9

Adaptive Kernel Learning for Signal Processing

Adaptive filtering is a central topic in signal processing. An adaptive filter is a filter structure provided with an adaptive algorithm that tunes the transfer function, typically driven by an error signal. Adaptive filters are widely applied in non-stationary environments because they can adapt their transfer function to match the changing parameters of the system that generates the incoming data (Hayes 1996; Widrow *et al.* 1975). They have become ubiquitous in current digital signal processing, mainly due to the increase in computational power and the need to process streamed data. Adaptive filters are now routinely used in all communication applications for channel equalization, array beamforming or echo cancellation, to cite some, and in other areas of signal processing as image processing or medical equipment.

By applying linear adaptive filtering principles in the kernel feature space, powerful nonlinear adaptive filtering algorithms can be obtained. This chapter introduces the wide topic of adaptive signal processing, and it explores the emerging field of kernel adaptive filtering (KAF). Its orientation is different from the preceding ones, as adaptive processing can be used in a variety of scenarios. Attention is paid to kernel LMS/RLS algorithms, to previous taxonomies of adaptive kernel methods, and to emergent kernel methods.

Matlab code snippets are included to illustrate the basic operations of the most common kernel adaptive filters. Tutorial examples are provided on applications including chaotic time-series prediction, respiratory motion prediction, and nonlinear system identification.

9.1 Linear Adaptive Filtering

Let us first define some basic concepts of linear adaptive filtering theory. The goal of adaptive filtering is to model an unknown, possibly time-varying system by observing the inputs and outputs of this system over time. We will denote the input to the system on time instant n as $\mathbf{x}_n$, and its output as d_n, see Fig. 9.1. The input signal $\mathbf{x}_n$ is assumed to be zero-mean. We represent it as a vector, and it will often represent a time-delay vector of L taps of a signal x_n on time instant n as $\mathbf{x}_n = [x_n, x_{n-1}, \dots, x_{n-L+1}]^\top$.

A diagram for a linear adaptive filter is depicted in Fig. 9.2. The input to the adaptive filter on time instant n is $\mathbf{x}_n$, and its response y_n is obtained through the following linear operation:

$$y_n = \boldsymbol{w}_n^H \boldsymbol{x}_n, \tag{9.1}$$

Cite as: Steven Van Vaerenbergh, "Adaptive Kernel Learning for Signal Processing". In J. L. Rojo-Álvarez, M. Martínez-Ramón, J. Muñoz-Marí, G. Camps-Valls (Eds.), *Digital Signal Processing with Kernel Methods*, pp. 387–431,
Wiley-IEEE Press: Hoboken, NJ, USA, 2018.

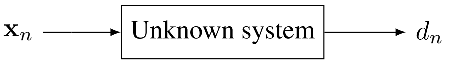

Figure 9.1 An unknown system with input $\boldsymbol{x}_n$ and output d_n at time instant n.

where H is the hermitic operator.

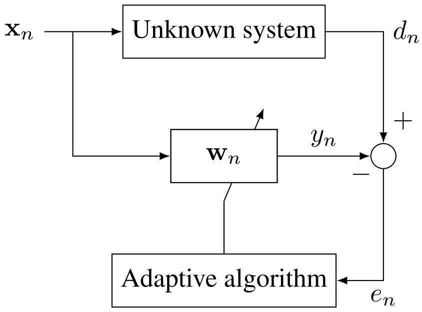

Figure 9.2 A linear adaptive filter for system identification.

Linear adaptive filtering follows the *online learning* framework, which consists of two basic steps that are repeated at each time instant n. First, the online algorithm receives an observation $\boldsymbol{x}_n$ for which it calculates the estimated image y_n, based on its current estimate of $\boldsymbol{w}_n$. Next, the algorithm receives the desired output d_n (also known as *symbol* in communications), which allows it to calculate the estimation error $e_n = d_n - y_n$ and update its estimate for $\boldsymbol{w}_n$. In some situations d_n is known a priori, i.e. the received signal $\boldsymbol{x}_n$ is one of a set of *training* signals provided with known labels. The procedure is then called *supervised*. When d_n belongs to a finite set of quantized labels, and it can be assumed that the error will be likely much smaller than the quantization step or minimum Euclidean distance between labels, the desired label is estimated by quantizing y_n to the closest label. In these cases, the algorithm is called *decision directed*.

9.1.1 LMS Algorithm

The classical adaptive optimization techniques have their roots in the theoretical approach called the steepest-descent algorithm. Assume that the expectation of the squared error signal, $J_n = E\{\|e_n\|^2\}$ can be computed. Since this error is a function of the vector $\boldsymbol{w}_n$, the idea of the algorithm is to modify this vector towards the direction of the steepest descent of J_n. This direction is just opposite to its gradient $\nabla_{\boldsymbol{w}} J_n$. Indeed, assuming complex stationary signals, the error expectation is

$$\begin{aligned} E\left\{|e_n|^2\right\} &= E\left\{|d_n - \boldsymbol{w}_n^H \boldsymbol{x}_n|^2\right\} \\ &= E\left\{|d_n|^2 + \boldsymbol{w}_n^H \boldsymbol{x}_n \boldsymbol{x}_n^H \boldsymbol{w}_n - 2\boldsymbol{w}_n^H \boldsymbol{x}_n d_n^*\right\} \\ &= \sigma_d^2 + \boldsymbol{w}_n^H \mathbf{R}_{xx} \boldsymbol{w}_n - 2\boldsymbol{w}_n^H \mathbf{p}_{xd}, \end{aligned} \tag{9.2}$$

where $\mathbf{R}_{xx}$ is the signal autocorrelation matrix, $\mathbf{p}_{xd}$ is the cross-correlation vector between the signal and the filter output, and σ_d^2 is the variance of the system output. Its gradient with respect to vector $\boldsymbol{w}_n$ is expressed as

$$\nabla_{\boldsymbol{w}} J_n = 2\mathbf{R}_{xx}\boldsymbol{w}_n - 2\mathbf{p}_{xd}. \tag{9.3}$$

The adaptation rule based on steepest descent thus becomes

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \eta \nabla_{\boldsymbol{w}} J_n, \tag{9.4}$$

where η is the *step size* or *learning rate* of the algorithm.

The Least Mean Squares (LMS) algorithm, introduced in 1960 by Widrow (Widrow *et al.* 1975), is a very simple and elegant method of training a linear adaptive system to minimize the mean square error that approximates the gradient $\nabla_{\boldsymbol{w}} J_n$ using an instantaneous estimate of the gradient. From Eq. (9.3), an approximation can be written as

$$\nabla_{\boldsymbol{w}} J_n \approx 2\boldsymbol{x}_n \boldsymbol{x}_n^H \boldsymbol{w}_n - 2\boldsymbol{x}_n d_n^* \tag{9.5}$$

Using this approximation in Eq. (9.4) leads to the well-known stochastic gradient descent update rule which is the core of the LMS algorithm:

$$\begin{aligned} \boldsymbol{w}_{n+1} &= \boldsymbol{w}_n - \eta \boldsymbol{x}_n \left(\boldsymbol{x}_n^H \boldsymbol{w}_n - d_n^* \right) \\ &= \boldsymbol{w}_n + \eta \boldsymbol{x}_n e_n^*. \end{aligned} \tag{9.6}$$

This optimization procedure is also the basis for tuning nonlinear filter structures such as neural networks Haykin (2001) and some of the kernel-based adaptive filters discussed later in this chapter. A detailed analysis including that of convergence and misadjustment is given in (Haykin 2001). The Matlab code for the LMS training step on a new data pair $(\boldsymbol{x}, d)$ is displayed in Listing 9.1.

```
y = x'*w; % evaluate filter output
err = d - y; % instantaneous error

w = w + mu*x*err'; % update filter coefficients
```

Listing 9.1 Training step of the LMS algorithm on a new datum $(\boldsymbol{x}, d)$.

Under the stationarity assumption, the LMS algorithm converges to the Wiener solution in mean, but the weight vector $\boldsymbol{w}_n$ shows a variance that converges to a value that is a function of η. Therefore, low variances are only achieved at low adaptation speed. A more sophisticated approach with faster convergence is found in the the Recursive Least-Squares (RLS) algorithm.

9.1.2 RLS Algorithm

The RLS algorithm was first introduced by Plackett in 1950 (Plackett 1950). In a stationary scenario, it converges to the Wiener solution in mean and variance, improving also the slow rate of adaptation of the LMS algorithm. Nevertheless, this gain in convergence speed comes at the price of a higher complexity, as we will see below.

Recursive update

The basis of the RLS algorithm consists of recursively updating the vector $\boldsymbol{w}$ that minimizes a regularized version of the cost function J_n

$$J_n = \sum_{i=1}^{n} |d_i - \boldsymbol{x}_i^H \boldsymbol{w}|^2 + \delta \boldsymbol{w}^H \boldsymbol{w}, \tag{9.7}$$

where δ is a positive constant *regularization factor*. The regularization factor penalizes the squared norm of the solution vector so that this solution does not apply to much weight to any specific dimension[1]. The solution that minimizes the least-squares cost function (9.7) is well known and given by

$$\boldsymbol{w} = (\mathbf{R}_{xx} + \delta\mathbf{I})^{-1}\mathbf{p}_{xd}. \tag{9.8}$$

The regularization δ guarantees that the inverse in Eq. (9.8) exists. In the absence of regularization, i.e. for $\delta = 0$, the solution requires to invert the matrix $\mathbf{R}_{xx}$, which may be rank-deficient.

For a detailed derivation of the RLS algorithm we refer the reader to Haykin (2001); Sayed (2003). In the sequel, we will provide its update equations and a short discussion of its properties compared to LMS.

We denote the autocorrelation matrix for the data $\boldsymbol{x}_1$ till $\boldsymbol{x}_n$ as $\mathbf{R}_n$,

$$\mathbf{R}_n = \sum_{i=1}^{n} \boldsymbol{x}_i^H \boldsymbol{x}_i. \tag{9.9}$$

RLS requires the introduction of an *inverse autocorrelation matrix* $\mathbf{P}_n$, defined as

$$\mathbf{P}_n = (\mathbf{R}_n + \delta\mathbf{I})^{-1}. \tag{9.10}$$

At step $n-1$ of the recursion, the algorithm has processed $n-1$ data, and its estimate $\boldsymbol{w}_{n-1}$ is the optimal solution for minimizing the squared cost function (9.7) at time step $n-1$. When a new datum $\boldsymbol{x}_n$ is obtained, the inverse autocorrelation matrix is updated as

$$\mathbf{P}_n = \mathbf{P}_{n-1} - \mathbf{g}_n\mathbf{g}_n^H\mathbf{P}_{n-1}, \tag{9.11}$$

where $\mathbf{g}_n$ is the *gain vector* of the RLS algorithm,

$$\mathbf{g}_n = \frac{\mathbf{P}_{n-1}}{1 + \boldsymbol{x}_n^H\mathbf{P}_{n-1}\boldsymbol{x}_n}. \tag{9.12}$$

The update of the solution itself reads

$$\boldsymbol{w}_n = \boldsymbol{w}_{n-1} + \mathbf{g}_n e_n, \tag{9.13}$$

in which e_n represents the usual prediction error $e_n = d_n - \boldsymbol{x}_n^H\boldsymbol{w}_{n-1}$. The RLS algorithm starts by initializing its solution to

$$\boldsymbol{w}_0 = 0, \tag{9.14}$$

and the estimate of the inverse autocorrelation matrix $\mathbf{P}_n$ to

$$\mathbf{P}_0 = \delta^{-1}\mathbf{I}. \tag{9.15}$$

Then, it recursively updates its solution by including one datum $\boldsymbol{x}_i$ at a time and performing the calculations from Eqs. 9.11 to (9.13).

Due to the matrix multiplications involved in the RLS updates, the computational complexity per time step for RLS is quadratic in terms of the data dimension, $\mathcal{O}(L^2)$, while the LMS algorithm has only linear complexity, $\mathcal{O}(L)$.

---

[1] A slightly more general formulation involves a *regularization matrix* which can penalize the individual elements of the solution differently Sayed (2003).

Exponentially-weighted RLS

The RLS algorithm takes into account all previous data when it updates its solution with a new datum. This kind of update yields a faster convergence than LMS, which guides its update based only on the performance on the newest datum. Nevertheless, by guaranteeing that its solution is valid for all previous data, the RLS algorithm is in essence looking for a stationary solution, and thus it cannot adapt to nonstationary scenarios, where a tracking algorithm is required. LMS, on the other hand, deals correctly with nonstationary scenarios, thanks to the instantaneous nature of its update which forgets older data and only adapts to the newest datum.

A tracking version of RLS can be obtained by including a *forgetting factor* $\lambda \in (0, 1]$ in its cost function, as follows

$$J_n = \sum_{i=1}^{n} \lambda^{n-i} \|d_i - \boldsymbol{x}_i^H \boldsymbol{w}\|^2 + \lambda^n \delta \boldsymbol{w}^H \boldsymbol{w}. \tag{9.16}$$

The resulting algorithm is called exponentially-weighted RLS Haykin (2001); Sayed (2003). The inclusion of the forgetting factor assigns lower weights to older data, which allows the algorithm to adapt gradually to changes. The update for the inverse autocorrelation matrix becomes

$$\mathbf{P}_n = \lambda^{-1}\mathbf{P}_{n-1} - \lambda^{-1}\mathbf{g}_n \boldsymbol{x}_n^H \mathbf{P}_{n-1}, \tag{9.17}$$

and the new gain vector becomes

$$\mathbf{g}_n = \frac{\lambda^{-1}\mathbf{P}_{n-1}}{1 + \lambda^{-1}\boldsymbol{x}_n^H \mathbf{P}_{n-1} \boldsymbol{x}_n}. \tag{9.18}$$

The Matlab code for the training step of the exponentially-weighted RLS algorithm is displayed in Listing 9.2.

```
y = x'*w; % evaluate filter output
err = d - y; % instantaneous error

g = P*x/(lambda+x'*P*x); % gain vector
w = w + g*err; % update filter coefficients
P = lambda\(P - g*x'*P); % update inverse autocorrelation matrix
```

Listing 9.2 Training step of the exponentially-weighted RLS algorithm on a new datum $(\boldsymbol{x}, d)$.

Recursive estimation algorithms play a crucial role for many problems in adaptive control, adaptive signal processing, system identification, and general model building and monitoring problems Ljung (1999). In the signal processing literature, great attention has been paid to their efficient implementation. Linear autoregressive models require relatively few parameters and allow closed-form analysis, while ladder or lattice implementation of linear filters has long been studied in signal theory. However, when the system generating the data is driven by nonlinear dynamics, the model specification and parameter estimation problems increase their complexity, and hence nonlinear adaptive filtering becomes strictly necessary.

Note that, in the field of control theory, a range of sequential algorithms for nonlinear filtering have been proposed since the sixties, notably the extended Kalman filter Lewis *et al.* (2007) and the unscented Kalman filter Julier and Uhlmann (1996), which are both nonlinear extensions of the celebrated Kalman filter Kalman (1960), and particle filters Del Moral (1996). These methods generally require knowledge of a state-space model, and while some of them are related to adaptive filtering, we will not deal with them explicitly in this chapter.

9.2 Kernel Adaptive Filtering

The nonlinear filtering problem and the online adaptation of model weights were first addressed by neural networks in the nineties Dorffner (1996); Narendra and Parthasarathy (1990). Throughout the last decade, a great interest has been devoted to developing nonlinear versions of linear adaptive filters by means of kernels (Liu *et al.* 2010). The goal is to develop machines that learn over time in changing environments, and at the same time adopt the nice characteristics of convexity, convergence and reasonable computational complexity, which was not successfully implemented in neural networks.

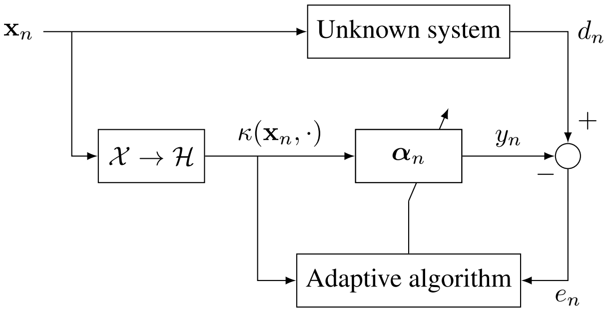

Figure 9.3 A kernel adaptive filter for nonlinear system identification.

Kernel adaptive filtering aims to formulate the classic linear adaptive filters in rkHs, such that a series of convex least-squares problems is solved. Several basic kernel adaptive filters can be obtained by applying a linear adaptive filter directly on the transformed data, as illustrated in Fig. 9.3. This requires the reformulation of scalar-product based operations in terms of *kernel evaluations*. The resulting algorithms typically consist of algebraically simple expressions, though they feature powerful nonlinear filtering capabilities. Nevertheless, the design of these online kernel methods requires to deal with some of the challenges that typically arise when dealing with kernels, such as overfitting and computational complexity issues.

In the sequel we will discuss two families of kernel adaptive filters in detail, namely kernel least mean squares and kernel recursive least-squares algorithms. Several related kernel adaptive filters will be reviewed briefly as well.

9.3 Kernel Least Mean Squares

The early approach to kernel adaptive filtering introduced a kernel version of the celebrated ADALINE in (Frieß and Harrison 1999), though this method was not online. Kivinen et al. proposed an algorithm to perform stochastic gradient descent in rkHs Kivinen *et al.* (2004): The so-called Naive Online regularized Risk Minimization Algorithm (NORMA) introduces a regularized risk that can be solved online and can be shown to be equivalent to a kernel version of leaky LMS, which itself is a regularized version of LMS.

9.3.1 Derivation of KLMS

As an illustrative guiding example of a kernel adaptive filter, we will take the kernelization of the standard LMS algorithm, known as Kernel Least Mean Squares (KLMS) (Liu *et al.*

2008). The approach employs the traditional kernel trick. Essentially, a nonlinear function $\boldsymbol{\phi}(\cdot)$ maps the data $\boldsymbol{x}_n$ from the input space to $\boldsymbol{\phi}(\boldsymbol{x}_n)$ in the feature space. Let $\boldsymbol{w}_{\mathcal{H}}$ be the weight vector in this space such that the filter output is $y_n = \boldsymbol{w}_{\mathcal{H},n}^{\top}\boldsymbol{x}_n$, where $\boldsymbol{w}_{\mathcal{H},n}$ is the estimate of $\boldsymbol{w}_{\mathcal{H}}$ at time instant n. Note that we will be taking scalar products of real-valued vectors from now on. Given a desired response d_n we wish to minimize squared loss, $J_{\boldsymbol{w}_{\mathcal{H},n}}$, with respect to $\boldsymbol{w}_{\mathcal{H}}$. Similar to Eq. (9.6), the obtained stochastic gradient descent update rule reads

$$\boldsymbol{w}_{\mathcal{H},n} = \boldsymbol{w}_{\mathcal{H},n-1} + \eta e_n \boldsymbol{\phi}(\boldsymbol{x}_n). \tag{9.19}$$

By initializing the solution as $\boldsymbol{w}_{\mathcal{H},0} = 0$ (and hence $e_0 = d_0 = 0$), the solution after n iterations can be expressed in closed form as

$$\boldsymbol{w}_{\mathcal{H},n} = \eta \sum_{i=1}^{n} e_i \boldsymbol{\phi}(\boldsymbol{x}_i). \tag{9.20}$$

By exploiting the kernel trick, one obtains the prediction function

$$y_* = \eta \sum_{i=1}^{n} e_i \boldsymbol{\phi}(\boldsymbol{x}_i)\boldsymbol{\phi}(\boldsymbol{x}_*) = \eta \sum_{i=1}^{n} e_i k(\boldsymbol{x}_i, \boldsymbol{x}_*), \tag{9.21}$$

where $\boldsymbol{x}_*$ represents an arbitrary input datum and $k(\cdot,\cdot)$ is the kernel function, for instance the commonly used Gaussian kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/2\sigma_k)$ with *kernel width* σ_k. Note that the weights $\boldsymbol{w}_{\mathcal{H},n}$ of the nonlinear filter are not used explicitly in the KLMS algorithm. Also, since the present output is determined solely by previous inputs and all the previous errors, it can be readily computed in the input space. These error samples are similar to innovation terms in sequential state estimation (Haykin 2001), since they add new information to improve the output estimate. Each new input sample results in an output, and hence a corresponding error, which is never modified further and incorporated in the estimate of the next output. This recursive computation makes KLMS especially useful for online (adaptive) nonlinear signal processing.

In Liu *et al.* (2008) it was shown that the KLMS algorithm is well-posed in rkHs without the need of an extra regularization term in the finite training data case, because the solution is always forced to lie in the subspace spanned by the input data. The lack of an explicit regularization term leads to two important advantages. First of all, it has a simpler implementation than NORMA, as the update equations are straightforward kernel versions of the original linear ones. Second, it can potentially provide better results because regularization biases the optimal solution. In particular, it was shown that a small enough step-size can provide a sufficient "self-regularization" mechanism. Moreover, since the space spanned by the mapped samples is possibly infinite-dimensional, the projection error of the desired signal d_n could be very small, as is well known from Cover's theorem Haykin (1999). On the downside, the speed of convergence and the misadjustment also depend upon the step-size. As a consequence, they conflict with the generalization ability.

9.3.2 Implementation challenges and dual formulation

Another important drawback of the KLMS algorithm becomes apparent when analyzing its update Eq. (9.21). In order to make a prediction, the algorithm requires to store all previous errors e_i and all processed input data $\boldsymbol{x}_i$, for $i = 1, 2, \ldots, n$. In online scenarios where data is continuously being received, the size of the KLMS network will continuously grow, posing implementation challenges. This becomes even more evident if we recast the weight update from Eq. (9.19) into a more standard filtering formulation, by relying on the Representer theorem Schölkopf *et al.* (2001). This theorem states that the solution $\boldsymbol{w}_{\mathcal{H},n}$ can be expressed as a linear combination of the transformed input data,

$$\boldsymbol{w}_{\mathcal{H},n} = \sum_{i=1}^{n} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i). \tag{9.22}$$

This allows the prediction function to be written as the familiar kernel expansion

$$y_* = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_*). \tag{9.23}$$

The expansion coefficients α_i are called the *dual variables* and the reformulation of the filtering problem in terms of α_i is called the *dual formulation*. The update from Eq. (9.19) now becomes

$$\sum_{i=1}^{n} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i) = \sum_{i=1}^{n-1} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i) + \eta e_n \boldsymbol{\phi}(\boldsymbol{x}_n), \tag{9.24}$$

and after multiplying both sides with the new datum $\boldsymbol{\phi}(\boldsymbol{x}_n)$ and adopting a vector notation, we obtain

$$\boldsymbol{\alpha}_n^\top \boldsymbol{k}_n = \boldsymbol{\alpha}_{n-1}^\top \boldsymbol{k}_{n-1} + \eta e_n k_{n,n}, \tag{9.25}$$

where $\boldsymbol{\alpha}_n = [\alpha_1, \alpha_2, \ldots, \alpha_n]^\top$, the vector $\boldsymbol{k}_n$ contains the kernels of the n data and the newest point, $\boldsymbol{k}_n = [k(\boldsymbol{x}_1, \boldsymbol{x}_n), k(\boldsymbol{x}_2, \boldsymbol{x}_n), \ldots, k(\boldsymbol{x}_n, \boldsymbol{x}_n)]$, and $k_{n,n} = k(\boldsymbol{x}_n, \boldsymbol{x}_n)$. KLMS resolves this relationship by updating $\boldsymbol{\alpha}_n$ as

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ \eta e_n \end{bmatrix}. \tag{9.26}$$

The Matlab code for the complete KLMS training step on a new data pair $(\boldsymbol{x}, d)$ is displayed in Listing 9.3.

```
k = kernel(dict,x,kerneltype,kernelpar); % kernels between dictionary and x
y = k'*alpha; % evaluate function output
err = d - y; % instantaneous error

kaf.dict = [kaf.dict; x]; % add base to dictionary
kaf.alpha = [kaf.alpha; kaf.eta*err]; % add new coefficient
```

Listing 9.3 Training step of the KLMS algorithm on a new datum $(\boldsymbol{x}, d)$.

The update (9.26) emphasizes the growing nature of the KLMS network, which precludes its direct implementation in practice. In order to design a practical KLMS algorithm, the number of terms in the kernel expansion (9.23) should stop growing over time. This can be achieved by implementing an *online sparsification technique*, whose aim is to identify terms in the kernel expansion that can be omitted without degrading the solution. We will discuss several different sparsification approaches in Section 9.5.

Finally, observe that the computational complexity and memory complexity of the KLMS algorithm are both linear in terms of the number of data it stores, $\mathcal{O}(n)$. Recall that the complexity of the LMS algorithm is also linear, though not in terms of the number of data but in terms of the data *dimension*.

9.3.3 Example: Prediction of the Mackey-Glass time series

We demonstrate the online learning capabilities of the KLMS kernel adaptive filter by predicting the Mackey-Glass time series, which is a classic benchmark problem Liu *et al.* (2010). The Mackey-Glass time series is well-known for its strong non-linearity. It corresponds to a high-dimensional chaotic system, and its output is generated by the following time-delay differential equation:

$$\frac{dx_n}{dn} = -bx_n + \frac{ax_{n-\Delta}}{1 + x_{n-\Delta}^{10}}. \tag{9.27}$$

We focus on the sequence with parameters $b = 0.1$, $a = 0.2$ and time delay $\Delta = 30$, better known as the MG30 time series. The prediction problem consists in predicting the n-th sample, given all samples of the time series up till the $n - 1$-th sample.

Time-series prediction with kernel adaptive filters is typically performed by considering a time-delay vector $\boldsymbol{x}_n = [x_n, x_{n-1}, \ldots, x_{n-L+1}]^\top$ as the input and the next sample of the time series as the desired output, $d_n = x_{n+1}$. This approach casts the prediction problem into the well-know filtering framework[2]. Prediction of several steps ahead can be obtained by choosing a prediction horizon $h > 1$, and $d_n = x_{n+h}$. For time series generated by a deterministic process, a principled tool to find the optimal embedding is Takens' theorem Takens (1981). In the case of the MG30 time series, Takens' theorem indicates that the optimal embedding is around $L = 7$ Van Vaerenbergh *et al.* (2012a).

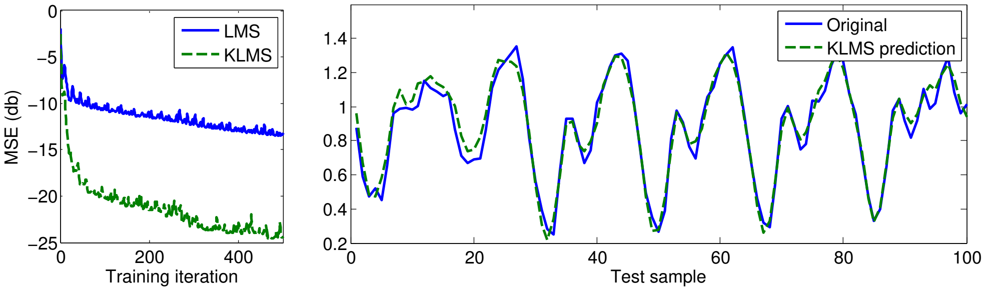

Figure 9.4 KLMS predictions on the Mackey-Glass time series. Left: Learning curve over 500 training iterations. Right: Test samples of the Mackey-Glass time-series and the predictions provided by KLMS.

We consider 500 samples for online training of the KLMS algorithm and use the next 100 data for testing. The step size of KLMS is fixed to 0.2, and we use the Gaussian kernel with $\sigma = 1$. Fig. 9.4 displays the prediction results after 500 training steps. The left plot shows the learning curve of the algorithm, obtained as the mean squared error (MSE) of the prediction on the test set, at each iteration of the online training process. As a reference, we include the learning curve of the linear LMS algorithm with a suitable step size. The right plot shows the 100 test samples of the original time series, as the full line, and KLMS' prediction on these test samples after 500 training steps. These predictions are calculated by evaluating the prediction equation (9.21) on the test samples. The code for this experiment and all subsequent ones is included in the accompanying material.

---

[2] Note that, recently, some authors have proposed to model time-series through a different approach based on explicit recursivity in the rKHs Li and Príncipe (2016); Tuia *et al.* (2014), as we will see later on in this chapter.

9.3.4 Practical KLMS algorithms

In the described Mackey-Glass experiment, the KLMS algorithm requires to store 500 coefficients α_i and the 500 corresponding data $\boldsymbol{x}_i$. The stored data $\boldsymbol{x}_i$ are referred to as its *dictionary*. If the online learning process were to continue indefinitely, the algorithm would require *evergrowing* memory and computation per time step. This issue has been identified as a major roadblock early on in the research on kernel adaptive filters, and it has led to the development of several procedures to slow down the dictionary growth by *sparsifying* the dictionary.

A sparsification procedure based on Gaussian elimination steps on the gram matrix was proposed in (Pokharel *et al.* 2009). This method is successful in limiting the dictionary size in the n-th training step to some $m < n$, but in order to do so it requires $\mathcal{O}(m^2)$ complexity, which defeats the purpose of using a KLMS algorithm.

Kernel normalized least-mean squares and the coherence criterion

Around the same time the KLMS algorithm was published, a kernelized version of the Affine Projection (AP) algorithm was proposed Richard *et al.* (2009). AP algorithms hold the middle ground between LMS and RLS algorithms by calculating an estimate of the correlation matrix based on the p last data. For $p = 1$ the algorithm reduces to a kernel version of the normalized LMS algorithm Haykin (2001), called Kernel Normalized Least-Mean Squares (KNLMS), and its update reads

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ 0 \end{bmatrix} + \frac{\eta}{\epsilon + \|\boldsymbol{k}_n\|} e_n \boldsymbol{k}_n. \tag{9.28}$$

Note that this algorithm updates all coefficients in each iteration, in contrast to KLMS which updates just one coefficient.

The kernel affine projection (KAP) and KNLMS algorithms introduced in Richard *et al.* (2009) also included an efficient dictionary sparsification procedure, called the *coherence criterion*. Coherence is a measure to characterize a dictionary in sparse approximation problems, defined in a kernel context as

$$\mu = \max_{i \neq j} |k(\boldsymbol{x}_i, \boldsymbol{x}_j)|. \tag{9.29}$$

The coherence of a dictionary will be large if it contains two bases $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ that are very similar, in terms of the kernel function. Due to their similarity, such bases contribute almost identical information to the algorithm, and one of them may be considered redundant. The online dictionary sparsification procedure based on coherence operates by only including a new datum $\boldsymbol{x}_n$ into the current dictionary $\mathcal{D}_{n-1}$ if it maintains the dictionary coherence below a certain threshold,

$$\max_{j \in \mathcal{D}_{n-1}} |k(\boldsymbol{x}_n, \boldsymbol{x}_j)| < \mu_0. \tag{9.30}$$

If the new datum fulfills this criterion, it is included in the dictionary, and the KNLMS coefficients are updated through Eq. (9.28). If the coherence criterion (9.30) is not fulfilled, the new datum is not included in the dictionary, and a *reduced* update of the KNLMS coefficients is performed,

$$\boldsymbol{\alpha}_n = \boldsymbol{\alpha}_{n-1} + \frac{\eta}{\epsilon + \|\boldsymbol{k}_n\|} e_n \boldsymbol{k}_n. \tag{9.31}$$

This update does not increase the number of coefficients and therefore it maintains the algorithm's computational complexity fixed during that iteration. The Matlab code for the complete KNLMS training step on a new data pair $(\boldsymbol{x}, d)$ is displayed in Listing 9.4.

```
k = kernel(dict,x,kerneltype,kernelpar); % kernels between dictionary and x
if (max(k) <= mu0), % coherence criterion
    dict = [dict; x]; % add base to dictionary
    alpha = [alpha; 0]; % reserve spot for new coefficient
end

k = kernel(dict,x,kerneltype,kernelpar); % kernels with new dictionary
y = k'*alpha; % evaluate function output
err = d - y; % instantaneous error

alpha = alpha + eta/(eps + k'*k)*err*k; % update coefficients
```

Listing 9.4 Training step of the KNLMS algorithm on a new datum $(\boldsymbol{x}, d)$.

The coherence criterion is computationally efficient in that it has a complexity that does not exceed that of the kernel adaptive filter itself, and it has demonstrated to be successful in practical situations Van Vaerenbergh and Santamaría (2013).

Quantized kernel least-mean squares

Recently, a kernel LMS algorithm was proposed that combines elements from the original KLMS algorithm and the coherence criterion, called Quantized Kernel Least Mean Squares (QKLMS) Chen *et al.* (2012). In particular, when the sparsification criterion decides to include a datum into the dictionary, the algorithm updates its coefficients as follows:

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ \eta e_n \end{bmatrix}. \tag{9.32}$$

When the datum does not fulfil the coherence criterion, it is not included in the dictionary. Instead, the closest dictionary element is retrieved, and the corresponding coefficient is updated as follows

$$\alpha_{n,j} = \alpha_{n-1,j} + \eta e_n, \tag{9.33}$$

where j is the dictionary index of the element that is closest. Though conceptually very simple, this algorithm obtains state-of-the-art results in several applications when only a low computational budget is available. The Matlab code for the complete QKLMS training step on a new data pair $(\boldsymbol{x}, d)$ is displayed in Listing 9.5.

```
k = kernel(dict,x,kerneltype,kernelpar); % kernels between dictionary and x
y = k'*alpha; % evaluate function output
err = d - y; % instantaneous error

[d2,j] = min(sum((dict - repmat(x,m,1)).^2,2)); % distance to dictionary
if d2 <= epsu^2,
    alpha(j) = alpha(j) + eta*err; % reduced coefficient update
else
    dict = [dict; x]; % add base to dictionary
    alpha = [alpha; eta*err]; % add new coefficient
end
```

Listing 9.5 Training step of the QKLMS algorithm on a new datum $(\boldsymbol{x}, d)$.

9.4 Kernel recursive least squares

In linear adaptive filtering, the RLS algorithm represents an alternative to LMS, with faster convergence and typically lower bias, at the expense of a higher computational complexity.

RLS is obtained by designing a recursive solution to the least-squares problem. Analogously, a recursive solution can be designed for the kernel ridge regression problem, yielding kernel recursive least-squares (KRLS) algorithms.

9.4.1 Kernel ridge regression

In order to obtain the kernel-based version of the regularized least-squares cost function (9.7), we first transform the data into the kernel feature space,

$$\begin{aligned} J_n &= \sum_{i=1}^{n} |d_i - \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}_{\mathcal{H}}|^2 + \delta \boldsymbol{w}_{\mathcal{H}}^\top \boldsymbol{w}_{\mathcal{H}} \\ &= \|\mathbf{d} - \mathbf{K}\boldsymbol{\alpha}\|^2 + \delta \boldsymbol{\alpha}^\top \mathbf{K} \boldsymbol{\alpha}, \end{aligned} \tag{9.34}$$

where we have applied the kernel trick to obtain the second equality. Here, vector $\mathbf{d}$ contains the n desired values, $\mathbf{d} = [d_1, d_2, \ldots, d_n]^\top$, and $\mathbf{K}$ is the kernel matrix with elements $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Eq. (9.34) represents the *kernel ridge regression* problem Saunders *et al.* (1998), and its solution is given by

$$\boldsymbol{\alpha} = (\mathbf{K} + \delta \mathbf{I})^{-1} \mathbf{d}. \tag{9.35}$$

The prediction for a new datum $\boldsymbol{x}_*$ is obtained as

$$y_* = \boldsymbol{k}_*^\top \boldsymbol{\alpha} = \boldsymbol{k}_*^\top (\mathbf{K} + \delta \mathbf{I})^{-1} \mathbf{d}. \tag{9.36}$$

9.4.2 Derivation of KRLS

The KRLS algorithm Engel *et al.* (2004) formulates a recursive procedure to obtain the solution of the regression problem (9.34) in the absence of regularization, $\delta = 0$. Without regularization, the solution (9.35) reads

$$\boldsymbol{\alpha} = \mathbf{K}^{-1} \mathbf{d}. \tag{9.37}$$

KRLS guarantees the invertibility of the kernel matrix $\mathbf{K}$ by excluding those data $\boldsymbol{x}_i$ from from the dictionary that are linearly dependent on the already included data, in the feature space. As we will see, this is achieved by applying a specific online sparsification procedure, which guarantees both that $\mathbf{K}$ is invertible and that the algorithm's dictionary stays compact.

Assume the solution after processing $n-1$ data is available, given by

$$\boldsymbol{\alpha}_{n-1} = \mathbf{K}_{n-1}^{-1} \mathbf{d}_{n-1}, \tag{9.38}$$

In the next iteration, n, a new data pair $(\boldsymbol{x}_n, d_n)$ is received and we wish to obtain the new solution $\boldsymbol{\alpha}_n$ by applying a low-complexity update on the previous solution (9.38). We first calculate the predicted output

$$y_n = \boldsymbol{k}_n^\top \boldsymbol{\alpha}_{n-1}, \tag{9.39}$$

and we obtain the a-priori error for this datum, $e_n = d_n - y_n$. The updated kernel matrix can be written as

$$\mathbf{K}_n = \begin{bmatrix} \mathbf{K}_{n-1} & \boldsymbol{k}_n \\ \boldsymbol{k}_n^\top & k_{nn} \end{bmatrix}. \tag{9.40}$$

By introducing the variables

$$\mathbf{a}_n = \mathbf{K}_{n-1}^{-1}\boldsymbol{k}_n, \tag{9.41}$$

and

$$\gamma_n = k_{nn} - \boldsymbol{k}_n^\top \mathbf{a}_n, \tag{9.42}$$

the update for the inverse kernel matrix can be written as

$$\mathbf{K}_n^{-1} = \frac{1}{\gamma_n}\begin{bmatrix} \gamma_n \mathbf{K}_{n-1}^{-1} + \mathbf{a}_n\mathbf{a}_n^\top & -\mathbf{a}_n \\ -\mathbf{a}_n & 1 \end{bmatrix}. \tag{9.43}$$

Eq. (9.43) is obtained by applying the Sherman-Morrison-Woodbury formula for matrix inversion, see for instance Golub and Van Loan (2012). Finally, the updated solution $\boldsymbol{\alpha}_n$ is obtained as

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ 0 \end{bmatrix} - e_n/\gamma_n \begin{bmatrix} \mathbf{a}_n \\ -1 \end{bmatrix}. \tag{9.44}$$

Equations (9.43) and (9.44) are efficient updates that allow to obtain the new solution in $\mathcal{O}(n^2)$ time and memory, based on the previous solution. Directly applying Eq. (9.37) at iteration n would require $\mathcal{O}(n^3)$ cost, so the recursive procedure is preferred in online scenarios. A detailed derivation of this result can be found in Engel *et al.* (2004); Van Vaerenbergh *et al.* (2012b).

Online sparsification by approximate linear dependency

The KRLS algorithm from Engel *et al.* (2004) follows the described recursive solution. In order to slow down the dictionary growth, shown in Eqs. (9.43) and (9.44), it introduces a sparsification criterion based on *approximate linear dependency* (ALD). According to this criterion, a new datum $\boldsymbol{x}_n$ should only be included in the dictionary if $\boldsymbol{\phi}(\boldsymbol{x}_n)$ cannot be approximated sufficiently well in feature space by a *linear combination* of the already present data.

Given a dictionary $\mathcal{D}$ of data $\boldsymbol{x}_j$ and a new training point $\boldsymbol{x}_n$, we need to find a set of coefficients $\mathbf{a} = [a_1, a_2, \ldots, a_m]^\top$ that satisfy the approximate linear dependency condition

$$\min_{\mathbf{a}} \left\| \sum_{j=1}^{m} a_j \boldsymbol{\phi}(\boldsymbol{x}_j) - \boldsymbol{\phi}(\boldsymbol{x}_n) \right\|^2 \leq \nu \tag{9.45}$$

where m is the cardinality of the dictionary. Interestingly, it can be shown that these coefficients are already calculated by the KRLS update itself, and they are available at each iteration n as $\mathbf{a}_n = \mathbf{K}_{n-1}^{-1}\boldsymbol{k}_n$, see Eq. (9.41). The ALD condition can therefore be verified by simply comparing γ_n to the ALD threshold,

$$\gamma_n = k_{nn} - \boldsymbol{k}_n^\top \mathbf{a}_n \leq \nu. \tag{9.46}$$

If $\gamma_n > \nu$, then we must add the newest datum $\boldsymbol{x}_n$ to the dictionary, $\mathcal{D}_n = \mathcal{D}_{n-1} \cup \{\boldsymbol{x}_n\}$, before updating the solution through Eq.(9.44). If $\gamma_n \leq \nu$ then the datum $\boldsymbol{x}_n$ is already represented sufficiently well by the dictionary. In this case the dictionary is not expanded, $\mathcal{D}_n = \mathcal{D}_{n-1}$, and a *reduced* update of the solution is performed, see Engel *et al.* (2004) for details. The Matlab code for the complete KRLS training step on a new data pair $(\boldsymbol{x}, d)$ is displayed in Listing 9.6.

```
k = kernel(dict,x,kerneltype,kernelpar); % kernels between dictionary and x
kxx = kernel(x,x,kaf.kerneltype,kaf.kernelpar); % kernel on x

a = Kinv*k; % coefficients of closest linear combination in feature space
gamma = kxx - k'*a; % residual of linear approximation in feature space

y = k'*alpha; % evaluate function output
err = d - y; % instantaneous error

if gamma>nu % new datum is not approximately linear dependent
    dict = [dict; x]; % add base to dictionary
    Kinv = 1/gamma*[gamma*Kinv+a*a',-a;-a',1]; % update inv. kernel matrix
    Z = zeros(size(P,1),1);
    P = [P Z; Z' 1]; % add linear combination coeff. to projection matrix
    ode = 1/gamma*err;
    alpha = [alpha - a*ode; ode]; % full update of coefficients
else % perform reduced update of alpha
    q = P*a/(1+a'*P*a);
    P = P - q*(a'*P); % update projection matrix
    alpha = alpha + Kinv*q*err; % reduced update of coefficients
end
```

Listing 9.6 Training step of the KRLS algorithm on a new datum $(\boldsymbol{x}, d)$.

9.4.3 Prediction of the Mackey-Glass time series with KRLS

The update equations for KRLS require substantially more computation than KLMS. In particular, KRLS has quadratic complexity, $\mathcal{O}(m^2)$, in terms of its dictionary size, and KLMS has linear complexity, $\mathcal{O}(m)$. On the other hand, KRLS has faster convergence and lower bias. We illustrate these properties by applying KRLS on the Mackey-Glass prediction experiment from Section 9.3.3.

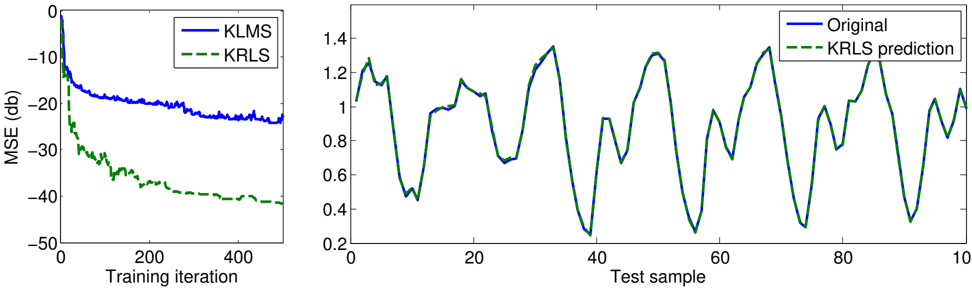

Figure 9.5 KRLS predictions on the Mackey-Glass time series. Left: Learning curve over 500 training iterations, including comparison to KLMS. Right: test samples and KRLS predictions.

Fig. 9.5 shows the results of training the KRLS algorithm on the Mackey-Glass time series. KRLS is applied with a Gaussian kernel with $\sigma_k = 1$, and its precision parameter was fixed to $\nu = 10^{-4}$. The left plot compares the learning curves of KLMS and KRLS, demonstrating a slightly faster initial convergence rate for KRLS, after which the algorithm converges to a much lower MSE than KLMS. The low bias is also visible in the right plot, which shows the prediction results on the test data.

9.4.4 Beyond the stationary model

One important limitation of the KRLS algorithm is that it always assumes a stationary model, and therefore it cannot track changes in the true underlying data model. This is a somewhat odd property for an *adaptive* filter, though note that this is also the case for the original RLS algorithm, see Section 9.1.2.

In order to enable tracking and make a truly adaptive KRLS algorithm, several modifications have been presented in the literature. An exponentially-weighted KRLS algorithm was proposed by including a forgetting factor, and an extended KRLS algorithm was designed by assuming a simple state-space model, though both algorithms show numerical instabilities in practice Liu *et al.* (2010). In the sequel we briefly discuss two different approaches that successfully allow KRLS to adapt to changing environments.

Sliding-Window KRLS

The KRLS algorithm summarizes past information into a compact formulation that does not allow easy manipulation. For instance, there does not exist a straightforward manner to include a forgetting factor to exclude the influence of older data.

In Van Vaerenbergh *et al.* (2006), a sliding-window based version of KRLS was proposed, called Sliding-Window Kernel Recursive Least-Squares (SW-KRLS). This algorithm stores a window of the last m data as its dictionary, and once a datum is older than m time steps it is simply discarded. In each step the algorithm adds the new datum and discards the oldest datum, leading to a sliding-window approach. The algorithm stores the inverse regularized kernel matrix, $(\mathbf{K}_n + \delta\mathbf{I})^{-1}$, calculated on its current dictionary, and a vector of the corresponding desired outputs, $\mathbf{d}_n$. By storing these variables it can calculate the solution vector by simply evaluating $\boldsymbol{\alpha}_n = (\mathbf{K}_n + \delta\mathbf{I})^{-1}\mathbf{d}_n$, see Eqs. (9.35) and (9.36).

The inverse kernel matrix is updated in two steps: First, the new datum is added, which requires expanding the matrix with one row and one column. This is carried out by performing the operation from Eq. (9.43), similar as in the KRLS algorithm. Second, the oldest datum is discarded, which requires removing one row and column from the inverse kernel matrix. This can be achieved by writing the kernel matrix and its inverse as follows,

$$\mathbf{K}_{n-1} = \begin{bmatrix} a & \mathbf{b}^T \\ \mathbf{b} & \mathbf{D} \end{bmatrix}, \quad \mathbf{K}_{n-1}^{-1} = \begin{bmatrix} e & \mathbf{f}^T \\ \mathbf{f} & \mathbf{G} \end{bmatrix}, \tag{9.47}$$

after which the inverse (regularized) kernel matrix is found as

$$\mathbf{D}^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e. \tag{9.48}$$

Details can be found in Van Vaerenbergh *et al.* (2006). Figure 9.6 illustrates the kernel matrix updates when using a sliding window, compared to the classical growing-window approach of KRLS. The Matlab code for the SW-KRLS training step on a new data pair $(\boldsymbol{x}, d)$ is displayed in Listing 9.7.

```
dict = [dict; x]; % add base to dictionary
dict_d = [dict_d; d];   % add d to output dictionary
k = kernel(dict,x,kerneltype,kernelpar); % kernels between dictionary and x
Kinv = grow_kernel_matrix(Kinv,k,c); % calculate new inverse kernel matrix

if (size(dict,1) > M) % prune
    dict(1,:) = []; % remove oldest base from dictionary
    dict_d(1) = []; % remove oldest d from output dictionary
```

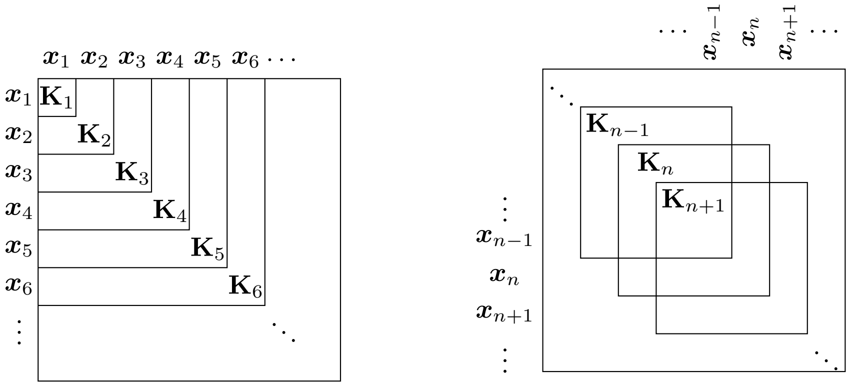

Figure 9.6 Different forms of updating the kernel matrix during online learning. In KRLS-type algorithms the update involves calculating the inverse of each kernel matrix, given the inverse of the previous matrix. Left: Growing kernel matrix, as constructed in KRLS (omitting sparsification for simplicity). Right: Sliding-window kernel matrix of a fixed size, as constructed in SW-KRLS.

```
    Kinv = prune_kernel_matrix(Kinv); % prune inverse kernel matrix
end

alpha = Kinv*dict_d; % obtain new filter coefficients
```

Listing 9.7 Training step of the SW-KRLS algorithm on a new datum $(\boldsymbol{x}, d)$. The functions “grow_kernel_matrix” and “prune_kernel_matrix” implement the operations (9.43) and (9.48).

SW-KRLS is a conceptually very simple algorithm that obtains reasonable performance in a wide range of scenarios, most notably in non-stationary environments. Nevertheless, its performance is limited by the quality of the bases in its dictionary, over which it has little control. In particular, it has no means to avoid redundancy in its dictionary or to maintain older bases that are relevant to its kernel expansion. In order to improve this performance, a Fixed-Budget KRLS (FB-KRLS) algorithm was proposed in Van Vaerenbergh *et al.* (2010). Instead of discarding the oldest data point in each iteration, FB-KRLS discards the data point that causes the least error upon being discarded, using a least a-posteriori error based pruning criterion we will discuss in Section 9.5. In stationary scenarios, FB-KRLS obtains significantly better results.

KRLS Tracker

The tracking limitations of previous KRLS algorithms were overcome by development of the Kernel Recursive Least-Squares Tracker (KRLS-T) algorithm Van Vaerenbergh *et al.* (2012b), which has its roots in the probabilistic theory of Gaussian Process (GP) regression. Similar to the FB-KRLS algorithm, this algorithm uses a fixed memory size and has a criterion to discard which data to discard in each iteration. But unlike FB-KRLS, the KRLS-T algorithm incoporates a forgetting mechanism to gradually downweigh older data.

We will provide more details on this algorithm in the discussion on probabilistic kernel adaptive filtering of Section 9.6. As a reference, we list the Matlab code for the KRLS-T training step on a new data pair $(\boldsymbol{x}, d)$ in Listing 9.8. While this algorithm has similar complexity as other KRLS-type algorithms, its implementation is more complex due its fully probabilistic treatment of the regression problem. Note that some additional checks to avoid numerical problems have been left out; The complete code can be found in the Kernel Adaptive Filtering toolbox, discussed in Section 9.8.

```
% perform one forgetting step
Sigma = lambda*Sigma + (1-lambda)*K; % forgetting on covariance matrix
mu = sqrt(lambda)*mu; % forgetting on mean vector

k = kernel(dict,x,kerneltype,kernelpar); % kernels between dictionary and x
kxx = kernel(x,x,kaf.kerneltype,kaf.kernelpar); % kernel on x

q = Q*k;
y_mean = q'*mu; % predictive mean of new datum
gamma2 = kxx - k'*q; % projection uncertainty
h = Sigma*q;
sf2 = gamma2 + q'*h; % noiseless prediction variance
sy2 = sn2 + sf2; % unscaled predictive variance of new datum
y_var = s02*sy2; % predictive variance of new datum

% include new sample and add a basis
Qold = Q; % old inverse kernel matrix
p = [q; -1];
Q = [Q zeros(m,1);zeros(1,m) 0] + 1/gamma2*(p*p'); % updated inverse matrix

err = d - y_mean; % instantaneous error
p = [h; sf2];
mu = [mu; y_mean] + err/sy2*p; % posterior mean
Sigma = [Sigma h; h' sf2] - 1/sy2*(p*p'); % posterior covariance
dict = [dict; x]; % add base to dictionary

% estimate scaling power s02 via ML
nums02ML = nums02ML + lambda*(y - y_mean)^2/sy2;
dens02ML = dens02ML + lambda;
s02 = nums02ML/dens02ML;

% delete a basis if necessary
m = size(dict,1);
if m>M
    % MSE pruning criterion
    errors = (Q*mu)./diag(Q);
    criterion = abs(errors);

    [~, r] = min(criterion); % remove element which incurs in the min. err.
    smaller = 1:m; smaller(r) = [];

    if r == m, % remove the element we just added (perform reduced update)
        Q = Qold;
    else
        Qs = Q(smaller, r);
        qs = Q(r,r); Q = Q(smaller, smaller);
        Q = Q - (Qs*Qs')/qs; % prune inverse kernel matrix
    end
    mu = mu(smaller); % prune posterior mean
    Sigma = Sigma(smaller, smaller); % prune posterior covariance
    dict = dict(smaller,:); % prune dictionary
end
```

Listing 9.8 Training step of the KRLS-T algorithm on a new datum $(\boldsymbol{x}, d)$.

9.4.5 *Example: Nonlinear channel identification and reconvergence*

In order to demonstrate the tracking capabilities of some of the reviewed kernel adaptive filters we perform an experiment similar to the setup described in Lázaro-Gredilla *et al.* (2011); Van Vaerenbergh *et al.* (2006). Specifically, we consider the problem of online identification of a communication channel in which an abrupt change (switch) is triggered at some point.

A signal $x_n \in \mathcal{N}(0,1)$ is fed into a nonlinear channel that consists of a linear finite impulse response (FIR) channel followed by the nonlinearity $y = \tanh(z)$, where z is the output of the linear channel. During the first 500 iterations the impulse response of the linear channel is chosen as $\mathcal{H}_1 = [1, -0.3817, -0.1411, 0.5789, 0.191]$, and at iteration 501 it is switched to $\mathcal{H}_2 = [1, -0.0870, 0.9852, -0.2826, -0.1711]$. Finally, 20dB of Gaussian white noise is added to the channel output.

We perform an online identification experiment with the algorithms LMS, QKLMS, SW-KRLS, and KRLS-T. Each algorithm performs online learning of the nonlinear channel, processing one input datum (with a time-embedding of 5 taps) and one output sample per iteration. At each step, the MSE performance is measured on a set of 100 data points that are generated with the current channel model. The results are averaged out over 10 simulations.

The kernel adaptive filters use a Gaussian kernel with $\sigma_k = 1$. LMS and QKLMS use a learning rate $\eta = 0.5$. The sparsification threshold of QKLMS is set to $\epsilon_{\mathbb{U}} = 0.3$, which leads to a final dictionary of size around $m = 300$ at the end of the experiment. The regularization of SW-KRLS and KRLS-T is set to match the true value of the noise-to-signal ratio, 0.01. Regarding memory, SW-KRLS and KRLS-T are given a maximum dictionary size of $m = 50$. Finally, KRLS-T uses a forgetting factor of $\lambda = 0.998$.

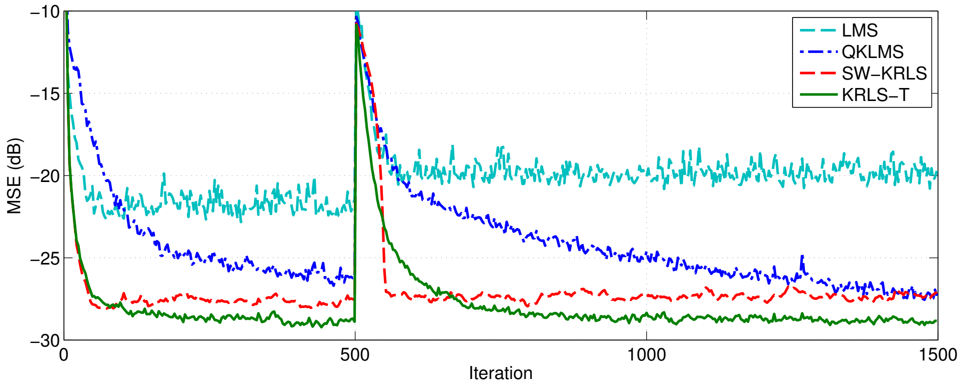

Figure 9.7 MSE learning curves of different kernel adaptive filters on a communications channel that shows an abrupt change at iteration 500.

The results are shown in Fig. 9.7. LMS performs worst, as it is not capable of modeling the nonlinearities in the system. QKLMS shows good results, given its low complexity, but a slow convergence. SW-KRLS and KRLS-T converge to a value which is mostly limited by its dictionary size, $m = 50$, and both show fast convergence rates. All algorithms are capable of reconverging after the switch, though their convergence rate is typically slower at that point.

9.5 Online Sparsification with Kernels

The idea behind sparsification methods is to construct a sparse dictionary of bases that represent the remaining data sufficiently well. As a general rule in learning theory, it is desirable to design a network with as few processing elements as possible. Sparsity reduces the complexity in terms of computation and memory, and it usually gives better generalization ability to unseen data Platt (1991); Vapnik (1995). In the context of kernel methods, sparsification aims to identify the bases in the kernel expansion $y_* = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_*)$, see Eq (9.23), that can be discarded without incurring a significant performance loss.

Online sparsification is typically performed by starting with an empty dictionary, $\mathcal{D}_0 = \emptyset$, and, in each iteration, adding the input datum $\boldsymbol{x}_i$ if it fulfills a chosen sparsification criterion. We denote the dictionary at time instant $n-1$ as $\mathcal{D}_{n-1} = \{\mathbf{c}_i\}_{i=1}^{m_{n-1}}$, where $\mathbf{c}_i$ is the i-th stored center, taken from the input data $\boldsymbol{x}$ received up till this instant, and m_{n-1} is the dictionary cardinality at this instant. When a new input-output pair $(\mathbf{x}_n, d_n)$ is received, a decision is made whether or not $\boldsymbol{x}_n$ should be added to the dictionary as a center. If the sparsification criterion is fulfilled, $\boldsymbol{x}_n$ is added to the dictionary, $\mathcal{D}_n = \mathcal{D}_{n-1} \cup \{\boldsymbol{x}_n\}$. If the criterion is not fulfilled, the dictionary is maintained, $\mathcal{D}_n = \mathcal{D}_{n-1}$, to preserve its sparsity.

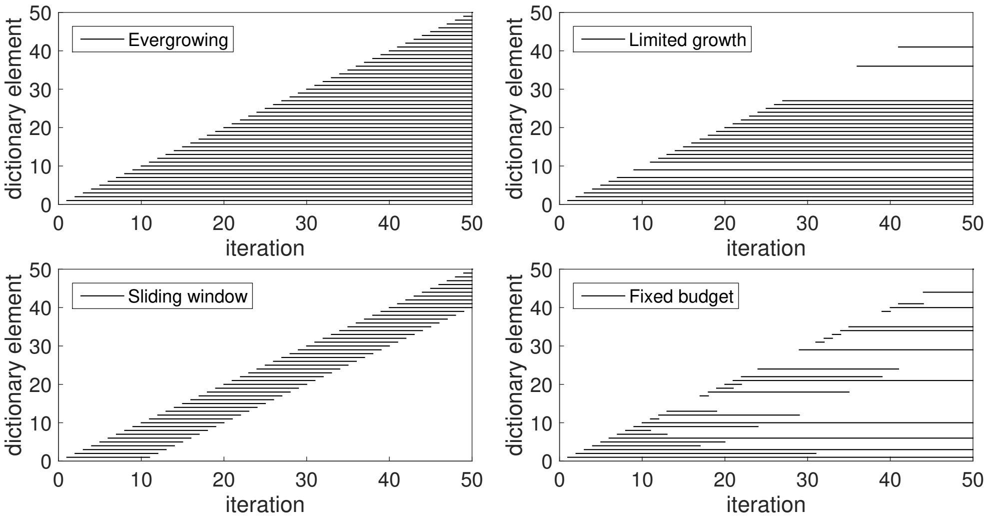

Figure 9.8 Dictionary construction processes for different sparsification approaches. Each horizontal line marks the presence of a center in the dictionary. Top left: The evergrowing dictionary construction, in which the dictionary contains n elements in iteration n; Top right: Online sparsification by slowing down the dictionary growth, as obtained by the coherence and ALD criteria; Bottom left: Sliding-window approach, displayed with 10 elements in the dictionary; Bottom right: Fixed-budget approach, in which the pruning criterion discards one element per iteration, displayed with dictionary size 10.

Fig. 9.8 illustrates the dictionary construction process for different sparsification approaches. Each horizontal line represents the presence of a center in the dictionary. At any given iteration, the elements in the dictionary are indicated by the horizontal lines that are present at that iteration. Below we discuss each approach in detail.

9.5.1 Sparsity by construction

We will first give an general overview of the different online sparsification methods in the literature, some of which we have already introduced in the context of the algorithms for which they were proposed. We distinguish three criteria that achieve sparsity by construction: novelty criterion, approximate linear dependency criterion, and coherence criterion. If the dictionary is not allowed to grow beyond a specified maximum size, it may be necessary to discard bases at some point. This process is referred to as *pruning*, and we will review the most important pruning criteria later.

Novelty Criterion. The novelty criterion is a data selection method introduced by Platt Platt (1991). It was used to construct resource allocating networks (RAN), which are essentially growing radial basis function networks. When a new data point $\boldsymbol{x}_n$ is obtained by the network, the novelty criterion calculates the distance of this point to the current dictionary, $\min_{j \in \mathcal{D}_{n-1}} \|\boldsymbol{x}_n - \mathbf{c}_j\|$. If this distance is smaller than some preset threshold, $\boldsymbol{x}_n$ is added to the dictionary. Otherwise, it computes the prediction error, and only if this error e_n is larger than another preset threshold, the datum $\boldsymbol{x}_n$ will be accepted as a new center.

Approximate Linear Dependency Criterion. A more sophisticated dictionary growth criterion was introduced for the KRLS algorithm in Engel *et al.* (2004): Each time a new datum $\boldsymbol{x}_n$ is observed, the approximate linear dependency (ALD) criterion measures how well the datum can be approximated in the feature space as a linear combination of the dictionary bases in that space. It does so by checking if the ALD condition holds, see Eq. (9.45),

$$\min_{\mathbf{a}} \left\| \sum_{j=1}^{m} a_j \boldsymbol{\phi}(\mathbf{c}_j) - \boldsymbol{\phi}(\boldsymbol{x}_n) \right\|^2 \leq \nu.$$

Evaluating the ALD criterion requires quadratic complexity, $\mathcal{O}(m^2)$ and therefore it is not suitable for algorithms with linear complexity such as KLMS.

Coherence Criterion. The coherence criterion is a straightforward criterion to check whether the newly arriving datum is sufficiently informative. It was introduced in the context of the KNLMS algorithm Richard *et al.* (2009). Given the dictionary D_{n-1} at iteration $n-1$ and the newly arriving datum $\boldsymbol{x}_n$, the coherence criterion to include the datum reads

$$\max_{j \in \mathcal{D}_{n-1}} |k(\boldsymbol{x}_n, \mathbf{c}_j)| < \mu_0. \tag{9.49}$$

In essence, the coherence criterion checks the similarity, as measured by the kernel function, between the new datum and the most similar dictionary center. Only if this similarity is below a certain threshold μ_0, the datum is inserted into the dictionary. The higher the threshold μ_0 is chosen, the more data will be accepted in the dictionary. It is an effective criterion that has linear computational complexity in each iteration: it only requires to calculate m kernel functions, making it suitable for KLMS-type algorithms.

In Chen *et al.* (2012) a similar criterion was introduced, $\min_{j \in \mathcal{D}_{n-1}} \|\boldsymbol{x}_n - \mathbf{c}_j\| > \epsilon_u$, which is essentially equivalent to the coherence criterion with a Euclidean distance based kernel.

9.5.2 *Sparsity by pruning*

In practice, it is often necessary to specify a maximum dictionary size m, or *budget*, that may not be exceeded, for instance due to limitations on hardware or execution time. In order to avoid exceeding this budget, one could simply stop including any data in the dictionary once the budget is reached, hence *locking* the dictionary. Nevertheless, it is very probable that at some point after locking the dictionary a new datum is received that is very informative. In this case, the quality of the algorithm's solution may improve by pruning the least relevant center of the dictionary and replacing it with the new, more informative datum.

The goal of a pruning criterion is to select a datum out of a given set, such that the algorithm's performance is least affected. This makes pruning criteria conceptually different from the previously discussed online sparsification criteria, whose goal is to decide whether or not to include a datum. Pruning techniques have been studied in the context of neural network design Hassibi *et al.* (1993); LeCun *et al.* (1989) and kernel methods De Kruif and De Vries (2003); Hoegaerts *et al.* (2004). We briefly discuss the two most important pruning criteria that appear in kernel adaptive filtering: sliding-window criterion and error criterion.

Sliding Window. In time-varying environments, it may be useful to discard the oldest bases, as these were observed when the underlying model was most different from the current model. This strategy is at the core of sliding-window algorithms such as NORMA Kivinen *et al.* (2004) and SW-KRLS Van Vaerenbergh *et al.* (2006). In every iteration, these algorithms include the new datum in the dictionary and discard the oldest datum, thereby maintaining a dictionary of fixed size.

Error Criterion. Instead of simply discarding the oldest datum, the error based criterion determines the datum that will cause the least increase of the squared-error performance after it is pruned. This is a more sophisticated pruning strategy that was introduced in Csató and Opper (2002); De Kruif and De Vries (2003) and requires quadratic complexity to evaluate, $\mathcal{O}(m^2)$. Interestingly, if the inverse kernel matrix is available, it is straightforward to evaluate this criterion. Given the i-th element on the diagonal of the inverse kernel matrix, $[\mathbf{K}^{-1}]_{ii}$, and the i-th expansion coefficient α_i, the squared error after pruning the i-th center from a dictionary is $\alpha_i/[\mathbf{K}^{-1}]_{ii}$. The error based pruning criterion therefore selects the index for which this quantity is minimized,

$$\arg\min_i \frac{\alpha_i}{[\mathbf{K}^{-1}]_{ii}}. \tag{9.50}$$

This criterion is used in the fixed-budget algorithms FB-KRLS Van Vaerenbergh *et al.* (2010) and KRLS-T Van Vaerenbergh *et al.* (2012b). An analysis performed in Lázaro-Gredilla *et al.* (2011) shows that the results obtained by this criterion are very close to the optimal approach, which is based on minimization of the Kullback–Leibler divergence between the original and the approximate posterior distributions.

Currently, the most successful pruning criteria used in the kernel adaptive filtering literature have quadratic complexity, $\mathcal{O}(m^2)$ and therefore they can only be used in KRLS-type algorithms. Optimal pruning in KLMS is a particularly challenging problem, as it is hard to define a pruning criterion that can be evaluated with linear computational complexity. A simple criterion is found in Rzepka (2012), where the center with the least weight is pruned, and weight is determined by the associated expansion coefficient,

$$\arg\min_i |\alpha_i|. \tag{9.51}$$

The design of more sophisticated pruning strategies is currently an open topic in KLMS literature. Some recently proposed criteria can be found in Zhao *et al.* (2013, 2016).

9.6 Probabilistic Approaches to Kernel Adaptive Filtering

In many signal processing applications, the problem of signal estimation is addressed. Probabilistic models have proven to be very useful in this context Arulampalam *et al.* (2002); Rabiner (1989). One of the advantages of probabilistic approaches is that they force the designer to specify all the prior assumptions of the model, and that they make a clear separation between the model and the applied algorithm. Another benefit is that they typically provide a measure of uncertainty about the estimation. Such an uncertainty estimate is not provided by classical kernel adaptive filtering algorithms, which produce a point estimate without any further guarantees.

In this section, we will review how the probabilistic framework of Gaussian Processes (GP) allows to extend kernel adaptive filters to probabilistic methods. The resulting GP-based algorithms not only produce an estimate of an unknown function, but an entire probability distribution over functions, see Fig. 9.9.

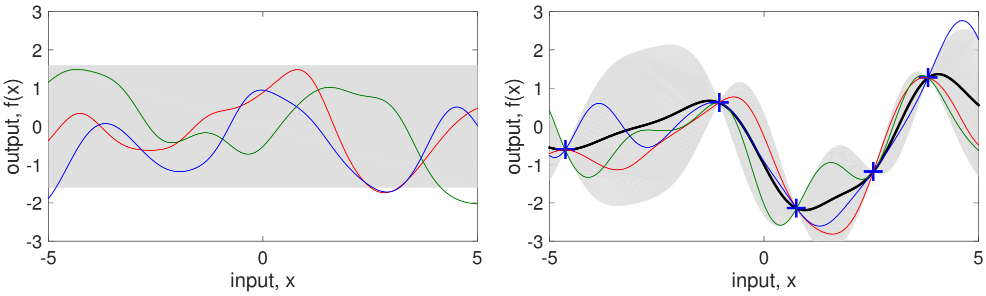

Figure 9.9 Functions drawn from a Gaussian process with a squared exponential covariance $k(\boldsymbol{x},\boldsymbol{x}') = \exp(-\|\boldsymbol{x}-\boldsymbol{x}'\|^2/2\sigma_k^2)$. The 95% confidence interval is plotted as the shaded area. Left: Draws from the prior function distribution. Right: Draws from the posterior function distribution, which is obtained after 5 data points (blue crosses) are observed. The predictive mean is displayed in black.

Before we describe any probabilistic kernel adaptive filtering algorithms, it is instructive to take a step back to the non-adaptive setting, and consider the kernel ridge regression problem (9.34). We will adopt the GP framework to analyze this problem from a probabilistic point of view.

9.6.1 Gaussian Processes and Kernel Ridge Regression

Let us assume that the observed data in a regression problem can be described by the following model,

$$d_n = f(\boldsymbol{x}_n) + \epsilon_n, \tag{9.52}$$

in which f represents an unobservable *latent function* and $\epsilon_n \sim \mathcal{N}(0,\sigma^2)$ is zero-mean Gaussian noise. We will furthermore assume a zero-mean GP prior on $f(\boldsymbol{x})$

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x},\boldsymbol{x}')), \tag{9.53}$$

and a Gaussian prior on the noise ϵ,

$$\epsilon \sim \mathcal{N}(0, \sigma^2). \tag{9.54}$$

In the GP literature, the kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$ is referred to as the *covariance*, since it specifies the a-priori relationship between values $f(\boldsymbol{x})$ and $f(\boldsymbol{x}')$ in terms of their respective locations, and its parameters are called *hyperparameters*.

By definition, the marginal distribution of a GP at a finite set of points is a joint Gaussian distribution, with its mean and covariance being specified by the functions $m(\boldsymbol{x})$ and $k(\boldsymbol{x}, \boldsymbol{x}')$ evaluated at those points Rasmussen and Williams (2006). Thus, the joint distribution of outputs $\mathbf{d} = [d_1, \ldots, d_n]^\top$ and the corresponding latent vector $\mathbf{f} = [f(\boldsymbol{x}_1), f(\boldsymbol{x}_2), \ldots, f(\boldsymbol{x}_n)]^\top$ is

$$\begin{bmatrix} \mathbf{d} \\ \mathbf{f} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} + \sigma^2\mathbf{I} & \mathbf{K} \\ \mathbf{K} & \mathbf{K} \end{bmatrix}\right). \tag{9.55}$$

By conditioning on the observed outputs $\mathbf{y}$, the posterior distribution over the latent vector can be inferred

$$\begin{aligned} p(\mathbf{f}|\mathbf{d}) &= \mathcal{N}(\mathbf{f}|\mathbf{K}(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{d}, \mathbf{K} - \mathbf{K}(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{K}) \\ &= \mathcal{N}(\mathbf{f}|\boldsymbol{\mu}, \boldsymbol{\Sigma}). \end{aligned} \tag{9.56}$$

Assuming this posterior is obtained for the data up till time instant $n-1$, the predictive distribution of a new output d_n at location $\boldsymbol{x}_n$ is computed as

$$p(d_n|\boldsymbol{x}_n, \mathbf{d}_{n-1}) = \mathcal{N}(d_n|\mu_{\text{GP},n}, \sigma^2_{\text{GP},n}) \tag{9.57a}$$

$$\mu_{\text{GP},n} = \boldsymbol{k}_n^\top(\mathbf{K}_{n-1} + \sigma^2\mathbf{I})^{-1}\mathbf{d}_{n-1} \tag{9.57b}$$

$$\sigma^2_{\text{GP},n} = \sigma^2 + k_{nn} - \boldsymbol{k}_n^\top(\mathbf{K}_{n-1} + \sigma^2\mathbf{I})^{-1}\boldsymbol{k}_n. \tag{9.57c}$$

The mode of the predictive distribution, given by $\mu_{\text{GP},n}$ in Eq. (9.57b), coincides with the prediction of kernel ridge regression, given by Eq. (9.36), showing that the regularization in KRR can be interpreted as a noise power σ^2. Furthermore, the variance of the predictive distribution, given by $\sigma^2_{\text{GP},n}$ in Eq. (9.57c), coincides with Eq. (9.42), which is used by the ALD dictionary criterion for KRLS.

9.6.2 *Online recursive solution for Gaussian process regression*

A recursive update of the complete GP (9.57) was proposed in Csató and Opper (2002), as the Sparse Online Gaussian Process (SOGP) algorithm. We will follow the notation of Van Vaerenbergh *et al.* (2012b), whose solution is equivalent to SOGP but whose choice of variables allows for an easier interpretation. Specifically, the predictive mean and covariance of the GP solution (9.57) can be updated as

$$p(\mathbf{f}_n|\mathbf{X}_n, \mathbf{d}_n) = \mathcal{N}(\mathbf{f}_n|\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n) \tag{9.58a}$$

$$\boldsymbol{\mu}_n = \begin{bmatrix} \boldsymbol{\mu}_{n-1} \\ \hat{d}_n \end{bmatrix} + \frac{e_n}{\hat{\sigma}^2_{dn}} \begin{bmatrix} \mathbf{h}_n \\ \hat{\sigma}^2_{fn} \end{bmatrix} \tag{9.58b}$$

$$\boldsymbol{\Sigma}_n = \begin{bmatrix} \boldsymbol{\Sigma}_{n-1} & \mathbf{h}_n \\ \mathbf{h}_n^\top & \hat{\sigma}^2_{fn} \end{bmatrix} - \frac{1}{\hat{\sigma}^2_{dn}} \begin{bmatrix} \mathbf{h}_n \\ \hat{\sigma}^2_{fn} \end{bmatrix} \begin{bmatrix} \mathbf{h}_n \\ \hat{\sigma}^2_{fn} \end{bmatrix}^\top, \tag{9.58c}$$

where $\mathbf{X}_n$ contains the n input data, $\mathbf{h}_n = \mathbf{\Sigma}_{n-1}\mathbf{K}_{n-1}^{-1}\boldsymbol{k}_n$, and $\hat{\sigma}_{fn}^2$ and $\hat{\sigma}_{dn}^2$ are the predictive variances of the latent function and the new output, respectively, calculated at the new input. Details can be found in Lázaro-Gredilla *et al.* (2011); Van Vaerenbergh *et al.* (2012b). In particular, the update of the predictive mean can be shown to be equivalent to the KRLS update. The advantage of using a full GP model is that not only does it allow to update the predictive mean, as does KRLS, but it keeps track of the entire predictive distribution of the solution. This allows, for instance, to establish confidence intervals when predicting new outputs.

Similar to KRLS, this online GP update assumes a stationary model. Interestingly however, the Bayesian approach (and in particular its handling of the uncertainty) does allow for a principled extension that performs tracking, as we briefly discuss in the sequel.

9.6.3 *Kernel Recursive Least Squares Tracker*

In Van Vaerenbergh *et al.* (2012b), a KRLS Tracker (KRLS-T) algorithm was presented that explicitly handles uncertainty about the data, based on the probabilistic GP framework. In stationary environments it operates identically to the earlier proposed Sparse Online GP algorithm (SOGP) from Csató and Opper (2002), though it includes a *forgetting mechanism* that enables it to handle non-stationary scenarios as well.

During each iteration, KRLS-T performs a forgetting operation in which the mean and covariance are replaced through

$$\boldsymbol{\mu} \leftarrow \sqrt{\lambda}\boldsymbol{\mu} \tag{9.59a}$$

$$\mathbf{\Sigma} \leftarrow \lambda\mathbf{\Sigma} + (1-\lambda)\mathbf{K}. \tag{9.59b}$$

The effect of this operation on the predictive distribution is shown in Fig. 9.10. For illustration purposes, the forgetting factor is chosen unusually low, $\lambda = 0.9$.

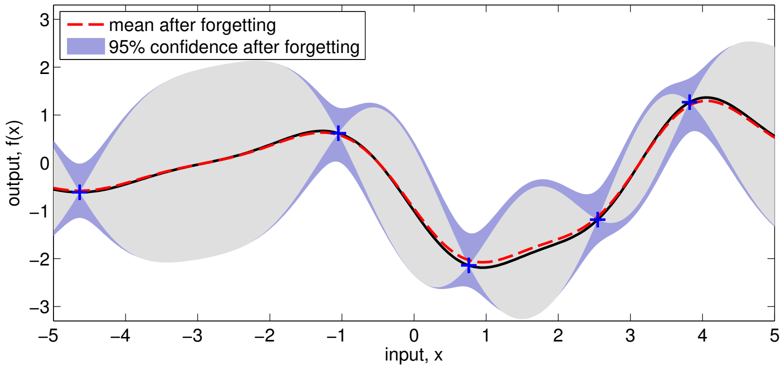

Figure 9.10 Forgetting operation of KRLS-T. The original predictive mean and variance are indicated as the black line and shaded grey area, as in Fig. 9.9. After one forgetting step, the mean becomes the dashed red curve, and the new 95% confidence interval is indicated in blue.

This particular form of forgetting corresponds to blending the informative posterior with a "noise" distribution that uses the same color as the prior. In other words, forgetting occurs by taking a step back towards the prior knowledge. Since the prior has zero mean, the mean is simply scaled by the square root of the forgetting factor λ. The covariance, which represents the posterior uncertainty on the data, is pulled towards the covariance of the prior. Interestingly, a regularized version of RLS (known as *extended RLS*) can be obtained by using a linear kernel with the B2P forgetting procedure. Standard RLS can be obtained by using a different forgetting rule, see Van Vaerenbergh *et al.* (2012b).

The KRLS-T algorithm can be seen as a probabilistic extension of KRLS that obtains confidence intervals and is capable of adapting to time-varying environments. It obtains state-of-the-art performance in several nonlinear adaptive filtering problems, see Van Vaerenbergh and Santamaría (2013) and the results of Fig. 9.7, though it has a more complex formulation than most other kernel adaptive filters and it requires a higher computational complexity. We will explore these aspects through additional examples in Section 9.8.

9.6.4 Probabilistic KLMS

The success of the probabilistic approach for KRLS-like algorithms has lead several researchers to investigate the design of probabilistic KLMS algorithms. The low complexity of KLMS-type algorithms makes them very popular in practical solutions. Nevertheless, this low computational complexity is also a limitation that makes the design of a probabilistic KLMS algorithm a particularly hard research problem.

Some advances have already been made in this direction. Specifically, in Park *et al.* (2014) a probabilistic KLMS algorithm was proposed, though it only considered the maximum-a-posteriori (MAP) estimate. In Van Vaerenbergh *et al.* (2016a), it was shown that several KLMS algorithms can be obtained by imposing a simplifying restriction on the full SOGP model, thereby linking KLMS algorithms and online GP approaches directly.

9.7 Further Reading

A myriad of different kernel adaptive filtering algorithms have appeared in the literature. We described the most prominent algorithms, which represent the state of the art. While we only focused on their online learning operation, several other aspects are worth studying. In this section, we briefly introduce the most interesting topics that are the subject of current research.

9.7.1 Selection of Kernel Parameters

A typical problem in kernel methods in general and kernel adaptive filtering in particular is the determination of the optimal kernel and other parameters, such as regularization, forgetting factor, embedding size, etc. These parameters are often are referred to as *hyperparameters* in order to distinguish them from the kernel expansion coefficients α_i. A standard approach to determine the optimal hyperparameters is to perform a grid search with cross-validation. Nevertheless, this approach quickly becomes infeasible when more than a few hyperparameters or parameter values are to be considered, due to the combinatorial explosion of possible grid points to evaluate.

A more efficient and principled method is offered by the Gaussian process framework. Specifically, the optimal hyperparameters of GP regression can be found by maximizing the log marginal likelihood, which has an analytic expression given by

$$\log p(\mathbf{d}|\mathbf{X}) = -\frac{1}{2}\mathbf{d}^\top \left(\mathbf{K} + \sigma^2\mathbf{I}\right)^{-1}\mathbf{d} - \frac{1}{2}\log\|\mathbf{K}\| - \frac{n}{2}\log 2\pi. \tag{9.60}$$

It is straightforward to compute this log marginal likelihood and its gradients, and one can choose any of the existing nonlinear optimization methods to perform the maximization. This procedure is commonly referred to as type-II maximum likelihood (ML). Details can be found in Rasmussen and Williams (2006).

The optimal hyperparameters obtained by type-II ML correspond to the optimal choices for kernel ridge regression, due to the correspondence between GP regression and KRR, and for several kernel adaptive filters. A case study for KRLS and KRLS-T can be found in Van Vaerenbergh *et al.* (2012a).

Finally, in online scenarios it would be interesting to perform an online estimation of the optimal hyperparameters. This, however, is a difficult open research problem for which only a handful of methods have been proposed, see for instance Soh and Demiris (2015). In practice it is still more appropriate to perform type-II ML offline on a batch of training data, before running the online learning procedure using the found hyperparameters.

9.7.2 Multi-Kernel Adaptive Filtering

In the last decade, several methods have been proposed to consider multiple kernels instead of a single one Bach *et al.* (2004); Sonnenburg *et al.* (2006). The different kernels may correspond to different notions of similarity, or they may address information coming from multiple, heterogeneous data sources.

On the other hand, in the field of linear adaptive filtering, it was recently shown that a convex combination of adaptive filters can improve the convergence rate and tracking performance Arenas-García *et al.* (2006) compared to running a single adaptive filter.

Multi-kernel adaptive filtering combines ideas from the above two approaches Yukawa (2012). Its learning procedure activates those kernels whose hyperparameters correspond best to the currently observed data, which could be interpreted as a form of hyperparameter learning. Furthermore, the adaptive nature of these algorithms allow them to track the importance of each kernel in time-varying scenarios, possibly giving them an advantage over single-kernel adaptive filtering.

Several multi-kernel adaptive filtering algorithms have been proposed in the recent literature, for instance Gao *et al.* (2014); Ishida and Tanaka (2013); Pokharel *et al.* (2013); Yukawa (2012). While they show promising performance gains over single-kernel adaptive filtering algorithms, their computational complexity is much higher. This is an important aspect inherent to the combination of multiple kernel methods, and it is a topic of current research.

9.7.3 Recursive Filtering in Kernel Hilbert Spaces

The modeling and prediction of time series with kernel adaptive filters is usually addressed by time-embedding the data, thus considering each time lag as a different input dimension. This approach presents some drawbacks: First, the optimal filter order may change over time, which would require an additional tracking mechanism; Second, if the optimal filter order is high, as for instance in audio applications Van Vaerenbergh *et al.* (2016b), the method be affected by the curse of dimensionality. For some problems, the concept of an optimal filter order may not even make sense.

An alternative approach to modeling and predicting time series is to construct *recursive* kernel machines, which implement recursive models explicitly in the rkHs. A preliminary work in this direction considered the design of a *recursive kernel* in the context of infinite recurrent neural networks Hermans and Schrauwen (2012). More recently, recursive versions of the autoregressive, moving-average and gamma filters in rkHs were proposed Tuia *et al.* (2014). By exploiting properties of functional analysis and recursive computation, this approach avoids the reduced-rank approximations that are required in standard kernel adaptive filters. Finally, a kernel version of the autoregressive-moving-average filter was presented in Li and Príncipe (2016).

9.8 Tutorial Examples

This section presents experiments in which kernel adaptive filters are applied to time series prediction and nonlinear system identification. These experiments are implemented using code based on the Kernel Adaptive Filtering Toolbox Van Vaerenbergh and Santamaría (2013), which is available at `https://github.com/steven2358/kafbox/`.

9.8.1 Kernel Adaptive Filtering Toolbox

The Kernel Adaptive Filtering Toolbox (KAFBOX) is a Matlab benchmarking toolbox to evaluate and compare kernel adaptive filtering algorithms. It includes a large list of algorithms that have appeared in the literature, and additional tools for hyperparameter estimation and algorithm profiling, among others.

The kernel adaptive filtering algorithms in KAFBOX are implemented as objects using the `classdef` syntax. Since all KAF algorithms are online methods, each of them includes two basic operations: 1) Obtaining the filter output, given a new input $\boldsymbol{x}_*$; and 2) Training on a new data pair $(\boldsymbol{x}_n, d_n)$. These operations are implemented as the methods `evaluate` and `train`, respectively.

As an example, we list the code for the KLMS algorithm in Listing 9.9. The object definition contains two sets of properties, one for the hyperparameters and one for the variables it will learn. The first method is the object's constructor method, which copies the specified hyperparameter settings. The second method is the `evaluate` function, which performs the operation $y_* = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_*)$. It includes an `if` clause to check if the algorithm has at least performed one training step yet. If not, zeroes are returned as predictions. Finally, the `train` method implements a single training step of the online learning algorithm. This method typically handles algorithm initialization as well, such that functions that operate on a KAF object do not have to worry about initializing. The training step itself is summarized in very few lines of Matlab code, for many algorithms.

```
% Kernel Least-Mean-Square algorithm
%
% W. Liu, P.P. Pokharel, and J.C. Principe, "The Kernel Least-Mean-Square
% Algorithm," IEEE Transactions on Signal Processing, vol. 56, no. 2, pp.
% 543-554, Feb. 2008, http://dx.doi.org/10.1109/TSP.2007.907881
%
% Remark: implementation includes a maximum dictionary size M
%
% This file is part of the Kernel Adaptive Filtering Toolbox for Matlab.
% https://github.com/steven2358/kafbox/

classdef klms < handle

    properties (GetAccess = 'public', SetAccess = 'private')
        eta = .5; % learning rate
        M = 10000; % maximum dictionary size
        kerneltype = 'gauss'; % kernel type
```

```
        kernelpar = 1; % kernel parameter
    end

    properties (GetAccess = 'public', SetAccess = 'private')
        dict = []; % dictionary
        alpha = []; % expansion coefficients
    end

    methods

        function kaf = klms(parameters) % constructor
            if (nargin > 0) % copy valid parameters
                for fn = fieldnames(parameters)',
                    if ismember(fn,fieldnames(kaf)),
                        kaf.(fn{1}) = parameters.(fn{1});
                    end
                end
            end
        end

        function y_est = evaluate(kaf,x) % evaluate the algorithm
            if size(kaf.dict,1)>0
                k = kernel(kaf.dict,x,kaf.kerneltype,kaf.kernelpar);
                y_est = k'*kaf.alpha;
            else
                y_est = zeros(size(x,1),1);
            end
        end

        function train(kaf,x,y) % train the algorithm
            if (size(kaf.dict,1)<kaf.M), % avoid infinite growth
                y_est = kaf.evaluate(x);
                err = y - y_est;
                kaf.alpha = [kaf.alpha; kaf.eta*err]; % grow
                kaf.dict = [kaf.dict; x]; % grow
            end
        end

    end
end
```

Listing 9.9 Matlab code for the KLMS algorithm object class, from KAFBOX.

For the experiments below, we will use v2.0 of KAFBOX, which can be downloaded from the “releases” page `https://github.com/steven2358/kafbox/releases/`.

9.8.2 Prediction of a Respiratory Motion Time Series

In the first experiment we apply KAF algorithms to predict a bio-medical time series, more specifically a respiratory motion trace. These data come from robotic radiosurgery, in which a photon beam source is used to ablate tumors. The beam is operated by a robot arm that aims to move the beam source to compensate for the motion of internal organs. Traditionally, this is achieved by recording the motion of markers applied to the body surface and by using this motion to draw conclusions about the tumor position. Although this method significantly increases the targeting accuracy, the system delay arising from data processing and positioning of the beam results in a systematic error. This error can be decreased by predicting the motion of the body surface Ernst (2012).

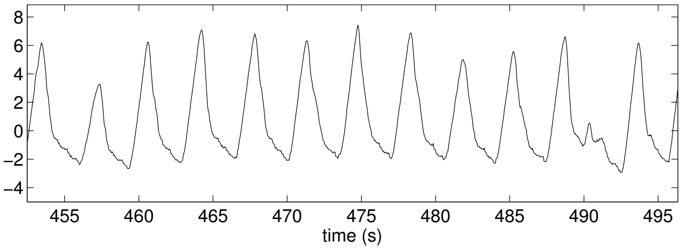

Figure 9.11 A snapshot of the respiratory motion trace.

The data was recorded at Georgetown University Hospital using CyberKnife® equipment, and it represents the recorded position of one of the markers attached to the body surface[3]. A snapshot of this motion trace is shown in Fig. 9.11. The delay to compensate totals 115 ms, which, at a sampling frequency of 26 Hz, corresponds to 3 samples. The task thus consists in three-step ahead prediction. We use a time-embedding of 8 samples. Since the breathing pattern may change over time, we employed only tracking algorithms. Their parameters are listed in Table 9.1. The MSE results of the four algorithms are displayed in the last column of this table. A comparison of the original series and the predictions of one of the algorithms is shown in Fig. 9.12. The code to reproduce these results can be found in Listing 9.10.

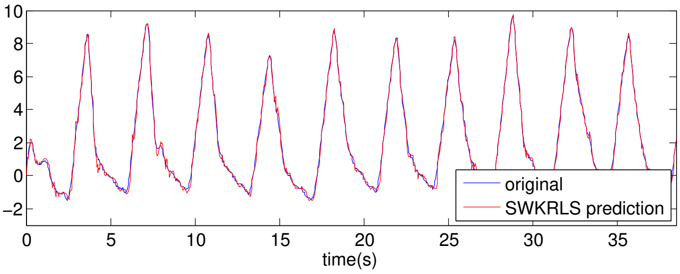

Figure 9.12 The respiratory motion trace and the 3-step ahead prediction of a KAF algorithm.

Table 9.1 Parameters used for predicting the respiratory motion trace, MSE result for 3-step ahead prediction, and measured training time.

Algorithm	Parameters	MSE performance	Training time
NORMA	$\lambda = 10^{-4}$, $\tau = 30$	-5.78 dB	0.16 s
QKLMS	$\eta = 0.99$, $\epsilon_{\mathbb{U}} = 1$	-8.14 dB	0.18 s
SWKRLS	$c = 10^{-4}$, $m = 50$	-13.35 dB	0.29 s
KRLS-T	$\sigma_n^2 = 10^{-4}$, $m = 50$, $\lambda = 0.999$	-18.16 dB	0.54 s

---

[3] Data available at `http://signals.rob.uni-luebeck.de/`

```matlab
% 3-step ahead prediction on the respiratory motion time series.
% Requires KAFBOX toolbox.
close all; clear

%% PARAMETERS
h = 3; % prediction horizon
L = 8; % embedding
n = 1000; % number of data

sigma = 7; % kernel parameter

% Uncomment one of the following algorithms. All use a Gaussian kernel.
% kaf = norma(struct('lambda',1E-4,'tau',30,'kernelpar',sigma,'eta',0.99));
% kaf = qklms(struct('epsu',1,'kernelpar',sigma,'eta',0.99));
kaf = swkrls(struct('M',50,'kernelpar',sigma,'c',1E-4));
% kaf = krlst(struct('M',50,'lambda',0.999,'sn2',1E-4,'kernelpar',sigma));

%% PREPARE DATA
data = load('respiratorymotion3.dat');
X = zeros(n,L);
for i = 1:L,
    X(i:n,i) = data(1:n-i+1,1); % time embedding
end
y = data((1:n)+h);

%% RUN ALGORITHM
MSE = zeros(n,1);
y_est_all = zeros(n,1);

title_ = upper(class(kaf)); % store algorithm name
fprintf('Training %s',title_)
for i=1:n,
    if ~mod(i,floor(n/10)), fprintf('.'); end % progress indicator

    xi = X(i,:);
    yi = y(i);

    y_est = kaf.evaluate(xi); % evaluate on test data
    MSE(i) = (yi-y_est)^2; % test error
    y_est_all(i) = y_est;

    kaf.train(xi,yi); % train with one input-output pair
end
fprintf('\n');

%% OUTPUT
fprintf('Mean MSE: %.2fdB\n\n',10*log10(mean(MSE)));

figure; hold all;
t = (1:n)/26; % sample rate is 26 Hz
plot(t,y)
plot(t,y_est_all,'r')
legend({'original',sprintf('%s prediction',title_)},'Location','SE');
```

Listing 9.10 Matlab code for running KAF algorithms on the respiratory motion prediction problem.

9.8.3 Online Regression on the KIN40K Data Set

In the second experiment we train the online algorithms to perform regression of the KIN40K data set Ghahramani (1996)[4], which is a standard regression problem in the machine learning literature. The KIN-40K data set is obtained from the forward kinematics of an 8-link all-revolute robot arm, similar to the one depicted in Fig. 9.13. It contains 40000 examples, each consisting of an 8-dimensional input vector and a scalar output. KIN40K was generated with maximum nonlinearity and little noise, representing a very difficult regression test.

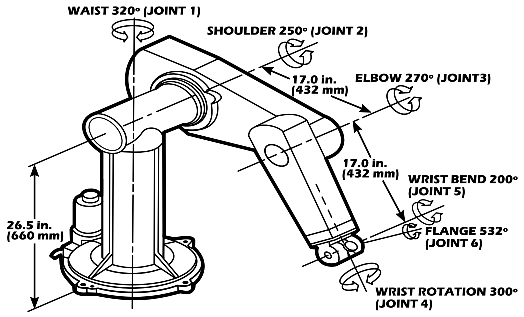

Figure 9.13 Sketch of a 5-link all-revolute robot arm. The data used in the KIN40K experiment were generated by simulating an 8-link extension of this arm.

In this experiment, we first determine the optimal hyperparameters for the kernel adaptive filters by running the tool `kafbox_parameter_estimation`, which is based on the GPML toolbox from Rasmussen and Williams (2006). We use 1000 randomly selected data points for the hyperparameter optimization. In the literature, an anisotropic kernel function, which has a different kernel width per dimension, is commonly used on these data. For simplicity, though, we employ an isotropic Gaussian kernel. The hyperparameters found by the optimization procedure are listed in Table 9.2. The forgetting factor is only used by KRLS-T, though its optimal value is determined to be 1, which indicates that no forgetting takes place in practice.

Table 9.2 Optimal hyperparameters found for the KIN40K regression problem.

Parameter	Optimal value
kernel width σ	1.66
regularization	$2.47 \cdot 10^{-6}$
forgetting factor λ	1

From the remaining data we randomly select 5000 data points for training and 5000 data for testing the regression. Apart from the hyperparameters that are determined automatically in this experiment, the kernel adaptive filters have some parameters relating to memory size and learning rate. The values chosen for these parameters are listed in Table 9.3. The values for

---

[4] Data available at `http://www.cs.toronto.edu/~delve/data/datasets.html`

QKLMS are chosen such that it obtains optimal performance after training with a dictionary size that is one order of magnitude larger than that of the KRLS algorithms. The precision parameter ν of KRLS is tuned to yield a dictionary size of around $m = 500$ at the end of the experiment, which is the budget of SWKRLS and KRLS-T.

The learning curves for this experiment are shown in Fig. 9.14. The code for reproducing this experiment is displayed in Listing 9.11.

Table 9.3 Additional parameters used in the KIN40K regression experiment, final dictionary size, and measured training time.

Algorithm	Parameters	Final dictionary size	Training time
QKLMS	$\eta = 0.99$, $\epsilon_{\mathbb{U}} = 1.2$	2750	18.51 s
KRLS	$\nu = 0.32$	510	12.09 s
FBKRLS	$m = 500$	500	33.49 s
KRLS-T	$m = 500$	500	86.41 s

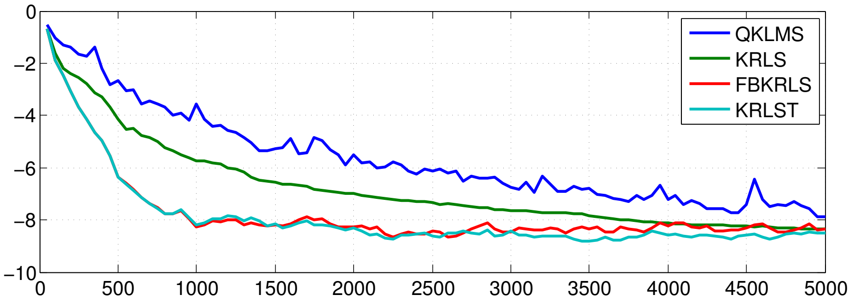

Figure 9.14 Learning curves of different algorithms on the KIN40K data.

```
% This demo estimates the optimal hyperparameters on the KIN40K data and
% performs online regression learning. The estimated parameters are:
% forgetting factor lambda, regularization c and Gaussian kernel width.

close all; clear
rng(1); % fix random seed, for reproducibility

%% PARAMETERS

n_hyper = 1000; % number of data to use for hyperparameter estimation
n_train = 5000; % number of data for online training
n_test = 5000; % number of data for testing
test_every = 50; % run test every time this number of iterations passes

% selected algorithms are defined below

%% PROGRAM
tic

```

```
fprintf('Loading KIN40K data...\n')
load('kin40k'); % data and hyperparameters

%% PREPARE DATA
indp = randperm(length(y)); % random permudation
ind_hyper = indp(1:n_hyper);
ind_train = indp(n_hyper+1:n_hyper+n_train);
ind_test = indp(n_hyper+n_train+1:n_hyper+n_train+n_test);
X_hyper = X(ind_hyper,:); % data for hyperparameter estimation
y_hyper = y(ind_hyper);
X_train = X(ind_train,:); % training data
y_train = y(ind_train);
X_test = X(ind_test,:); % test data
y_test = y(ind_test);

fprintf('Estimating KRLS-T parameters...\n\n')
[sigma,reg,ff] = kafbox_parameter_estimation(X_hyper,y_hyper);

% select algorithms
i=1;
algos{i} = qklms(struct('eta',0.5,'epsu',1.2,'kernelpar',sigma)); i=i+1;
algos{i} = krls(struct('nu',.32,'kernelpar',sigma)); i=i+1;
algos{i} = fbkrls(struct('M',500,'lambda',reg,'kernelpar',sigma)); i=i+1;
algos{i} = ...
    krlst(struct('lambda',ff,'M',500,'sn2',reg,'kernelpar',sigma)); i=i+1;

n_algos = length(algos);
MSE = nan*zeros(n_train,1);
titles = cell(n_algos,1);
final_dict_size = zeros(n_algos,1);

for j=1:n_algos
    kaf = algos{j};

    titles{j} = upper(class(kaf));
    fprintf(sprintf('Running %s with estimated parameters...\n',titles{j}))

    for i=1:n_train,
        if ~mod(i,floor(n_train/10)), fprintf('.'); end % progress ...
            indicator, 10 dots

        kaf.train(X_train(i,:),y_train(i)); % train with one input-output ...
            pair

        if mod(i,test_every) == 0 % run test only every
            y_est = kaf.evaluate(X_test); % predict on test set
            MSE(i,j) = mean((y_test-y_est).^2);
        end
    end
    final_dict_size(j) = size(kaf.dict,1);
    fprintf('\n');
end

toc
%% OUTPUT

figure;
xs=find(~isnan(MSE(:,1)));
plot(xs,10*log10(MSE(xs,:)))
legend(titles)

fprintf('\n');
```

```
fprintf('        Estimated\n');
fprintf('sigma:  %.4f\n',sigma)
fprintf('c:      %e\n',reg)
fprintf('lambda: %.4f\n\n',ff)

final_dict_size
```

Listing 9.11 Matlab code for determining the optimal hyperparameters and running online regression on the KIN40K data.

9.8.4 *The Mackey-Glass Time Series*

The Mackey-Glass time-series prediction is a benchmarking problem in nonlinear time-series modelling. We discussed this time series and the prediction results for KLMS and KRLS in Sections 9.3.3 and 9.4.3, respectively.

The learning curves, shown in Fig. 9.5, indicate that KLMS converges very slowly, and that KRLS can obtain a much lower MSE in less iterations. On the other hand, KRLS requires an order of magnitude more computation and memory. These results are in line with the intuitions from linear adaptive filtering, in which LMS and RLS represent two different choices in the compromise between complexity and convergence rate.

Nevertheless, there is a fundamental difference between the complexity analyses of linear and kernel adaptive filtering algorithms. While in linear adaptive filters the complexity depends on the data dimension, in KAF algorithms it depends on the dictionary size. And, importantly, the latter is a parameter that can be controlled.

A KRLS-type algorithm with a large dictionary can converge faster than a KLMS-type algorithm with a similarly sized dictionary, at the expense of a higher computational complexity. But it would be instructive to ask how a KRLS-type algorithm with a small dictionary compares to a KLMS-type algorithm with a large dictionary. Can the KRLS algorithm obtain similar complexity as KLMS, while maintaining its better convergence rate? This question is answered in the diagrams of Fig. 9.15. We have included two KAf algorithms, QKLMS and KRLS-T, that allow easy control over their dictionary size.

Table 9.4 Parameters used in the Mackey-Glass time series prediction. A Gaussian kernel with $\sigma_k = 1$ was used.

Algorithm	Fixed parameters	Varying parameter
QKLMS	$\eta = 0.5$	$\epsilon_{\mathbb{U}} \in \{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 0.2, 0.3, 0.5, 0.7, 1\}$
KRLS-T	$\lambda = 1, \sigma_n^2 = 10^{-6}$	$m \in \{3, 5, 7, 10, 20, 30, 50, 150\}$

Fig. 9.15 represents the results obtained by the KAF profiler tool included in KAFBOX. The Matlab code to reproduce this figure is displayed in Listing 9.12. The profiler tool runs each algorithm several times with different configurations, whose parameters are shown in Table 9.4, producing one point in the plot per algorithm configuration. It calculates several variables, such as the number of floating point operations (FLOPS), the used memory (in bytes), and execution time.

By plotting the MSE vs. the FLOPS or memory, we get an idea of the resources required to obtain a desired MSE result. If, for instance, we are working in a scenario with a restriction on computational complexity, we should select the algorithm that performs best under this restriction by determining which performance curve is most to the left for the amount of

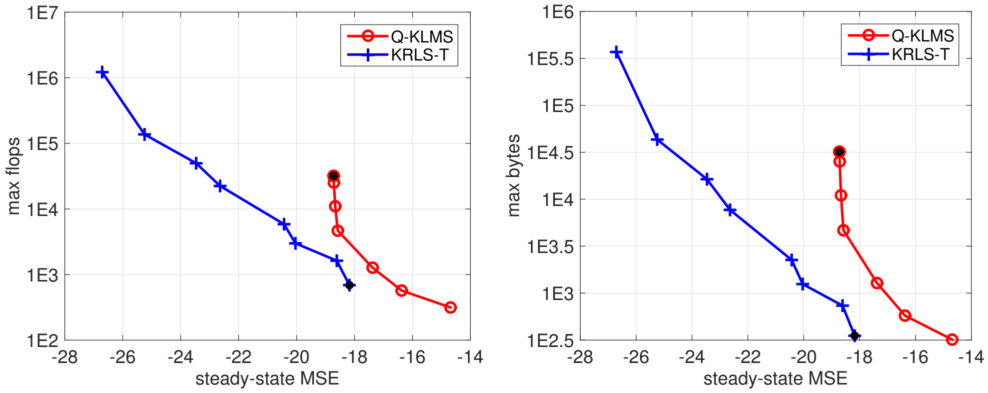

Figure 9.15 MSE vs. complexity trade-off comparisons for prediction of the Mackey-Glass time-series. Left: Maximum number of FLOPS per iteration as a function of the steady-state MSE. Right: Maximum number of bytes per iteration as a function of the steady-state MSE. Each marker represents a single run of one of the algorithms with a single set of parameters. The start of each parameter sweep is indicated by a black dot.

FLOPS available. In the same manner, by fixing a maximum on MSE we obtain the FLOPS and memory required by each algorithm. In the left plot of Fig. 9.15 we observe that if the available computational complexity is very limited, it may be more interesting to use QKLMS. In other cases, KRLS-T is preferred as it obtains better MSE for the same amount of FLOPS. In terms of memory used, it appears that it is always advantageous to use KRLS-T, as can be seen in the right plot.

```
% Experiment: kernel adaptive filter algorithm profiler.
% Compares the cost vs prediction error tradeoffs and convergence speeds
% for several algorithms on the MG30 data set.
% Requires KAFBOX toolbox.

clear
close all

%% PARAMETERS

% data and algorithm setup
data.name = 'mg30';
data.n_train = 500; % number of data points
data.n_test = 100; % number of data points
data.embedding = 7; % time embedding
data.offset = 50; % apply offset per simulation

sim_opts.numsim = 5; % 10 seconds per simulation on a 2016 PC
sim_opts.error_measure = 'MSE';

i=0; % initialize setups

%% QKLMS
i=i+1;
algorithms{i}.name = 'QKLMS';
algorithms{i}.class = 'qklms';
algorithms{i}.figstyle = struct('color',[1  0  0],'marker','o');
```

```
algorithms{i}.options = ...
    struct('eta',0.5,'sweep_par','epsu','sweep_val',[1E-4 1E-3 1E-2 .1 ...
    .2 .3 .5 .7 1],...
    'kerneltype','gauss','kernelpar',1);

%% KRLS-T
i=i+1;
algorithms{i}.name = 'KRLS-T';
algorithms{i}.class = 'krlst';
algorithms{i}.figstyle = struct('color',[0  0  1],'marker','+');
algorithms{i}.options = struct('sn2',1E-6,'lambda',1,'sweep_par','M',...
    'sweep_val',[3 5 7 10 20 30 50 150],...
    'kerneltype','gauss','kernelpar',1);

%% PROGRAM

fprintf('Running profiler for %d algorithms on %s data.\n',i,data.name);
output_dir = fullfile(mfilename('fullpath'),'..','results');

t1 = tic;
[data,algorithms,results] = ...
    kafbox_profiler(data,sim_opts,algorithms,output_dir);
t2 = toc(t1);

fprintf('Elapsed time: %d seconds\n',ceil(t2));

%% OUTPUT

mse_curves = kafbox_profiler_msecurves(results);

kafbox_profiler_plotresults(algorithms,mse_curves,results,{'ssmse','flops'});

kafbox_profiler_plotresults(algorithms,mse_curves,results,{'ssmse','bytes'});

resinds = [1,2;2,8]; % result indices
kafbox_profiler_plotconvergence(algorithms,mse_curves,resinds);
```

Listing 9.12 Matlab code for profiling QKLMS and KRLS-T on the Mackey-Glass time series.

9.9 Questions and Problems

Exercise 9.9.1 *When is it be more useful to employ a KLMS-like algorithm and when a KRLS-like algorithm?*

Exercise 9.9.2 *Demonstrate that the computational complexity of KLMS is $\mathcal{O}(m)$, where m is the number of data in its dictionary.*

Exercise 9.9.3 *Demonstrate that the computational complexity of KRLS is $\mathcal{O}(m^2)$, where m is the number of data in its dictionary.*

Exercise 9.9.4 *List the advantages and disadvantages of using a sliding-window approach for determining a filter's dictionary, as used for instance by SW-KRLS and NORMA.*

Exercise 9.9.5 *In the tracking experiment of Section 9.4.5, the slope of the learning curve for some algorithms is less steep just after the switch than in the beginning of the experiment. Identify for which algorithms this happens, in Fig. 9.7, and explain for each of these algorithms why this is the case.*

References

Arenas-García, J., Figueiras-Vidal, A. R., and Sayed, A. H. (2006). Mean-square performance of a convex combination of two adaptive filters. IEEE transactions on signal processing, **54***(3), 1078–1090.*

Arulampalam, M. S., Maskell, S., Gordon, N., and Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. IEEE Trans. Signal Process., **50***(2), 174–188.*

Bach, F. R., Lanckriet, G. R., and Jordan, M. I. (2004). Multiple kernel learning, conic duality, and the SMO algorithm. In Proceedings of the twenty-first international conference on Machine learning*, page 6. ACM.*

Chen, B., Zhao, S., Zhu, P., and Príncipe, J. C. (2012). Quantized kernel least mean square algorithm. IEEE Transactions on Neural Networks and Learning Systems, **23***(1), 22–32.*

Csató, L. and Opper, M. (2002). Sparse online Gaussian processes. Neural Computation, **14***(3), 641–668.*

De Kruif, B. J. and De Vries, T. J. A. (2003). Pruning error minimization in least squares support vector machines. IEEE Transactions on Neural Networks, **14***(3), 696–702.*

Del Moral, P. (1996). Non-linear filtering: interacting particle resolution. Markov processes and related fields, **2***(4), 555–581.*

Dorffner, G. (1996). Neural networks for time series processing. Neural Network World, **6***, 447–468.*

Engel, Y., Mannor, S., and Meir, R. (2004). The kernel recursive least squares algorithm. IEEE Transactions on Signal Processing, **52***(8), 2275–2285.*

Ernst, F. (2012). Compensating for quasi-periodic motion in robotic radiosurgery. *Springer.*

Frieß, T.-T. and Harrison, R. F. (1999). A kernel-based adaline. In Proceedings of the 7th European Symposium on Artificial Neural Networks (ESANN 1999)*, pages 245–250, Bruges, Belgium.*

Gao, W., Richard, C., Bermudez, J.-C. M., and Huang, J. (2014). Convex combinations of kernel adaptive filters. In 2014 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–5. IEEE.*

Ghahramani, Z. (1996). The kin datasets. URL: `http://www.cs.utoronto.ca/~delve/data/kin/kin.ps.gz`.

Golub, G. H. and Van Loan, C. F. (2012). Matrix computations*, volume 3. JHU Press.*

Hassibi, B., Stork, D. G., and Wolff, G. J. (1993). Optimal brain surgeon and general network pruning. In Neural Networks, 1993., IEEE International Conference on*, pages 293–299. IEEE.*

Hayes, M. H. M. H. (1996). Statistical digital signal processing and modeling / Monson H. Hayes. *New York : John Wiley & Sons.*

Haykin, S. (1999). Neural Networks – A Comprehensive Foundation. *Prentice Hall, 2nd edition.*

Haykin, S. (2001). Adaptive Filter Theory (4th Edition). *Prentice Hall.*

Hermans, M. and Schrauwen, B. (2012). Recurrent kernel machines: Computing with infinite echo state networks. Neural Computation, **24***(1), 104–133.*

Hoegaerts, L., Suykens, J., Vandewalle, J., and De Moor, B. (2004). A comparison of pruning algorithms for sparse least squares support vector machines. Lecture Notes in Computer Science*, pages 1247–1253.*

Ishida, T. and Tanaka, T. (2013). Multikernel adaptive filters with multiple dictionaries and regularization. In Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2013 Asia-Pacific*, pages 1–6. IEEE.*

Julier, S. J. and Uhlmann, J. K. (1996). A general method for approximating nonlinear transformations of probability distributions. Technical report, Technical report, Robotics Research Group, Department of Engineering Science, University of Oxford.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. Journal of basic Engineering, **82***(1), 35–45.*

Kivinen, J., Smola, A. J., and Williamson, R. C. (2004). Online learning with kernels. IEEE Transactions on Signal Processing, **52***(8), 2165–2176.*

Lázaro-Gredilla, M., Van Vaerenbergh, S., and Santamaría, I. (2011). A Bayesian approach to tracking with kernel recursive least-squares. In 2011 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6.*

LeCun, Y., Denker, J. S., Solla, S. A., Howard, R. E., and Jackel, L. D. (1989). Optimal brain damage. In NIPs*, volume 2, pages 598–605.*

Lewis, F. L., Xie, L., and Popa, D. (2007). Optimal and robust estimation: with an introduction to stochastic control theory*, volume 29. CRC press.*

Li, K. and Príncipe, J. C. (2016). The kernel adaptive autoregressive-moving-average algorithm. IEEE Transactions on Neural Networks and Learning Systems, **27***(2), 334–346.*

Liu, W., Pokharel, P. P., and Príncipe, J. C. (2008). The kernel least-mean-square algorithm. IEEE Transactions on Signal Processing, **56***(2), 543–554.*

Liu, W., Príncipe, J. C., and Haykin, S. (2010). Kernel Adaptive Filtering: A Comprehensive Introduction. *Wiley.*

Ljung, L. (1999). System identification: theory for the user. *Prentice-Hall, Upper Saddle River, NJ, USA.*

Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. IEEE Trans. Neur. Net., **1***(1), 4–27.*

Park, I. M., Seth, S., and Van Vaerenbergh, S. (2014). Probabilistic kernel least mean squares algorithms. In 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8272–8276. IEEE.*

Plackett, R. L. (1950). Some theorems in least squares. Biometrika, **37***, 149–157.*

Platt, J. (1991). A resource-allocating network for function interpolation. Neural computation, **3***(2), 213–225.*

Pokharel, P. P., Liu, W., and Príncipe, J. C. (2009). Kernel least mean square algorithm with constrained growth. Signal Processing, **89***(3), 257–265.*

Pokharel, R., Seth, S., and Principe, J. C. (2013). Mixture kernel least mean square. In The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE.*

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE, **77***(2), 257–286.*

Rasmussen, C. E. and Williams, C. K. I. (2006). Gaussian Processes for Machine Learning. *MIT Press.*

Richard, C., Bermudez, J. C. M., and Honeine, P. (2009). Online prediction of time series data with kernels. IEEE Transactions on Signal Processing, **57***(3), 1058–1067.*

Rzepka, D. (2012). Fixed-budget kernel least mean squares. In Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–4. IEEE.*

Saunders, C., Gammerman, A., and Vovk, V. (1998). Ridge regression learning algorithm in dual variables. In Proceedings of the 15th International Conference on Machine Learning (ICML)*, pages 515–521, Madison, WI, USA.*

Sayed, A. H. (2003). Fundamentals of Adaptive Filtering. *Wiley-IEEE Press.*

Schölkopf, B., Herbrich, R., and Smola, A. J. (2001). A generalized representer theorem. In Computational learning theory*, pages 416–426. Springer.*

Soh, H. and Demiris, Y. (2015). Spatio-temporal learning with the online finite and infinite echo-state gaussian processes. IEEE transactions on neural networks and learning systems, **26***(3), 522–536.*

Sonnenburg, S., Rätsch, G., Schäfer, C., and Schölkopf, B. (2006). Large scale multiple kernel learning. Journal of Machine Learning Research, **7***(Jul), 1531–1565.*

Takens, F. (1981). Detecting strange attractors in turbulence. Dynamical Systems and Turbulence, **898***, 366–381.*

Tuia, D., Muñoz-Marí, J., Rojo-Álvarez, J. L., Martínez-Ramón, M., and Camps-Valls, G. (2014). Explicit recursive and adaptive filtering in reproducing kernel hilbert spaces. IEEE Transactions on Neural Networks and Learning Systems, **25***(7), 1413–1419.*

Van Vaerenbergh, S. and Santamaría, I. (2013). A comparative study of kernel adaptive filtering algorithms. In 2013 IEEE Digital Signal Processing (DSP) Workshop and IEEE Signal Processing Education (SPE)*, Napa, CA, USA.*

Van Vaerenbergh, S., Vía, J., and Santamaría, I. (2006). A sliding-window kernel RLS algorithm and its application to nonlinear channel identification. In 2006 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 789–792, Toulouse, France.*

Van Vaerenbergh, S., Santamaría, I., Liu, W., and Príncipe, J. C. (2010). Fixed-budget kernel recursive least-squares. In 2010 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Dallas, TX, USA.*

Van Vaerenbergh, S., Santamaría, I., and Lázaro-Gredilla, M. (2012a). Estimation of the forgetting factor in kernel recursive least squares. In 2012 IEEE International Workshop on Machine Learning for Signal Processing (MLSP).

Van Vaerenbergh, S., Lázaro-Gredilla, M., and Santamaría, I. (2012b). Kernel recursive least-squares tracker for time-varying regression. IEEE Transactions on Neural Networks and Learning Systems, **23***(8), 1313–1326.*

Van Vaerenbergh, S., Fernandez-Bes, J., and Elvira, V. (2016a). On the relationship between online Gaussian process regression and kernel least mean squares algorithms. In 2016 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, Salerno, Italy. IEEE.*

Van Vaerenbergh, S., Comminiello, D., and Azpicueta-Ruiz, L. A. (2016b). A split kernel adaptive filtering architecture for nonlinear acoustic echo cancellation. In 24th European Signal Processing Conference (EUSIPCO 2016)*, Budapest, Hungary.*

Vapnik, V. N. (1995). The Nature of Statistical Learning Theory. *Springer-Verlag New York, Inc., New York, NY, USA.*

Widrow, B., McCool, J., and Ball, M. (1975). The complex LMS algorithm. Proceedings of the IEEE, **63***(4), 719–720.*

Yukawa, M. (2012). Multikernel adaptive filtering. IEEE Transactions on Signal Processing, **60***(9), 4672–4682.*

Zhao, S., Chen, B., Zhu, P., and Príncipe, J. C. (2013). Fixed budget quantized kernel least-mean-square algorithm. Signal Processing, **93***(9), 2759–2770.*

Zhao, S., Chen, B., Cao, Z., Zhu, P., and Principe, J. C. (2016). Self-organizing kernel adaptive filtering. EURASIP Journal on Advances in Signal Processing, **2016***(1), 106.*

INDEX

Cite as: Steven Van Vaerenbergh, "Adaptive Kernel Learning for Signal Processing". In J. L. Rojo-Álvarez, M. Martínez-Ramón, J. Muñoz-Marí, G. Camps-Valls (Eds.), Digital Signal Processing with Kernel Methods, pp. 387–431, Wiley-IEEE Press: Hoboken, NJ, USA, 2018.