Kernel Recursive Least Squares for the CMAC Neural Network

C. W. Laufer and G. Coghill

***Abstract*—The Cerebellar Model Articulation Controller (CMAC) neural network is an associative memory that is biologically inspired by the cerebellum, which is found in the brains of animals. The standard CMAC uses the least mean squares algorithm to train the weights. Recently, the recursive least squares algorithm was proposed as a superior algorithm for training the CMAC online as it can converge in one epoch, and does not require tuning of a learning rate. However, the RLS algorithms computational speed is dependant on the number of weights required by the CMAC which is often large and thus can be very computationally inefficient. Recently also, the use of kernel methods in the CMAC was proposed to reduce memory usage and improve modeling capabilities. In this paper the Kernel Recursive Least Squares (KRLS) algorithm is applied to the CMAC. Due to the kernel method, the computational complexity of the CMAC becomes dependant on the number of unique training data, which can be significantly less than the weights required by non-kernel CMACs. Additionally, online sparsification techniques are applied to further improve computational speed.**

***Index Terms*—CMAC, kernel recursive least squares.**

I. Introduction

The Cerebellar Model Articulation Controller (CMAC) is a neural network that was invented by Albus [1] in 1975. The CMAC is modeled after the cerebellum which is the part of the brain responsible for fine muscle control in animals. It has been used with success extensively in robot motion control problems [2].

In the standard CMAC, weights are trained by the least mean square (LMS) algorithm. Unfortunately, the LMS algorithm requires many training epochs to converge to a solution. In addition, a learning rate parameter needs to be carefully tuned for optimal convergence. Recently, CMAC-RLS [3] was proposed where the recursive least squares (RLS) algorithm is used in place of the LMS algorithm. CMAC-RLS is advantageous as it does not require tuning of a learning rate, and will converge in just one epoch. This is especially advantageous in methods such as feed-back error learning [2] where online learning is used. In order to achieve such advantages, the price paid is an $O(n_w^2)$ computational complexity, where n_w is the number of weights required by the CMAC. Unfortunately, the number of weights required by the CMAC can be quite large for high dimensional problems. In [4] the inverse QR-RLS (IQRRLS) algorithm was used with the CMAC allowing real time RLS learning of low dimensional problems (less than three dimensions) on a PC, although the algorithm is still too computationally demanding for the real time learning of higher dimensional problems.

In [5] the kernel CMAC (KCMAC) trained with LMS was proposed. An advantage of the KCMAC is that it requires significantly fewer weights without the use of hashing methods. In the KCMAC only n_d weights are needed, where n_d is the number of unique training points presented. In most situations n_d is significantly less than n_w. Another advantage to the KCMAC is that the full overlay of basis functions can be implemented without requiring an unmanageable amount of memory space for the weights.

In [6] it was shown that the multivariate CMAC is not a universal approximator, and can only reproduce functions from the additive function set. The work in [5] showed that the reason for this is the reduced number of basis functions in the multivariate CMAC. When the full overlay of basis functions is used the CMAC becomes a universal approximator, with improved modeling capabilities. The full overlay of basis functions is typically not used as it would require a huge memory space. However, with the KCMAC the number of weights needed does not depend on the overlay, thus allowing the full overlay to be used. In this paper we show that the kernel RLS (KRLS) [7] algorithm can be used in the CMAC neural network. The proposed CMAC-KRLS algorithm combines the one epoch convergence and no learning rate selection advantages of the CMAC-RLS algorithms, whilst offering superior computational complexity, a smaller memory footprint and better modeling capabilities.

This paper is organized as follows. In Section II a brief introduction to the CMAC, CMAC-RLS and KCMAC is presented. In Section III the obvious CMAC-KRLS implementation is presented. In section IV optimizations to the obvious implementation are shown, and two 'discarding' methods which drastically improve computational performance at the expense of noise rejection are presented. Section V provides some results and comparisons against the discarding and non-discarding methods and against a CMAC-RLS implementation. Finally Section VI presents some conclusions.

II. Brief Introduction to the CMAC

A. Standard CMAC

The CMAC can be considered as a mapping $S \to M \to A \to P$. Where $S \to M$ is a mapping from an n_d-dimensional input vector $y = [y_1 \quad y_2 \quad \cdots \quad y_{n_d}]^T$ where

Manuscript received September 29, 2012; revised December 6, 2012.

The authors are with the Department of Electrical and Electronic Engineering, University of Auckland, Auckland, New Zealand (e-mail: clau070@aucklanduni.ac.nz, g.coghill@auckland.ac.nz)

$y_i \in R$ to a quantized vector $q = [q_1 \quad q_2 \quad \cdots \quad q_{n_d}]^T$ where $q_i \in Z$.

The mapping $M \rightarrow A$ is a non-linear recoding from vector q into a higher dimensional binary vector called the association vector, $X = [x_1 \quad x_2 \quad \cdots \quad x_{n_w}]^T$ where n_w is the number of weights in the CMAC and $x_i \in \{0,1\}$. The number of weights in the CMAC can be large but the association vector will only contain m '1's, where m is the number of layers in the CMAC.

In the mapping $A \rightarrow P$ the association vector is used to select and add together m values from an array of weights $W = [w_1 \quad w_2 \quad \cdots \quad w_{n_w}]^T$ where $w_i \in R$ to form the output. This can be viewed as an inner product calculation $X^T W$.

Learning in the CMAC corresponds to adjusting the value of the weights in order to produce a correct output for an input. In the standard CMAC, the LMS algorithm shown in (1) is used for this purpose, where μ is the learning rate, d_t is the desired output for training sample t, and $X^T W_{old}$ is the actual CMAC output.

$$W_{new} = W_{old} + \frac{\mu}{m} X^T \left(d_t - X^T W_{old} \right) \tag{1}$$

In Fig. 1 a visualization of a two input ($n_d = 2$) CMAC is shown with current quantized input $q = [4 \quad 8]^T$, quantizing resolution $r = 13$ in both dimensions, and $n_w = 64$. Here $m = 4$ layers are used, which correspond to the four weight tables on the right of the Fig. We can see that the input vector slices through the four layers on both axes. The sliced letters for each layer activate a certain weight in its corresponding weight table. Each individual weight corresponds to a hypercube in the input space, which for the 2D CMAC is simply a square. The activated hypercubes for the problem in Fig. 1 is shown as four squares diagonally arranged in the input space. Here weights *Bc*, *Fg*, *Jk* and *No* are activated. If put into activation vector form it will appear as,

$$\begin{array}{rccccccccccccc} & Aa & Ba & \cdots & Bc & \cdots & Fg & \cdots & Jk & \cdots & No & \cdots & Pp \\ x = \varphi(y_t) = [0 & 0 & 0 \cdots 0 & 1 & 0 \cdots 0 & 1 & 0 \cdots 0 & 1 & 0 \cdots 0 & 1 & 0 \cdots 0 & 0]^T \end{array}$$

The number of weights required by the CMAC grows exponentially with the input dimension, and can thus be very large. The number of weights in a CMAC is given by

$$n = \sum_{i=1}^{m} \prod_{j=1}^{n_d} \left\lfloor \frac{\left(r_j - 1\right) + d_j^i}{h} \right\rfloor + 1 \tag{2}$$

where r_j is the quantizing resolution in dimension j, d_j^i dictates how many quantization grid squares layer i in dimension j is displaced and h is the length of a 'full block'. An example of a full block in Fig. 1 is letter A which spans $h = 4$ four quantization grid squares, whereas letter D is not a full block because it does not span h grid squares. The number of layers m is usually given by $m = h$ for the diagonal and uniform overlay.

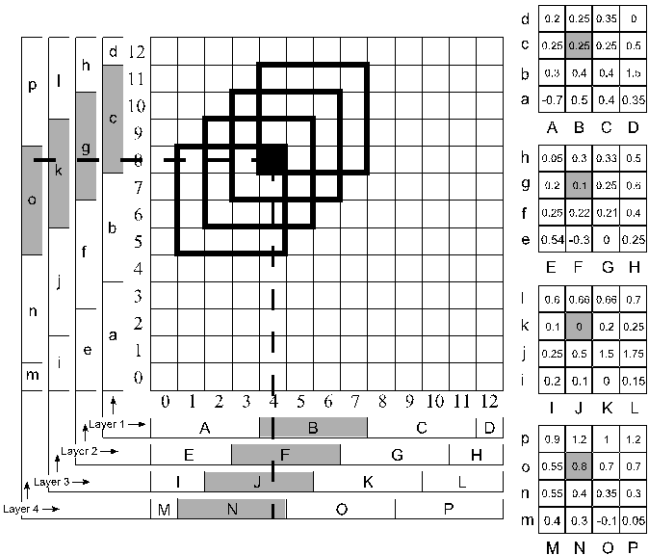

Fig. 1. A two-input CMAC example with four layers, diagonal overlay and requiring 64 weights.

1) Overlays

The displacement/arrangement of the layers/hypercubes plays a large role in the modeling performance of the CMAC. The standard Albus CMAC uses a diagonal overlay arrangement, and this is used in the CMAC example in Fig. 1, and is also shown in Fig. 2a. In [8] the so called 'uniform' arrangement shown in Fig. 2b is found which is an overlay yielding improved modeling performance. With the diagonal and uniform arrangements, the number of layers required is given by $m = h$. The parameter h is adjusted to control the amount of local generalization in the CMAC.

It is now well known that the multivariate CMAC can only approximate functions from the additive function set [6], and is thus not a universal approximator. In [5] it was shown that the reason for this is the limited number of basis functions in the multivariate CMAC. To fix this, the full overlay, shown in Fig. 2c, should be used where here the number of layers is given by $m = h^{n_d}$. The full overlay poses a problem however, as the number of weights required by the CMAC increases dramatically and often becomes too large to manage for high dimensional problems.

B. CMAC-RLS

Recently it was shown that the RLS algorithm can be used in the CMAC in place of the LMS update equation [3]. The use of CMAC-RLS is advantageous especially for online motion control learning in a stationary environment as the RLS algorithm allows the CMAC to learn in one epoch, and does not require tuning of a learning rate.

Other RLS algorithm implementation variants such as

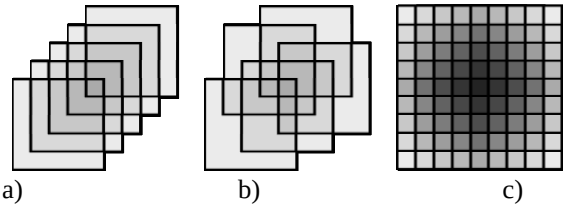

Fig. 2. The a) diagonal (m=h=5), b) uniform (m=h=5), and c) full (h=5, m=25) 2D overlay arrangements.

QR-decomposition RLS (QRRLS) [9] and inverse

QR-decomposition RLS (IQR-RLS) [4] have been used to improve the computational speed making CMAC-RLS feasible for low dimensional problems that require few weights.

C. Kernel CMAC

In a kernel machine, the input vector is non-linearly transformed into a higher dimensional 'feature vector' by a kernel function. The work in [5] makes the connection that the CMAC is essentially a kernel machine where the $M \to A$ mapping to the association vector is the non-linear transform to a higher dimension where the kernel used is a first order b-spline function. Using this knowledge, a common method used in kernel machines called the 'kernel trick' can be applied where the weights are then evaluated in the 'kernel space' rather than the feature space. Since the dimensionality of the kernel space is equal to the number of unique training data presented to the algorithm rather than n_w, significantly less memory is required for weight storage. Therefore, the number of weights used becomes independent of the type of overlay used, so it is feasible to use the full CMAC overlay. From [5] the output of the KCMAC is given as,

$$op = k^T \beta \tag{3}$$

where β is the weight vector in the kernel space, and K is the kernel vector given by $K = Xx$, where X is the dictionary and consists of the previously seen association vectors arranged as row vectors and x is the current association vector. The KCMAC LMS training algorithm from [5] is given as,

$$\beta_{new} = \beta_{old} + 2\mu K^T e \tag{4}$$

where K is a matrix consisting of previous kernel vectors arranged as row vectors, and e is a vector of errors. The vectors β, K, e and matrix K grow in size as more unique training data are presented to the KCMAC.

III. The Kernel RLS Algorithm for the CMAC

The online sparsifying KRLS algorithm is derived and presented in [7]. The KRLS algorithm will be a better choice than the RLS algorithm for training the CMAC, as the computational complexity will be dependent on the number of unique training data seen, rather than the number of training data possible. Hence, the full overlay of basis functions can be used. With sparsification techniques the number of training data required can be reduced even further. Here we quote the algorithm from [7] with slight alterations to specialize it for the CMAC. Note that the method for calculating the association vector is not shown here, but a good description of how it is calculated can be found in [10].

Algorithm 1 features an online sparsification technique that sparsifies by preventing feature vectors that are approximately linearly dependant on the dictionary, X from being added. The full concept and derivation behind this sparsification method can be found in [7]. Using this method the dictionary size can be limited, whilst still making use of training points not added to the dictionary. In (12) the scalar value δ is calculated which is a measure of how linearly dependant x is on the dictionary X. If δ is greater than some threshold v, this means that x was not approximately linear dependant on the dictionary, and will be added to the dictionary. Otherwise, if the threshold is not met, the update equations (20) – (22) will be used instead. The elements of vector a represent a weighting on how linearly dependant a vector in the dictionary is to the current feature vector. If the current feature vector is already in the dictionary, the entries of vector a will be all zero except for a single unity entry at the index of the matching dictionary point.

ALGORITHM I: KRLS-CMAC

$K^{-1} = [1/m],\ \beta = [d_1/m],$ $X = \varphi(quant(y_1)),\ P = [1],\ c = 1$		(5)
for $t = 2,3...n_t$		(6)
Get new sample: (y_t, d_t)		(7)
Quantize sample: $q = quant(y_t)$		(8)
Calculate association vector: $x = \varphi(q)$		(9)
$k = Xx$	$O(cm)$	(10)
$a = K^{-1}k$	$O(c^2)$	(11)
$\delta = m - k^T a$	$O(c)$	(12)
$if\ \delta > v$	$O(1)$	(13)
$X = \begin{bmatrix} X & x^T \end{bmatrix}^T$	$O(1)$	(14)
$K_{new}^{-1} = \frac{1}{\delta}\begin{bmatrix} \delta K_{old}^{-1} + aa^T & -a \\ -a^T & 1 \end{bmatrix}$	$O(c^2)$	(15)
$P_{new} = \begin{bmatrix} P_{old} & 0 \\ 0 & 1 \end{bmatrix}$	$O(c)$	(16)
$\beta_{new} = \frac{1}{\delta}\begin{bmatrix} \delta\beta_{old} - a\left(d_t - k^T\beta_{old}\right) \\ d_t - k^T\beta_{old} \end{bmatrix}$	$O(c)$	(17)
$c = c + 1$		(18)
else		
$q = \frac{P_{old}a}{1 + a^T P_{old} a}$	$O(c^2)$	(19)
$P_{new} = P_{old} - qa^T P_{old}$	$O(c^2)$	(20)
$\beta_{new} = \beta_{old} + K^{-1}q\left(d_t - k^T\beta_{old}\right)$	$O(c^2)$	(21)

The sparsification threshold should be set to some percentage of m. It was found that usually setting it to 10%-30% of m worked well.

IV. An Optimized Kernel RLS Algorithm for the CMAC

The CMAC-KRLS algorithm is still fairly computationally complex, with major bottlenecks at (10), (11), (12), and (20) - (22). Fortunately, most of these bottlenecks can be reduced by some optimizations presented below. First for reference, the optimized discarding CMAC-KRLS is presented in Algorithm 2.

A. Generation of the Kernel Vector

If the full overlay is used in a high dimensional CMAC, m can become extremely large. For example if $h = 20$, and $n_d = 4$, then $m = 20^4 = 160\,000$. This causes a computational burden for the KCMAC as the calculation of the kernel vector given by [5] is $k = Xx$. This requires $c \times m$ comparisons if the first order b-spline is used as the kernel function (binary CMAC) and x and X are stored sparsely. Although comparisons are efficient, if m is very large the computation will still be demanding.

ALGORITHM II: OPTIMIZED DISCARDING CMAC-KRLS

$$K^{-1} = [1/m],\ \beta = [d_1/m],\ Q = quant(y_1),\ P = [1],\ 0.98 \le \lambda \le 1,\ c = 1, \quad (22)$$

$$for\ t = 2,3...n_t \quad (23)$$

Get new sample: (y, d_t) (24)

Quantize sample: $q = quant(y_t)$ (25)

$$for\ i = 1:c \quad (26)$$

$$k_i = \max\left[\prod_{j=1}^{n_d} h - |Q_i - q|, 0\right] \quad O(n_d c) \quad (27)$$

$$if\ (Q.contains(q)) \quad O(1) \quad (28)$$

$$b = Q.indexof\ (q) \quad O(1) \quad (29)$$

$$q = \frac{P_{b,b}}{\lambda + P_{b,b}} \quad O(1) \quad (30)$$

$$P_{b,b} = \lambda^{-1}\left(P_{b,b} - qP_{b,b}\right) \quad O(1) \quad (31)$$

$$\beta_{new} = \beta_{old} + K^{-1}_{:,b} q\left(d_t - k^T \beta_{old}\right) \quad O(c) \quad (32)$$

$$elseif\ (!rejectDict.Contains(q)) \quad O(1) \quad (33)$$

$$a = K^{-1}k \quad O(c^2) \quad (34)$$

$$\delta = m - k^T a \quad O(c) \quad (35)$$

$$if\ \delta > v \quad O(1) \quad (36)$$

$$Q = \begin{bmatrix} Q & a^T \end{bmatrix}^T \quad O(1) \quad (37)$$

$$K^{-1}_{new} = \frac{1}{\delta}\begin{bmatrix} \delta K^{-1}_{old} + aa^T & -a \\ -a^T & 1 \end{bmatrix} \quad O(c^2) \quad (38)$$

$$P_{new} = \begin{bmatrix} P_{old} & 0 \\ 0 & 1 \end{bmatrix} \quad O(1) \quad (39)$$

$$\beta_{new} = \frac{1}{\delta}\begin{bmatrix} \delta\beta_{old} - a\left(d_t - k^T \beta_{old}\right) \\ d_t - k^T \beta_{old} \end{bmatrix} \quad O(c) \quad (40)$$

$$c = c + 1 \quad (41)$$

else

$$rejectDict = \begin{bmatrix} rejectDict \\ q^T \end{bmatrix} \quad O(1) \quad (42)$$

NOTE: $K^{-1}_{:,b}$ indicates the b'th column of K^{-1}

Here another method to calculate the kernel vector for the first order b-spline kernel is shown which is very efficient for the full overlay. By realizing that the individual kernel vector entries, k_i, are actually the number of shared hypercubes between dictionary point Q_i (where the dictionary Q stores quantized input vectors instead of association vectors), and current quantized input q, we can reduce the number of calculations required to calculate the kernel vector to $n_d \times c$. In Fig. 3 we see a 2D CMAC full overlay, where $h = 3$, and thus $m = 9$. We can view this Fig as having the dictionary point Q_i at the center, and the numbers in the surrounding grid squares give the number of shared hypercubes for nearby possible values of q. Equation (27) can be used to calculate number of overlaps k_i for a particular quantized dictionary point Q_i and the current quantized input q.

Using this method means that instead of storing the association vector in the dictionary, the quantized input vector should be stored instead. There is also no need to evaluate the association vector using this method, since k is now directly a function of the quantized input q rather than association vector x.

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

Fig. 3. Kernel vector values for Q_i at the center, and nearby possible q.

B. Discarding Sparsification

In any kernel machine used in an online learning environment, it is important to keep the dictionary size small so that it will be able to provide a response to input data in real time. In Algorithm 1 a sparsification technique was used which only added data that was not approximately linearly dependant on the dictionary. However, it still made use of every training point to adjust the weights, even if it was not in the dictionary.

In what we will call the discarding CMAC-KRLS implementation, data not added to the dictionary is simply discarded and not made use of. Thus, when performing (20) – (21) we see that the a vector is always all zero except for a single unity entry at the index where the matching dictionary entry is stored and thus the P matrix remains diagonal. So if the dictionary index for input q is known to be b, we only need to update scalars q_b (which is simply notated as q in Algorithm 2) and $P_{b,b}$. Thus, equations (20) to (21) can be simplified significantly as can be seen in equations (29) – (32). The disadvantage however is that, in a non-stationary environment the CMAC may be slower to adapt, or in a noisy environment the CMAC will be slower to converge as only if the dictionary points are re-visited will the CMAC update. If the CMAC must be used in a non-stationary or noisy environment, the non discarding Algorithm 1 can be used, the sparsification threshold can be reduced, or the semi-discarding algorithm presented next in section IV.C can be used.

This algorithm can also be written such that instead of performing the computationally demanding approximate linear dependence threshold test every iteration, it only need be performed if the current input is not a member of the dictionary already. This is because instead of using the test, a simple hashtable lookup as seen in (28) can be performed to see if the current quantized input vector is already a member of the dictionary. A hashtable lookup is a very efficient $O(1)$

operation. The threshold test will still need to be carried out in the case that the current input is not already in the dictionary.

Furthermore, if a point has been previously discarded and thus not added to the dictionary, it will never be added to the dictionary in the future. This is because as more points are added to the dictionary, the rejected point can only become more linearly dependant on the dictionary. This prevents the need to compute the approximate linear dependence test when seeing previously rejected points and is reflected by equations (33) and (43).

C. Semi-Discarding Sparsification

If increased noise performance is required, whilst retaining some of the good computational properties of the discarding method, a semi-discarding method shown in Algorithm 3 can be used. With the semi-discarding method, in the update section of Algorithm 1 (after the else statement) every value in the a vector is forcefully set to zero, except for the largest absolute value, which is the most contributing value and indicates the value in the dictionary most like the current input. The update algorithm is then performed with the masked a vector. This keeps the P matrix diagonal – the reason for fast computational performance. Modifications to get the semi-discarding algorithm are shown in Algorithm 3.

ALGORITHM III: SEMI-DISCARDING CMAC-KRLS

Same as Algorithm 2 but,
Replace (34) with an else statement
Replace (43) with four new lines:

$$b = a.index\left(\max|a|\right) \qquad O(c) \qquad (43)$$

$$q = \frac{P_{b,b}a_b}{\lambda + \left(P_{b,b}a_b^2\right)} \qquad O(1) \qquad (44)$$

$$P_{b,b} = \lambda^{-1}\left(P_{b,b} - qP_{b,b}a_b\right) \qquad O(1) \qquad (45)$$

$$\beta_{new} = \beta_{old} + K_{:,b}^{-1}q\left(d_t - k^T\beta_{old}\right) \qquad O(c) \qquad (46)$$

D. Forgetting Factor

A forgetting factor is typically used to allow RLS algorithms to track in non-stationary environments. The forgetting factor λ has been integrated into the update equations (30) and (31) in Algorithm 2, and also in (34) and (35) in Algorithm 3. A forgetting factor of around 0.98 to 1 is useful. Smaller values give better tracking performance, but decreased noise rejection.

E. Additional Computational Optimizations

The kernel vector is a sparse vector, and thus equation (34) can be sped up significantly by performing sparse vector matrix multiplication.

V. KRLS-CMAC RESULTS

In the following experiments each CMAC used a resolution of $r = 100$ for each dimension, and a local generalization parameter of $h = 10$. The experiments were run on an Intel i5 4-core CPU. The algorithm was written in C# and parallelization was applied where possible. The algorithms were tested on a two input sinc function, and various results are discussed below.

As mentioned previously, algorithms that rely on the kernel trick use as many weights as there are unique training points. They can use even less if sparsification methods are used. In Fig. 4 the number of weights used to learn the sinc function for any sparsifying CMAC-KRLS algorithm with full overlay is plotted against different sparsification thresholds. A total of 1681 unique training points were presented sequentially. The number of weights used by a CMAC-IQRLS algorithm is also plotted for diagonal and uniform overlays which are similar. The full overlay cannot be used in CMAC-IQRLS as it would require 11,881 weights which is not computationally feasible. Also note that if the problem was a higher dimensional problem, the number of weights required for CMAC-RLS would be much larger even with only the diagonal or uniform overlays.

In Fig. 5 the average time taken over ten epochs for each CMAC-KRLS algorithm with full overlay to complete learning of a single training point is plotted for various sparsification thresholds which are recorded as a percentage of m between 0 and 90%. Take note of the logarithmic scale. The number of weights admitted to the dictionary for a specific sparsification threshold can be seen on Fig. 4. Fig. 5 shows that the two discarding algorithms perform the fastest. This is because after the first epoch all the dictionary points have been added, thus the discarding algorithms use their very efficient update algorithm in subsequent epochs bringing the average down. The non-discarding algorithm is the slowest due to its more complex update equations. For comparison, the CMAC-IQRLS algorithm is shown for the same problem. It should be noted that although the CMAC-IQRLS algorithm is competitive with the non-discarding algorithm here, in a higher dimensional problem it would likely be infeasible whereas the CMAC-KRLS would perform at a similar speed no matter the dimension given the same number of unique training data.

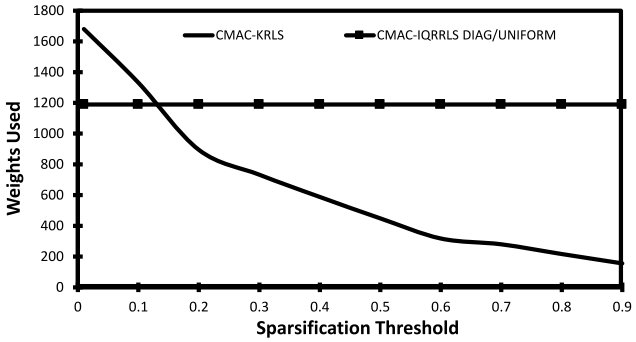

Fig. 4. Number of weights used for any CMAC-KRLS for different sparsification thresholds compared against the CMAC-IQRLS.

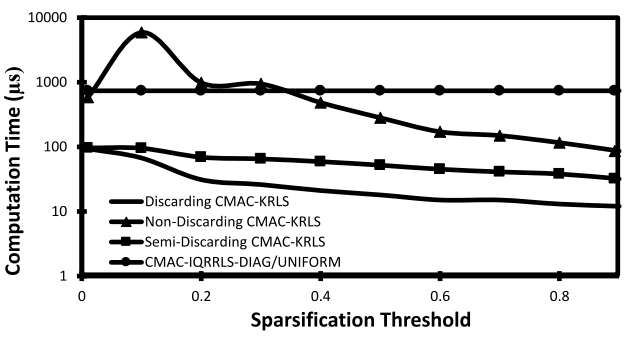

Fig. 5. Average time taken per iteration over ten training epochs.

The total absolute error for modeling the two input sinc function with no noise was measured for each CMAC-KRLS variant with full overlay and recorded in Fig. 6. Note that it was found that the non-discarding and semi-discarding algorithms required additional epochs to fully converge when trained sequentially, and thus the algorithm was run for 10 epochs before measuring the error. The discarding and non-discarding CMAC-KRLS algorithms were similar in performance up till a sparsification threshold of 0.5. The semi-discarding algorithm was only slightly worse than the non-discarding algorithm. For comparison the errors from the CMAC-IQRLS algorithm with the diagonal and uniform overlays are shown.

In Fig. 7 a comparison between the discarding, semi-discarding and non-discarding CMAC-KRLS for noisy data and random training points training under a sparsification threshold of 0.2 is shown. The non-discarding CMAC-KRLS performs significantly better as the number of training data increases due to its ability to make use of every data point. The discarding CMAC-KRLS only makes use of training points already in the dictionary, so it has less ability to average over time. The semi-discarding algorithm has improved performance over the discarding algorithm due to its ability to make some use of the discarded data.

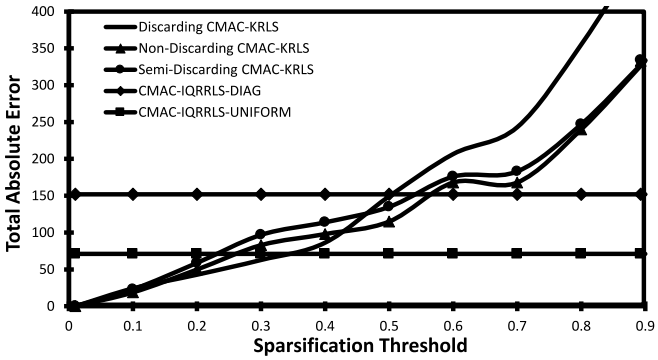

Fig. 6. Total absolute error for different sparsification thresholds.

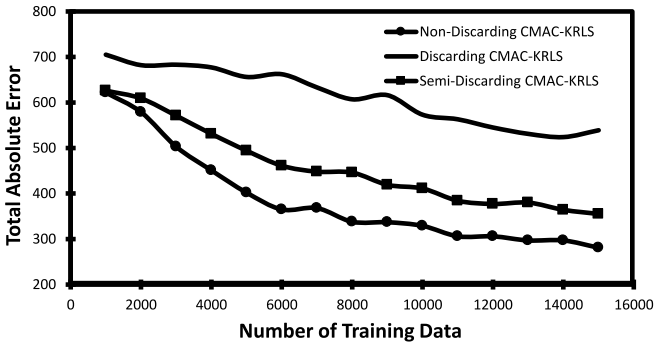

Fig. 7. Comparison with noisy random data between each CMAC-KRLS under a sparsification threshold of 0.2.

VI. Conclusion

In this paper the CMAC-KRLS algorithm was presented. It was shown that the CMAC-KRLS is more computationally efficient for high dimensional problems compared with the CMAC-RLS as its complexity is only ultimately dependant on the number of unique training data and not the number training points as it is with CMAC-RLS. It was also shown that the full overlay can be used in the CMAC-KRLS efficiently to improve the modeling capabilities if the kernel vector is calculated in a way that does not require computation of the association vector. A very efficient discarding and semi-discarding CMAC-KRLS algorithm was also presented that trades off noise rejection for computational speed. Overall, the CMAC-KRLS is shown to be a superior alternative to the CMAC-RLS algorithms for high dimensional problems in terms of modeling, computational, and memory performance.

References

[1] J. S. Albus, "New approach to manipulator control: The cerebellar model articulation controller (CMAC)," *Journal of Dynamic Systems, Measurement and Control, Transactions of the ASME*, vol. 97, pp. 220-227, 1975.

[2] M. K. H. Gomi, "Learning control for a closed loop system using feedback-error-learning," in *Proceedings of the 29th Conference on Decision and Control Honolulu*, Hawaii, 1990.

[3] T. Qin, *et al.*, "A learning algorithm of CMAC based on RLS," *Neural Processing Letters*, vol. 19, pp. 49-61, 2004.

[4] C. W. Laufer, "A regularized inverse QR decomposition based recursive least squares algorithm for the CMAC neural network," Unpublished.

[5] G. Horvath and T. Szabo, "Kernel CMAC with improved capability, systems, man, and cybernetics, Part B: Cybernetics," *IEEE Transactions on*, vol. 37, pp. 124-138, 2007.

[6] M. Brown, C. J. Harris, and P. C. Parks, "The interpolation capabilities of the binary CMAC," *Neural Networks*, vol. 6, pp. 429-440, 1993.

[7] Y. Engel, S. Mannor, and R. Meir, "The kernel recursive least-squares algorithm," *IEEE Transactions on Signal Processing*, vol. 52, pp. 2275-2285, 2004.

[8] P. C. Parks and J. Militzer, "Improved allocation of weights for associative memory storage in learning control systems," *1st IFAC symposium on Design Methods of Control Systems*, pp. 777-782, 1991.

[9] T. Qin, H. Zhang, Z. Chen, and W. Xiang, "Continuous CMAC-QRLS and its systolic array," *Neural Processing Letters*, vol. 22, pp. 1-16, 2005.

[10] R. L. Smith, "Intelligent motion control with an artificial cerebellum," Doctorate, Electrical and Electronic Engineering, University of Auckland, Auckland, 1998.