Efficient Recursive Least Squares Methods for the CMAC Neural Network

C. Laufer, 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 (RLS) algorithm was proposed as a superior algorithm for training the CMAC online as it can converge in just one epoch, and does not require tuning of a learning rate. However, the RLS algorithm was found to be very computationally demanding as its computational complexity is dependent on the square of the number of weights required which can be huge for the CMAC. Here, we show a more efficient RLS algorithm that uses inverse QR decomposition and additionally provides a regularized solution, improving generalization. However, while the inverse QR decomposition based RLS algorithm reduces computation time significantly; it is still not fast enough for use in CMACs greater than two dimensions. To further improve efficiency we show that by using kernel methods the CMAC computational complexity can be transformed to become dependent on the number of unique training data. Additionally, it is shown how modeling error can be improved through use of higher order basis functions.**

***Index Terms*—artificial neural networks, CMAC, kernel methods, recursive least squares**

I. Introduction

The Cerebellar Model Articulation Controller (CMAC) 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 $O(n^2)$ computational complexity, where n 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.

However, it is shown in [4] that by using a QR-decomposition based RLS algorithm, computation time can be reduced by half for a univariate CMAC. In this paper we show that the computation time can be further reduced for univariate and additionally multivariate CMACs by using an *inverse* QR decomposition RLS (IQR-RLS) algorithm, tailoring it for the CMAC and finally parallelizing it for use on multi-core CPUs. While the complexity remains $O(n^2)$, the new algorithm is fast enough to solve problems up to two dimensions at a reasonable speed.

For higher dimensional problems the IQR-RLS algorithm still falls short in terms of computation as its complexity remains $O(n^2)$. 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 at most only n_d weights are needed, where n_d is the number of unique quantized training points presented. In most situations n_d is significantly less than n, and additionally not every training point needs to be used. 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 a superior computational complexity, a smaller memory footprint and better modeling capabilities.

II. Brief Introduction to the CMAC

A. Standard CMAC

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

Manuscript received March 22, 2011.

Carl Werner Laufer is a PhD student in the Electrical and Electronic Engineering Department of the University of Auckland, New Zealand. (e-mail: clau070@aucklanduni.ac.nz)

George Coghill is a Senior Lecturer in the Electrical and Electronic Engineering Department of the University of Auckland, New Zealand. (e-mail: g.coghill@auckland.ac.nz)

$y_i \in$ □ to a quantized vector $\mathbf{q} = [q_1 \quad q_2 \quad \cdots \quad q_{n_y}]^T$ where $q_i \in$ □. Quantization is performed by the function

$$\mathbf{q}_j = \left\lfloor \mathbf{r}_j \times \frac{\mathbf{i}_j - \mathbf{min}_j}{\mathbf{max}_j - \mathbf{min}_j} \right\rfloor \qquad (2.1)$$

where j is the dimension, $\mathbf{q}_j$ is the quantized value, $\mathbf{r}_j$ is the desired quantization resolution, $\mathbf{i}_j$ is the original real valued input, and $\mathbf{max}_j$ and $\mathbf{min}_j$ are the known bounds of $\mathbf{i}_j$.

The mapping $M \to A$ is a non-linear recoding from vector $\mathbf{q}$ into a higher dimensional binary vector called the association vector, $\mathbf{x} = [x_1 \quad x_2 \quad \cdots \quad x_n]^T$ where n 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 $A \to P$ the association vector is used to select and add together m values from an array of weights $\mathbf{w} = [w_1 \quad w_2 \quad \cdots \quad w_n]^T$ where $w_i \in$ □ to form the output. This can be viewed as an inner product calculation $\mathbf{x}^T\mathbf{w}$.

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 (2.2) is used for this purpose, where μ is the learning rate, d_t is the desired output for training sample t, and $\mathbf{x}^T\mathbf{w}_{old}$ is the actual CMAC output.

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \frac{\mu}{m}\mathbf{x}^T\left(d_t - \mathbf{x}^T\mathbf{w}_{old}\right) \qquad (2.2)$$

In Fig 1 a visualization of a two input ($n_y = 2$) CMAC is shown with current quantized input $\mathbf{q} = [4 \quad 8]^T$, quantizing resolution $r = 13$ in both dimensions, and $n = 64$. Here $m = h = 4$ layers are used, which correspond to the four weight tables on the right of the figure. 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,

$$\begin{array}{cccccccccccccc} & Aa & Ba & \cdots & Bc & \cdots & Fg & \cdots & Jk & \cdots & No & \cdots & Pp & \\ \mathbf{x} = \varphi(\mathbf{q}) = [& 0 & 0 & 0\cdots0 & 1 & 0\cdots0 & 1 & 0\cdots0 & 1 & 0\cdots0 & 1 & 0\cdots0 & 0 &]^T \end{array}$$

where φ is the CMAC addressing function which is not shown here but can be found in [11]. We can also sparsely store this vector by simply storing the addresses of the activated weights,

$$activatedAddresses = \begin{bmatrix} 9 & 25 & 41 & 57 \end{bmatrix} \qquad (2.3)$$

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

$$n = \sum_{i=1}^{m} \prod_{j=1}^{n_y} \left\lfloor \frac{\left(\mathbf{r}_j - 1\right) + \mathbf{d}_j^i}{h} \right\rfloor + 1 \qquad (2.4)$$

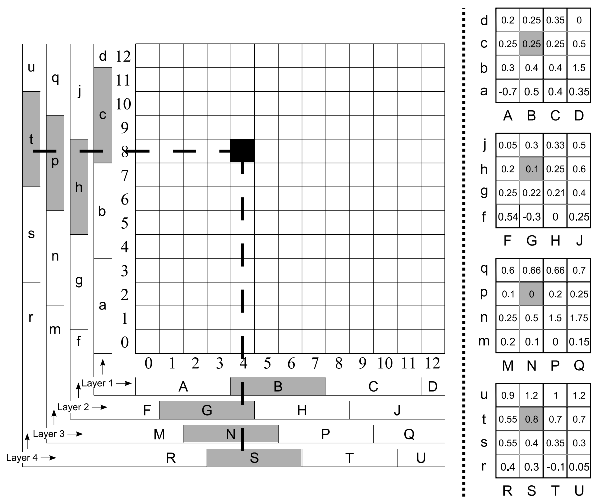

Fig 1. A two-input CMAC with four layers, diagonal overlay and requiring 64 weights.

where 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 the maximum $h = 4$ quantization grid squares. The number of layers m is usually given by $m = h$, however, as is seen in the next section this is not always the case.

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 2b. In [8] the so called 'uniform' arrangement shown in Fig 2a 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 is not a universal approximator. To fix this, the full overlay, shown in Fig 2c, should be used where here the number of layers is given by $m = h^{n_y}$. Using the full overlay poses a problem however, as the number of weights required by the CMAC increases dramatically as can be seen by (2.4), and often becomes too large to manage for high dimensional problems.

III. The Inverse QR-RLS Algorithm for the CMAC

QR-decomposition is a method for decomposing a matrix into two matrices, one orthogonal and the other upper triangular. It is useful for solving the linear least squares problem recursively [9] in a more numerically stable manner compared with standard RLS. Usually, using QR methods will degrade computational performance. However, the work in [4] tailors the QR-RLS algorithm specifically for the CMAC resulting in halving the computation time. Unfortunately, the tailored

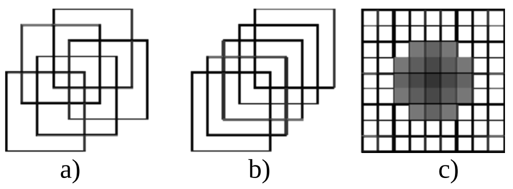
a) b) c)

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

algorithm is only suitable for univariate CMACs as the authors assume the association vector uses a method where the m '1's are contiguous, which cannot be the case for multivariate CMACs.

If the weight vector is required to be updated after every training sample presented, as it is required in the CMAC, a costly matrix back substitution step of $O(n^2)$ time complexity needs to be carried out each time. We can avoid this back substitution step entirely by using an *inverse* QR-RLS (IQR-RLS) algorithm, which instead allows the weights to be calculated directly. In Algorithm I we present an IQR-RLS algorithm that was derived in [10] and uses the Givens rotation method to perform the QR-decomposition, but has here been tailored for the CMAC in order to increase computational speed.

Where $a_i(k)$, $r_{ij}(k)$ and $u_j^{(i)}(k)$ are the individual entries of $\mathbf{a}(k)$, $\mathbf{R}(k)$ and $\mathbf{u}(k)$ respectively. Note that δ is a constant that is usually set between 10 and 10000. Larger values give theoretically better results, though it was found that setting δ too large causes floating point inaccuracies. A value of 100 was found to work well.

A. Optimizations

There are three speed improvements that are implemented in ALGORITHM I. The first improvement involves (3.9). The activatedAddresses array contains the array addresses of the m '1's in the association vector like what is shown in (2.3). This array is calculated by function φ^* in (3.6) which is a slight modification to function φ. In (3.9) the address where the first '1' appears in the association vector is recorded. The for loop in (3.10) then begins its computation from this address. This is because the $\mathbf{a}(k)$ vector calculated in (3.7) will be zero up until the address of the first '1' in $\mathbf{x}(k)$, as $\mathbf{R}^{-T}$ is lower triangular. If $a_i(k)$ is zero then $s(k)$ will equal zero, and $c(k)$ will equal one resulting in no change for $r_{ij}(k)$ and $u_j(k)$, rendering any calculation redundant.

ALGORITHM I: IQR-RLS ALGORITHM FOR THE CMAC

$\mathbf{w}(0)=\mathbf{0},\ \mathbf{x}(0)=\mathbf{0},$ $\mathbf{R}^{-T}(0)=\delta,\ \mathbf{I}_{n\times n}(\delta >> 1),\ \rho < 0.001$		(3.1)
for $k=1,2...n_d$		(3.2)
Get new sample: $(\mathbf{y}_k, d_k)$		(3.3)
Quantize sample: $\mathbf{q}=quant(\mathbf{y}_k)$	$O(n_y)$	(3.4)
Calculate association vector: $\mathbf{x}=\varphi(\mathbf{q})$	$O(mn_y)$	(3.5)
$activatedAddresses=\varphi^*(\mathbf{q})$		(3.6)
$\mathbf{a}(k)=\mathbf{R}^{-T}(k-1)\mathbf{x}(k)$	$O(mn)$	(3.7)
$\mathbf{u}(k)=\mathbf{0},\ \alpha^{(0)}(k)=1$	$O(1)$	(3.8)
$start=activatedAddresses_1(k)$		(3.9)
for $i=start:n$	$O(n^2)$	(3.10)
$givens()$		(3.11)
$e(k)=d(k)-\mathbf{x}^T(k)\mathbf{w}(k-1)$	$O(m)$	(3.12)
$z(k)=\dfrac{e(k)}{\alpha^{(n)}(k)}$	$O(1)$	(3.13)
$\mathbf{w}(k)=\mathbf{w}(k-1)-z(k)\mathbf{u}(k)$	$O(n)$	(3.14)
MACRO: $givens()$		
$if\ (\lvert a_i(k)\rvert) > \rho)$	$O(1)$	(3.15)
$\alpha^{(i)}(k)=\sqrt{\left[\alpha^{(i-1)}(k)\right]^2+a_i^2(k)}$		(3.16)
$s(k)=\dfrac{-a_i(k)}{\alpha^{(i)}(k)}$		(3.17)
$c(k)=\dfrac{\alpha^{(i-1)}(k)}{\alpha^{(i)}(k)}$		(3.18)
for $j=1:i$	$O(n)$	(3.19)
$r_{ij}(k)=c(k)r_{ij}(k-1)-s(k)u_j^{(i-1)}(k-1)$		(3.20)
$u_j^{(i)}(k)=c(k)u_j^{(i-1)}(k)+s(k)r_{ij}(k-1)$		(3.21)

The second improvement follows on from this where the calculation of (3.16) – (3.21) is gated by (3.15), and is thus only performed if the absolute value of $a_i(k)$ is greater than ρ which is set to a small value just above zero. Values of $a_i(k)$ are often zero due to the sparseness of the CMAC input which leaves $\mathbf{R}^{-T}$ sparse, and the sparseness of the association vector. We set ρ to be slightly larger than zero because during the matrix-vector multiplication in (3.7), values are often added and subtracted to form the sum of zero and due to floating point inaccuracies the result will not equal exactly zero. Furthermore, increasing ρ beyond the floating point inaccuracy boundary acts to decrease the accuracy of the solution and increase computation speed. Generally, a value for ρ between 0.000001 and 0.001 worked well.

Thirdly, a sparse matrix-vector multiplication can be performed with (3.7) because $\mathbf{x}(k)$ or the association vector is sparse, and the addresses of the '1's are known from the activatedAddresses array. Thus, only m values for each row of $\mathbf{R}^{-T}$ need to be added.

It can be seen from (3.10) that computation time will significantly increase with an increase in the number of weights required by the CMAC. We can combat this disadvantage for larger problems by using hash mapping to specify the number of weights to use.

B. Results

A two input sinc function was modeled on an Intel i5 CPU using the CMAC. Fig 4 shows the computation times recorded for a particular number of weights compared against other RLS algorithms used in the CMAC. The number of weights used by the CMAC was controlled by modifying the quantization resolution used. IQR-RLS was found to be the fastest of the RLS algorithms. Although it was previously said that the QR-RLS algorithm can halve the computation time of the standard RLS algorithm, we did not implement those speed enhancements from [4] as they would restrict the CMAC to a single input only. The QR-RLS algorithm was then many times slower than standard RLS as is evidenced in Fig 4.

Compared with the LMS algorithm which requires less than one microsecond per iteration, RLS algorithms are much slower. However, many epochs are required for the LMS algorithm to converge, which is not desirable in online learning.

IV. PARALLELIZED IQR-RLS ALGORITHM

The QR-RLS algorithm is naturally and optimally parallelized on a systolic array as is seen in [4]. The IQR-RLS algorithm from ALGORITHM I can also be parallelized in the same manner. A systolic array implementation of IQR-RLS is shown in Fig 3. In this figure, each circle represents a processing element that also stores the values of $\mathbf{R}^{-T}$ ($\mathbf{R}^{-T}$ is lower triangular). The circles in the left most column implement (3.16) – (3.21), and all other circular processing elements implement (3.20) and (3.21). The square boxes calculate the weights using (3.12) – (3.14). Often access to systolic array hardware is not available, and only PCs are available. Parallelization on a PC may be performed by emulating the systolic array computation structure with threading. However, this would introduce many threading overheads, and would potentially perform poorly. Here a simpler method is proposed to parallelize the IQR-RLS algorithm on a PC with a multi-core CPU by using the systolic array visualization.

In a visual sense, ALGORITHM I sequentially updates the $u_j^{(i)}$ and r_{ij} values in the systolic array row by row. It is, however, equally valid to update the $u_j^{(i)}$ and r_{ij} values column by column instead. If the column by column method is used, since each column is independent from one another in terms of other value dependencies (apart from c and s), it is possible to update each column simultaneously and without memory sharing bottlenecks. As it is a simple exercise to parallelize ALGORITHM I, we do not present the algorithm explicitly, but instead describe how it may be parallelized in two steps below.

The first step that needs to be performed is sequential in nature. First, realize that the c and s values are constant across each row. Thus the values of c and s must first be sequentially calculated for each row and stored in an array. The value $\alpha^{(n)}$ is also calculated and stored as a by-product from calculating c and s.

In step two we realize that we can update $u_j^{(i)}$ and r_{ij} column by column. Since each column is independent of one another, each column can be updated in a separate thread. We can further optimize by combining computation of shorter columns together to equalize thread computation times and by skipping calculations on rows where the a_i value is below the threshold ρ.

Additionally, fine grained parallelism on a modern CPU can be achieved by using 'Streaming SIMD Extensions' (SSE). Currently, SSE instructions allow two double precision, and four single precision multiplications to be performed simultaneously. This is especially useful for the inner loop calculations (3.20) and (3.21).

A. Results

The algorithm was used to model a two input sinc function and was run on a 4-core Intel i5 processor. It was found that parallelization slightly slowed down computation for small problems due to threading overheads, but decreased computation times for larger problems. We can expect for this algorithm to become automatically faster as processor core counts increase. A computational time comparison between the sequential and parallel versions of IQR-RLS is plotted in Fig 4. The parallel algorithm implements both threading and SSE based parallelization.

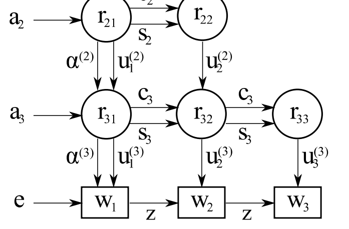

Fig 3. Parallel systolic array implementation of the IQR-RLS algorithm

V. REGULARIZED IQR-RLS

The work in [5, 6, 11, 12] show that the generalization error of the CMAC can be significant. In [5] a method called 'regularization' is presented for the LMS algorithm, which considerably reduces the generalization error. Regularization combats a design flaw in the CMAC by forcing activated weights to be similar, thus preventing certain weights dominating the contribution to the output calculation. Here we apply the same regularization concept but instead to the IQR-RLS algorithm. A partial mathematical derivation is given below based of the derivations found in [10]. First from [5] we use the least squares cost function $\varepsilon(k)$ where k is the current training sample iteration,

$$\varepsilon(k)=\sum_{i=1}^{k}\left(\begin{array}{l}\left[d(i)-\mathbf{x}^T(i)\mathbf{w}(k)\right]^2\\ +\eta\sum_{j:x_j(k)=1}\left[\dfrac{d(i)}{m}-w_j(k)\right]^2\end{array}\right) \tag{5.1}$$

The first term in (5.1) is the error between the desired and actual CMAC output. The second term is the regularization term which adds to the cost if the activated weights are different to one another in value. The constant η is used to control the amount of regularization and it was found setting it to the reciprocal of δ generally worked well. In order to easily minimize (5.1), it must be written in vector-matrix form. Define vectors $\mathbf{w}(k)$, $\mathbf{d}(k)$, $\mathbf{h}(k)$ and matrices $\mathbf{X}(k)$, $\mathbf{\Sigma}(k)$ as,

$$\mathbf{w}(k)=\begin{bmatrix}w_1(k) & w_2(k) & \cdots & w_n(k)\end{bmatrix}_{n\times1}^T \tag{5.2}$$

$$\mathbf{d}(k)=\begin{bmatrix}d(1) & d(2) & \cdots & d(k)\end{bmatrix}_{k\times1}^T \tag{5.3}$$

$$\mathbf{h}(k)=\begin{bmatrix}\mathbf{G}(1)\mathbf{q}(1) & \mathbf{G}(2)\mathbf{q}(2) & \cdots & \mathbf{G}(k)\mathbf{q}(k)\end{bmatrix}_{nk\times1}^T \tag{5.4}$$

$$\mathbf{X}(k)=\begin{bmatrix}\mathbf{x}(1) & \mathbf{x}(2) & \cdots & \mathbf{x}(k)\end{bmatrix}_{k\times n}^T \tag{5.5}$$

$$\mathbf{\Sigma}(k)=\begin{bmatrix}\mathbf{G}(1) & \mathbf{G}(2) & \cdots & \mathbf{G}(k)\end{bmatrix}_{nk\times n}^T \tag{5.6}$$

where,

$$\mathbf{q}(k)=\begin{bmatrix}\dfrac{d(k)}{m} & \dfrac{d(k)}{m} & \cdots & \dfrac{d(k)}{m}\end{bmatrix}_{n\times1}^T \tag{5.7}$$

$$\mathbf{G}(k)=\left[diag\left(\mathbf{x}(k)\right)\right]_{n\times n} \tag{5.8}$$

$diag\left(\mathbf{x}(k)\right)$ creates an $n\times n$ zero matrix with the entries of $\mathbf{x}(k)$ along the main diagonal. Using (5.2) – (5.8) we can rewrite the cost function as,

$$\varepsilon(k)=\left\|\mathbf{d}(k)-\mathbf{X}(k)\mathbf{w}(k)\right\|^2+\eta\left\|\mathbf{h}(k)-\mathbf{\Sigma}(k)\mathbf{w}(k)\right\|^2 \tag{5.9}$$

Where $\|\mathbf{a}\|=\sqrt{\mathbf{a}^T\mathbf{a}}$. Equation (5.9) can be rewritten as a single term by defining matrix $\mathbf{A}(k)$ and vector $\mathbf{y}(k)$ as

$$\mathbf{A}(k)=\begin{bmatrix}\mathbf{X}(k)\\ \sqrt{\eta}\mathbf{\Sigma}(k)\end{bmatrix}_{(k+nk)\times n} \tag{5.10}$$

$$\mathbf{y}(k)=\begin{bmatrix}\mathbf{d}(k)\\ \sqrt{\eta}\mathbf{h}(k)\end{bmatrix}_{(k+nk)\times 1} \tag{5.11}$$

using (5.10) and (5.11) to rewrite $\varepsilon(k)$ then gives,

$$\varepsilon(k)=\left\|\mathbf{y}(k)-\mathbf{A}(k)\mathbf{w}(k)\right\|^2 \tag{5.12}$$

Now from [10] we see that since $\mathbf{A}(k)$ is $(k+nk)\times n$, there exists a $(k+nk)\times(k+nk)$ orthogonal matrix $\mathbf{Q}(k)$ such that,

$$\mathbf{Q}(k)\mathbf{A}(k)=\begin{bmatrix}\mathbf{R}(k)\\ \mathbf{0}\end{bmatrix} \tag{5.13}$$

Where $\mathbf{R}(k)$ is the $n\times n$ upper triangular Cholesky factor, and $\mathbf{0}$ is an $((k+nk)-n)\times n$ zero matrix. Similarly,

$$\mathbf{Q}(k)\mathbf{y}(k)=\begin{bmatrix}\mathbf{z}(k)\\ \mathbf{v}(k)\end{bmatrix} \tag{5.14}$$

Where $\mathbf{z}(k)$ is a $n\times 1$ vector, and $\mathbf{v}(k)$ is a $((k+nk)-n)\times 1$ vector. Since $\mathbf{Q}(k)$ is orthogonal, pre-multiplying each term in (5.12) does not change the value of the norm,

$$\varepsilon(k)=\left\|\mathbf{Q}(k)\mathbf{y}(k)-\mathbf{Q}(k)\mathbf{A}(k)\mathbf{w}(k)\right\|^2 \tag{5.15}$$

Substitute (5.13) & (5.14) into (5.15) to get the desired form,

$$\varepsilon(k)=\left\|\begin{bmatrix}\mathbf{z}(k)-\mathbf{R}(k)\mathbf{w}(k)\\ \mathbf{v}(k)\end{bmatrix}\right\|^2 \tag{5.16}$$

It can be seen that the norm in (5.16) will be minimized if,

$$\mathbf{R}(k)\mathbf{w}(k)=\mathbf{z}(k) \tag{5.17}$$

With (5.17) the weights can be solved for with back substitution. Now the problem becomes how to update $\mathbf{R}(k-1)$ to $\mathbf{R}(k)$ and $\mathbf{z}(k-1)$ to $\mathbf{z}(k)$. First, consider the non-regularized solution. In the non-regularized solution, $\mathbf{A}(k)$ and $\mathbf{y}(k)$ in (5.12) are replaced with $\mathbf{X}(k)$ and $\mathbf{d}(k)$. Now, in [10] it is shown that an $(n+1)\times(n+1)$ orthogonal matrix $\mathbf{T}(k)$ exists that will perform the non-regularized update by updating using the latest entry of $\mathbf{X}(k)$ and $\mathbf{d}(k)$, which are $\mathbf{x}(k)$ and $d(k)$ respectively,

$$\mathbf{T}(k)\begin{bmatrix}\mathbf{R}(k-1)\\ \mathbf{x}^T(k)\end{bmatrix}=\begin{bmatrix}\mathbf{R}(k)\\ \mathbf{0}^T\end{bmatrix} \tag{5.18}$$

$$\mathbf{T}(k)\begin{bmatrix}\mathbf{z}(k-1)\\ d(k)\end{bmatrix}=\begin{bmatrix}\mathbf{z}(k)\\ \zeta(k)\end{bmatrix} \tag{5.19}$$

However, with regularization, (5.12) uses $\mathbf{A}(k)$ and $\mathbf{y}(k)$ which is a composition of two matrices and two vectors respectively, so we must update with the latest entries of $\mathbf{X}(k)$, $\mathbf{d}(k)$ *and* the latest entries of $\mathbf{\Sigma}(k)$, $\mathbf{h}(k)$ multiplied by $\sqrt{\eta}$ which are $\sqrt{\eta}\mathbf{G}(k)$ and $\sqrt{\eta}\mathbf{G}(k)\mathbf{q}(k)$ respectively. There must then exist an $(2n+1)\times(2n+1)$ matrix $\mathbf{T}(k)$ such that,

$$\mathbf{T}(k)\begin{bmatrix}\mathbf{R}(k-1)\\ \mathbf{x}^T(k)\\ \sqrt{\eta}\mathbf{G}(k)\end{bmatrix}=\begin{bmatrix}\mathbf{R}(k)\\ \mathbf{0}^T\\ \mathbf{0}\end{bmatrix} \tag{5.20}$$

and similarly to update $\mathbf{z}(k-1)$ to $\mathbf{z}(k)$,

$$\mathbf{T}(k)\begin{bmatrix}\mathbf{z}(k-1)\\ d(k)\\ \sqrt{\eta}\mathbf{G}(k)\mathbf{q}(k)\end{bmatrix}=\begin{bmatrix}\mathbf{z}(k)\\ \zeta(k)\\ \boldsymbol{\varphi}(k)\end{bmatrix} \tag{5.21}$$

Unfortunately, if (5.20) and (5.21) are used, we cannot proceed with the same derivations found in [10] as the derivations are suited only to a $(n+1)\times(n+1)$ $\mathbf{T}(k)$ matrix. We, however, realize that $\mathbf{R}(k)$ and $\mathbf{z}(k)$ may be calculated iteratively if we define $\mathbf{R}_n(k)\equiv\mathbf{R}(k)$, $\mathbf{z}_n(k)\equiv\mathbf{z}(k)$ and write $\mathbf{G}(k)$ and $\mathbf{q}(k)$ row by row as,

$$\mathbf{G}(k)=\begin{bmatrix}\mathbf{g}_1(k) & \mathbf{g}_2(k) & \cdots & \mathbf{g}_n(k)\end{bmatrix}^T \tag{5.22}$$

$$\mathbf{G}(k)\mathbf{q}(k)=\begin{bmatrix}\mathbf{g}_1(k)\mathbf{q}(k) & \mathbf{g}_2(k)\mathbf{q}(k) & \cdots & \mathbf{g}_n(k)\mathbf{q}(k)\end{bmatrix}^T \tag{5.23}$$

then we first update using $\mathbf{x}(k)$ and $d(k)$,

$$\mathbf{T}_0(k)\begin{bmatrix}\mathbf{R}(k-1)\\ \mathbf{x}^T(k)\end{bmatrix}=\begin{bmatrix}\mathbf{R}_0(k)\\ \mathbf{0}^T\end{bmatrix} \tag{5.24}$$

$$\mathbf{T}_0(k)\begin{bmatrix}\mathbf{z}(k-1)\\ d(k)\end{bmatrix}=\begin{bmatrix}\mathbf{z}_0(k)\\ \zeta(k)\end{bmatrix} \tag{5.25}$$

secondly, using $\mathbf{g}_1(k)$ to $\mathbf{g}_n(k)$ we iteratively update until we have $\mathbf{R}_n(k)$,

$$\mathbf{T}_1(k)\begin{bmatrix}\mathbf{R}_0(k)\\ \sqrt{\eta}\mathbf{g}_1(k)\end{bmatrix}=\begin{bmatrix}\mathbf{R}_1(k)\\ \mathbf{0}^T\end{bmatrix} \tag{5.26}$$

$$\vdots$$

$$\mathbf{T}_n(k)\begin{bmatrix}\mathbf{R}_{n-1}(k)\\ \sqrt{\eta}\mathbf{g}_n(k)\end{bmatrix}=\begin{bmatrix}\mathbf{R}_n(k)\\ \mathbf{0}^T\end{bmatrix} \tag{5.27}$$

and using $\mathbf{g}_1(k)\mathbf{q}(k)$ to $\mathbf{g}_n(k)\mathbf{q}(k)$ we iteratively update until we have $\mathbf{z}_n(n)$,

$$\mathbf{T}_1(k)\begin{bmatrix}\mathbf{z}_0(k)\\ \sqrt{\eta}\mathbf{g}_1(k)\mathbf{q}(k)\end{bmatrix}=\begin{bmatrix}\mathbf{z}_1(k)\\ \boldsymbol{\varphi}_1(k)\end{bmatrix} \tag{5.28}$$

$$\vdots$$

$$\mathbf{T}_n(k)\begin{bmatrix}\mathbf{z}_{n-1}(k)\\ \sqrt{\eta}\mathbf{g}_n(k)\mathbf{q}(k)\end{bmatrix}=\begin{bmatrix}\mathbf{z}_n(k)\\ \boldsymbol{\varphi}_n(k)\end{bmatrix} \tag{5.29}$$

It is now clear that we need not continue with the derivation, as (5.26) – (5.29) are in the same form as (5.18) and (5.19). Instead we infer that we can simply perform the final IQR-RLS update algorithm given in [10] for $\mathbf{x}(k)$ and $d(k)$ as is done in ALGORITHM I and then perform the algorithm n more times, but by replacing $\mathbf{x}(k)$ with $\mathbf{g}_1(k)$ to $\mathbf{g}_n(k)$ and $d(k)$ with $\mathbf{g}_1(k)\mathbf{q}(k)$ to $\mathbf{g}_n(k)\mathbf{q}(k)$.

In ALGORITHM II the regularized IQR-RLS algorithm for the CMAC is presented.

A. Optimizations

Running the IQR-RLS algorithm an extra n times would slow the entire algorithm down significantly. However, an important observation to make is that only m rows of $\mathbf{G}(k)$ will *not* be the zero vector. The zero vector rows can be ignored as they would produce a zero $\mathbf{a}(k)$ vector. Thus instead of running the algorithm an extra n times for regularization, it need only be run m more times, which is reflected in the for loop in (5.32).

Another major optimization performed in ALGORITHM II is related to (5.33). Here we realize that we can start loop

ALGORITHM II: REGULARIZED IQR-RLS ALGORITHM FOR THE CMAC

First perform one run of ALGORITHM I, but replace code line (3.14) with (5.30) instead.		
$\mathbf{Temp}(k) = z(k)\mathbf{u}(k)$	$O(n)$	(5.30)
Then while still inside training sample loop (3.2),		
$\eta = 1/\delta$		(5.31)
$for\, h = 1:m$	$O(mn^2)$	(5.32)
$p = activatedAddresses_h(k)$		(5.33)
$\mathbf{a}(k) = \eta\mathbf{R}^{-T}(k)\mathbf{g}_p(k)$		(5.34)
$\mathbf{u}(k) = \mathbf{0},\ \alpha^{(0)}(k) = 1$		(5.35)
$for\, i = p:n$		(5.36)
$givens()$		(5.37)
$\mathbf{Temp}(k) += \frac{\eta\left[\mathbf{g}_p(k)\mathbf{q}(k) - \mathbf{g}_p(k)\mathbf{w}(k-1)\right]\mathbf{u}(k)}{\alpha^{(n)}(k)}$		(5.38)
$\mathbf{w}(k) = \mathbf{w}(k-1) - \mathbf{Temp}(k)$		(5.39)

(5.36) from address p of the $h'th$ '1' in the association vector. This is because $\mathbf{g}_p(k)$ is essentially the association vector with every entry, other than the $p'th$ entry masked as zero, therefore every $a_i(k)$ value before the $p'th$ address will be zero making performing the givens macro redundant, as was explained in section III.A.

B. Results

It was found that the regularized RLS algorithm is able to compute the regularized weight vector in one epoch. In Fig 5 we see the output of a non-regularized CMAC on the left, modeling a sine function with the IQR-RLS algorithm. The CMAC sampled the sine function every 30 degrees, used a quantization resolution of 100, and had 10 layers. There is severe interpolation/generalization error between training samples. The figure on the right shows the CMAC trained on the same sine wave, but with regularization turned on. The CMAC output is now almost a perfect sine wave. The improvement in total absolute error (TAE) is shown in TABLE I. The error was also tested on a 2D sinc plot in which the CMAC was trained with three times less points than it was tested with in order to test generalization. With regularization, training times increase. Fig 4 shows a computation time comparison for the regularized IQR-RLS algorithm.

TABLE I: TAE OF REGULARIZED AND NON-REGULARIZED CMAC-IQRRLS

	Non-Regularized	Regularized
1D Sine Plot	51.8	19.5
2D Sinc Plot	400	206

VI. INTRODUCTION TO THE 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 binary 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 size of a dictionary (which stores previously admitted feature vectors) where the maximum size is the number of unique training data presented

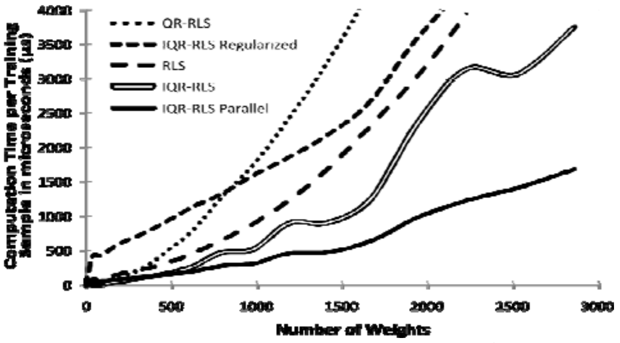

Fig 4. Computation time per training sample vs. number of weights in the CMAC for various RLS implementations. The number of weights was controlled by altering the quantization resolution.

to the algorithm rather than n, 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. The size of the dictionary is denoted c, and is what controls computational complexity.

VII. THE KERNEL RLS ALGORITHM FOR THE CMAC

Although it was shown how the IQR-RLS algorithm can be used to speed up CMAC-RLS, it is still not fast enough for use on high dimensional problems on a PC. Here we use a kernel-RLS (KRLS) algorithm which allows the CMAC-RLS to be used for higher dimensional problems. 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, and computational complexity will no longer be dependent on the result of (2.4). With sparsification techniques the number of training data required can be reduced even further. Here we quote the KRLS algorithm from [7] with slight alterations to specialize it for the CMAC.

ALGORITHM III features an online sparsification technique that sparsifies by preventing feature vectors that are approximately linearly dependent on the dictionary, $\mathbf{X}$ 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 (6.8) the scalar value δ is calculated which is a measure of how linearly dependent $\mathbf{x}$ is on the dictionary $\mathbf{X}$. If δ is greater than some threshold v, $\mathbf{x}$ will be added to the dictionary as this means that it was not approximately linearly dependent on the dictionary. Otherwise, if the threshold is not met, the update equations (6.16) – (6.18) (shaded) will be used instead. The elements of vector $\mathbf{a}$ represent a weighting on how linearly dependent a vector in the dictionary is to the current feature vector. If the current feature vector is already in the

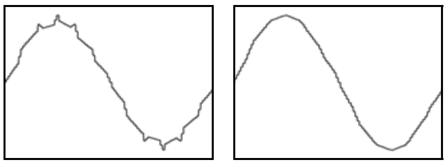

Fig 5. Non-regularized CMAC output (left) and regularized CMAC output (right).

ALGORITHM III: CMAC-KRLS

$\mathbf{K}^{-1}=[1/m], \boldsymbol{\beta}=[d_1/m],$ $\mathbf{X}=\varphi(quant(\mathbf{y}_1)), \mathbf{P}=[1], c=1$		(6.1)
for $t=2,3...n_d$		(6.2)
Get new sample: $(\mathbf{y}_t, d_t)$		(6.3)
Quantize sample: $\mathbf{q}=quant(\mathbf{y}_t)$	$O(n_y)$	(6.4)
Calculate association vector: $\mathbf{x}=\varphi(\mathbf{q})$	$O(mn_y)$	(6.5)
$\mathbf{k}=\mathbf{X}\mathbf{x}$	$O(cm)$	(6.6)
$\mathbf{a}=\mathbf{K}^{-1}\mathbf{k}$	$O(c^2)$	(6.7)
$\delta=m-\mathbf{k}^T\mathbf{a}$	$O(c)$	(6.8)
if $\delta>v$ $\mathbf{X}=\begin{bmatrix}\mathbf{X} & \mathbf{x}^T\end{bmatrix}^T$	$O(1)$	(6.9) (6.10)
$\mathbf{K}_{new}^{-1}=\frac{1}{\delta}\begin{bmatrix}\delta\mathbf{K}_{old}^{-1}+\mathbf{a}\mathbf{a}^T & -\mathbf{a}\\ -\mathbf{a}^T & 1\end{bmatrix}$	$O(c^2)$	(6.11)
$\mathbf{P}_{new}=\begin{bmatrix}\mathbf{P}_{old} & 0\\ 0 & 1\end{bmatrix}$ $\boldsymbol{\beta}_{new}=\frac{1}{\delta}\begin{bmatrix}\delta\boldsymbol{\beta}_{old}-\mathbf{a}\left(d_t-\mathbf{k}^T\boldsymbol{\beta}_{old}\right)\\ d_t-\mathbf{k}^T\boldsymbol{\beta}_{old}\end{bmatrix}$	$O(c)$	(6.12) (6.13)
$c=c+1$		(6.14)
else		(6.15)
$\mathbf{q}=\frac{\mathbf{P}_{old}\mathbf{a}}{1+\mathbf{a}^T\mathbf{P}_{old}\mathbf{a}}$	$O(c^2)$	(6.16)
$\mathbf{P}_{new}=\mathbf{P}_{old}-\mathbf{q}\mathbf{a}^T\mathbf{P}_{old}$		(6.17)
$\boldsymbol{\beta}_{new}=\boldsymbol{\beta}_{old}+\mathbf{K}^{-1}\mathbf{q}\left(d_t-\mathbf{k}^T\boldsymbol{\beta}_{old}\right)$		(6.18)

dictionary, the entries of vector **a** will be all zero except for a single unity entry at the index of the matching dictionary point.

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.

VIII. AN OPTIMIZED KRLS ALGORITHM FOR THE CMAC

The CMAC-KRLS algorithm is still fairly computationally complex, with major bottlenecks at (6.6), (6.7), (6.11), and (6.16) - (6.18). Fortunately, most of these bottlenecks can be reduced by some optimizations presented below. The optimized discarding CMAC-KRLS algorithm is presented as ALGORITHM IV.

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 $n_y=4$, then $m=20^4=160\,000$. This causes a computational burden as the calculation of the kernel vector given by [5] is $\mathbf{k}=\mathbf{X}\mathbf{x}$. 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.

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, $\mathbf{k}_i$, are actually the number of shared hypercubes between dictionary point $\mathbf{Q}_i$ (where the dictionary $\mathbf{Q}$ stores quantized input vectors instead of association vectors), and current quantized input $\mathbf{q}$, we can reduce the number of

ALGORITHM IV: OPTIMIZED DISCARDING CMAC-KRLS

$\mathbf{K}^{-1}=[1/m], \boldsymbol{\beta}=[d_1/m], \mathbf{Q}=quant(\mathbf{y}_1),$ $\mathbf{P}=[1], 0.98\le\lambda\le1, c=1,$		(7.1)
for $t=2,3...n_d$		(7.2)
Get new sample: $(\mathbf{y}_t, d_t)$		(7.3)
Quantize sample: $\mathbf{q}=quant(\mathbf{y}_t)$	$O(n_y)$	(7.4)
for $i=1:c$		(7.5)
$\mathbf{z}=\lvert\mathbf{Q}_i-\mathbf{q}\rvert$	$O(n_yc)$	(7.6)
$\mathbf{k}_i=\prod_{j=1}^{n_y}\max\left[h-\mathbf{z}_j,0\right]$	$O(n_yc)$	(7.7)
if $(\mathbf{Q}.contains(\mathbf{q}))$ $b=\mathbf{Q}.indexof(\mathbf{q})$ $q=\frac{\mathbf{P}_{b,b}}{\lambda+\mathbf{P}_{b,b}}$ $\mathbf{P}_{b,b}=\lambda^{-1}\left(\mathbf{P}_{b,b}-q\mathbf{P}_{b,b}\right)$	$O(1)$	(7.8) (7.9) (7.10) (7.11)
$\boldsymbol{\beta}_{new}=\boldsymbol{\beta}_{old}+\mathbf{K}_{:,b}^{-1}q\left(d_t-\mathbf{k}^T\boldsymbol{\beta}_{old}\right)$	$O(c)$	(7.12)
elseif $(!rejectDict.Contains(\mathbf{q}))$	$O(1)$	(7.13)
$\mathbf{a}=\mathbf{K}^{-1}\mathbf{k}$	$O(c^2)$	(7.14)
$\delta=m-\mathbf{k}^T\mathbf{a}$	$O(c)$	(7.15)
if $\delta>v$ $\mathbf{Q}=\begin{bmatrix}\mathbf{Q} & \mathbf{q}^T\end{bmatrix}^T$	$O(1)$	(7.16) (7.17)
$\mathbf{K}_{new}^{-1}=\frac{1}{\delta}\begin{bmatrix}\delta\mathbf{K}_{old}^{-1}+\mathbf{a}\mathbf{a}^T & -\mathbf{a}\\ -\mathbf{a}^T & 1\end{bmatrix}$	$O(c^2)$	(7.18)
$\mathbf{P}_{new}=\begin{bmatrix}\mathbf{P}_{old} & 0\\ 0 & 1\end{bmatrix}$	$O(1)$	(7.19)
$\boldsymbol{\beta}_{new}=\frac{1}{\delta}\begin{bmatrix}\delta\boldsymbol{\beta}_{old}-\mathbf{a}\left(d_t-\mathbf{k}^T\boldsymbol{\beta}_{old}\right)\\ d_t-\mathbf{k}^T\boldsymbol{\beta}_{old}\end{bmatrix}$	$O(c)$	(7.20)
$c=c+1$		(7.21)
else		(7.22)
$rejectDict=\begin{bmatrix}rejectDict\\ \mathbf{q}^T\end{bmatrix}$	$O(1)$	(7.23)
NOTE: $\mathbf{K}_{:,b}^{-1}$ indicates the b'th column of $\mathbf{K}^{-1}$		

calculations required to calculate the kernel vector to $n_y\times c$. In Fig 6 a snapshot of the kernel function generated by equation (7.7) or equivalently (6.6) is shown for a single input CMAC ($n_y=1$) where $h=3$. In reality, this function extends continuously from *min* to *max*. Calculating the kernel function for a higher dimensional CMAC is simple. Simply multiply each kernel value obtained in each dimension together. To visualize this further in Fig 7 we see a 2D CMAC full overlay, where $h=3$, and thus $m=9$. We can view this figure as having the dictionary point $\mathbf{Q}_i$ at the center, and the numbers in the surrounding grid squares give the number of shared hypercubes for nearby possible values of **q**. Equation (7.7) can be used to calculate number of overlaps $\mathbf{k}_i$ for a particular quantized dictionary point $\mathbf{Q}_i$ and the current quantized input $\mathbf{q}$. As an example say $\mathbf{Q}_i=[3\quad 4]$ and $\mathbf{q}=[3\quad 2]$. In dimension $j=1$, $\mathbf{Q}_{i,1}=3$ and $\mathbf{q}_1=3$, and in dimension two we have $\mathbf{Q}_{i,2}=4$ and $\mathbf{q}_2=2$. Thus, $\mathbf{k}_i=\left(3-\lvert 3-3\rvert\right)\times\left(3-\lvert 4-2\rvert\right)=3\times1=3$. We can confirm by using Fig 7 where the center point of this snapshot is [3 4]. We can then look up point [3 2] and see that it gives $\mathbf{k}_i=3$.

There is no need to evaluate the association vector using

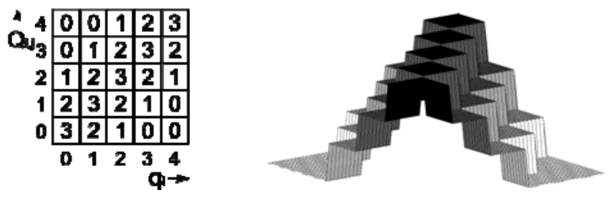

Fig 6: Kernel function where $h = 3$. Topological view (left) and profile view (right).

this method, since $\mathbf{k}$ is now directly a function of the quantized input $\mathbf{q}$ rather than association vector $\mathbf{x}$.

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 III a sparsification technique was used which only added data that was not already approximately linearly dependent on the dictionary. However, it still made use of every training point to adjust the weights, even if it was not admitted to 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 (6.16) – (6.18) we see that the $\mathbf{a}$ vector is always all zero except for a single unity entry at the index where the matching dictionary entry is stored and thus the $\mathbf{P}$ matrix remains diagonal. So if the dictionary index for input $\mathbf{q}$ is known to be b, we only need to update scalars $\mathbf{q}_b$ (which is simply notated as q in ALGORITHM IV) and $\mathbf{P}_{b,b}$. Thus, equations (6.16) to (6.18) can be simplified significantly as can be seen in equations (7.9) – (7.12). 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 revisited will the CMAC update. If the CMAC must be used in a non-stationary or noisy environment, the non discarding ALGORITHM III can be used, the sparsification threshold can be reduced, or the semi-discarding algorithm presented next in section 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 (7.8) 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 dependent on the dictionary. This removes the need to compute the approximate linear dependence test when seeing previously rejected points and is reflected by equations (7.13) and (7.23).

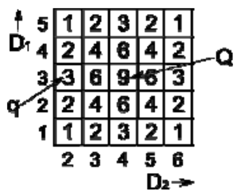

Fig 7. Kernel vector values for a 2D CMAC with $\mathbf{Q}$ at the center, and nearby possible $\mathbf{q}$. Arrows point to example values mentioned in section VIII.A.

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 can be used. With the semi-discarding method, in the update section of ALGORITHM III (shaded) every value in the $\mathbf{a}$ 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 $\mathbf{a}$ vector. This keeps the $\mathbf{P}$ matrix diagonal – the reason for fast computational performance. Modifications to get the semi-discarding algorithm are shown in ALGORITHM V.

D. Forgetting Factor

A forgetting factor is typically used to allow RLS algorithms to track better in non-stationary environments. The forgetting factor λ has been integrated into the update equations (7.10) and (7.11) in ALGORITHM IV, and also in (7.25) and (7.26) in ALGORITHM V. 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 (7.14) can be sped up significantly by performing sparse vector matrix multiplication. Also, since the $\mathbf{P}$ matrix remains diagonal in the discarding and semi-discarding algorithms, it can be stored as a vector. Thus expanding $\mathbf{P}$ in (7.19) becomes an $O(1)$ operation.

IX. HIGHER ORDER BASIS FUNCTIONS FOR THE CMAC-KRLS

The standard CMAC, and also the CMAC-KRLS shown so far uses a quantized binary kernel function. This unfortunately produces a staircase like output. This can be resolved by making the CMAC resolution as high as possible. In the standard CMAC there is a trade-off as increasing the resolution causes an increase in the number of weights required. However, in the CMAC-KRLS increasing resolution has no adverse affects as the number of weights required does not increase. Only the generalization parameter

ALGORITHM V: SEMI-DISCARDING CMAC-KRLS

Same as ALGORITHM IV but, Replace (7.13) with an else statement Replace (7.23) with four new lines:		
$b = \mathbf{a}.index\left(\max\lvert\mathbf{a}\rvert\right)$	$O(c)$	(7.24)
$q = \dfrac{\mathbf{P}_{b,b}\mathbf{a}_b}{\lambda + \left(\mathbf{P}_{b,b}\mathbf{a}_b^2\right)}$	$O(1)$	(7.25)
$\mathbf{P}_{b,b} = \lambda^{-1}\left(\mathbf{P}_{b,b} - q\mathbf{P}_{b,b}\mathbf{a}_b\right)$	$O(1)$	(7.26)
$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} + \mathbf{K}_{:,b}^{-1}q\left(d_t - \mathbf{k}^T\boldsymbol{\beta}_{old}\right)$	$O(c)$	(7.27)

h needs to be made larger to compensate, if the resolution doubles, the generalization parameter needs to double too. When increasing the resolution, in the limit, the kernel function becomes triangular in shape as is seen in Fig 8, rather than staircase like, as was shown in Fig 6. We can obtain this kernel function either by increasing the resolution to a very large value or directly by simply by removing the floor operation from (2.2) and storing the non-floored values in the dictionary. In the latter case only the ratio between the resolution and generalization parameter *h* becomes important in tuning the generalization of the CMAC.

We can further improve the modeling of the CMAC-KRLS by using a b-spline curve as the kernel function. This is easily performed by using the equation found in [13]

$$\mathbf{k}_i = \prod_{j=1}^{n_y} \left(\frac{1}{n!} \sum_{\phi=1}^{n+1} \binom{n+1}{\phi} (-1)^{\phi} (\delta x + \frac{n+1}{2} - \phi)_+^n \right) \qquad (8.1)$$

where

$$x_+^d = \begin{cases} x^d, & if \ x > 0 \\ 0, & otherwise \end{cases} \qquad (8.2)$$

and where the order *n* for the b-spline is given by

$$n = 2o + 1 \qquad (8.3)$$

where *o* is the chosen spline order and where δ is the dilation constant which is used to control the width of the spline. The dilation constant plays the same role as *m* when used in the CMAC. A profile example of a b-spline kernel function where $o = 1$ is shown in Fig 9.

X. CMAC-KRLS 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.

Algorithms that rely on the kernel trick use as many weights as there are unique training points added to the dictionary. This number can be controlled if sparsification methods are used. In Fig 10 the number of weights used to learn the 2D sinc function for any sparsifying CMAC-KRLS algorithm is plotted against different sparsification thresholds. A total of 1681 unique training points were presented sequentially. The number of weights used by a CMAC-IQR-RLS algorithm is also plotted for diagonal and uniform overlays which are similar. The full overlay cannot be used in CMAC-IQR-RLS 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-IQR-RLS would be much

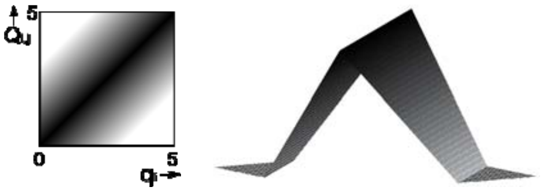

Fig 8. Linear kernel function where h = 3. Topological view (left) and profile view (right).

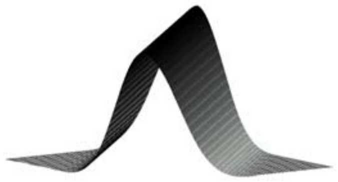
Fig 9: B-Spline Kernel Function where $o = 1$

larger even with only the diagonal or uniform overlays.

In Fig 11 the average time per iteration 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. Fig 11 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-IQR-RLS algorithm is shown for the same problem. It should be noted that although the CMAC-IQR-RLS algorithm is competitive with the non-discarding algorithm here, in a higher dimensional problem its use would be infeasible whereas the CMAC-KRLS would perform at a similar speed no matter the dimension given the same number of unique training data.

The total absolute error for modeling a noise free two input sinc function was measured for each CMAC-KRLS variant with full overlay and recorded in Fig 12. 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 ten 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 higher in error than the non-discarding algorithm. For comparison the errors from the CMAC-IQR-RLS algorithm with the diagonal and uniform overlays are shown. In Fig 13 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

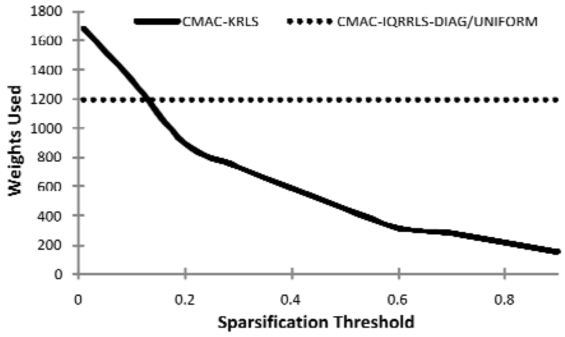

Fig 10. Number of weights used for any 2D CMAC-KRLS for different sparsification thresholds compared against the 2D CMAC-IQR-RLS.

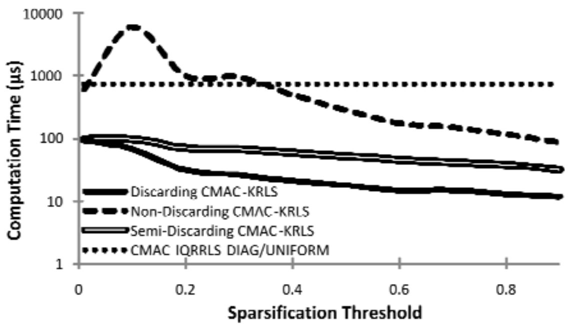

Fig 11. Average time taken per iteration over ten training epochs.

algorithm has improved performance over the discarding algorithm due to its ability to make some use of the discarded data.

In Table II the total absolute error (TAE) results of the different basis functions are shown. The sine wave was sampled every 30 degrees, while the sinc wave was sampled with three times less points that it was tested with. The generalization parameter was varied to get the lowest TAE. The binary basis function used a resolution of 100. The results show that the b-spline produces better results than the linear and binary basis functions. The results for the spline basis function are better as its smoothness better approximates the smooth curves of the sine and sinc waves. It was found that using these higher order basis functions introduces almost no noticeable increase in computation time.

XI. Conclusions

In this paper two methods for incorporating RLS into the CMAC neural network were shown. The first method used the IQR-RLS algorithm to simplify and parallelize computation resulting in much faster computation times. Additionally, the IQR-RLS algorithm was able to be easily regularized resulting in greatly improved CMAC generalization, but at the expense of computation time. Although IQR-RLS was shown to be much faster than standard CMAC-RLS it was still not fast enough for CMACs with greater than two inputs. In order to overcome this problem the KRLS algorithm was introduced which transformed the computational complexity to become dependent on the number of training data, rather than the number of weights required by the CMAC. The results show that the CMAC-KRLS is significantly faster than other CMAC-RLS algorithms, and can in fact model better as it can use the full overlay of basis functions. Additionally, it was shown that higher order basis functions can easily be implemented into the CMAC-KRLS with little to no increase in computational complexity. However, although the CMAC-KRLS is faster than the CMAC-IQR-RLS, IQR-RLS has an implementation that is highly suitable to parallel hardware, and may be a better choice for hardware implementation, or highly parallel CPUs.

TABLE II: TAE of Higher Order Basis Functions

	Binary	Linear	Spline $o = 1$
1D Sine Wave	16.3	10.51	3.97
2D Sinc Wave	277	257	127

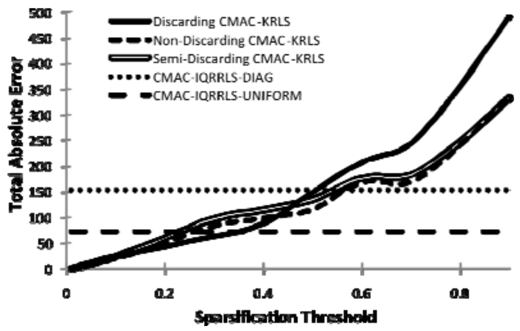

Fig 12. Total absolute error for different sparsification thresholds

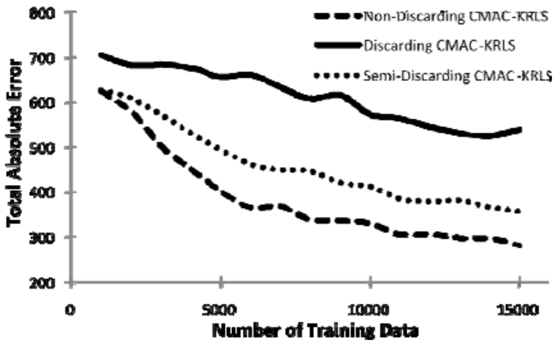

Fig 13. Comparison with noisy random data between each CMAC-KRLS under a sparsification threshold of 0.2.

XII. Future Work

Currently there is no regularization method for the CMAC-KRLS, however, this research is currently being undertaken. Additionally, other improvements such as CMAC eligibility [11] will be implemented into the CMAC-KRLS to achieve improved performance in motion control learning situations. Also, more investigation needs to be undertaken to learn more about the error the semi-discarding method introduces.

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 Ser G, pp. 220-227, 1975.

[2] M. K. Hiroaki 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] 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.

[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] J. A. Apolinário and M. D. Miranda, "Conventional and Inverse QRD-RLS Algorithms," in QRD-RLS Adaptive Filtering, J. A. Apolinário, Ed.: Springer US, 2009, pp. 1-35.

[10] S. T. Alexander and A. L. Ghirnikar, "Method for recursive least squares filtering based upon an inverse QR decomposition," IEEE Transactions on Signal Processing, vol. 41, pp. 20-30, 1993.

[11] R. L. Smith, "Intelligent Motion Control with an Artificial Cerebellum," Doctorate, Electrical and Electronic Engineering, University of Auckland, Auckland, 1998.

[12] J. Pallotta and L. G. Kraft, "Two dimensional function learning using CMAC neural network with optimized weight smoothing," in Proceedings of the American Control Conference, San Diego, CA, USA, 1999, pp. 373-377.

[13] S. Fomel, "Inverse B-spline interpolation," 2000.