FIXED-BUDGET KERNEL RECURSIVE LEAST-SQUARES

Steven Van Vaerenbergh[*], *Ignacio Santamaría*[*], *Weifeng Liu*[†] *and José C. Príncipe*[‡]

[*]Dept. of Communications Engineering, University of Cantabria, Santander, Spain
[†]Forecasting Team, Amazon.com, 701 5th Ave, Seattle, WA 98104 USA
[‡]Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA
Email: {steven,nacho}@gtas.dicom.unican.es, weifeng@ieee.org, principe@cnel.ufl.edu

ABSTRACT

We present a kernel-based recursive least-squares (KRLS) algorithm on a fixed memory budget, capable of recursively learning a nonlinear mapping and tracking changes over time. In order to deal with the growing support inherent to online kernel methods, the proposed method uses a combined strategy of growing and pruning the support. In contrast to a previous sliding-window based technique, the presented algorithm does not prune the oldest data point in every time instant but it instead aims to prune the least significant data point. We also introduce a label update procedure to equip the algorithm with tracking capability. Simulations show that the proposed method obtains better performance than state-of-the-art kernel adaptive filtering techniques given similar memory requirements.

Index Terms— Kernel methods, machine learning, recursive least-squares, nonlinear filtering, fixed budget

1. INTRODUCTION

During the past decade there has been a growing interest in kernel methods, motivated by their successful applications in many fields such as image processing, biomedical engineering and communications. Similar to neural networks, kernel methods are universal nonlinear approximators, but they show the advantage of yielding convex optimization problems. Popular kernel-based algorithms include support vector machines (SVM) and kernel principal component analysis (KPCA) [1].

An online scenario assumes that data is received as a stream of input-output patterns $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots\}$, in which $\mathbf{x}_i$ is a vector and y_i is a scalar, for $i = 1, 2, \ldots$ Notice that we assume a supervised setting in which input and output data are provided. The goal of online algorithms is to update their solution in every iteration n, based on the new available data $(\mathbf{x}_n, y_n)$, while maintaining a low computational complexity. In their classic formulation, kernel methods are not suitable for online applications, as their functional representation grows linearly with the number of processed patterns. This poses both computational as well as memory issues. Therefore, substantial effort has gone into designing "sparsification" techniques that curb the growth of the networks constructed by these methods.

Recently a number of kernel adaptive filtering algorithms have been presented that introduce sparsification procedures similar to the resource-allocating networks (RAN) proposed by Platt [2]. In [3], Engel et al. introduce an *approximate linear dependency* (ALD) criterion that constructs a dictionary of significant patterns based on the dependency of feature vectors. In [4], Liu et al. propose a criterion that measures the information a point can contribute to the learning system, based on Gaussian process theory. Unlike ALD, this "surprise"-based criterion also takes into account the output data y_i.

As a counterpart to criteria that optimally construct a sparse network, a number of *pruning* criteria have been developed that discard redundant patterns from existing large networks. Pruning techniques have been well studied in the context of neural network design, for instance in optimal brain damage [5] and optimal brain surgeon [6], which are based on an analysis of the Hessian of the error surface. In [7, 8] a number of different, easier to evaluate criteria were presented to prune least-squares support vector machines (LS-SVM).

A few methods have been proposed that combine growing and pruning procedures, for instance generalized growing and pruning (GGAP) [9], the forgetron [10], and sliding-window algorithms [11, 12, 13], which limit the memory to the M newest patterns. These approaches are interesting with practical applications in mind, such as implementations on a microchip, since they allow to put an exact upper bound on the memory size and the number of computations needed. Moreover, they are capable of forgetting past data, which makes them suitable for operating in time-varying environments.

The contributions of this paper are threefold. First, we show how the solution of the kernel recursive least-squares (KRLS) algorithm can be updated when one new point is added to the support and one chosen point is discarded. Second, we show how to choose the point to discard by using a suitable, easy to evaluate pruning criterion. And third, we introduce a *label update* procedure that equips the proposed algorithm with tracking capability.

The rest of this paper is organized as follows. In section 2 we briefly overview the basics of KRLS, followed by a description of the proposed method in section 3. The results of two numerical experiments are reported in section 4 and, finally, the main conclusions of this work are listed in section 5.

2. KERNEL RECURSIVE LEAST-SQUARES

2.1. Kernel Methods

Kernel methods are powerful nonlinear techniques based on a nonlinear transformation of the data $\mathbf{x}_i$ into a high-dimensional *feature space*, in which it is more likely that the transformed data $\Phi(\mathbf{x}_i)$ is linearly separable. In feature space, inner products can be calculated by using a positive definite kernel function satisfying Mercer's condition [14]: $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$. This simple and elegant idea, also known as the "kernel trick", allows to perform inner-product based algorithms implicitly in feature space by replacing all inner products by kernels. A commonly used kernel function is the

This work was supported by MEC of Spain under grant TEC2007-68020-C04-02 TCM (MultiMIMO) and FPU grant AP2005- 5366.

Gaussian kernel

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma^2). \quad (1)$$

In kernel-based regression techniques, a nonlinear mapping is evaluated as a linear combination of kernels of *support vectors* $\mathbf{x}_i$

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x}). \quad (2)$$

Thanks to the Representer Theorem [1], the nonlinearity f can be represented sufficiently well by choosing the training vectors as the support of this expansion.

2.2. Recursive Least-Squares in Feature Space

In an offline scenario where M input-output patterns are available, the standard kernel-based least-squares (LS) problem [1] can be defined as finding the coefficients α_i that minimize

$$\min_{\boldsymbol{\alpha}} |\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}|^2 + \lambda \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha}, \quad (3)$$

where $\mathbf{y} \in \mathbb{R}^{M\times 1}$ contains the outputs y_i of the training data, $\mathbf{K} \in \mathbb{R}^{M\times M}$ is the Gram matrix (or *kernel matrix*), with elements $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$, and λ is a regularization parameter which introduces a penalization on the solution norm and therefore imposes smoothness. The solution of (3) is found as

$$\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y}, \quad (4)$$

where $\mathbf{I}$ represents the unit matrix.

The goal of kernel recursive least-squares is to update this solution recursively as new data become available [15]. However, in contrast to linear RLS, which is based on the covariance matrix, KRLS is based on the Gram matrix $\mathbf{K}$, whose dimensions depend on the number of input patterns, not on their dimensionality. As a consequence, the inclusion of new data into the solution (4) causes the kernel matrix to grow without bound.

2.3. Curbing the Growth of the Support

Several techniques have been proposed to curb this growth, including the ALD criterion [3], the *surprise information measure* [4] and sliding-window techniques [11, 12, 13]. These methods assemble a limited *dictionary* of input-output patterns $(\mathbf{x}_i, y_i)$ which are used to construct the nonlinear mapping (2). In order to obtain a confident estimate, these patterns should represent the complete input-output data distribution sufficiently well. Simultaneously to this selection process, KRLS performs kernel least-squares regression on these patterns to obtain the optimal nonlinear mapping.

The proposed method builds upon ideas presented in [12, 13], in which the memory size is fixed to M patterns. However, it takes a more active role in the building of the dictionary. Specifically, in every iteration it first adds a new point to the memory, and then it determines the least relevant data point present in the memory, which is subsequently pruned. The result of this active learning strategy is that at any time instant the memory will contain only the M most relevant patterns seen up till that moment.

3. FIXED-BUDGET KERNEL RLS

An important aspect of online algorithms is that their computational complexity should be moderate in every iteration. Therefore, one of the design goals for the proposed method was to obtain a complexity not higher than $O(M^2)$, where M is the number of patterns stored in memory.

In the following, we describe the different parts of the algorithm, starting by the algebraic operations necessary to update the KRLS solution efficiently. The criterion used to determine the least significant pattern in memory is discussed in section 3.2, and in section 3.3 we propose a simple memory update formula that allows to deal with time-varying nonlinear mappings. By $\mathbf{K}_n$ we will denote the regularized kernel matrix $\mathbf{K} + \lambda\mathbf{I}$ obtained in the n-th iteration.

3.1. Update of the Kernel LS Solution

The inversion of the $M \times M$ matrix in (4) is computationally expensive, requiring $O(M^3)$ calculations. However, in [12], a matrix update procedure of $O(M^2)$ was proposed that allows to compute the inverse matrix $\mathbf{K}_n^{-1}$ given the previous inverse kernel matrix $\mathbf{K}_{n-1}^{-1}$.

3.1.1. Adding a row and column to the kernel matrix

In the n-th iteration, a new pattern $(\mathbf{x}_n, y_n)$ is first added to the memory, which corresponds to adding one row and one column to the kernel matrix $\mathbf{K}_{n-1}$. We call this operation "upsizing" the matrix, and the result is denoted as $\breve{\mathbf{K}}_n$. Given the inverse matrix $\mathbf{K}_{n-1}^{-1}$, the inverse of the upsized matrix, $\breve{\mathbf{K}}_n^{-1}$, can be obtained by calculating

$$\breve{\mathbf{K}}_n = \begin{bmatrix} \mathbf{K}_{n-1} & \mathbf{b} \\ \mathbf{b}^T & d \end{bmatrix} \Rightarrow \breve{\mathbf{K}}_n^{-1} = \begin{bmatrix} \mathbf{K}_{n-1}^{-1} + g\mathbf{e}\mathbf{e}^T & -g\mathbf{e} \\ -g\mathbf{e}^T & g \end{bmatrix}, \quad (5)$$

in which $\mathbf{e} = \mathbf{K}_{n-1}^{-1}\mathbf{b}$, $g = (d - \mathbf{b}^T\mathbf{e})^{-1}$, and $\mathbf{b}$ and d contain kernels between $\mathbf{x}_n$ and the other points in memory (see [12]).

3.1.2. Removing the i-th row and column from the kernel matrix

In the sliding-window approach from [12], the *first* row and column of the upsized kernel matrix $\breve{\mathbf{K}}_n$ are removed in every iteration, yielding the "downsized" matrix $\mathbf{K}_n$. The inverse of this matrix can be obtained efficiently based on the knowledge of $\breve{\mathbf{K}}_n^{-1}$, as follows

$$\breve{\mathbf{K}}_n = \begin{bmatrix} a & \mathbf{b}^T \\ \mathbf{b} & \mathbf{K}_n \end{bmatrix}, \quad \breve{\mathbf{K}}_n^{-1} = \begin{bmatrix} e & \mathbf{f}^T \\ \mathbf{f} & \mathbf{G} \end{bmatrix} \Rightarrow \mathbf{K}_n^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e. \quad (6)$$

The proposed method requires to remove an *arbitrary* row and column of the matrix $\breve{\mathbf{K}}_n$. In order to do this, the matrix inversion formula can be extended by applying a few permutations, based on

$$\mathbf{P}_i = \begin{bmatrix} 0 & \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{i-2} & \mathbf{0} & \mathbf{0} \\ 1 & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_{M-i+1} \end{bmatrix}, \quad \mathbf{Q}_i = \begin{bmatrix} \mathbf{0} & \mathbf{I}_{i-1} & \mathbf{0} \\ 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_{M-i} \end{bmatrix}, \quad (7)$$

in which $\mathbf{I}_j$ is the unit matrix of size j and $\mathbf{0}$ is the all-zeroes matrix of adequate dimensions. Notice that $\mathbf{P}_i^{-1} = \mathbf{P}_i$ and $\mathbf{Q}_i^{-1} = \mathbf{Q}_i^T$. Algorithm 1 summarizes the steps necessary to obtain the required inverse matrix. In the first step, the result of pre-and post-multiplying by $\mathbf{P}_i$ is an exchange of the first and i-th row and column. In the last step, pre- and post-multiplying a matrix by $\mathbf{Q}_i$ puts its i-th row and column in front of the others. In practice, these calculations can be implemented as fast matrix operations.

Algorithm 1 Procedure to obtain the inverse of a matrix $\breve{\mathbf{K}}_n$ whose i-th row and column are removed

Compute $\breve{\mathbf{K}}_n^i = \mathbf{P}_i \breve{\mathbf{K}}_n \mathbf{P}_i$ and $(\breve{\mathbf{K}}_n^i)^{-1} = \mathbf{P}_i \breve{\mathbf{K}}_n^{-1} \mathbf{P}_i$ with (7).
Remove the first row and column of $\breve{\mathbf{K}}_n^i$ to obtain $\mathbf{K}_n^i$.
Calculate the inverse $(\mathbf{K}_n^i)^{-1}$ by applying (6).
Obtain $\mathbf{K}_n^{-1} = \mathbf{Q}_i (\mathbf{K}_n^i)^{-1} \mathbf{Q}_i$ using (7).

3.2. Discarding criterion

In the n-th iteration, the pattern $(\mathbf{x}_n, y_n)$ is first added to the memory, which now contains $M+1$ patterns. The inverse kernel matrix is updated as described in section 3.1.1, and the regression coefficients α_i can be recalculated according to (4). In this section we discuss a criterion that determines the least significant of the $M+1$ stored patterns. Once it is found, the kernel matrix is downsized as in Alg. 1, and α_i is recalculated through (4).

Ideally, the pruning criterion should take into account the information present in the stored input data $\mathbf{x}_i$ and the stored labels y_i. A simple criterion consists in selecting the pattern that bears least error after it is omitted. As shown in [7], this error can be obtained as

$$d(\mathbf{x}_i, y_i) = \frac{|\alpha_i|}{[\breve{\mathbf{K}}_n^{-1}]_{i,i}}, \tag{8}$$

which is easily found since $\boldsymbol{\alpha}$ and $\breve{\mathbf{K}}_n^{-1}$ have been calculated previously. Moreover, experiments in [7, 8] show that this criterion obtains significant better performance than various related criteria.

3.3. Memory Update for Tracking Time-Varying Mappings

The above described procedure is capable of identifying a static nonlinear mapping, by selecting patterns to store in memory and performing regression on these patterns. If the nonlinear mapping changes over time, however, it is likely that the memory contains patterns that do not reflect the current mapping well. Since regression is performed only on the memory, these invalid patterns can remain in the memory and affect the algorithm's performance.

On the other hand, it is reasonable to assume that after a number of iterations the input space will be sufficiently well sampled. Since the change in the observed system's response is reflected only on the output data y_i, we only need to adjust the outputs stored in the memory in order to achieve tracking capability. We propose to use the following update for all stored data labels y_i, whenever a new input-output point $(\mathbf{x}_n, y_n)$ is received

$$y_i \leftarrow y_i - \mu \kappa(\mathbf{x}_i, \mathbf{x}_n)(y_i - y_n), \quad \forall i, \tag{9}$$

where $\mu \in [0, 1]$ is a step-size parameter.

The "relabeling" equation (9) takes into account the similarities in input *and* output space, measured respectively by the kernel function and the difference $y_i - y_n$. As a consequence, it only affects patterns $\mathbf{x}_i$ that are close enough to the new pattern $\mathbf{x}_n$ in the sense measured by the kernel. Concordantly, the change in the labels will be proportional to $y_i - y_n$. For instance, if the new point $\mathbf{x}_n$ coincides with some stored $\mathbf{x}_i$ and its label y_i is very different from y_n, this label will be changed proportionally to the difference $y_i - y_n$. Notice that if $\mu = 0$ this update has no effect, and the algorithm assumes the observed nonlinear system to be static. The final algorithm is summarized in Alg. 2.

Algorithm 2 Fixed-Budget Kernel Recursive Least-Squares

initialize
  Memory $= \{(\mathbf{x}_1, y_1)\}$. Calculate $\mathbf{K}_1^{-1}$ and $\boldsymbol{\alpha}$ with (4).
for $n = 2, 3, \ldots$ **do**
  Update all stored labels y_i with (9).
  Add $(\mathbf{x}_n, y_n)$ to memory and obtain $\breve{\mathbf{K}}_n^{-1}$ with (5).
  if memory size $> M$ **then**
    Determine least significant stored point $(\mathbf{x}_L, y_L)$ with (8).
    Prune $(\mathbf{x}_L, y_L)$ from memory and obtain $\mathbf{K}_n^{-1}$ with Alg. 1.
  end if
  Obtain KRLS solution based on updated memory, with (4).
end for

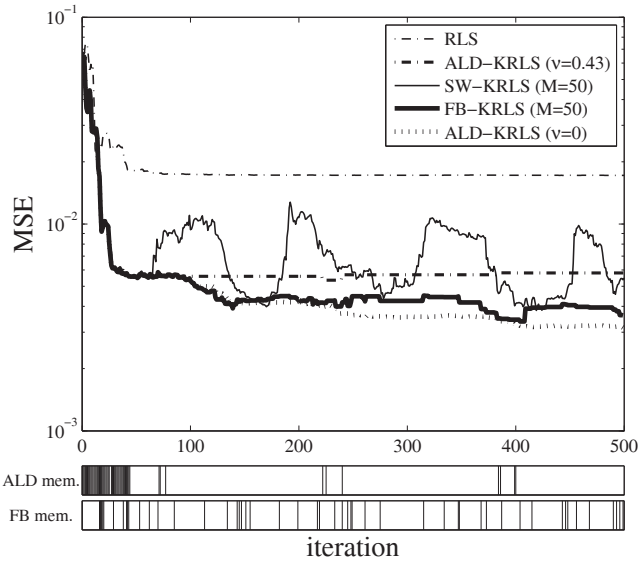

Fig. 1. Top: Learning curves for one-step prediction on the Mackey-Glass time-series. Bottom: indices of the patterns stored in memory by ALD-KRLS ($\nu = 0.43$) and FB-KRLS. Note that the final memory of FB-KRLS consists of patterns selected over the whole series.

4. SIMULATIONS

4.1. Prediction of a Steady-State Nonlinear Time-Series

In a first experiment we perform one-step prediction on the nonlinear Mackey-Glass time-series with a number of online algorithms. The data are corrupted by zero-mean Gaussian noise with 0.001 variance. The algorithms are trained online on 500 points of this series, and in each iteration the MSE performance is calculated on a test set of 100 points. A time-embedding of 7 is chosen, i.e. $\mathbf{x}_n = [x_n, \ldots, x_{n-6}]^T$ and the desired output is $y_n = x_{n+1}$.

The learning curves of the different algorithms are shown in Fig. 1. For the kernel-based algorithms, the Gaussian kernel with $\sigma = 1$ and regularization $\lambda = 0.1$ is chosen. As a lower bound for the MSE, the results of the ALD-KRLS algorithm with $\nu = 0$ are included, which uses a growing memory and has complexity $O(n^2)$. For SW-KRLS and FB-KRLS the memory size is fixed to 50 patterns. It is remarkable that the FB-KRLS technique obtains results that are very close to the lower bound. By setting $\nu = 0.43$, ALD-KRLS stores 53 patterns in memory, which is similar to FB-KRLS. Nevertheless, in this case ALD-KRLS performs worse.

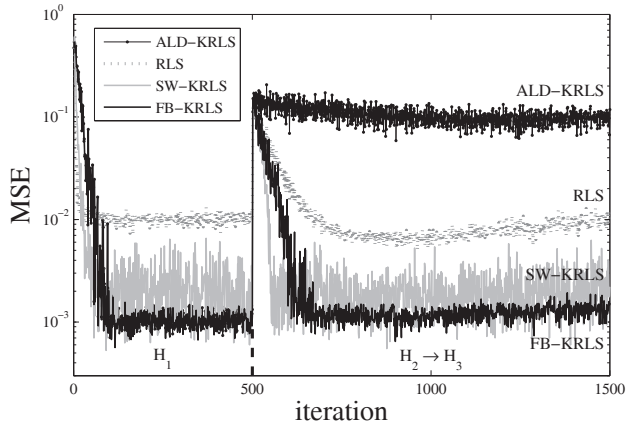

Fig. 2. Performance on a time-varying Wiener system. In the first zone ("H_1") all algorithms reach optimal steady-state performance, and ALD-KRLS and FB-KRLS coincide. After the abrupt change at iteration 500, only the tracking algorithms are able to recover.

Table 1. Performance comparison of final MSE values.

Algorithm	MSE	memory size
ALD-KRLS	0.0951 ± 0.1363	16
RLS	0.0084 ± 0.0099	n/a
SW-KRLS	0.0020 ± 0.0087	50
FB-KRLS	0.0012 ± 0.0018	16

4.2. Identification of a Time-varying Nonlinear System

For the second experiment, we consider a nonlinear communications channel composed of a linear filter followed by a static nonlinearity, also known as a Wiener system. As the system input we choose a binary signal, $x_i \in \{-1, +1\}$, and 30dB of white Gaussian noise is added at its output. The binary input signal is time-embedded with 4 taps, resulting in the input space showing 16 clusters. The static nonlinearity is $f(x) = \tanh(x)$. During the first 500 iterations the linear filter is fixed as $H_1(z) = 1 - 0.2663z^{-1} - 0.5541z^{-2} + 0.1420z^{-3}$. On iteration 501, the channel is abruptly switched to $H_2 = 1 + 0.1050z^{-1} - 0.3760z^{-2} - 0.4284z^{-3}$, which is then changed linearly until becoming $H_3(z) = 1 - 0.4326z^{-1} - 0.1656z^{-2} - 0.3153z^{-3}$ on the 1500-th iteration.

The algorithms use the following parameters: A Gaussian kernel with $\sigma = 0.1$ and $\lambda = 0.01$ are chosen for ALD-KRLS and FB-KRLS. Both methods only require 16 points in their dictionary, which is obtained for ALD-KRLS by setting $\nu = 0.1$, and for FB-KRLS by fixing $M = 16$. ALD-KRLS and RLS use a forgetting factor $\beta = 0.99$, and for FB-KRLS μ is set to 0.8. SW-KRLS uses a Gaussian kernel with $\sigma = 2$ and $M = 50$. The results, averaged out over 100 Monte Carlo simulations, are shown in Fig. 2. FB-KRLS obtains better performance than SW-KRLS, since the latter does not actively select significant patterns. Moreover, due to the fact that ALD-KRLS is not designed to be a tracking algorithm, it performs worst in this experiment. Table 1 displays the MSE averaged out over the last 500 iterations.

5. CONCLUSIONS

We presented a new fixed-budget kernel recursive least-squares algorithm for online identification of nonlinear systems. To maintain its memory size, it combines a growing memory with a discarding criterion previously proposed for LS-SVM. We also presented an efficient updating method for pruning an arbitrary point from the dictionary, and a label update procedure to provide tracking capability.

The proposed method represents a significant improvement over the previously proposed SW-KRLS algorithm, and given similar memory requirements it also outperforms ALD-KRLS. Moreover, it is capable of tracking changes of a time-varying nonlinear mapping. Future research topics include the study of more sophisticated pruning criteria, and a comparison with other nonlinear trackers.

6. REFERENCES

[1] B. Schölkopf and A. J. Smola, *Learning with Kernels*, The MIT Press, Cambridge, MA, USA, 2002.

[2] J. Platt, "A resource-allocating network for function interpolation," *Neural Computation*, vol. 3, no. 2, pp. 213–225, 1991.

[3] Y. Engel, S. Mannor, and R. Meir, "The kernel recursive least-squares algorithm," *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2275–2285, 2004.

[4] W. Liu, I. Park, and J. C. Príncipe, "An information theoretic approach of designing sparse kernel adaptive filters," *IEEE Transactions on Neural Networks*, vol. 20, no. 12, pp. 1950–1961, Dec. 2009.

[5] Yann Le Cun, John S. Denker, and Sara A. Solla, "Optimal brain damage," in *2nd Annual Conf. on Neural Information Proc. Syst. (NIPS)*, Denver, USA, Nov. 1989, pp. 598–605.

[6] B. Hassibi, D.G. Stork, and G.J. Wolff, "Optimal brain surgeon and general network pruning," in *IEEE International Conference on Neural Networks*, 1993, vol. 1, pp. 293–299.

[7] B.J. de Kruif and T.J.A. de Vries, "Pruning error minimization in least squares support vector machines," *IEEE Transactions on Neural Networks*, vol. 14, no. 3, pp. 696–702, May 2003.

[8] L. Hoegaerts, J.A.K. Suykens, J. Vandewalle, and B. De Moor, "A comparison of pruning algorithms for sparse least squares support vector machines," *Lecture Notes in Computer Science*, pp. 1247–1253, 2004.

[9] G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 57–67, Jan. 2005.

[10] O. Dekel, S. Shalev-Shwartz, and Y. Singer, "The forgetron: A kernel-based perceptron on a budget," *SIAM Journal on Computing*, vol. 37, no. 5, pp. 1342–1372, 2008.

[11] J. Kivinen, A.J. Smola, and R.C. Williamson, "Online learning with kernels," *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2165–2176, Aug. 2004.

[12] S. Van Vaerenbergh, J. Vía, and I. Santamaría, "A sliding-window kernel RLS algorithm and its application to nonlinear channel identification," in *IEEE Int. Conf. on Acoustics, Speech, and Sig. Proc. (ICASSP)*, Toulouse, France, May 2006.

[13] S. Van Vaerenbergh, J. Vía, and I. Santamaría, "Nonlinear system identification using a new sliding-window kernel RLS algorithm," *Journal of Comm.*, vol. 2, no. 3, pp. 1–8, May 2007.

[14] V. N. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag New York, Inc., 1995.

[15] J. C. Príncipe, W. Liu, and S. Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*, Wiley New York, 2010, in press.