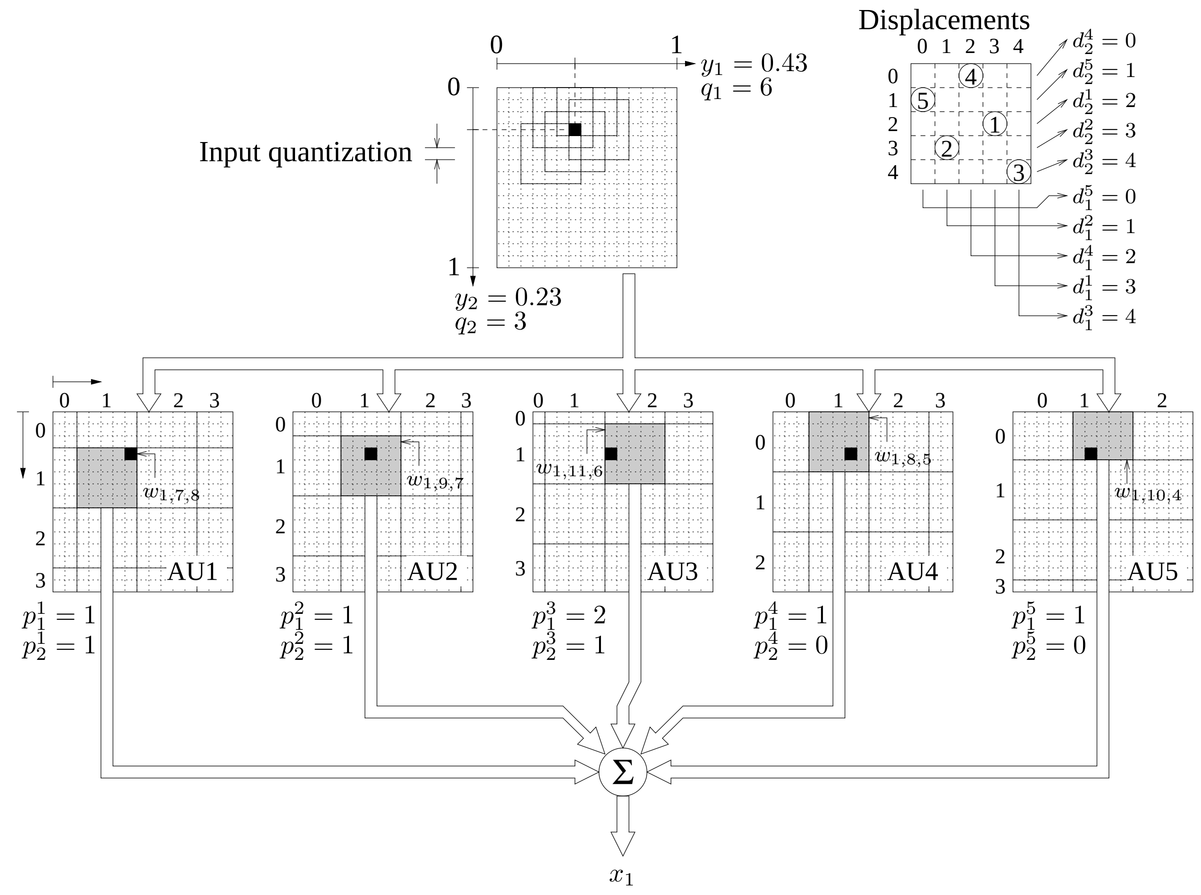


Figure 3.2: *A two input CMAC (table form) equivalent to figure 3.1, $n_a = 5$.*

THE CMAC ALGORITHM

Parameters

n_y — Number of inputs (integer ≥ 1)
n_x — Number of outputs (integer ≥ 1)
n_a — Number of association units (integer ≥ 1)
d^i_j — The displacement for AU i along input y_j (integer, $0 \leq d^i_j < n_a$)
min_i and max_i — Input ranges; the minimum and maximum values for y_i (real).
res_i — Input resolution; the number of quantization steps for input y_i (integer > 1).
n_w — The total number of physical CMAC weights per output.

Internal state variables

μ_i — An index into the weight tables for association unit i ($i = 1 \ldots n_a$).
$W_j[i]$ — Weight i in the weight table for output x_j, $i = 0 \ldots (n_w - 1)$, $j = 1 \ldots n_x$.

ALGORITHM: CMACMAP — Maps the CMAC input to its output.
Inputs: $y_1 \ldots y_{n_y}$ (scalars)
Outputs: $x_1 \ldots x_{n_x}$ (scalars)

```
⇒    CMACQUANTIZEANDASSOCIATE ($y_1 \ldots y_{n_y}$)      —this sets $\mu_1 \ldots \mu_{n_a}$
     for $i \leftarrow 1 \ldots n_x$                               —for all outputs
         $x_i \leftarrow 0$
         for $j \leftarrow 1 \ldots n_a$ : $x_i \leftarrow x_i + W_i[\mu_j]$        —for all AUs add table entry to $x_i$
```

ALGORITHM: CMACQUANTIZEANDASSOCIATE — Get weight table indexes.
Inputs: $y_1 \ldots y_{n_y}$ (scalars)
Outputs: $\mu_1 \ldots \mu_{n_a}$ (scalars)

```
⇒    for $i \leftarrow 1 \ldots n_y$                                —for all inputs
         if $y_i < \text{min}_i$ then : $y_i \leftarrow \text{min}_i$                 —limit $y_i$
         if $y_i > \text{max}_i$ then : $y_i \leftarrow \text{max}_i$                 —limit $y_i$
         $q_i \leftarrow \left\lfloor \text{res}_i \frac{y_i - \text{min}_i}{\text{max}_i - \text{min}_i} \right\rfloor$                 —quantize input
         if $q_i \geq \text{res}_i$ then : $q_i \leftarrow \text{res}_i - 1$             —enforce $0 \leq q_i < \text{res}_i$
     for $i \leftarrow 1 \ldots n_a$                                —for all AUs
         for $j \leftarrow 1 \ldots n_y$ : $p^i_j \leftarrow \left\lfloor \frac{q_j + d^i_j}{n_a} \right\rfloor$          —find tables indexes for all inputs
         $\mu_i \leftarrow$ CMACHASH $(i, p^i_1, p^i_2, \ldots, p^i_{n_y})$          —Get weight index for AU $i$
```

ALGORITHM: CMACTARGETTRAIN — Train the CMAC output to reach a given target.
(It is assumed that CMACMAP has already been called.)
Inputs: $t_1 \ldots t_{n_x}$, α (scalars)

```
⇒    for $i \leftarrow 1 \ldots n_x$
         increment $\leftarrow \alpha \frac{t_i - x_i}{n_a}$
         for $j \leftarrow 1 \ldots n_a$ : $W_i[\mu_j] \leftarrow W_i[\mu_j]$ + increment
```

Table 3.1: *The algorithms for computing the CMAC output and training the CMAC.*

3.3 Training

The training process takes a set of desired input to output mappings (training points) and adjusts the weights so that the global mapping fits the set. It will be useful to distinguish between two different CMAC training modes:

- **Target training**: First an input is presented to the CMAC and the output is computed. Then each of the n_a referenced weights (one per lookup table) is increased so that the outputs x_i come closer to the desired target values t_i, i.e.:

$$w_{ijk} \leftarrow w_{ijk} + \frac{\alpha\, a_{jk}}{n_a}(t_i - x_i) \tag{3.5}$$

  where α is the learning rate constant $(0 \leq \alpha \leq 1)$. If $\alpha = 0$ then the outputs will not change. If $\alpha = 1$ then the outputs will be set equal to the targets. This is implemented in the CMACTARGETTRAIN procedure shown in table 3.1.

- **Error training**: First an input is presented to the CMAC and the output is computed. Then each of the n_a referenced weights is incremented so that the output vector $[x_1 \ldots x_{n_x}]$ increases in the direction of the error vector $[e_1 \ldots e_{n_x}]$, i.e.:

$$w_{ijk} \leftarrow w_{ijk} + \frac{\alpha\, a_{jk}}{n_a} e_i \tag{3.6}$$

  This results in a trivial change to the CMACTARGETTRAIN procedure given in table 3.1.

Target training causes the CMAC output to seek a given *target*, error training causes it to grow in a given *direction*. The input/desired-output pairs (training points) are normally presented to the CMAC in one of two ways:

- **Random order**: If the CMAC is to be trained to represent some function that is known beforehand, then a selection of training points from the function can be presented in random order to minimize learning interference [122].

- **Trajectory order**: If the CMAC is being used in an online controller then the training point inputs will probably vary gradually, as they are tied to the system sensors. In this case each training point will be close to the previous one in the input space.

3.4 Hashing

3.4.1 The number of CMAC weights

How are the weights in the weight tables stored in the computer's memory? The naïve approach, storing each weight as an separate number in a large floating point array, is usually not possible. To see why, consider the number of weights required for the CMAC parameters given in table 3.1. The maximum value of the table index p_j^i is (if the largest displacements are assumed):

$$\text{maximum } p_j^i = \left\lfloor \frac{(\text{maximum } q_j) + (\text{maximum } d_j^i)}{n_a} \right\rfloor \tag{3.7}$$

$$= \left\lfloor \frac{(\text{res}_j - 1) + (n_a - 1)}{n_a} \right\rfloor \tag{3.8}$$

$$= \left\lfloor \frac{\text{res}_j - 2}{n_a} \right\rfloor + 1 \tag{3.9}$$

The total number of CMAC weights is the number of AU tables times the number of weights stored per AU:

$$\text{weights in CMAC} = n_a \times \prod_{j=1}^{n_y} \left\{ \text{maximum } p_j^i + 1 \right\} \tag{3.10}$$

$$= n_a \times \prod_{j=1}^{n_y} \left\{ \left\lfloor \frac{\text{res}_j - 2}{n_a} \right\rfloor + 2 \right\} \tag{3.11}$$

To simplify this, assume that res_j is sufficiently large and the same for all j, so p_j^i can be approximated by

$$\text{maximum } p_j^i \approx \frac{\text{res}}{n_a} \tag{3.12}$$

which gives

$$\text{weights in CMAC} \approx n_a \left(\frac{\text{res}}{n_a} \right)^{n_y} \tag{3.13}$$

$$= \frac{\text{res}^{(n_y)}}{n_a{}^{(n_y - 1)}} \tag{3.14}$$

There are two things to note about equation 3.14. First, increasing n_a *reduces* the number of weights, even though it increases the number of weight tables. This fact will be used later to try to get a smoother output from the CMAC.

The second thing to note is that, for typical values of res, n_a and n_y the number of weights is huge. For example, suppose res $= 200$, $n_a = 10$ and $n_y = 6$ (these values are not atypical) then the number of weights is 640 million. At four bytes per floating point number this would require 2441 megabytes of memory, which is presently unreasonable. The problem is that the number of weights is exponential in the number of input dimensions n_y. One way to reduce the number of weights is to use hashing.

3.4.2 What is hashing?

Hashing is a commonly used technique in computer science [109]. The idea is to take an address into a large "virtual" memory and map it into an address into a smaller physical memory. For example, suppose a program is required to quickly store and recall 100 seven digit telephone numbers[2]. In would be wasteful and inefficient to use the telephone number as an index into an array of 10^7 entries. Instead an array of only (say) 251 entries[3] could be used, and the index into this array could be computed with

$$\text{index} = (\text{phone number}) \bmod 251 \tag{3.15}$$

The modulo function hashes the large phone number address into a smaller index in the range $0 \ldots 250$. This makes more efficient use of memory, but now there is the problem that some of the phone numbers will hash to the same physical memory location (a problem that gets worse as more numbers that have to be stored). This is known as "hash collision". A realistic database program would have to resolve hash collision using collision lists or secondary probing techniques [109].

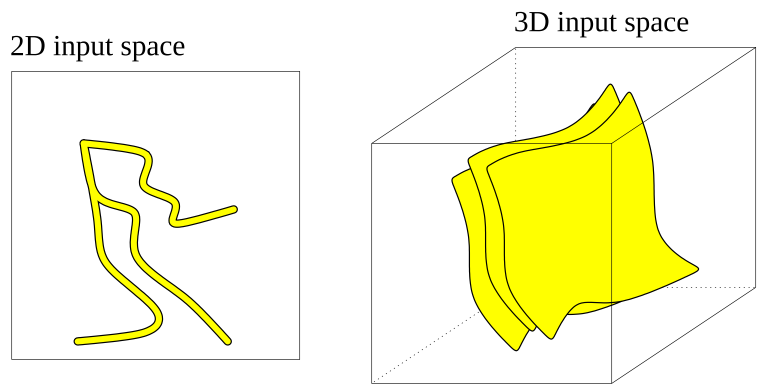


Figure 3.3: *An illustration of how CMAC inputs derived from control processes can tend to fall along low dimensional surfaces (one and two dimensional examples).*

3.4.3 How the CMAC uses hashing

The CMAC uses a hashing scheme to map the large "virtual" weight table address into a smaller physical one. Hash collisions are ignored—if any two weights with the same physical address are written, the second one will simply replace the first. Consider using a 100 kilobyte physical memory for the example above which has a 2441 megabyte virtual memory requirement. In this case almost 25000 virtual weights would map to a single physical weight, which would seem to hopelessly compromise any training that was performed.

In practice this is not a problem if the inputs to a CMAC only venture over a fraction of the entire input space volume, because then only a subset of the virtual weights will be referenced. This is usually true for inputs that come from physical sensors attached to some controlled process. Sensor values may be correlated with one another, and the process may only be observed in some fixed modes of operation (or in limit cycles), which means that only a small subset of all possible input trajectories are seen. This usually means that the CMAC inputs only lie on a few low dimensional surfaces in the input space (figure 3.3).

The hash collision problem is intolerable if the inputs roam over the *entire* input space. But if the CMAC input parameters are correctly chosen then it will be tolerable over the set of *likely* inputs. Thus the benefit of hashing is that it allows the relatively small number of physical weights to be allocated *where they are needed* in the input space.

Hash collisions are manifested as noise in the CMAC output. As a larger volume of the input space is used, more hash collisions will be experienced, the CMAC output will get noisier and it will be harder to get the CMAC to retain the training data.

3.4.4 CMAC hashing algorithms

A CMAC hashing algorithm (CMACHASH) generates a physical weight table index i from the AU number j and the tables indexes $p^j_1, p^j_2, \ldots, p^j_{n_y}$ (from table 3.1), i.e.

$$i = f_h\Big(f_v(p^j_1, p^j_2, \ldots, p^j_{n_y}), j\Big) \qquad \text{—variant 1} \qquad (3.16)$$

[2]Any programmer can think of many ways to tackle this problem (e.g. linked lists and binary trees) but only hashing is interesting here.

[3]251 is a prime number. Prime numbers have been shown to be particularly good with the modulo hash function [109].

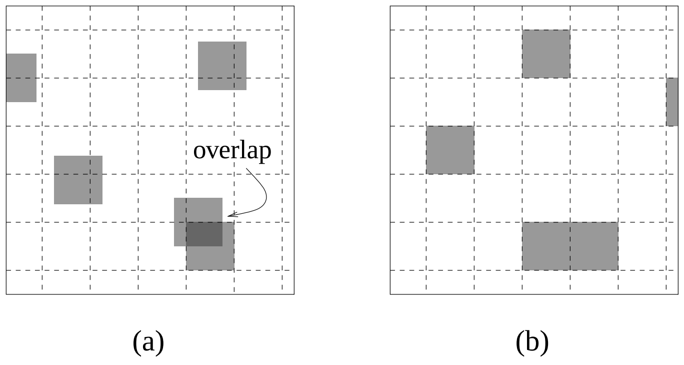


Figure 3.4: *When hashing is used, a single physical weight influences multiple virtual weights, resulting in disjointed "basis functions" — (a) shows an example of variant 1 hashing (AU number hashed) and (b) shows an example of variant 2 (AU number not hashed).*

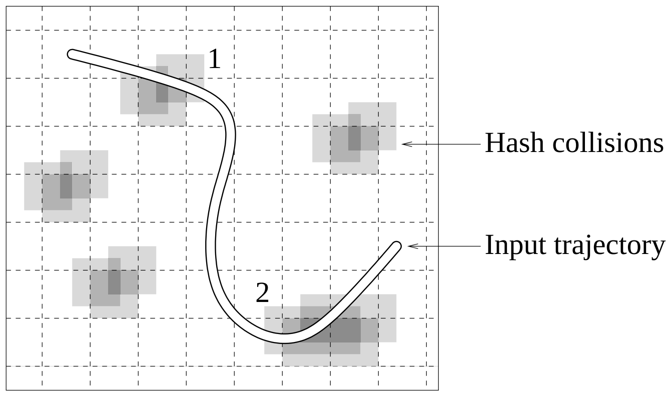


Figure 3.5: *This is what happens when variant-2 hashing does not hash on the AU index (the trajectory intersects two hash-collision clusters).*

$$\text{or} \quad i = f_h\Big(f_v(p_1^j, p_2^j, \ldots, p_{n_y}^j), j\Big) + n_p\, j \qquad \text{—variant 2} \tag{3.17}$$

where f_v is a function that generates a unique virtual address $(0 \ldots n_v - 1)$ from the weight table indexes, and f_h is a hashing function that produces a physical address $(0 \ldots n_p - 1)$. A single physical weight will control multiple virtual weights, as shown in figure 3.4—this set is called the basis function for that weight[4]. Figure 3.4a shows how variant 1 hashes on the AU index, so a basis function can have weights in multiple AUs. Thus the basis function can have overlapping AU cells. Figure 3.4b shows how variant 2 uses the AU index j to index a separate n_p-sized area for each AU, ensuring that each basis function contains only cells from a single AU.

Variant 2 must also hash on the AU index j, because otherwise the basis functions for corresponding weights in each AU will have the same pattern. Consider figure 3.5, which shows what can happen when variant 2 hashing does not include the AU index. Basis functions localized in more than one useful area of the input space are clustered together, so training performed at point 1 will cause points 1 and 2 to change by the same amount. If the AU index is included in the hash function then the hash collisions will

[4]This terminology is not strictly correct, mathematically speaking.

be widely distributed, so the training effect at point 2 will have only $1/n_a$ of the full effect. Either variant will work, but it will be seen later that variant 2 makes some aspects of the FOX algorithm simpler.

There are numerous choices for f_v and f_h. The major requirements are that f_v should produce a different index for every virtual weight, and f_h should not result in a physical weight having some systematic pattern of virtual weights that will lead to greater than expected hash collisions (i.e. the more "random" the hash function is the better).

Two examples of f_v are shown in table 3.2, one suited for software implementation and the other for hardware. Two examples of f_h are shown in figure 3.6.

3.5 Properties of the CMAC

3.5.1 Limited input space

Each CMAC input has a minimum and maximum value, beyond which the weight tables do not hold any weight values. Thus the designer needs to know beforehand what the probable input ranges are. The mapping algorithm in table 3.1 clamps the inputs to be between the minimum and maximum values. The CMAC will not perform as expected if the input goes outside this valid range, but valid weight table indexes will still be generated.

3.5.2 Piecewise constant

The CMAC input to output mapping is piecewise constant because of the quantized input. In other words the mapping contains many discontinuous steps. This may seem likely to result in a poor mapping, but in fact it is not a great disadvantage in many applications. If the input resolutions are chosen to be large enough then the discontinuities will be small. There is a trade off, however, because more input resolution results in a larger virtual weight space.

3.5.3 Local generalization

The concept of generalization is explained in Appendix A. In contrast to the multi-layer perceptron (MLP), the CMAC has *local* generalization (LG). This means that when each data point is presented to the CMAC training algorithm, only a small region of the mapping around that point is adjusted. This occurs, of course, because only the weights which affect the output are adjusted, and each of those weights can only influence a small area of the mapping. Hashing increases the area that each physical weight influences, but this does not count as generalization as it is unpredictable. Figure 3.7 shows a possible result of one CMAC training iteration for $\alpha = 1$.

Local generalization can also be regarded as interpolation. The interpolation capabilities of the binary CMAC are explored further in [22].

3.5.4 Training sparsity

To ensure good generalization, how far apart (in the input space) can the training points be in trajectory-ordered data? Figure 3.8 shows the CMAC mapping that results from training with sparse data, ranging from 6 to 21 training points spaced evenly between zero and one. In each case the training was performed on each training point in turn with a learning rate $\alpha = 0.1$, repeating this until convergence was achieved. This CMAC had 100 quantization steps in the [0,1] interval and $n_a = 10$, so the LG area had 20% of the [0,1] interval width.

ALGORITHMS TO COMPUTE VIRTUAL ADDRESS (f_v)

Purpose

Compute a virtual address for weight table indexes $p^j_1, p^j_2, \ldots, p^j_{n_y}$.

ALGORITHM 1 (software)

Definitions

$r_k = \left\lfloor \frac{\text{res}_k - 2}{n_a} \right\rfloor + 2$ — the number of different values of p^j_k.

Inputs: $p^j_1, p^j_2, \ldots, p^j_{n_y}$.

Output: h—A virtual address in the range $0 \ldots \left(\prod_{k=1}^{n_y} r_k \right) - 1$

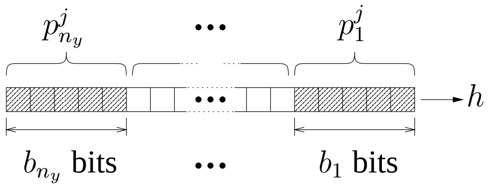
```
⇒   $h \leftarrow 0$
    for $i \leftarrow 1 \ldots n_y$
        $h \leftarrow h\, r_i + p^j_i$
```

ALGORITHM 2 (hardware)

Inputs: $p^j_1, p^j_2, \ldots, p^j_{n_y}$.

Output: h—A virtual address.

⇒ Insert the indexes $p^j_1 \ldots p^j_{n_y}$ directly into a hardware register:



$b_1 \ldots b_{n_y}$ are such that $2^{b_i} \geq r_i$.

Table 3.2: *Two algorithms for computing a virtual weight address from the weight table indexes.*

Here is a highly effective hash function in C, adapted from [100]. It does a pseudo-DES hashing of the two 32 bit arguments to a 32 bit result. The argument `virtual` *is the virtual address returned by f_v (it must have a maximum value less than 2^{32}),* `au_num` *is the association unit number, and* `pbits` *is the number of bits to allow in the physical address ($0 \ldots 31$). This does a much better job of nonlinear pseudo-random bit mixing than some other hash functions (like mod-prime or linear congruential) but at the expense of speed (it's still pretty quick, though).*

```
unsigned long pdes_hash (unsigned long virtual, unsigned long au_num,
                         int pbits)
{
  unsigned long i,ia,ib,iswap,itmph=0,itmpl=0;
  static unsigned long c1 [4] = {
    0xbaa96887L,0x1e17d32cL,0x03bcdc3cL,0x0f33d1b2L};
  static unsigned long c2 [4] = {
    0x4b0f3b58L,0xe874f0c3L,0x6955c5a6L,0x55a7ca46L};
  /* Perform 4 iterations of DES logic, using a simpler       */
  /* (non-cryptographic) nonlinear function instead of DES's. */
  for (i=0; i<4; i++) {
    ia = (iswap=au_num)^c1[i];
    itmpl = ia & 0xffff;
    itmph = ia >> 16;
    ib = itmpl*itmpl + ~(itmph*itmph);
    au_num = virtual^(((ia=(ib>>16)|((ib&0xffff)<<16))^c2[i])+
             itmpl*itmph);
    virtual = iswap;
  }
  /* Chop off the lower 'pbits' bits of the hashed value */
  return virtual & ((1 << pbits)-1);
}
```

This hash function is particularly useful for hardware CMAC implementations. The weight table indexes and the AU number j are loaded into a linear feedback shift register [100] which is then shifted sufficiently to scramble the bits. The physical address is taken as some subset of the shift register bits. Enough shifts must be performed so that each original bit affects a number of bits in the physical address. See [54] for further information.

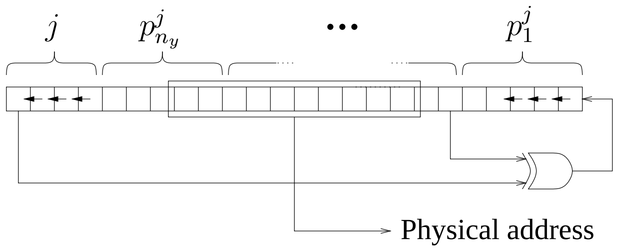


Figure 3.6: *Some hashing algorithms.*

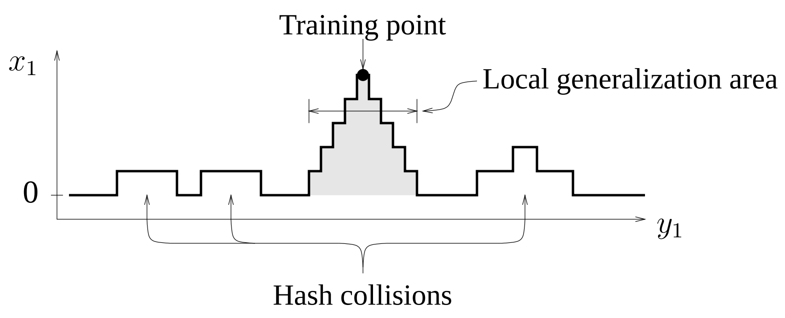


Figure 3.7: *An example CMAC output after one training iteration starting from zero ($n_a = 5$).*

For points where the LG regions don't overlap (the graphs of 5 and 6 points) the mapping is a series of triangular bumps. As the points get closer together (7,8,9,10 points) the regions between them become better interpolated, until at 11 points the LG regions overlap exactly and the inter-point spaces are linearly interpolated—this is a reasonably good mode of generalization. As the training points get closer still (12,13,. . .) the inter-point spaces are no longer linearly interpolated, but the error there remains relatively small. Linear interpolation is achieved again at 21 points. Obviously, better generalization is achieved if the training points are closer together.

The bottom three graphs in figure 3.8 show the same CMAC after training with 10, 100 and 1000 randomly positioned points in the interval [0,1] ($\alpha = 0.5$). 10 points is too few to get a good approximation with $\alpha < 1$. The approximation becomes smoother with more training points because the points tend to be closer together. Note the region on the left of the 100 point graph where the lack of training points causes a small interval of large error.

In conclusion, for a small number of fixed training points the generalization performance is inferior to the MLP, as the inter-point regions are hardly ever perfectly interpolated. As the inter-point space decreases below the size of the LG region, the generalization becomes better. For a large number of "randomly" spaced training points (perhaps generated by some function or process) the CMAC is a good functional approximator.

3.5.5 Training interference

Consider training a CMAC at two points, first A then B, with some learning rate α. If point B is within the LG area of point A it can make the mapping error at point A worse. This effect is called training interference.

Figure 3.9 shows the cause of the problem. If points A and B have overlapping LG areas then they share some weights (v weights in the figure). Training at B will affect the value of x_A because v of its n_a weights will be altered. If the "training interference" is defined to be the degree of influence over x_A that training at B has then:

$$\text{training interference (\%)} = \frac{v}{n_a} \times 100\% \tag{3.18}$$

If $v = 0$ then there is obviously no interference (because there is no overlap) and if $v = n_a$ then A and B are the same point and the interference is maximized. This can be particularly problematic during trajectory training, because then successive training points are very likely to be close to each other. The

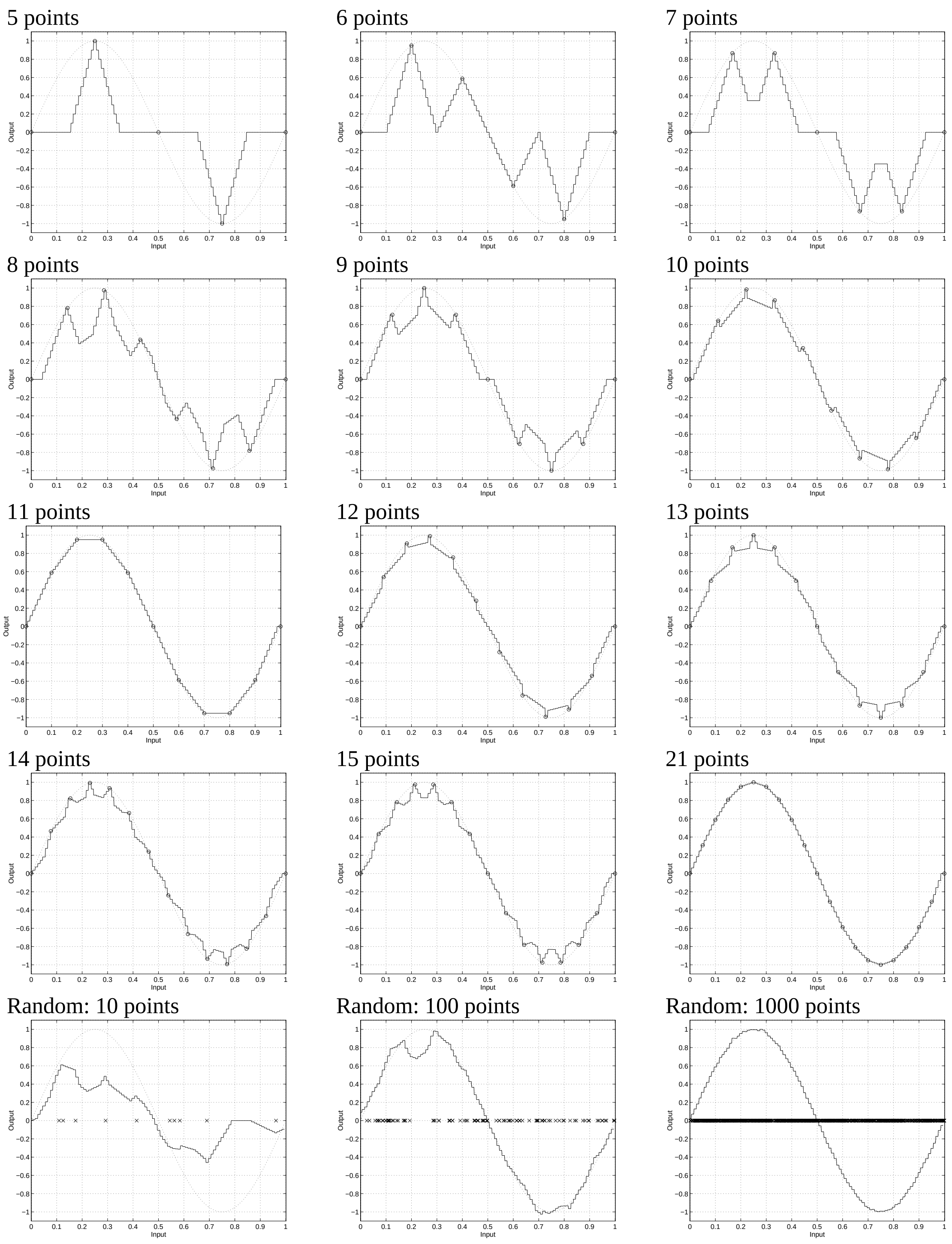


Figure 3.8: *Top four rows: the CMAC mapping resulting from training with sparse data, ranging from 6 to 21 training points spaced evenly between zero and one (marked with $\circ$'s). Bottom row: the CMAC mapping resulting from training with 10, 100 and 1000 randomly positioned points (input positions marked with $\times$'s, $\alpha = 0.5$). In both cases there are 100 quantization steps in the [0,1] interval, $n_a = 10$ and the training target for input y is $\sin 2\pi y$.*

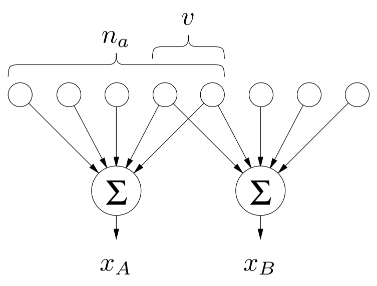


Figure 3.9: *The CMAC outputs for two different inputs that are within each other's local generalization areas.*

problem can be reduced with a lower learning rate. This effect is further explored in [122] for different training techniques.

3.5.6 Multidimensional inflexibility

A single input CMAC has enough weight parameters so that it can generate an independent output for each quantized input. In standard CMACs with more than one input this can not be done, because (as shown in figure 3.1) the association neurons (or weight table entries) are distributed rather sparsely in the quantized input space.

However, it was said above that for n_a weight tables, the table cell size was n_a times as coarse as the quantized input cells—but this is merely a convention, not a requirement, and so there could be n_T weight tables with a cell size that is n_c times coarser than the input cells. These tables would still have to be displaced from each other (because if two tables are perfectly aligned then one is redundant). The pattern of displacement is given in a displacement diagram (an example is shown in figure 3.2). Figure 3.10 shows some other displacement diagrams, along with their local generalization areas and association neuron layouts. In practice, $n_c = n_T = n_a$ is almost always used.

Figure 3.10c is fully populated, and so it is capable of representing a different output for each input cell, but normally this flexibility is not required. To reduce the number of weights, sparser distributions like figure 3.10a or figure 3.10b are commonly used. The displacement table should have at least one entry for each row and column. Figure 3.10b shows the simplest arrangement ($d_j^i = i$) which was suggested by Albus [2] and has been commonly used.

3.5.7 Comparison with the MLP

To represent a given set of training data the CMAC normally requires more parameters than an MLP. The CMAC is often over-parameterized for the training data, so the MLP has a smoother representation which ignores noise and corresponds better to the data's underlying structure. The number of CMAC weights required to represent a given training set is proportional to the volume spanned by that data in the input space.

The CMAC can be more easily trained on-line than the MLP, for two reasons. First, the training iterations are fast. Second, the rate of convergence to the training data can be made as high as required, up to the instant convergence of $\alpha = 1$ (although such high learning rates are never used because of training interference and noise problems). The MLP always requires many iterations to converge, regardless of its learning rate.

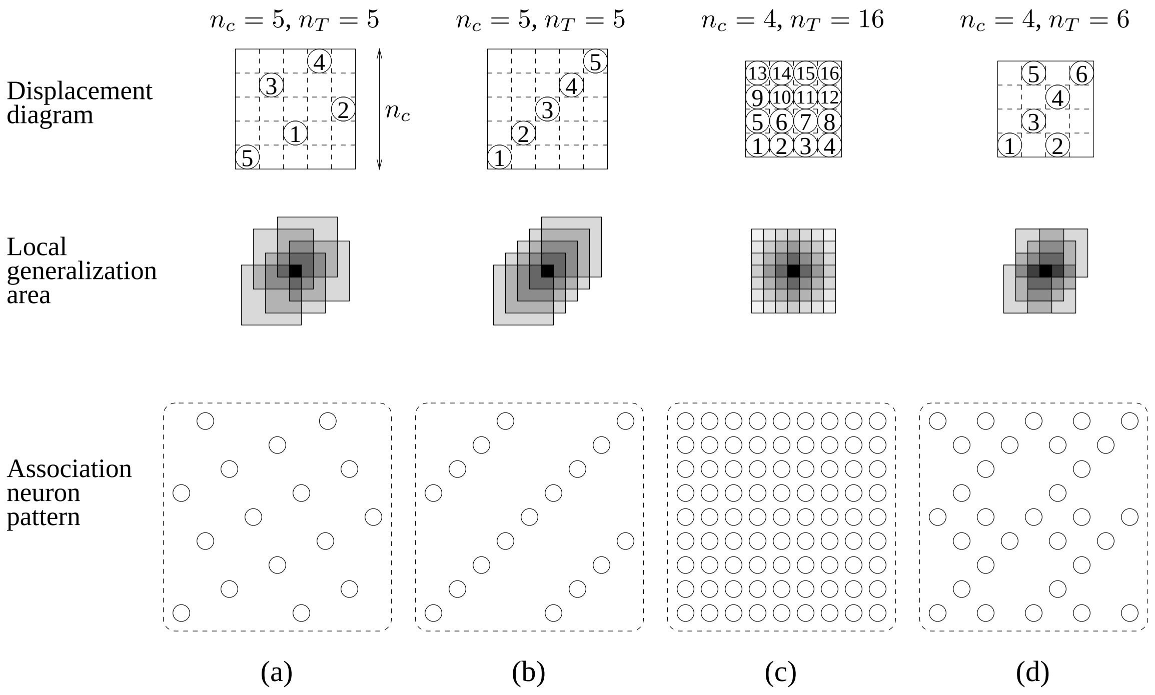


Figure 3.10: *Some example CMAC displacement diagrams and how they affect the local generalization area and association neuron layout.*

3.6 Design decisions

3.6.1 Input issues

When training with trajectory data, the input resolutions and n_a should be chosen to ensure that there is no training sparsity, i.e. so that successive training points are within the LG area of each other.

It can be useful to apply some function to an input before presenting it to the CMAC. These functions normally compress or expand the input space in certain areas so that the mapping detail can be concentrated where it is needed. A common choice is some variant of the sigmoid function

$$f(x) \quad = \quad \frac{1}{1+e^{-x}} \tag{3.19}$$

3.6.2 Overlay displacements

There are two common choices when choosing the overlay displacement values d_j^i:

1. Choose $d_j^i = i$ which means that the AU tables are aligned along a hyperdiagonal in the input space. This is reasonably effective and particularly easy, especially for hardware CMAC implementations.

2. Choose the d_j^i to get AU clustering that is optimal in some sense. For example, Parks and Militzer ([94] and [21, appendix B]) computed overlay displacement tables for n_a up to 100 and n_y up to fifteen, such that the minimum hamming distance between any two overlay displacement vectors

is maximized. It was found that particularly good overlay displacements are achieved when n_a is prime and equal to $2n_y + 1$, and bad overlay displacements are achieved when n_a is divisible by 6.

3.6.3 Hashing performance

Hash collisions limit the CMAC mapping accuracy. This problem will now be analyzed quantitatively. Let V be the fraction of the input space hypercube that contains the set of likely CMAC inputs. Note that this includes the local generalization area around that set (so one dimensional trajectories occupy a finite volume). Once the CMAC is trained, hashing will cause the area outside V to map to random noise. Define the "noise factor" to be:

$$\text{noise factor} \overset{\triangle}{=} \text{virtual weights mapped to each physical weight} \quad (3.20)$$

$$= \frac{V \cdot (\text{virtual weights})}{(\text{physical weights})} \quad (3.21)$$

$$= \frac{V \cdot \text{res}^{(n_y)}}{n_a^{(n_y-1)}\, n_w} \quad \text{(from equation 3.14)} \quad (3.22)$$

The noise factor for a particular configuration limits the accuracy which the CMAC mapping can achieve. Higher noise factors mean that more hash collision noise can be expected. For example, figure 3.11 shows a single-input single-output CMAC trained on a simple target function using 10,000 random training points and $\alpha = 0.3$. The graphs show the result for four different numbers of physical weights, with less weights giving a higher noise factor and a poorer mapping.

Suppose that the input trajectory is one dimensional and of length ℓ, and that the input space hypercube has side length s. Then

$$V = \frac{\ell \cdot (\text{trajectory width})^{n_y-1}}{s^{n_y}} \quad (3.23)$$

$$= \frac{\ell \cdot (n_a \frac{s}{\text{res}})^{n_y-1}}{s^{n_y}} \quad (3.24)$$

Which gives the following noise factor:

$$\text{noise factor} = \frac{\ell}{s} \cdot \frac{\text{res}}{n_w} \quad (3.25)$$

Note that this is independent of n_a and n_y, so for this ideal case the designer is free to choose those parameters based on other considerations.

The noise factor is not a very useful concept in practice, because V is hardly ever known beforehand—it is completely dependent on the system in which the CMAC is used. In many circumstances it is better to use a trial and error method to determine good CMAC parameters. Some reasonable CMAC parameters are initially chosen, and if the CMAC output is too noisy then the number of weights or n_a is increased, or the input resolution is reduced. Computer memory is cheap, so in many situations it is acceptable to initially use far more weights than are really needed.

3.6.4 Number of AUs

As n_a is increased, the number of parameters controlling the CMAC mapping is reduced. A single input CMAC has at least as many virtual weights as there are quantized input cells, so it is capable of representing any function. But standard multiple input CMACs do not have this property, so as n_a is

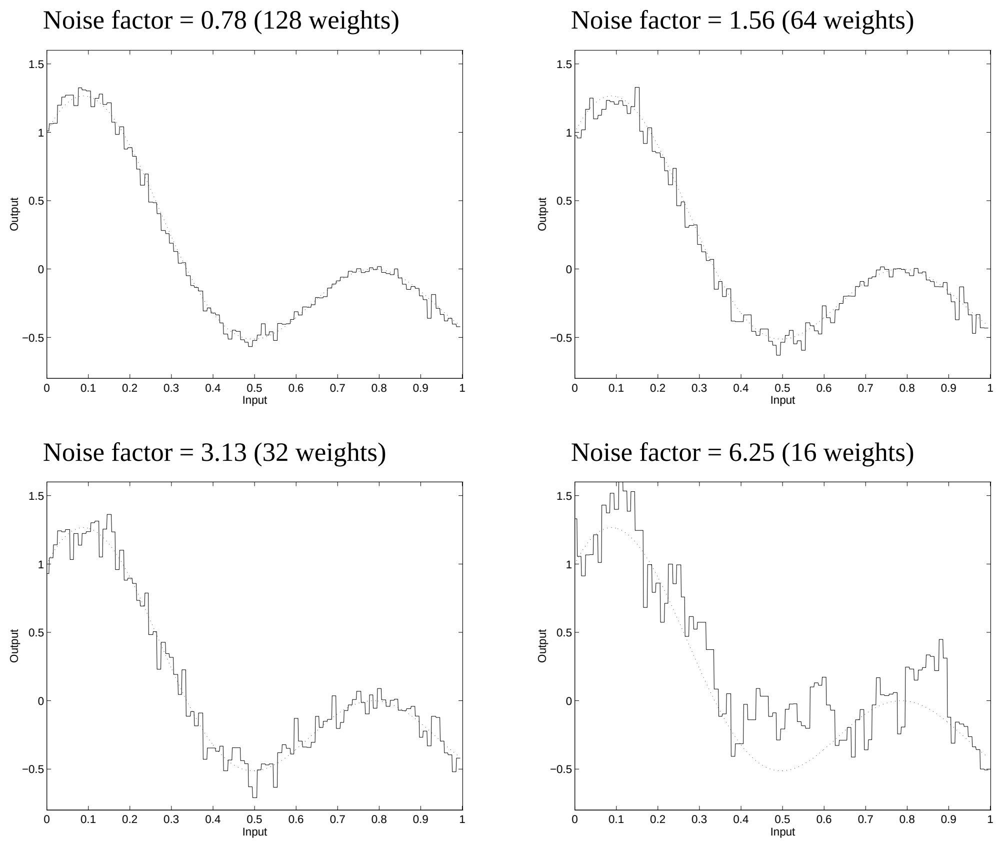


Figure 3.11: *Example CMAC mappings with different noise factors. The target function is* $x_1 = \sin 6y_1 + \cos 8y_1$, *and* $n_a = 10$.

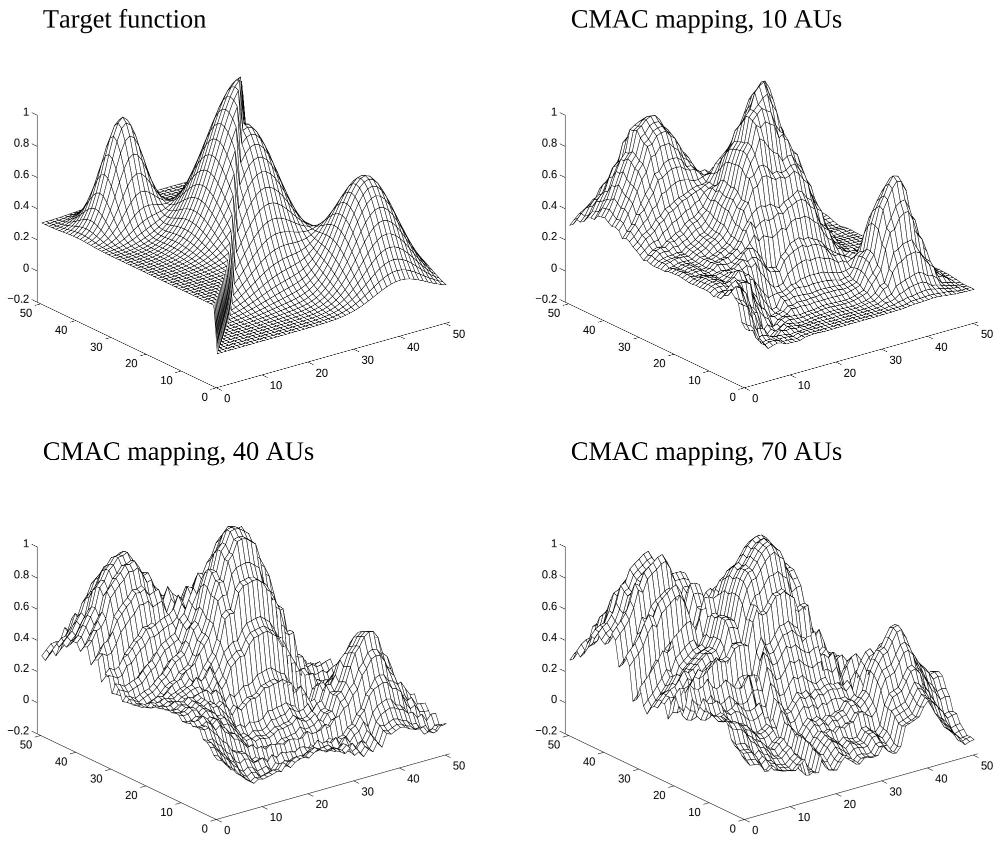


Figure 3.12: *Example 2-input CMAC mappings. The top left graph is the target function, the others show the best CMAC mappings for different AU numbers (n_a). The input resolution is 100 quantization steps along each input.*

increased the mapping flexibility is reduced. Figure 3.12 demonstrates what happens with two inputs. The top left graph is the target function—notice that it has a step discontinuity. The others graphs show CMAC mappings for n_a =10, 40 and 70 (these were trained using 10,000 random training points and $\alpha = 0.3$).

A higher n_a makes the mapping less accurate—for example it is less able to track quick variations. These inaccuracies are not the result of hash collisions (The number of weights was set to 2^{20} in this example to prevent that). With less parameters, the mapping is simply more constrained by the CMAC architecture. This can be useful to ensure a certain degree of "smoothness" in the CMAC output to compensate for an underconstraining training process. Note however that the CMAC can always perfectly represent training points along any *one-dimensional* trajectory, even for a high-dimensional CMAC.

3.6.5 Weight smoothing

As shown in figure 3.8, even if training points are closer together than the LG distance, perfect linear interpolation may not be achieved. A useful technique called "weight smoothing" was developed by the author to correct this problem, although it was never used in any application. After each training iteration the n_a indexed weights are adjusted according to

$$\text{for } j \leftarrow 1 \dots n_a : W_i[\mu_j] \quad \leftarrow \quad (1-\alpha_n) W_i[\mu_j] + \frac{\alpha_n}{n_a} \sum_{k=1}^{n_a} W_i[\mu_k] \tag{3.26}$$

In other words, each weight is moved towards the average of all the weights by an amount proportional to α_n. Repeated training with a small value of α_n distributes the weight more evenly, reducing the gaps between the largest and smallest weights.

Figure 3.13 shows the effects. These graphs were generated the same way as in figure 3.8, but the weight smoothing training algorithm was used. The top four rows, which use $\alpha_n = 0.01$, show that points closer than the LG distance are almost perfectly linearly interpolated. The bottom row, which uses $\alpha_n = 0.2$, show some benefit as well. The 100 point graph has a smaller error than figure 3.8 in the region where the training points are sparse. But there is a slight negative side-effect: the 1000 point graph shows a slight distortion from the desired function.

3.7 Extensions

Many modifications and extensions have been made to the basic CMAC concept. For example:

- Lane, Handelman and Gelfand [63] describe "higher order" CMAC networks. In these networks the layer 1 feature detecting neuron outputs[5] respond with n'th order spline functions (e.g. triangular or cubic) instead of just a binary 0/1, although the region of nonzero activation is still bounded the same way. With careful selection of the spline parameters, the CMAC mapping becomes smooth with analytical derivatives. Higher order CMAC networks can be chained together in multi-layer and hierarchical architectures that can be trained using conventional backpropagation techniques. However, they can require more weight parameters than a binary CMAC to achieve a given level of accuracy.

- Brown and Harris [21] provide a unified description which links the CMAC to related neural and fuzzy networks, including the B-spline network. They analyze the common features of these

[5] Refer to figure 3.1b.

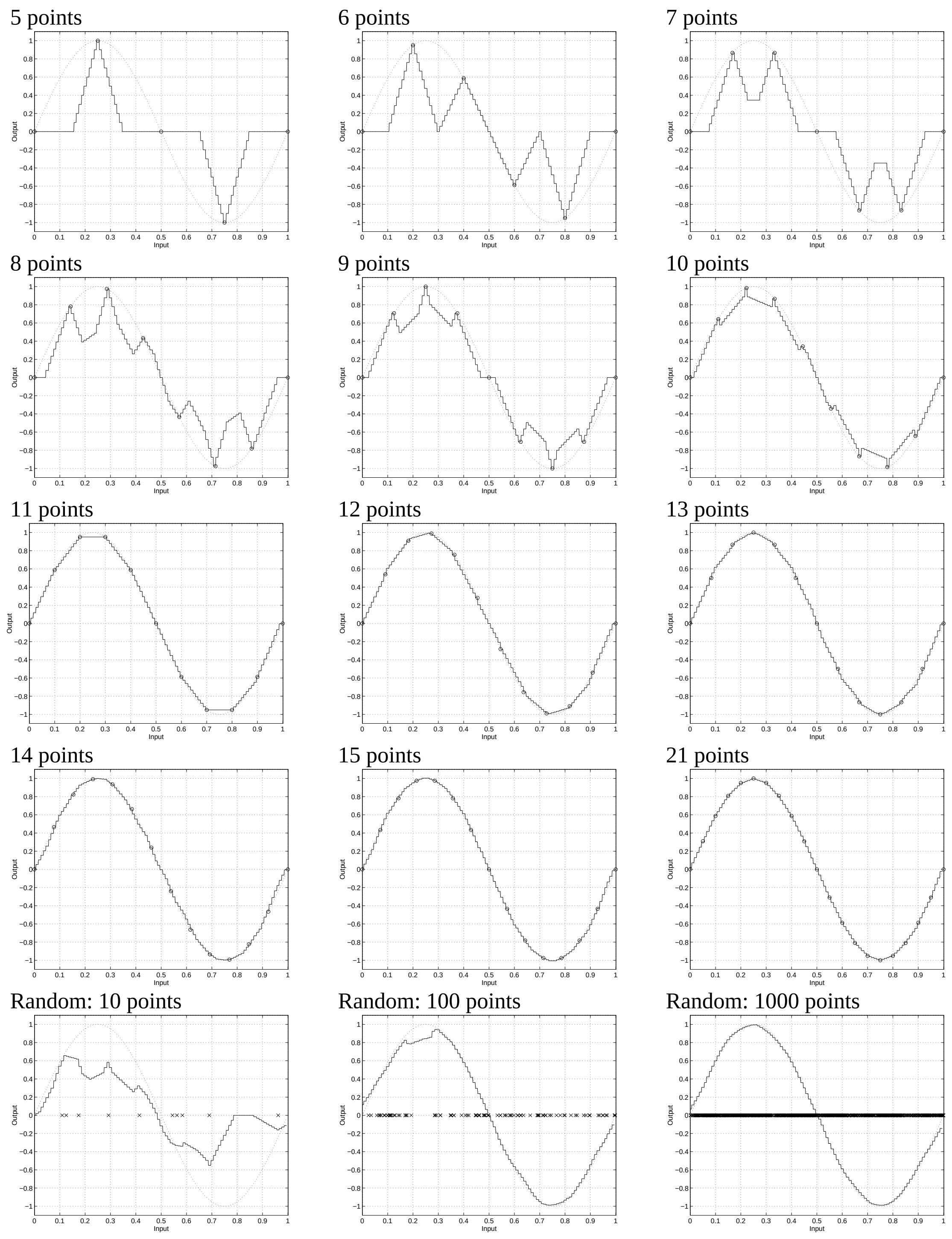


Figure 3.13: *This shows the same results as figure 3.8, but the weight smoothing training algorithm has been used. The top four rows use $\alpha_n = 0.01$, and the bottom row uses $\alpha_n = 0.2$.*

networks (such as local learning and interpolation) and compare them on a number of benchmark problems.

- Adaptive encoding of the CMAC inputs has been used to focus the representational detail on hard-to-model areas of the input space [33].
- There have been several efforts at CMAC hardware implementation, for example [54] which uses programmable logic devices. Hardware is less advantageous for the CMAC than it is for other neural network architectures, because the CMAC is so fast in software. The most difficult hardware design aspect is usually the hashing algorithm. Solutions like that in table 3.2 are typically used.

3.8 Conclusion

The CMAC neural network has the following advantages:

- The mapping and training operations are extremely fast. The time taken is proportional to the number of association units.
- The algorithms are easy to implement.
- Local generalization prevents over-training in one area of the input space from degrading the mapping in another (unless there are too few physical weights).

It has the following disadvantages:

- Many more weight parameters are needed than for, say, the multi-layer perceptron.
- The generalization is not global, so useful interpolation will only occur if there are enough training points—points further apart than the local generalization distance will not be correctly interpolated.
- The input-to-output mapping is discontinuous, without analytical derivatives, although this can be remedied with higher order CMACs [63].
- Selection of CMAC parameters to prevent excessive hash collision can be a large design problem. If there are a limited number of physical weights available then it is difficult to do without knowledge of the input signal coverage, so trial-and-error must usually be used. If physical weights are plentiful then it is acceptable to use far more than are really necessary and not worry about hash collisions.

Despite these problems, the CMAC is extremely useful in real-time adaptive control because of its speed. Far greater processing power would be required when using, say, an MLP network where every weight is involved in every mapping and training operation.

The next chapter will show how the CMAC can be used as the adaptive component in a simple “feedback-error” control system, and the chapter after that will show how the CMAC can be extended to make the FOX controller, which can be used to solve some problems that are beyond the capabilities of feedback-error.