Fixed Order Implementation Method of Kernel Adaptive Filters with Lower Computational Complexity

Kiyoshi NISHIKAWA, and Koji MAKIZAKI
Dept. of Information and Communications Systems, Tokyo Metropolitan University
6-6 Asahigaoka, Hino-shi, Tokyo 191-0065 JAPAN
E-mail: knishikawa@m.ieice.org Tel: +81-42-585-8423

***Abstract*—In this paper, we propose an implementation method of kernel adaptive filters by fixing the filter order with lower computational complexity. Kernel adaptive filters are used for adaptive learning of non-linear systems. Although they enable us to estimate non-linear systems, computational load required for implementing the kernel method becomes relatively high. Moreover, the conventional methods require the order of the adaptive filter to be incremented as time n increases. The increment of the filter order results in variation of processing time for updating the filter at each time. These features could cause a problem when we implement them in a system with limited computational resources, such as embedded systems like mobile terminals. We propose, in this paper, a fixed order implementation method of kernel adaptive filters. The proposed method also includes a method to reduce the computational complexity to calculate the Gaussian kernel function. Through the simulation, we show that the proposed method could provide almost same convergence characteristics with less than half of the processing time under certain conditions.**

I. Introduction

Kernel adaptive filters enable us to estimate non-linear systems, and are expected to be used in applications such as non-linear channel equalization[1].

Kernel adaptive filters are derived by applying the kernel method to linear adaptive filters. Although, they are effective for non-linear learning, required computational load for kernel method is relatively high. Another problem of the conventional kernel adaptive filters is that the order of the adaptive filter increases as time n increases. The increment of the filter order could cause problems in some environments where computational resources are limited such as mobile terminals. Besides, the processing time for updating the filter coefficients will also increase. This characteristics also might cause problems in those environments.

In this paper, we propose an implementation method of the kernel adaptive filters with fixing the filter order at lower computation complexity. The proposed method is consisting of two parts. The one is for reducing the number of calculation required for obtaining the values of the inner products using the kernel functions. We show that, by expanding the kernel function, a part of the calculations could be omitted so that the total number of calculation could be decreased. Then, the second part is to fix the filter order. By combining these two methods, we can implement the kernel adaptive filters with lower computational complexity and the fixed order. Through the computer simulations, we verified the validity of the proposed method.

II. Preparation

Here, we briefly describe the conventional kernel method and kernel adaptive filters[1], [2].

A. Kernel method

The input signal $x(n)$ is transformed into a high-dimensional feature space F. By denoting the transformation to the space Fas $\Phi(x)$, the output signal of the adaptive filter is expressed as

$$f\left(\boldsymbol{x}_n\right) = \Phi^T\left(\boldsymbol{x}_n\right)\boldsymbol{w} \tag{1}$$

where $\boldsymbol{w}$ and $\boldsymbol{x}_n$ are filter coefficient vector of the adaptive filter, and tap-input vector at time n respectively. They are given as

$$\boldsymbol{w} = [w_0, \ldots, w_{M-1}], \tag{2}$$

$$\boldsymbol{x}_n = [x(n), \ldots, x(n-M+1)] \tag{3}$$

where w_i, $x(n)$ and M show the i-th coefficient of the filter at time n, the input sample at n and the length of the filter respectively. Note that we drop the time index n in the expressions for simplifying the notation.

Here, let us assume that the filter vector $\boldsymbol{w}$ can be expressed as a linear combination of m training vectors $\Phi(\boldsymbol{y}_j)$ as

$$\boldsymbol{w} = \sum_{j=1}^{m} \alpha_j \Phi\left(\boldsymbol{y}_j\right). \tag{4}$$

The vectors $\boldsymbol{y}_j$ are subset of $\boldsymbol{x}_\ell$ $(\ell = 0, 1, \ldots, n-1)$ and the detail will be described in the next sub-section, and α_j is the weight corresponding to $\boldsymbol{y}_j$. Then, the output in (1) is expressed[1] as

$$f\left(\boldsymbol{x}_n\right) = \sum_{j=1}^{m} \left(\Phi^T\left(\boldsymbol{x}\right)\Phi\left(\boldsymbol{y}_j\right)\right)\alpha_j. \tag{5}$$

By defining $\boldsymbol{\alpha}$ as $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_m]^T$, we can regard $\boldsymbol{\alpha}$ as the coefficient vector of a filter, and the kernel adaptive algorithms are derived[2], [3] to estimate the optimum $\boldsymbol{\alpha}$. For that, we should calculate the inner product $\Phi^T(\boldsymbol{x}(n))\Phi(\boldsymbol{y}_j)$ in Eq. (5).

The kernel adaptive filters use the kernel function to calculate the inner product. A kernel function $k(\cdot,\cdot)$ is given as

$$\forall \mathbf{a},\mathbf{b} \in X \qquad k(\boldsymbol{a},\boldsymbol{b}) = \Phi^T(\mathbf{a})\Phi(\mathbf{b}) \tag{6}$$

and is used to calculate the inner product in the space F. This method is referred to as the kernel trick[1]. Note that, we do not need to know the transformed vectors $\Phi(\boldsymbol{a})$, or $\Phi(\boldsymbol{b})$ themselves to calculate the inner product. For a kernel function to be used in the kernel adaptive algorithm, the condition $k(\mathbf{a},\mathbf{a}) = 1$ must be satisfied[1].

As the kernel function, the Gaussian kernel defined as below is used in conventional kernel adaptive filters[2], [3].

$$k(\boldsymbol{a},\boldsymbol{b}) = \exp\left(-\|\boldsymbol{a}-\boldsymbol{b}\|^2\right) \tag{7}$$

where $\|\cdot\|$ show the Euclidean norm. In this paper, we assume that the Gaussian kernel is used as the kernel function.

B. Kernel adaptive filter

By applying the kernel method to the linear adaptive filters, the concept of the kernel adaptive filter is derived[1], [2]. The description below, and also our proposed method, are based on the method for designing sparse kernel adaptive filters proposed in [2].

First, we rewrite Eq. (5) as

$$f(\boldsymbol{x}_n) = \begin{bmatrix} \Phi(\boldsymbol{x}_n)^T\Phi(\mathbf{y}_1) \\ \Phi(\boldsymbol{x}_n)^T\Phi(\mathbf{y}_2) \\ \vdots \\ \Phi(\mathbf{x_n})^T\Phi(\mathbf{y}_m) \end{bmatrix}^T \boldsymbol{\alpha} = \begin{bmatrix} k(\boldsymbol{x}_n,\boldsymbol{y}_1) \\ k(\boldsymbol{x}_n,\boldsymbol{y}_2) \\ \vdots \\ k(\boldsymbol{x}_n,\boldsymbol{y}_m) \end{bmatrix}^T \boldsymbol{\alpha} \tag{8}$$

$$= \boldsymbol{h}_n\boldsymbol{\alpha} \tag{9}$$

where we set $\boldsymbol{h}_n$ as $\boldsymbol{h}_n = [k(\boldsymbol{x}_n,\boldsymbol{y}_1),\ldots,k(\boldsymbol{x}_n,\boldsymbol{y}_m)]^T$. Then, the filter $\boldsymbol{\alpha}$ can be updated using a linear adaptive algorithm by regarding $\boldsymbol{h}_n$ as the input vectors to $\boldsymbol{\alpha}$[1].

Here, we define the matrix D which is consisting of the vectors $[\boldsymbol{y}_1,\ldots,\boldsymbol{y}_m]$ as

$$\mathbf{D} = \begin{bmatrix} \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_m \end{bmatrix} \tag{10}$$

and $\boldsymbol{D}$ is called the dictionary. The vectors stored in the dictionary $\boldsymbol{D}$ are m $(m \le n)$ past input vectors $\boldsymbol{x}_\ell$ where m is a variable determined by the algorithm below, and, in general, m increases as n.

Let us denote $\boldsymbol{D}$ at time n by $\boldsymbol{D}_n$. Then, $\boldsymbol{D}_n$ and $\boldsymbol{h}_n$ are updated according to the following pseudo algorithm:

Initialization
$\quad \boldsymbol{D}_1 = \boldsymbol{y}_1 = \boldsymbol{x}_1$
$\quad \boldsymbol{h}_1 = k(\boldsymbol{x}_1,\boldsymbol{y}_1)$
$\quad \boldsymbol{\alpha}_1 = 0 \quad , \quad m = 1$
for $n = 2,3,\cdots$
$\quad$ if $\max_{j=1,\ldots,m} |k(\boldsymbol{x}_n,\boldsymbol{y}_j)| > \mu_0$ (11)
$\quad\quad \boldsymbol{D}_n = \boldsymbol{D}_{n-1}$
$\quad\quad \boldsymbol{h}_n = \begin{bmatrix} k(\boldsymbol{x}_n,\boldsymbol{y}_1) & \cdots & k(\boldsymbol{x}_n,\boldsymbol{y}_m) \end{bmatrix}^T$ (12)
$\quad$ else
$\quad\quad m = m+1$
$\quad\quad \boldsymbol{D}_n = \boldsymbol{D}_{n-1} \cup \{\boldsymbol{x}_n\}$
$\quad\quad \boldsymbol{h}_n = \begin{bmatrix} k(\boldsymbol{x}_n,\boldsymbol{y}_1) & \cdots & k(\boldsymbol{x}_n,\boldsymbol{y}_m) \end{bmatrix}^T$ (13)
$\quad$ end if
end for

In Eq. (11), μ_0 is a threshold in the range $0 < \mu_0 < 1$ and its value is determined according to the sparseness of the filter. The input vector $\boldsymbol{x}_n$ will be compared with the vectors in $\boldsymbol{D}_n$ by the condition shown in Eq. (11). If the condition met, $\boldsymbol{x}_n$ will be stored in $\boldsymbol{D}_n$ as a new training vector.

For updating the filter coefficients, the conventional linear adaptive algorithm can be used, namely, NLMS[2], RLS[3], ERLS-DCD[5] algorithms and so on.

III. Proposed method

Here, we describe the proposed method. The proposed method is consisting of two parts.

In kernel adaptive filter, the computational complexity to implement the kernel method is relatively high. This could pose a problem when we employ low complexity adaptive algorithms such as the NLMS algorithm. Hence, we first propose a method to implement the kernel function with reduced computational complexity. Then, we consider another problem, the increase of the filter order m. Increase of m in the conventional methods would make it difficult to implement kernel adaptive filters in embedded systems such as mobile terminals. Hence, we propose an implementation method with the fixed m.

We should note that the proposed method could be used with any adaptive algorithm.

A. Reduction of Computational Complexity by Modification of Gaussian Kernel

In the proposed method, we assume to use the Gaussian kernel. Under this assumption, we show that we can reduce the amount of calculation required to implement the kernel function.

First, we expand the Gaussian kernel as below.

$$\begin{aligned} k(\boldsymbol{a},\boldsymbol{b}) &= \exp\left(-\|\boldsymbol{a}-\boldsymbol{b}\|^2\right) \\ &= \exp\left(2\boldsymbol{a}^T\boldsymbol{b} - \boldsymbol{a}^T\boldsymbol{a} - \boldsymbol{b}^T\boldsymbol{b}\right) \end{aligned} \tag{14}$$

By using this, $\boldsymbol{h}_n$ in Eq. (12) could be rewritten as

$$\mathbf{h}_n = \exp\left(\begin{bmatrix} 2\boldsymbol{x}_n^T\boldsymbol{y}_1 - \boldsymbol{x}_n^T\boldsymbol{x}_n - \boldsymbol{y}_1^T\boldsymbol{y}_1 \\ 2\boldsymbol{x}_n^T\boldsymbol{y}_2 - \boldsymbol{x}_n^T\boldsymbol{x}_n - \boldsymbol{y}_2^T\boldsymbol{y}_2 \\ \vdots \\ 2\boldsymbol{x}_n^T\boldsymbol{y}_m - \boldsymbol{x}_n^T\boldsymbol{x}_n - \boldsymbol{y}_m^T\boldsymbol{y}_m \end{bmatrix}\right). \tag{15}$$

It is shown that there are three columns in $\exp(\cdot)$, namely $2\boldsymbol{x}_n^T\boldsymbol{y}_j$, $\boldsymbol{x}_n^T\boldsymbol{x}_n$, and $\boldsymbol{y}_j^T\boldsymbol{y}_j$ where $j = 1, 2, \ldots, m$. Of these, the first and the second columns contain $\boldsymbol{x}_n$, and these two columns should be updated at each time n.

We notice the following points.

a) For calculating the first column $\boldsymbol{x}_n^T\boldsymbol{y}_j$, we need to calculate m inner products at each time.
b) For the second column, only one inner product $\boldsymbol{x}_n^T\boldsymbol{x}_n$ is required.
c) The third column contains the inner products of past input vectors in the dictionary, namely, $\boldsymbol{y}_j^T\boldsymbol{y}_j$.

From c), we see that the inner products in the third column were calculated in past time j ($j = 1, 2, \ldots, m$). Hence, by saving these values in the dictionary $\boldsymbol{D}_n$ as additional information, we could decrease the amount of calculation required to calculate Eq. (14).

B. Construction of dictionary in the proposed method

As described in the previous subsection, we propose to store the inner products at each time $\boldsymbol{x}_n^T\boldsymbol{x}_n$ in the dictionary as additional information. Then we can avoid the calculation of the third term in Eq. (15).

The construction of the dictionary in the proposed method is described as below.

Initialization

$$\chi_1 = \boldsymbol{x}_1^T\boldsymbol{x}_1 \tag{16}$$

$$\boldsymbol{D}_1 = \boldsymbol{y}_1 = \begin{bmatrix}\boldsymbol{x}_1 \\ \chi_1\end{bmatrix} \tag{17}$$

$$\boldsymbol{h}_1 = 1 \quad , \quad \boldsymbol{\alpha}_1 = 0 \quad , \quad m = 1$$

for $n = 2, 3, \cdots$

$$\chi_n = \boldsymbol{x}_n^T\boldsymbol{x}_n \tag{18}$$

$$\boldsymbol{h}_n = \exp\left(\boldsymbol{D}_{n-1}^T\begin{bmatrix}2\boldsymbol{x} \\ -1\end{bmatrix} - \chi_n\mathbf{1}\right) \tag{19}$$

if $\max_{j=1,\ldots,m}|\boldsymbol{h}_n(j)| > \mu_0$

$$\boldsymbol{D}_n = \boldsymbol{D}_{n-1}$$

else

$$m = m + 1$$

$$\boldsymbol{h}_n = \begin{bmatrix}\boldsymbol{h}_n \\ 1\end{bmatrix} \tag{20}$$

$$\boldsymbol{D}_n = \boldsymbol{D}_{n-1} \cup \left\{\begin{bmatrix}\boldsymbol{x}_n \\ \chi_n\end{bmatrix}\right\} \tag{21}$$

end if

end for

where $\mathbf{1}$ shows a vector of length m whose components are set as 1.

By comparing the direct calculation in Sec. II-B, we can reduce the number of calculation. In TABLE I, we show the comparison in terms of the number of calculation to implement the kernel part. Note that, in the table, we assumed that the multiplication and addition could be regarded as equal load. This seems likely true for the recently used processors, or DSP chips based on the pipeline processing.

From the table, we can see that the total amount of calculation of the proposed method become smaller than the direct calculation when $M > 2$ and $m \gg 1$. Besides, the difference becomes large as m, the filter length, increases as in the case of the kernel adaptive filters.

TABLE I
COMPARISON OF NUMBER OF CALCULATION FOR KERNEL PART

	Multiplication	Addition	Total
Proposed	$Mm + m + 2$	$M + 2m + 1$	$(M+3)(m+1)$
Direct calculation	Mm	$2Mm - m$	$(3M-1)m$

C. Fixed order implementation of the kernel adaptive filters

Next, we propose a method to fix the filter order m. As described in the above, m would be incremented when a new training vector added to $\boldsymbol{D}_n$, and this could cause a problem when we implement it in the environments with limited computational resources, such as mobile terminals.

We propose to fix the order by setting the allowable maximum order of the filter as $m_{\max}$. Besides, the initial state of the dictionary is set as the zero matrix of $(M+1) \times m_{\max}$, the filter coefficient vector as a zero-vector of $m_{\max}$.

For enabling these fixed order matrix and vectors, we slightly modify the update equation of filter, and that of $\boldsymbol{D}_n$ as the following.

Initialization

$$\boldsymbol{D} = \mathbf{0}^{(M+1)\times m_{max}} \quad , \quad \boldsymbol{\alpha}_{-1} = \mathbf{0}^{m_{max}} \quad , \quad m = 1$$

for $n = 0, 1, \cdots$

$$\chi_n = \boldsymbol{x}_n^T\boldsymbol{x}_n \quad , \quad \boldsymbol{h}_n = \exp\left(\boldsymbol{D}^T\begin{bmatrix}2\boldsymbol{x}_n \\ -1\end{bmatrix} - \chi_n\mathbf{1}^{m_{max}}\right) \tag{22}$$

if $\left(\max_{j=1,\ldots,m}|\boldsymbol{h}_n(j)| \le \mu_0\right) \wedge (m < m_{max})$ (23)

$$m = m + 1$$

$$\boldsymbol{D}(m-1) = \begin{bmatrix}\boldsymbol{x}_n \\ \chi_n\end{bmatrix} \quad , \quad \boldsymbol{h}_n(m-1) = 1$$

$$\boldsymbol{\alpha}_{n-1}(0) = \boldsymbol{\alpha}_{n-1}(0) + \boldsymbol{\alpha}_{n-1}(m-1) \tag{24}$$

$$\boldsymbol{\alpha}_{n-1}(m-1) = 0$$

end if

$$e_n = d_n - \boldsymbol{h}_n^T\boldsymbol{\alpha}_{n-1} \quad , \quad \boldsymbol{\alpha}_n = \boldsymbol{\alpha}_{n-1} + \Delta\boldsymbol{\alpha}_n \tag{25}$$

end for

In the proposed method, we compare m and $m_{\max}$ at Eq. (23) so that the filter order is upper bounded by $m_{\max}$. Besides, when we insert an input vector as i-th training vector, we

set i-th filter coefficient $\boldsymbol{\alpha}_{n-1}$ to be zero. Note that, even we fix the order, the dictionary should be maintained to keep the sufficient training by the methods such as proposed in [6].

IV. Simulation Results

Here, we show simulation results of adaptive prediction using the proposed method. The computer environments of the simulations are shown in Table II.

In the simulations, we generated the input signal using the equation

$$x_n = \left(0.8 - 0.5\exp\left(-x_{n-1}^{-2}\right)\right) x_{n-1} - \left(0.3 + 0.9\exp\left(-x_{n-1}^{2}\right)\right) x_{n-2} + 0.1\sin\left(x_{n-1}\pi\right) \tag{26}$$

and the initial values of x_{-1} and x_{-2} were given as random numbers of uniform distribution in the region $(0, 1)$. The order M of the input signal was set as four, and a white Gaussian noise of SNR 40dB was added to the signal. The threshold value μ_0 was set as 0.8, the length of the signal was 10000. The values of coefficients in Eq. (26) were changed at $n = 4000$ to demonstrate the tracking ability of the method. We evaluated in terms of mean squared error (MSE) and the ensemble average of 1000 independent trials are shown. The KNLMS algorithm was used as the learning algorithm, and compared with the linear NLMS, and the conventional KNLMS algorithms. We set $m_{\max}$ as $m_{\max} = 32$ in the simulations and the value was determined by trials.

The results are shown in Fig. 1. From the figure, we could confirm that the proposed method provides almost identical convergence characteristic as that of the conventional. In TABLE III, we show the processing time required for updating filter coefficients at each time. Note that, for conventional KNLMS algorithm, the time for updating varies as the filter order increases, and hence, the average time is shown in the table. It is shown that the computational time of the proposed method is almost a half of that of the conventional one. We note that this reduction is largely due to the fact that the conventional method requires higher filter order than the proposed method. Hence, with the optimum selection of the filter order, the proposed method could be implemented with a lower computational complexity.

Moreover, filter order in the proposed method is fixed so that the processing time for each time is also fixed. This feature would be preferred when it is implemented in embedded systems, or systems with limited computational resources. Note that the difference become larger as M increases as shown in Sec III-B.

V. Conclusion

In this paper, we proposed a method to implement the kernel adaptive filters with the fixed order. The proposed method is composed of two parts. The first part is to reduce the computational complexity to calculate the Gaussian kernel function by re-arranging the inner product. The second part is a method to fix the order of the adaptive filter. By fixing the order, we could also fix the processing time for updating at each time, and the amount of calculation required to implement the method. These features are important in the systems with limited computational resources. Also, the proposed method has an advantage that it could be used with any adaptive algorithms.

Through computational simulations, we confirmed that the proposed method could maintaining the almost equivalent convergence characteristics by appropriately selecting the filter order. Besides, it was shown that the possibility of reducing the processing time by fixing the order. The consideration on the optimum selection of $m_{\max}$ will be one of our future works.

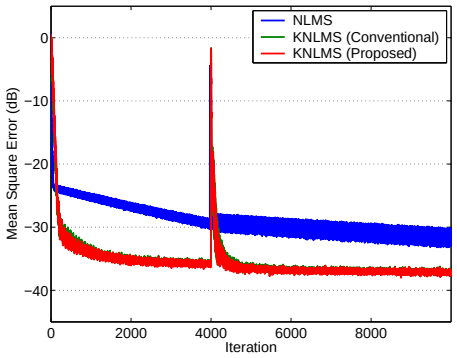

Fig. 1. Comparison of convergence characteristics of the NLMS, conventional KNLMS, and the proposed algorithms. The proposed algorithm provides almost identical characteristics to that of the conventional KLMS although its computational load is lower. At $n = 4000$, the values of the coefficients in Eq. (26) was changed to show the tracking capabilities of the algorithms.

TABLE II
Computer environments for simulations

CPU	Intel Core 2 Duo 3.33 GHz
RAM	4.0 GB
Software	MATLAB 6.5.1 Release 13

TABLE III
Comparison of computational time

Algorithm	Time for an update of the filter
NLMS	26.0 μs
KNLMS (Conventional)	103.6 μs (average)
KNLMS (Proposed)	40.0 μs

References

[1] Weifeng Liu, José C. Príncipe and Simon Haykin, "Kernel Adaptive Filtering," Wiley: 2010.

[2] Cédric Richard, José Carlos M. Bermudez and Paul Honeine, "Online Prediction of Time Series Data With Kernels," IEEE Transactions on Signal Processing, Vol. 57, No. 3, pp.1058-1067, Mar., 2009

[3] Yaakov Engel, Shie Mannor and Ron Meir, "The Kernel Recursive Least-Squares Alrorithm," IEEE Transactions on Signal Processing, Vol. 52, No. 8, pp.2275-2285, Aug., 2004

[4] Puskal P. Pokharel, Weifeng Liu and Jose C. Principe, "Kernel LMS," *IEEE International Conf. Acoustics, Speech and Signal Proc.*, 2007

[5] Yoshiki OGAWA, and Kiyoshi NISHIKAWA, "A Kernel Adaptive Filter based on ERLS-DCD Algorithm," *Proc. of Intl Tech. Conf. Circuits Systems, Computer, Communications 2011*, June 2011.

[6] Weifeng Liu, Ii Park, and José C. Príncipe, "An Information Theoretic Approach of Designing Sparse Kernel Adaptive Filters," IEEE Trans. on Neural Networks, Vol. 20, No. 20, pp. 1950–1961, 2009