A Regularized Inverse QR Decomposition Based Recursive Least Squares Algorithm 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 (LMS) 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 one epoch, and does not require tuning of a learning rate. However, the RLS algorithm was found to be very computationally demanding. In this work, the RLS computation time is reduced by using an inverse QR decomposition based RLS (IQR-RLS) algorithm which is also parallelized for multi-core CPUs. Furthermore, this work shows how the IQR-RLS algorithm may be regularized which greatly improves the generalization capabilities of the CMAC.**

***Index Terms*—CMAC, inverse QR-RLS, regularization, 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, 3].

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, the recursive least squares (RLS) algorithm was proposed for use in the CMAC [2]. The RLS algorithm does not require tuning of a learning rate, and will converge in just one epoch. This is especially advantageous for online learning used in methods such as feedback error learning [3]. 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 in the CMAC. While fast $O(n_w)$ RLS algorithms exist, they are only suitable for input vectors which exhibit a time-shifting property [4]. This property does not exist in the CMAC. However, it is shown in [5] 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 $O(n_w^2)$, the new algorithm is fast enough to solve small problems at a reasonable speed.

Secondly, it is well known that the standard CMAC can have significant generalization error [6-9]. In [9] a regularization term was applied to the LMS cost function which reduced the generalization error. In this paper we apply the regularization term to the RLS cost function, and derive a new IQR-RLS algorithm which computes a regularized weight vector in one epoch.

This paper is organized as follows. In Section II a brief introduction to the CMAC is presented. In Section III, the IQR-RLS algorithm is explained, and a special algorithm tailored for the CMAC is presented. Section III presents a method for parallelizing the IQR-RLS algorithm for multi-core CPUs. In Section V the regularized IQR-RLS algorithm is derived, and the algorithm and results presented. Finally, in section VI the conclusions are presented.

II. Brief Overview of the 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 a n_d -dimensional input vector $\mathbf{y} = [y_1 \quad y_2 \quad \cdots \quad y_{n_d}]^T$ where $y_i \in \mathbb{R}$ to a quantized vector $\mathbf{q} = [q_1 \quad q_2 \quad \cdots \quad q_{n_d}]^T$ where $q_i \in \mathbb{Z}$.

The mapping $M \rightarrow A$ is a non-linear recoding from vector $\mathbf{q}$ into a higher dimensional binary vector called the association vector, $\mathbf{x} = [x_1 \quad x_2 \quad \cdots \quad x_{n_w}]^T$ 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 $\mathbf{x}$ will only contain m '1's, where m is the number of layers in the CMAC (a selectable parameter which controls generalization).

In the mapping $A \rightarrow P$ the association vector is used to select and add m values from an array of weights $\mathbf{w} = [w_1 \quad w_2 \quad \cdots \quad w_{n_w}]^T$ where $w_i \in \mathbb{R}$ 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 (2.1) is used for this purpose, where k is the training sample iteration, β is the learning rate, $d(k)$ is the desired output, and $\mathbf{x}^T(k)\mathbf{w}(k-1)$ is the actual CMAC output.

Manuscript received May 29, 2012; revised July 27, 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).

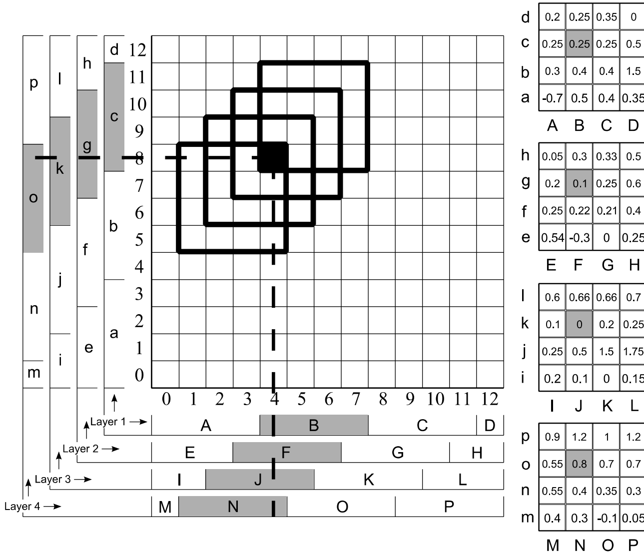

Fig. 1. A two-input CMAC example with four layers and 64 weights.

$$\mathbf{w}(k) = \mathbf{w}(k-1) + \frac{\beta}{m}\mathbf{x}^T(k)\left(d(k) - \mathbf{x}^T(k)\mathbf{w}(k-1)\right) \quad (2.1)$$

In Fig. 1 a visualization of a two input ($n_d = 2$) CMAC is shown with quantized input $\mathbf{q} = [4 \quad 8]^T$, and $n_w = 64$. Here $m = 4$ 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. Here weights Bc, Fg, Jk and No are activated. If put into activation vector form it will appear as,

$$\mathbf{x} = \begin{bmatrix} Aa & Ba & \cdots & Bc & \cdots & Fg & \cdots & Jk & \cdots & No & \cdots & Pp \\ 0 & 0 & 0\cdots0 & 1 & 0\cdots0 & 1 & 0\cdots0 & 1 & 0\cdots0 & 1 & 0 & 0 \end{bmatrix}^T \quad (2.2)$$

We can also sparsely store this vector by simply storing the addresses of the activated weights,

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

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 [10] in a more numerically stable manner compared with standard RLS. Usually, using QR methods will degrade computational performance. However, the paper in [5] tailors the QR-RLS algorithm specifically for the CMAC resulting in halving the computation time. Unfortunately, the tailored 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 multi-input CMACs.

If the weight vector is required to be updated after every presented training sample, as it is required in the CMAC, a costly matrix back substitution step of $O(n_w^2)$ time complexity needs to be carried out each time. We can avoid this back substitution step entirely by using an *inverse* QR-RLS algorithm, which instead allows the weights to be calculated directly. In Algorithm I we present an IQR-RLS algorithm that was derived in [11] which 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 (3.5). 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 will have been calculated previously as part of the CMAC addressing algorithm which is not shown here but can be found in [6]. Here the address where the first '1' appears in the association vector is recorded. The for loop in (3.6) then begins its computation from this address. This is because the $\mathbf{a}(k)$ vector calculated in (3.3) 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

$$\mathbf{w}(0) = \mathbf{0},\ \mathbf{x}(0) = \mathbf{0},\ \mathbf{R}^{-T}(0) = \delta\mathbf{I}_{n_w \times n_w}\ (\delta >> 1),\ \rho < 0.001 \quad (3.1)$$

for each training sample k : (3.2)

$$\mathbf{a}(k) = \mathbf{R}^{-T}(k-1)\mathbf{x}(k) \quad (3.3)$$

$$\mathbf{u}(k) = \mathbf{0},\ \alpha^{(0)}(k) = 1 \quad (3.4)$$

$$start = activatedAddresses_1(k) \quad (3.5)$$

$$for\ i = start : n_w \quad (3.6)$$

$$givens() \quad (3.7)$$

$$e(k) = d(k) - \mathbf{x}^T(k)\mathbf{w}(k-1) \quad (3.8)$$

$$z(k) = \frac{e(k)}{\alpha^{(n_w)}(k)} \quad (3.9)$$

$$\mathbf{w}(k) = \mathbf{w}(k-1) - z(k)\mathbf{u}(k) \quad (3.10)$$

Macro: *givens*()

$$if\ (|a_i(k)|) > \rho) \quad (3.11)$$

$$\alpha^{(i)}(k) = \sqrt{\left[\alpha^{(i-1)}(k)\right]^2 + a_i^2(k)} \quad (3.12)$$

$$s(k) = \frac{-a_i(k)}{\alpha^{(i)}(k)} \quad (3.13)$$

$$c(k) = \frac{\alpha^{(i-1)}(k)}{\alpha^{(i)}(k)} \quad (3.14)$$

$$for\ j = 1 : i \quad (3.15)$$

$$r_{ij}(k) = c(k)r_{ij}(k-1) - s(k)u_j^{(i-1)}(k-1) \quad (3.16)$$

$$u_j^{(i)}(k) = c(k)u_j^{(i-1)}(k) + s(k)r_{ij}(k-1) \quad (3.17)$$

The second improvement follows on from this where the calculation of (3.12) - (3.17) is gated by (3.11), and thus is 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 (3.3), values are often added and subtracted to form the sum of zero. Due to floating point inaccuracies the result will not equal exactly zero, hence the threshold. 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 (3.3) 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 (3.6) that computation time will 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. 3 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 [5] as they would restrict the CMAC to a single input only. The QR-RLS algorithm was then many times slower than standard RLS.

Compared with the LMS algorithm which requires less than one microsecond per iteration, RLS is 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 [5]. 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. 2 In this figure, each circle represents a processing element that also stores the values of $\mathbf{R}^{-T}$. The circles in the left most column implement (3.12) - (3.17), and all other circular processing elements implement (3.16) and (3.17). The square boxes calculate the weights using (3.8) - (3.10).

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 $\alpha^{(n_w)}$ 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, one for each core on the CPU. We can further optimize by combining computation of shorter columns together to equalize thread computation times and by writing code to skip any 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 (3.16) and (3.17).

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. 3. The parallel algorithm implements both threading and SSE based parallelization.

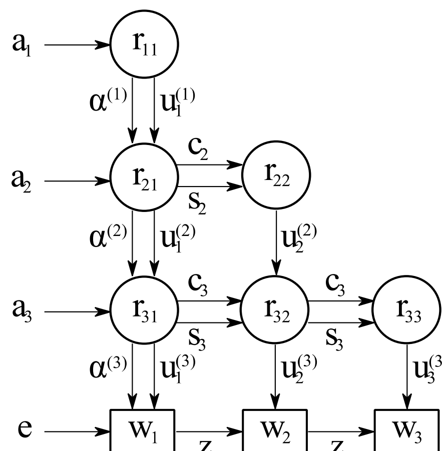

Fig. 2. Parallel systolic array implementation of the IQR-RLS algorithm

V. REGULARIZED IQR-RLS

The work in [6-9] shows that the generalization error of the CMAC can be significant. In [9] 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 [11]. First from [9] we use the least squares cost function $\varepsilon(k)$ where k is the number of training samples as,

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

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 λ 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 solve this problem, (5.1) 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,

$$\mathbf{w}(k)=\left[w_1(k)\quad w_2(k)\quad\cdots\quad w_{n_w}(k)\right]^T_{n_w\times 1} \tag{5.2}$$

$$\mathbf{d}(k)=\left[d(1)\quad d(2)\quad\cdots\quad d(k)\right]^T_{k\times 1} \tag{5.3}$$

$$\mathbf{h}(k)=\left[\mathbf{G}(1)\mathbf{q}(1)\quad \mathbf{G}(2)\mathbf{q}(2)\quad\cdots\quad \mathbf{G}(k)\mathbf{q}(k)\right]^T_{n_wk\times 1} \tag{5.4}$$

$$\mathbf{X}(k)=\left[\mathbf{x}(1)\quad \mathbf{x}(2)\quad\cdots\quad \mathbf{x}(k)\right]^T_{k\times n_w} \tag{5.5}$$

$$\mathbf{\Sigma}(k)=\left[\mathbf{G}(1)\quad \mathbf{G}(2)\quad\cdots\quad \mathbf{G}(k)\right]^T_{n_wk\times n_w} \tag{5.6}$$

where,

$$\mathbf{q}(k)=\left[\frac{d(k)}{m}\quad\frac{d(k)}{m}\quad\cdots\quad\frac{d(k)}{m}\right]^T_{n_w\times 1} \tag{5.7}$$

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

$diag\left(\mathbf{x}(k)\right)$ creates an $n_w\times n_w$ 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,

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

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

$$\mathbf{A}(k)=\begin{bmatrix}\mathbf{X}(k)\\ \sqrt{\lambda}\mathbf{\Sigma}(k)\end{bmatrix}_{(k+n_wk)\times n_w} \tag{5.10}$$

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

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 [11] we see that since $\mathbf{A}(k)$ is $(k+n_wk)\times n_w$, there exists a $(k+n_wk)\times(k+n_wk)$ 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 $n_w\times n_w$ upper triangular Cholesky factor, and $\mathbf{0}$ is an $((k+n_wk)-n_w)\times n_w$ 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 $n_w\times 1$ vector, and $\mathbf{v}(k)$ is a $((k+n_wk)-n_w)\times 1$ vector. Since $\mathbf{Q}(k)$ is orthogonal, pre-multiplying each term in (5.12) does not change the value of the norm,

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

Substituting (5.13) and (5.14) into (5.15) gives 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 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. For IQR-RLS we need the $\mathbf{R}^{-1}(k)$ matrix however, so the derivation continues. 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 $\mathbf{d}(k)$. Thus, in [11] it is shown that an $(n_w+1)\times(n_w+1)$ 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 $\sqrt{\lambda}$ which are $\sqrt{\lambda}\mathbf{G}(k)$ and $\sqrt{\lambda}\mathbf{G}(k)\mathbf{q}(k)$ respectively. There must then exist an $(2n_w+1)\times(2n_w+1)$ matrix $\mathbf{T}(k)$ such that,

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

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

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

Unfortunately, if (5.20) and (5.21) are used, we cannot proceed with the same derivations found in [11] as the derivations are suited only to a $(n_w+1)\times(n_w+1)$ $\mathbf{T}(k)$ matrix. We, however, realize that $\mathbf{R}(k)$ may be calculated iteratively if we define $\mathbf{R}_{n_w}(k)\equiv\mathbf{R}(k)$, $\mathbf{z}_{n_w}(k)\equiv\mathbf{z}(k)$ and write $\mathbf{G}(k)$ and $\mathbf{q}(k)$ in column form,

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

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

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

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

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

and using $\mathbf{g}_1(k)$ to $\mathbf{g}_{n_w}(k)$ we iteratively update until we have $\mathbf{R}_{n_w}(k)$,

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

$$\vdots$$

$$\mathbf{T}_{n_w}(k)\begin{bmatrix}\mathbf{R}_{n_w-1}(k)\\ \sqrt{\lambda}\mathbf{g}_{n_w}(k)\end{bmatrix}=\begin{bmatrix}\mathbf{R}_{n_w}(k)\\ \mathbf{0}^T\end{bmatrix} \quad (5.27)$$

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

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

$$\vdots$$

$$\mathbf{T}_{n_w}(k)\begin{bmatrix}\mathbf{z}_{n_w-1}(k)\\ \sqrt{\lambda}\mathbf{g}_{n_w}(k)\mathbf{q}(k)\end{bmatrix}=\begin{bmatrix}\mathbf{z}_{n_w}(k)\\ \boldsymbol{\varphi}_{n_w}(k)\end{bmatrix} \quad (5.29)$$

It is now clear that we need not continue with the derivation, as the regularizing equations (5.26) - (5.29) are in the same form as (5.18) and (5.19). Instead we can simply perform the final IQR-RLS update algorithm given in [11] for $\mathbf{x}(k)$ and $d(k)$ as is done in ALGORITHM I and then perform the algorithm n_w more times, but by replacing $\mathbf{x}(k)$ with $\mathbf{g}_1(k)$ to $\mathbf{g}_{n_w}(k)$ and $d(k)$ with $\mathbf{g}_1(k)\mathbf{q}(k)$ to $\mathbf{g}_{n_w}(k)\mathbf{q}(k)$.

ALGORITHM II: REGULARIZED IQR-RLS ALGORITHM FOR THE CMAC

First perform one run of ALGORITHM I, but replace line (3.10) with (5.30) instead.

$\mathbf{Temp}(k)=z(k)\mathbf{u}(k)$ (5.30)

Then while still inside training sample loop (3.2),

$\lambda=1/\delta$ (5.31)

for $h=1:m$ (5.32)

  $p=activatedAddresses_h(k)$ (5.33)

  $\mathbf{a}(k)=\lambda\mathbf{R}^{-T}(k)\mathbf{g}_p(k)$ (5.34)

  $\mathbf{u}(k)=\mathbf{0},\ \alpha^{(0)}(k)=1$ (5.35)

  for $i=p:n_w$ (5.36)

    $givens()$ (5.37)

  $\mathbf{Temp}(k)+=\dfrac{\lambda\left[\mathbf{g}_p(k)\mathbf{q}(k)-\mathbf{g}_p(k)\mathbf{w}(k-1)\right]\mathbf{u}(k)}{\alpha^{(n_w)}(k)}$ (5.38)

$\mathbf{w}(k)=\mathbf{w}(k-1)-\mathbf{Temp}(k)$ (5.39)

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

A. Optimizations

Running the IQR-RLS algorithm n_w times for each row 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 n_w times more 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 (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. 4 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.

With regularization, training times take a hit. Fig. 3 shows a computation time comparison for the regularized IQR-RLS

algorithm.

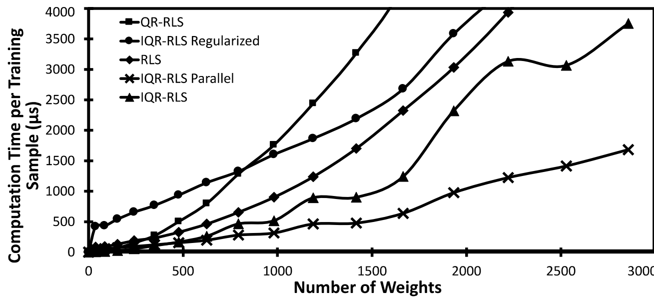

Fig. 3. Computation time per training sample vs. number of weights in the CMAC for various RLS-CMAC implementations. The number of weights was controlled by altering the quantization resolution.

Fig. 4. Non-regularized CMAC output (left) and regularized CMAC output (right) for a sine wave modelling test.

VI. Conclusion

In this paper it was shown that the IQR-RLS algorithm is a superior choice over QR-RLS and standard RLS for training the CMAC neural network in one epoch. It was shown how the IQR-RLS algorithm can be optimized for use in the CMAC, and also how it can be parallelized for multicore CPUs through multithreading, and through the use of SSE instructions. The final experimental results show that the algorithm runs at improved speeds compared to previously suggested RLS algorithm for the CMAC. This paper also presented a newly derived IQR-RLS algorithm that implements regularization which helps to reduce the generalization error of the CMAC.

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] T. Qin, et al., "A Learning Algorithm of CMAC Based on RLS," *Neural Processing Letters*, vol. 19, pp. 49-61, 2004.

[3] 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.

[4] D. T. M. Slock and T. Kailath, "Numerically stable fast transversal filters for recursive least squares adaptive filtering," *IEEE Transactions on Signal Processing*, vol. 39, pp. 92-114, 1991.

[5] 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.

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

[7] 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.

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

[9] G. Horvath and T. Szabo, "Kernel CMAC With Improved Capability," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics,* vol. 37, pp. 124-138, 2007.

[10] 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.

[11] 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.