Dynamic Memory Allocation for CMAC using Binary Search Trees

PETER SCARFE EUAN LINDSAY
Department of Mechanical Engineering
Curtin University of Technology
Bentley, Western Australia
AUSTRALIA
peter.scarfe@postgrad.curtin.edu.au, e.lindsay@curtin.edu.au
http://www.mech-eng.curtin.edu.au/lindsay

Abstract: - Cerebellar Model Articulation Controllers (CMACs) are a biologically-inspired neural network system suitable for trajectory control. Traditionally, CMACs have been implemented using hash-coding for their memory allocation, requiring static allocation of fixed amounts of memory in advance to the training of the system. This paper presents a method for implementing CMACs using Binary Search Trees to provide dynamic memory allocation, allowing for lower memory usage without compromising the functionality of the CMAC.

Key-Words: - CMAC, binary search trees, dynamic memory allocation, memory algorithms

1 Introduction

The Cerebellar Model Articulation Controller (CMAC) designed by Albus [1, 2], was originally designed to be used as a robotic controller. The CMAC was inspired by the biological cerebellum, the part of the brain found in 'higher animals' that can learn and control fine and precise movements of the animal's muscles. Playing the piano or even speaking are both muscle actuations in humans that require precise control over certain muscles, either individually or together, as a function of time. Originally designed by Albus to perform rote learning of movements of an artificial arm, CMACs have since been used extensively in a wide variety of systems from controllers to classifiers. The CMAC is able to learn the dynamic properties of a manipulator or actuator, and then respond to a desired input trajectory by producing the desired control response on the connected actuator. The more diverse the dynamic training data presented, the more skilled the CMAC will become. A solid biological explanation of the brains cerebellar and motor control complexity and behaviour is provided in [3-5].

Since Albus proposed the CMAC in 1975, several research groups have been using the CMAC with considerable success in a variety of applications from data mining classifiers [6-8] to unmanned aerial vehicle controllers [9], as well as several other uses. For a comprehensive list of CMAC uses and applications, Miller and Glanz's implementation of the CMAC paper provides a good list of CMAC uses prior to 1996 [10] and themselves utilized the CMAC in a number of real-time applications [11].

2 The CMAC Model

The CMAC uses a series of mappings (Fig 1), starting from an input vector **S** to an output vector or some scalar output value *p*, with various mappings in-between.

$$\mathbf{S} \rightarrow \mathrm{M} \rightarrow \mathrm{A} \rightarrow p$$

'S' represents an input vector of multiple dimensions. 'M' represents the single dimensional components in CMAC input space. 'A' represents the multi-dimensional representation receptive fields of '**S**'. 'A' directly corresponds to the weights associated with '**S**'. The summation of weights in 'A' then produces the output, '*p*'. Albus's 1981 book [4] gives a comprehensive explanation of the CMAC.

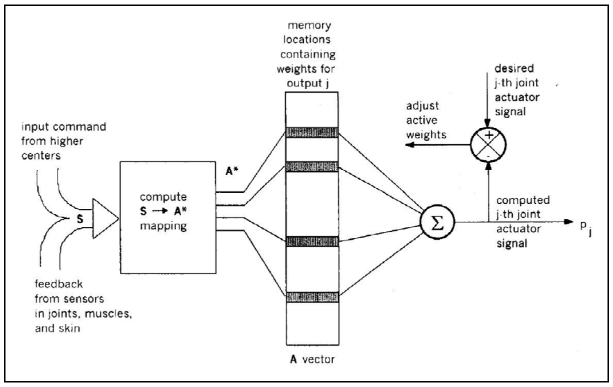

memory locations containing weights for output j
input command from higher centers
S
compute S → A* mapping
A*
adjust active weights
desired j-th joint actuator signal
+
-
Σ
computed j-th joint actuator signal
p_j
feedback from sensors in joints, muscles, and skin
A vector

Fig 1 - Block Diagram of Albus's CMAC for a single joint actuator [4].

Every input vector for each dimension in a CMAC excites several receptive field (RF) vectors for the same dimension. These receptive fields are parallel overlapping layers that exist over the input space (Fig 2). In this example the input vector (3,9) corresponds to receptive fields Ac, Gj and Nq for RF Layer 1, 2 and 3 respectively. The receptive fields that overlap the excited input vector are thus the corresponding excited receptive fields.

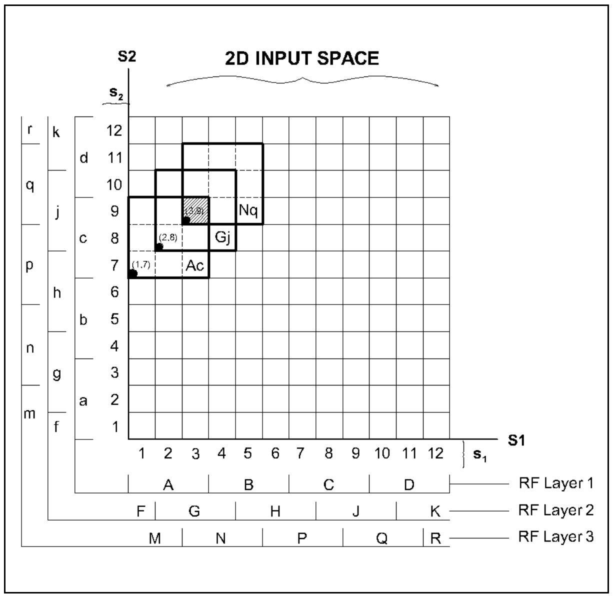


Fig 2 – From Input Vector to Receptive Fields

Each receptive field is directly linked to a weight in memory. Thus the address of the corresponding weight can be accessed by the address of the receptive field. In the example above the receptive field vectors are (1,7), (2,8) and (3,9) for receptive fields Ac, Gj and Nq respectively.

CMACs require memory addresses to store each corresponding receptive field weight. There are two major obstacles that determine how this memory allocation is preformed. If there were no memory limitations, then each memory address could be accessed directly for every possible weight. If there were no time limitations, then it would be possible to use minimal amounts of memory using sequential search methods. However, this is not the case.

Traditionally the CMAC memory requirement has been compressed by a method called hashing (or 'hash-coding') to reduce the memory requirements, making the CMAC practical to implement [12]. Efficient use of the available memory and fast access to the memory are the prime concerns of any hashing method [13]. This method was suggested to be used with the CMAC by Albus [1, 2] as a way to make the CMAC implementable on the hardware available at the time.

The downside to using hashing is that hash collisions may result. A hash collision occurs when two sets of data are stored at the same memory address. Hash collisions provide unfavourable effects on the convergence of the CMAC [14] and result as noise at the output. While methods have been designed to deal with hash collisions [10, 13], the static memory addressing and limited capability to utilize 100% of only the previously addressed memory weights provides only a partial solution to efficient memory handling.

In addition to hash coding, Hsu and Hwang, et al. [15] proposed a CMAC with a Content Addressable Memory (CAM) which is used in place of the hash coding method. This method uses a memory content search method to search for identical data in memory which has been previously assigned to a particular input vector. If no match is found, then the next available memory location is assigned to the new input vector. This way 100% of the memory is utilised, and grows unsupervised whenever the CMAC is presented with new input vectors.

The CMAC memory storage method here provides a dynamic solution over previous static methods. The code was written in the C programming language, and adapted from the freely available University of New Hampshire CMAC code [10], with the hashing memory allocation removed and replaced with the dynamic implementation presented in this paper.

3 Dynamic CMAC Memory Allocation

Dynamic allocation of memory when using CMACs on conventional personal computers is a most desirable property, as it is often not known how much memory is required to be allocated for a particular CMAC prior to its usage. Once the CMAC matures, if more memory is needed than was originally anticipated, unless the memory can be allocated when needed, the CMAC will fail to adapt further while retaining previously learnt knowledge.

Dynamic allocation of memory is particularly useful when a CMAC goes offline and data is required to be stored or transferred from its fast access online memory, to its slow access offline memory, usually a hard disk. Storing only the weights which have actually been used is preferable over storing large amounts of unused data predefined by the size of the static table allocated. For large systems utilizing an array of CMACs, the unused portions of memory produce an unnecessary memory storage overhead both in time and space.

4 Binary Search Trees

Binary Search Trees (BSTs) are a widely used method for storing data sets. Binary search trees are dynamic, requiring no advance information on the number of insertions needed. They also provide guaranteed worst case performance [13].

Binary search trees are named from their binary search property. A tree is made up of nodes as illustrated in Fig 3, with each node having two lower legs, with each leg linking to another node, which in turn connects to more nodes. Thus a parent-child structure is apparent, with the contents of the parent node always searched before the contents of its child nodes. The structure of the BST is such that all keys that are less than the parent node are stored in one sub-tree, with all keys that are greater than the parent node stored in the other tree. Thus when the parent node is searched, if the value of the key is not equal to the key being searched, then a less than or greater than comparison is made. The result then leads to the next comparison with one of the child nodes. And so the process continues throughout the tree until the correct value is found.

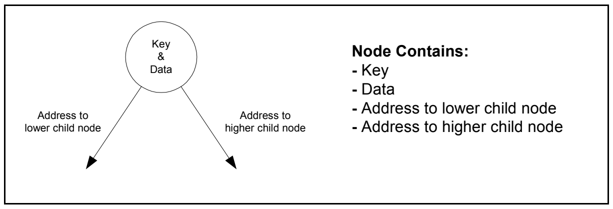

Key
&
Data
Address to lower child node
Address to higher child node
Node Contains:
- Key
- Data
- Address to lower child node
- Address to higher child node

Fig 3 - Single Binary Search Tree Node

The efficiency of data access times throughout a BST is dependent on the number of nodes to be searched before the required node is found. For optimal data access times, a BST should be balanced. If perfectly balanced the maximum number of nodes to be traversed will be no more than $|log_2(N)+1|$ comparisons, where N is the total number of nodes in the tree.

However when a BST is to be created, it is often impossible to create a perfectly balanced tree. The nodes would have to be inserted in a perfect order. Since most data sets are not known before insertion into a tree, randomly inserting the nodes into a tree will produce a close to balanced tree. On average, a search in a BST built from N random keys requires about $2log_e(N)$ comparisons [13].

Data in a BST can be easily manipulated; the nodes can be rearranged in any order required for optimal search efficiency. Thus building a BST by randomly inserting nodes can be rearranged into a balanced BST by simply sorting the nodes in order and then building a balanced BST. Fig 4 illustrates this process.

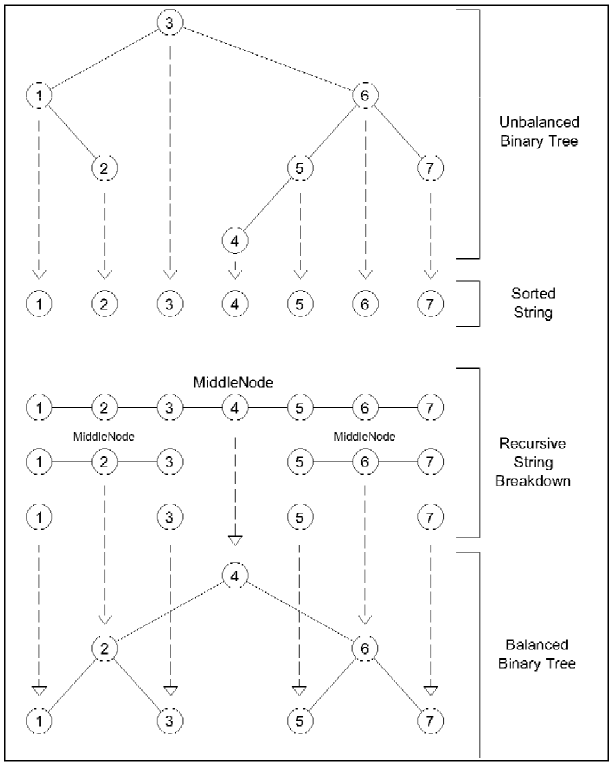


Fig 4 - Binary Tree Balancing – From Unbalanced to Balanced Tree

The BST with CMAC was implemented on a 32-bit processor, using four separate segments of data, each of 32 bit size. This requires 16 bytes of memory per node. Each node contains a key to locate the data in the BST, a CMAC weight, and two memory addresses (pointers in C) to the child nodes of the current parent node. This is illustrated in Fig 5.

Physical Memory Address (in bytes)	Data Stored in Memory Address
0x00000000	32bit integer (key)
0x00000004	32bit integer (weight)
0x00000008	32bit address (pointer to lower child node)
0x00000012	32bit address (pointer to higher child node)

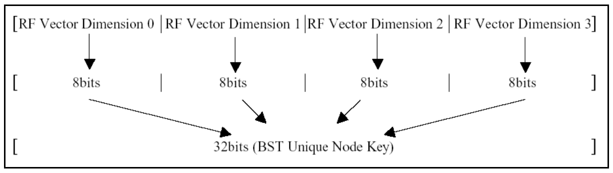
Fig 5 - Structure of data contained in each node within physical memory

Nodes were created in blocks of 1 MB as required, though this is totally customizable. Whilst there is the potential to allocate memory for each and every node, this adds considerable overhead to the insertion process. Allocating blocks reduces this, at the cost of having up to 1 MB of unused memory at any given time.

5 Node Key Generation

It is essential for each node in a BST to have a unique identifying key. The simplest way to form this key is to use the information regarding which receptive field in the CMAC is excited. The most obvious choice is to use the vector of an excited receptive field (RF) in input space resolution. However this is not the most memory efficient way to identify the receptive field key in a BST.

As illustrated in Fig 5, the key in each node is a 32-bit number. Thus there is a limitation as to how much data can be stored in 32 bits. As each key is a vector containing the dimensional vectors for each dimension, the data in the 32 bits needs to be both unique for any receptive field and hold as much information as possible. For a CMAC with a four dimensional input space, a 32 bit number can hold four lots of 8 bit data (Fig 6). Thus if each input space dimension has a resolution of 256 (8bits) then 32 bits can adequately hold a unique key for each excited input vector.


[RF Vector Dimension 0 | RF Vector Dimension 1 | RF Vector Dimension 2 | RF Vector Dimension 3]
[8bits | 8bits | 8bits | 8bits]
[32bits (BST Unique Node Key)]
Fig 6 - Four Vectors Represented as a 32bit Integer

Using the receptive field vector is also another way of representing unique keys in a BST. As each receptive field layer is independent from all other layers, a set of parallel data sets is generated. Each data set is to be assigned its own BST and thus this will produce in effect unique keys for each receptive field layer BST. While the vector will not be globally unique as in the case when using input space vector keys, the keys will still be unique to their own layer, and thus their own BST.

The local generalization property of the CMAC is a result of mapping an input space vector to a set of overlapping yet offset larger receptive fields. As each receptive field is larger than an input space vector, the resolution of each receptive field is less than the resolution of the input space. For example, take a four dimensional CMAC with input space resolution of 2048 (11bits) quantized units per dimension. With a CMAC generalization factor of 8, meaning that the CMAC will have 8 receptive field layers, each receptive field layer will be 8 times larger per input space unit, per dimension. This means that the resolution of each receptive field per dimension will be 2048/8 = 256. The number 256 can be stored in 8 bits of data, and thus since it is the receptive field vectors which are to be used to identify the keys in the BST, a CMAC with an four dimensional input space resolution of 11 bits per dimension can now be stored in 32 bits of data. This increases the input space resolution from 256 to 2048 per dimension.

The extraction of the receptive field layer vectors from the input space receptive field vectors is illustrated in Fig 8, overleaf.

The inherent nature of trajectory data posed a potential problem for the BST method of data storage. Because trajectories are continuous, consecutive data points will activate substantially similar receptive fields, and thus generate substantially similar keys. Building a BST from sequential keys leads to highly unbalanced trees, which do not present the desired advantages of efficient BST build and search speeds.

Inserting the nodes in a random order will achieve a more balanced tree, however this is not possible when the data is not all known in advance. The appearance of randomness, however, can be achieved through a scrambling process.

For each receptive field vector, a scrambling array was generated (see Fig 7 below). This array uniquely maps each of the original keys to a random number between 0 to the maximum RF vector value.

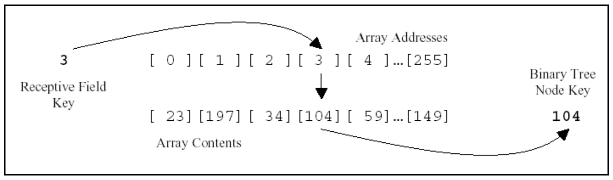


Fig 7 - Random Node Key Generator Array

The scrambled keys generated for each dimension i are the keys used as the receptive field vectors for their appropriate dimensions in Fig 6. The final 32bit key is then used directly as the key for a node

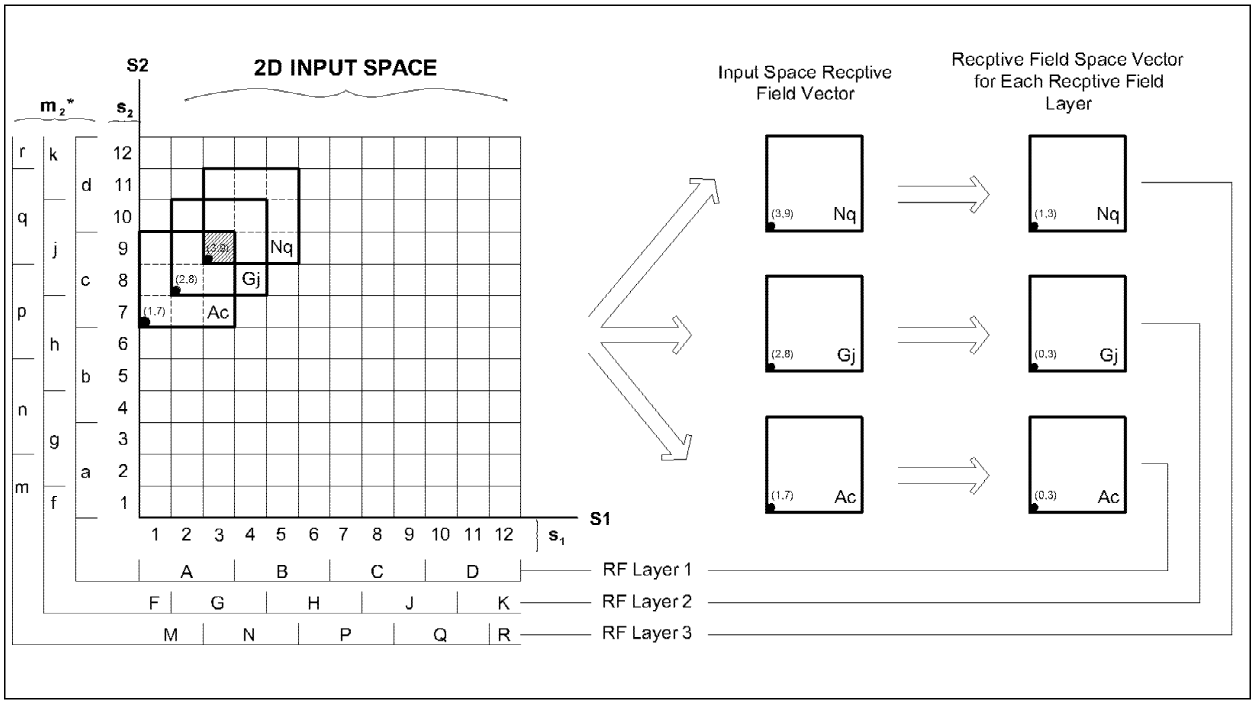


Fig 8 – Input Vector to Unique RF Vectors in Receptive Field Space

in the binary search tree for the key's corresponding receptive field layer. These keys are sufficiently 'random' for the tree to ensure that the BST that is generated is close to a balanced tree for whatever sequence of input vectors the CMAC encounters.

For optimal results however, close to balanced is not enough – the tree must be fully balanced. This process as illustrated in Fig 4 then can only be efficiently implemented whilst the CMAC is offline.

6 Conclusion

The CMAC controller architecture has been successfully developed in a 32-bit implementation of C using Binary Search Trees for memory storage. This allows for dynamic memory allocation, rather than the static memory allocation offered by the traditional hash-code memory implementations.

An innovative approach to key generation utilises the receptive field properties of the CMAC to uniquely identify nodes, and to exploit the structure of the CMAC to simplify the storage process. A process of key scrambling overcomes the challenges involved in building Binary Search Trees from near-sorted data, allowing for balanced (and thus efficient) trees to be constructed from trajectory data inputs.

This implementation allows for CMAC data to be stored in a smaller memory allocation – only the used weights are stored, without the need to store blank space for 'future' weights. In addition, the implementation allows mature CMACs to continue learning, without the constraint of a pre-determined maximum memory size.

The BST implementation allows for the advantages of dynamic memory allocation without compromising the functionality of the CMAC. This allows for CMACs to be implemented using fewer memory resources without a loss of capability.

References:

[1] J. S. Albus, "A new approach to manipulator control: The cerebellar model articulation controller (CMAC)," *Transactions of the ASME: Journal of Dynamic Systems, Measurement, and Control*, pp. 220-227, 1975.
[2] J. S. Albus, "Data Storage in the Cerebellar Model Articulation Controller (CMAC)," *Transactions of the ASME: Journal of Dynamic Systems, Measurement, and Control*, pp. 228-233, 1975.
[3] J. C. Houk, J. T. Buckingham, and A. G. Barto, "Models of the Cerebellum and Motor Learning," *Behavioral and Brain Sciences*, vol. 19, pp. 368-383, 1996.

[4] J. S. Albus, *Brains, Behavior and Robotics*. Massachusetts: Byte Publications, 1981.

[5] R. L. Smith, "Intelligent Motion Control with an Artificial Cerebellum," *PhD Thesis*, Department of Electrical and Electronic Engineering. University of Auckland, 1998.

[6] D. Cornforth, "The Kernel Addition Training Algorithm: Faster Training for CMAC based Neural Networks," presented at the Conference of Artificial Neural Networks and Expert Systems, Otago, 2001.

[7] G. Horvath, "CMAC: Reconsidereing an Old Neural Network," presented at The Intelligent Control Systems and Signal Processing (ICONS), Faro, Portugal, 2003.

[8] G. Horvath, "Kernal CMAC: an Efficient Neural Network for Classification and Regression," *Acta Polytechnica Hungarica*, vol. 3, pp. 5-20, 2006.

[9] F. G. Harmon, A. A. Frank, and S. S. Joshi, "The Control of a Parallel Hybrid-Electric Propulsion System for a Small Unmanned Aerial Vehicle using a CMAC Neural Network," *Neural Networks*, vol. 18, pp. 772-780, 2005.

[10] W. T. Miller and F. H. Glanz, "The University of New Hampshire Implementation of the Cerebellar Model Arithmetic Computer - CMAC," Robotics Laboratory, University of New Hampshire, Durham, New Hampshire 1996.

[11] W. T. Miller, F. Glanz, and L. G. Kraft, "CMAC: An associative neural network alternative to backpropagation," *Proceedings of the IEEE*, vol. 78, 1990.

[12] Z.-Q. Wang, J. L. Schiano, and M. Ginsberg, "Hash-coding in CMAC neural networks," presented at IEEE International Conference on Neural Networks, Washington, DC, USA, 1996.

[13] R. Sedgewick, *Algorithms in C*. USA: Addison-Wesley Publishing Company, Inc., 1990.

[14] T. Szabo and G. Horvath, "CMAC and its Extensions for Efficient System Modeling and Diagnosis," *International Journal of Applied Mathematics and Computer Science*, vol. 9, pp. 571-598, 1999.

[15] Y.-P. Hsu, K.-S. Hwang, and J.-S. Wang, "An Associative Architecture of CMAC for Mobile Robot Motion Control," *Journal of Information Science and Engineering*, vol. 18, pp. 145-161, 2002.