Online Regression with Kernels

Steven Van Vaerenbergh and Ignacio Santamaría
University of Cantabria

March 2014

Contents

This text appeared as chapter 21 in J.A.K. Suykens, M. Signoretto, and A, Argyriou, eds. "Regularization, Optimization, Kernels, and Support Vector Machines", CRC Press, 2014.

Introduction

Online machine learning algorithms are designed to learn from one data instance at a time. They are typically used in real-time scenarios, such as prediction or tracking problems, where data arrive sequentially and instant decisions must be made. The real-time nature of these settings implies that shortly after the decision is made, the true label will be made available, which allows the learning algorithm to adjust its solution before a new datum is received.

Online kernel methods extend the nonlinear learning capabilities of standard batch kernel methods to online environments. Especially important for these techniques is that they maintain their computational load moderate during each iteration, in order to perform fast updates in real time. Ideally, they should not only be able to learn in a stationary environment but also in non-stationary settings, where they must forget outdated information and adapt their solution to respond to changes in time. Online kernel methods also find use in batch scenarios where the amount of data is too high to fit in the machine's memory, and one or several passes over the data are to be performed.

In this chapter we focus on the problem of online regression. We will give an overview of the most important kernel-based methods for this problem, which have been developed over the span of the last decade. We start by formulating the online solution to the kernel ridge regression problem, and we point out different strategies to overcome the bottlenecks associated to using kernels in online methods. The discussed techniques are often referred to as *kernel adaptive filtering* algorithms, due to their close relationship with classical adaptive filters from the signal processing literature. After reviewing the most relevant algorithms in this area, we introduce an evaluation framework that allows us to compare their performance. We finish the discussion with a brief overview of the recent and future research directions.

1 Basic principles of kernel machines

Kernel-based methods have had considerable success in a wide range of areas over the past decade, since they allow to reformulate many nonlinear problems as convex optimization problems. In this section, we briefly outline the basic principles behind kernel methods, and we introduce the main ideas behind constructing online kernel-based algorithms.

1.1 Kernel methods

Kernel methods rely on a nonlinear transformation of the input data $\mathbf{x}$ into a high-dimensional reproducing kernel Hilbert space (RKHS), in which it is more likely that the transformed data $\phi(\mathbf{x})$ are linearly separable. In this space, denoted the *feature space*, inner products can be calculated by using a positive definite kernel function $\kappa(\cdot,\cdot)$ satisfying Mercer's condition:

$$\kappa(\mathbf{x},\mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle, \tag{1}$$

where $\mathbf{x}$ and $\mathbf{x}'$ represent two different data points. This duality between positive definite kernels and feature spaces allows to transform any inner-product based linear algorithm to an alternative, nonlinear algorithm by replacing the inner products with kernels [1, 15]. The solution, obtained as a linear functional in the feature space, then corresponds to the solution of a nonlinear problem in the input space.

Thanks to the Representer Theorem [14], a large class of optimization problems in RKHS have solutions that can be expressed as kernel expansions in terms of the training data only. Specifically,

kernel-based learning aims at finding a nonlinear relationship $f : \mathcal{X} \to \mathbb{R}$ that can be expressed as the kernel expansion

$$f(\mathbf{x}) = \sum_{n=1}^{N} \alpha(n)\kappa(\mathbf{x}_n, \mathbf{x}). \tag{2}$$

In this relationship, N is the number of training data, and $\alpha(n) \in \mathbb{R}$ are denoted the *expansion coefficients*. The training data $\mathbf{x}_n$ used in this expansion are sometimes referred to as *bases*.

Kernel methods are non-parametric techniques, since they do not assume any specific model. Indeed, their solution is expressed as the functional representation (2) that relies explicitly on the training data.

As an introductory example algorithm, we will now describe the batch approach to the standard kernel regression problem, known as kernel ridge regression. Its formulation will lie at the core of the online algorithms we will review later in this chapter.

1.2 Kernel ridge regression

Assume we are given N input data points $\mathbf{x}_n$, $n = 1, \ldots, N$, in a D-dimensional space, and the corresponding outputs y_n. In order to adopt a matrix-based formulation, we stack the input data into an $N \times D$ matrix $\mathbf{X}$ and the output data into the vector $\mathbf{y}$. In linear regression, the regularized least-squares problem consists in seeking a vector $\mathbf{w} \in \mathbb{R}^{D\times 1}$ that solves

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + c\|\mathbf{w}\|^2, \tag{3}$$

where c is a positive Tikhonov regularization constant. The solution to this problem is given by

$$\hat{\mathbf{w}} = (\mathbf{X}^\top\mathbf{X} + c\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}. \tag{4}$$

In the absence of regularization, i.e. $c = 0$, the solution is only well defined when $\mathbf{X}$ has full column rank, and thus $(\mathbf{X}^\top\mathbf{X})^{-1}$ exists.

In order to obtain the kernel-based version of Eq. (3), we first transform the data and the solution into the feature space,

$$\min_{\phi(\mathbf{w})} \|\mathbf{y} - \phi(\mathbf{X})\phi(\mathbf{w})\|^2 + c\|\phi(\mathbf{w})\|^2. \tag{5}$$

Here, the shorthand notation $\phi(\mathbf{X})$ refers to the data matrix that contains the transformed data, stacked as rows, $\phi(\mathbf{X}) = [\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_N)]^\top$. The Representer Theorem [14] states that the solution $\phi(\mathbf{w})$ of this problem can be expressed as a linear combination of the training data, in particular

$$\phi(\mathbf{w}) = \sum_{n=1}^{N} \alpha(n)\phi(\mathbf{x}_n) = \phi(\mathbf{X})^\top\boldsymbol{\alpha}, \tag{6}$$

where $\boldsymbol{\alpha} = [\alpha(1), \ldots, \alpha(N)]^\top$. After substituting Eq. (6) into Eq. (5), and defining the matrix $\mathbf{K} = \phi(\mathbf{X})\phi(\mathbf{X})^\top$, we obtain

$$\min_{\boldsymbol{\alpha}} \|\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}\|^2 + c\boldsymbol{\alpha}^\top\mathbf{K}\boldsymbol{\alpha}. \tag{7}$$

The matrix $\mathbf{K}$ is denoted as the *kernel matrix*, and its elements represent the inner products in the feature space, calculated as kernels $k_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$.

Eq. (7) represents the kernel ridge regression problem [12], and its solution is given by

$$\hat{\boldsymbol{\alpha}} = (\mathbf{K} + c\mathbf{I})^{-1}\mathbf{y}. \tag{8}$$

2 Framework for online kernel methods

Online learning methods update their solution iteratively. In the standard online learning framework, each iteration consists of several steps, as outlined in Alg. 1. During the n-th iteration, the algorithm first receives an input datum, $\mathbf{x}_n$. Then, it calculates the estimated output $\hat{y}_n$ corresponding to this datum. The learning setup is typically supervised, in that the true outcome y_n is made available next, which enables the algorithm to calculate the loss $L(\cdot)$ incurred on the data pair $(\mathbf{x}_n, y_n)$, and, finally, to update its solution. Initialization is typically performed by setting the involved weights to zero.

Algorithm 1 Protocol for online, supervised learning.

Initialize variables as empty or zero.
for $n = 1, 2, \ldots$ **do**
  Observe input $\mathbf{x}_n$.
  Predict output $\hat{y}_n$.
  Observe true output y_n.
  Update solution based on $L(e_n)$, with $e_n = y_n - \hat{y}_n$.
end for

A typical setup for online system identification with a kernel-based method is depicted in Fig. 1. It represents an unknown nonlinear system, whose input data $\mathbf{x}_n$ and response y_n (including additive noise r_n) can be measured at different time steps, and an adaptive kernel-based algorithm, which is used to identify the system's response.

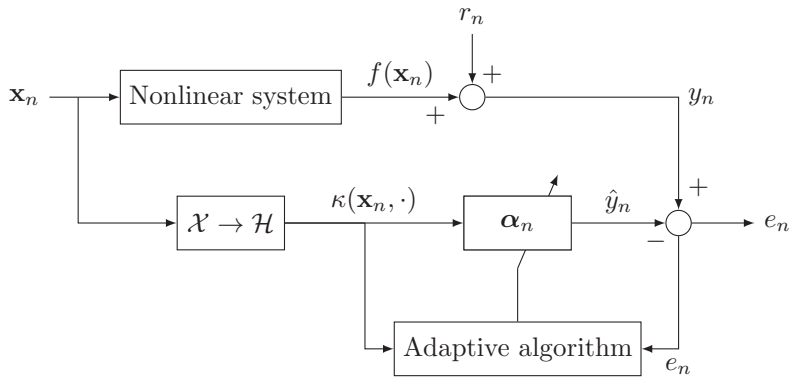

Figure 1: Kernel-based adaptive system identification. Figure adapted from [8].

Online algorithms should be capable of operating during extended periods of time, processing large amounts of data. Kernel methods rely on the functional representation (2), which grows as the amount of observations increases. A naive implementation of an online kernel method will therefore require growing computational resources during operation, leading to performance issues once the available memory is insufficient to store the training data or once the computations for one update take more time than the interval between incoming data [6].

The standard approach to overcome this issue is to limit the number of terms in Eq. (2) to

a representative subset of observations, called a *dictionary*. In the sequel, we will review several dictionary-learning methods that can be carried out online. They will serve as building blocks for online kernel methods.

2.1 Online dictionary learning

The dictionary learning process aims to identify the bases in the expansion (2) that can be discarded without incurring a significant performance loss. After discarding these bases, a new, approximate expansion is obtained as

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{m} \alpha(i)\kappa(\mathbf{u}_i, \mathbf{x}), \tag{9}$$

where $m < N$. The dictionary $\mathcal{D} = \{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$ consists of several data vectors $\mathbf{u}_i$ which are selected from the input data $\mathbf{x}_n$. They should be chosen carefully so that they represent the entire input data set sufficiently well.

In the online setting, the classical way of constructing a dictionary is by *growing*, i.e. by starting from an empty dictionary and gradually adding those bases that fulfill a certain 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*. In Table 1 we have listed several criteria to grow dictionaries, in the top half, and several criteria to prune them, in the lower half. We will review these criteria here briefly, and revisit them in detail when discussing the learning algorithms that use them.

Table 1: Standard criteria for deciding whether to include new data when growing a dictionary, and which data to prune.

criterion	type	complexity
all	growing	—
coherence	growing	$\mathcal{O}(m)$
ALD	growing	$\mathcal{O}(m^2)$
oldest	pruning	—
least weight	pruning	$\mathcal{O}(m)$
least a posteriori SE	pruning	$\mathcal{O}(m^2)$

A simple criterion to check whether the newly arriving datum is sufficiently informative is called the *coherence criterion* [11]. Given the dictionary $\mathcal{D}$ at some iteration and the newly arriving datum $\mathbf{x}$, this criterion defines the *coherence* of the datum as the quantity

$$\mu = \max_{\mathbf{u}_i \in \mathcal{D}} \kappa(\mathbf{u}_i, \mathbf{x}). \tag{10}$$

In essence, the coherence criterion checks the similarity, as measured by the kernel function, between the new datum and the most similar dictionary point. Only if the coherence is below a certain predefined threshold, $\mu \leq \mu_0$, 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 a simple and effective criterion that has low computational complexity: it only requires to calculate m kernel functions.

A more sophisticated dictionary growth criterion was introduced in [5]. Whenever a new datum is observed, this 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 following approximate linear dependence (ALD) condition holds:

$$\delta := \min_{\mathbf{a}} \left\| \sum_{i=1}^{m} a(i)\phi(\mathbf{u}_i) - \phi(\mathbf{x}) \right\|^2 \leq \nu, \tag{11}$$

where ν is a precision threshold. The lower ν is chosen, the more data will be accepted into the dictionary. In contrast to the coherence criterion, which only compares the new datum to one dictionary basis at a time, the ALD criterion looks for the best combination of all bases. This search corresponds to a least-squares problem, which, if solved iteratively, requires quadratic complexity in terms of M, per iteration. In comparison to the coherence criterion, ALD is computationally more complex. In return, it is able to construct more compact dictionaries that represent the same information with fewer bases.

In practice, it is often necessary to specify a maximum dictionary size M, or *budget*, that may not be exceeded. In order to avoid exceeding the budget, one could simply stop allowing any inclusions in the dictionary once the budget is reached, hence locking the dictionary. Nevertheless, it is not unimaginable that after reaching the budget some new datum may still be observed that is more informative than a currently stored dictionary basis. In this case, the quality of the algorithm's solution will improve by pruning the said dictionary basis and by adding the new, more informative datum.

At this point it is interesting to remark that there exists a conceptual difference between growing and pruning strategies: While growth criteria are concerned with determining *whether or not* to include a new datum, pruning criteria deal with determining *which* datum to discard.

In time-varying environments, it may be useful to simply discard the oldest bases, as these were observed when the underlying model was possibly most different from the current model. This strategy is at the core of sliding-window algorithms, which, in every iteration, accept the new datum and discard the oldest basis, thereby maintaining a dictionary of fixed size [22].

A different pruning strategy is obtained by observing that the solution takes the form of the functional representation (9). In this kernel expansion, each dictionary element $\mathbf{u}_i$ has an associated weight $\alpha(i)$. Hence, a low-complexity pruning strategy simply consists in discarding the dictionary element that has the least weight $|\alpha(i)|$ associated to it, as proposed in [16].

A more sophisticated pruning strategy was introduced in [3] and [4]. It consists in selecting the element that causes the least squared error (SE) to the solution after being pruned. The relevance of a dictionary basis, according to this criterion, can be calculated as

$$\frac{|\alpha(i)|}{[\mathbf{K}^{-1}]_{ii}}, \tag{12}$$

where $[\mathbf{K}^{-1}]_{ii}$ is the i-th element on the diagonal of the inverse kernel matrix, calculated for the dictionary bases. We will refer to this criterion as the *least a posteriori SE criterion*. Similarly to the ALD criterion, $\mathbf{K}^{-1}$ and $\boldsymbol{\alpha}$ can be updated iteratively, yielding a complexity per iteration of $\mathcal{O}(m^2)$.

3 Kernel recursive least-squares regression methods

We now formulate the recursive update for the kernel ridge regression solution (8), known as kernel recursive least-squares (KRLS). First, we describe the evergrowing approach, in which all training

data appear in the solution. By introducing dictionary strategies into this technique, we then obtain several different practical algorithms that limit their solution's growth.

3.1 Recursive updates of the least-squares solution

Assume an online scenario in which $n-1$ data have been processed at the $n-1$-th iteration. The regression solution from Eq. (8) reads

$$\boldsymbol{\alpha}_{n-1} = \dot{\mathbf{K}}_{n-1}^{-1}\mathbf{y}_{n-1}, \tag{13}$$

where we denote the regularized kernel matrix with a dot, $\dot{\mathbf{K}} = \mathbf{K} + c\mathbf{I}$, in order to simplify the notation. In the next iteration, n, a new data pair $(\mathbf{x}_n, y_n)$ is received and we wish to update the solution (13) recursively. Following the online protocol from Alg. 1, we first calculate the predicted output

$$\hat{y}_n = \mathbf{k}_n^\top \boldsymbol{\alpha}_{n-1}, \tag{14}$$

in which $\mathbf{k}_n = [\kappa(\mathbf{x}_n, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}_n, \mathbf{x}_{n-1})]^\top$, and we obtain the a priori error for this datum, $e_n = y_n - \hat{y}_n$. The updated kernel matrix is

$$\dot{\mathbf{K}}_n = \begin{bmatrix} \dot{\mathbf{K}}_{n-1} & \mathbf{k}_n \\ \mathbf{k}_n^\top & k_{nn} + c \end{bmatrix}, \tag{15}$$

with $k_{nn} = \kappa(\mathbf{x}_n, \mathbf{x}_n)$. By introducing the variables

$$\mathbf{a}_n = \dot{\mathbf{K}}_{n-1}^{-1}\mathbf{k}_n, \tag{16}$$

and

$$\gamma_n = k_{nn} + c - \mathbf{k}_n^\top \mathbf{a}_n, \tag{17}$$

the new inverse kernel matrix is calculated as

$$\dot{\mathbf{K}}_n^{-1} = \frac{1}{\gamma_n}\begin{bmatrix} \gamma_n \dot{\mathbf{K}}_{n-1}^{-1} + \mathbf{a}_n\mathbf{a}_n^\top & -\mathbf{a}_n \\ -\mathbf{a}_n & 1 \end{bmatrix}. \tag{18}$$

Finally, the updated solution $\boldsymbol{\alpha}_n$ is obtained as

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} - \mathbf{a}_n e_n/\gamma_n \\ e_n/\gamma_n \end{bmatrix}. \tag{19}$$

Equations (18) and (19) 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. (13) at iteration n would require $\mathcal{O}(n^3)$ cost, so the recursive procedure is preferred in online scenarios. For a detailed derivation of this evergrowing formulation we refer to [5, 18].

3.2 Approximate Linear Dependency KRLS algorithm

The KRLS algorithm from [5] uses the recursive solution we described previously, and it introduces the ALD criterion in order to reduce the growth of the functional representation. While the name KRLS was coined for the algorithm proposed in [5], this algorithm is only one of the many possible

implementations of the KRLS principle. We will therefore refer to it as ALD-KRLS. An outline of this algorithm is given in Alg. 2.

At every iteration, ALD-KRLS decides whether or not to increase its order, based on its dictionary growth criterion. If the criterion is fulfilled, it performs a *full update*, consisting of an order increase of the dictionary and the algorithm variables. If the criterion is not fulfilled, the dictionary is maintained, but instead of simply discarding the data pair $(\mathbf{x}_n, y_n)$ altogether, the solution coefficients are updated (though not expanded) with the information contained in this datum. We denote this type of update as a *reduced update*.

Algorithm 2 KRLS algorithm with reduced growth.

```
Initialize variables as empty or zero.
for n = 1, 2, ... do
  Observe input x_n.
  Predict output: Eq. (14)
  Observe true output y_n.
  if dictionary growth criterion is fulfilled then
    Expand dictionary: D_n = D_{n-1} ∪ {x_n}
    Update inverse kernel matrix: Eq. (18)
    Update expansion coefficients: Eq. (19)
  else
    Maintain dictionary: D_n = D_{n-1}
    Perform reduced update of expansion coefficients.
  end if
end for
```

ALD-KRLS does not include regularization, $c = 0$. In this case, the coefficients $\mathbf{a}_n = [a_n(1), \ldots, a_n(m)]^\top$ that minimize the ALD condition (11) are given by Eq. (16), and the norm of the best linear combination is given by $\delta_n = \gamma_n$. In order to perform a reduced update, ALD-KRLS keeps track of a projection matrix that is used to project the information of redundant bases onto the current solution, before discarding them. For detail, refer to [5].

3.3 Sliding-Window KRLS algorithm

ALD-KRLS assumes a stationary model, and, therefore, it is not suitable as a tracking algorithm. In addition, there does not seem to be an obvious extension that allows for tracking, mainly because it summarizes past information into a compact formulation that allows for little manipulation. This is a somewhat surprising fact, taking into account that it is derived from the linear recursive least-squares (RLS) algorithm, which is easily extendible into several tracking formulations, for instance by considering a forgetting factor.

In order to address this issue, a Sliding-Window KRLS (SW-KRLS) algorithm was proposed in [22]. Instead of summarizing previous data, it simply stores a window of the last M data as its dictionary. In each step it adds the new datum and discards the oldest datum, leading to a sliding-window approach. In order to expand the inverse kernel matrix with a new datum it uses Eq. (18). To discard the oldest datum it relies on the following relationship. By breaking up the

kernel matrix and its inverse as

$$\dot{\mathbf{K}}_{n-1} = \begin{bmatrix} a & \mathbf{b}^T \\ \mathbf{b} & \mathbf{D} \end{bmatrix}, \quad \dot{\mathbf{K}}_{n-1}^{-1} = \begin{bmatrix} e & \mathbf{f}^T \\ \mathbf{f} & \mathbf{G} \end{bmatrix}, \tag{20}$$

the inverse of the reduced kernel matrix is found as

$$\mathbf{D}^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e. \tag{21}$$

Finally, the coefficients $\boldsymbol{\alpha}_n$ are obtained through Eq. (13), in which the vector $\mathbf{y}_n$ now contains only the M last outputs. This vector is stored along with the dictionary.

SW-KRLS is a conceptually very simple algorithm that obtains reasonable performance in a wide range of scenarios, including non-stationary environments. Nevertheless, its performance is limited by the quality of the bases in its dictionary, over which it has no 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 [21]. Instead of discarding the oldest data point in each iteration, it discards the data point that causes the least error upon being discarded, using the least a posteriori SE pruning criterion from Table 1. In stationary scenarios, this extension obtains significantly better results. In non-stationary cases, however, it does not offer any advantage, and a different approach is required, as we discuss in the sequel.

3.4 Bayesian interpretation

The standard KRLS equations, as described above, can also be derived from a Bayesian perspective. As we will see, the obtained solution is equivalent to the KRLS update, though the Bayesian approach does not only provide the mean value for the predicted solution but also its entire posterior distribution. The full derivation, which is based on the framework of Gaussian Processes (GPs) [10], can be found in [18, 9].

A Bayesian setting requires a model that describes the observations, and priors on the parameters of this model. The GP regression model assumes that the outputs can be modeled as some noiseless latent function of the inputs plus an independent noise component

$$y = f(\mathbf{x}) + r, \tag{22}$$

and then sets a zero mean[1] GP prior on $f(\mathbf{x})$ and a Gaussian prior on r:

$$f(\mathbf{x}) \sim \mathcal{GP}(\mathbf{0}, \kappa(\mathbf{x}, \mathbf{x}')), \qquad r \sim \mathcal{N}(0, \sigma^2), \tag{23}$$

where σ^2 is a hyperparameter that specifies the noise power. The notation $\mathcal{GP}(m(\mathbf{x}), \kappa(\mathbf{x}, \mathbf{x}'))$ refers to a GP distribution over functions in terms of a mean function $m(\mathbf{x})$ (zero in this case) and a *covariance* function $\kappa(\mathbf{x}, \mathbf{x}')$, equivalent to a kernel function. The covariance function specifies the a priori relationship between values $f(\mathbf{x})$ and $f(\mathbf{x}')$ in terms of their respective locations, and it is parameterized by a small set of hyperparameters, grouped in vector $\boldsymbol{\theta}$.

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 homonymous functions evaluated

[1] It is customary to subtract the sample mean from the data $\{y_n\}_{n=1}^N$, and then to assume a zero-mean model.

at those points. Thus, the joint distribution of outputs $\mathbf{y} = [y_1, \ldots, y_N]^\top$ and the corresponding latent vector $\mathbf{f} = [f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N)]^\top$ is

$$\begin{bmatrix}\mathbf{y}\\ \mathbf{f}\end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix}\mathbf{K}+\sigma^2\mathbf{I} & \mathbf{K}\\ \mathbf{K} & \mathbf{K}\end{bmatrix}\right). \tag{24}$$

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

$$\begin{aligned} p(\mathbf{f}|\mathbf{y}) &= \mathcal{N}(\mathbf{f}|\mathbf{K}(\mathbf{K}+\sigma^2\mathbf{I})^{-1}\mathbf{y}, \mathbf{K}-\mathbf{K}(\mathbf{K}+\sigma^2\mathbf{I})^{-1}\mathbf{K}) \\ &= \mathcal{N}(\mathbf{f}|\boldsymbol{\mu}, \boldsymbol{\Sigma}). \end{aligned} \tag{25}$$

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

$$p(y_n|\mathbf{x}_n, \mathbf{y}_{n-1}) = \mathcal{N}(y_n|\mu_{\mathrm{GP},n}, \sigma^2_{\mathrm{GP},n}) \tag{26a}$$

$$\mu_{\mathrm{GP},n} = \mathbf{k}_n^\top(\mathbf{K}_{n-1}+\sigma^2\mathbf{I})^{-1}\mathbf{y}_{n-1} \tag{26b}$$

$$\sigma^2_{\mathrm{GP},n} = \sigma^2 + k_{nn} - \mathbf{k}_n^\top(\mathbf{K}_{n-1}+\sigma^2\mathbf{I})^{-1}\mathbf{k}_n. \tag{26c}$$

The mode of the predictive distribution, given by $\mu_{\mathrm{GP},n}$ in Eq. (26b), coincides with the solution of KRLS, given by Eq. (18), showing that the regularization in KRLS can be interpreted as a noise power σ^2. Furthermore, the variance of the predictive distribution, given by $\sigma^2_{\mathrm{GP},n}$ in Eq. (26c), coincides with Eq. (17), which is used by the dictionary criterion for ALD-KRLS.

Using Eqs. (26), a recursive update of the complete GP can be found as

$$p(\mathbf{f}_n|\mathbf{X}_n, \mathbf{y}_n) = \mathcal{N}(\mathbf{f}_n|\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n) \tag{27a}$$

$$\boldsymbol{\mu}_n = \begin{bmatrix}\boldsymbol{\mu}_{n-1}\\ \hat{y}_n\end{bmatrix} + \frac{e_n}{\hat{\sigma}^2_{yn}}\begin{bmatrix}\mathbf{h}_n\\ \hat{\sigma}^2_{fn}\end{bmatrix} \tag{27b}$$

$$\boldsymbol{\Sigma}_n = \begin{bmatrix}\boldsymbol{\Sigma}_{n-1} & \mathbf{h}_n\\ \mathbf{h}_n^\top & \hat{\sigma}^2_{fn}\end{bmatrix} - \frac{1}{\hat{\sigma}^2_{yn}}\begin{bmatrix}\mathbf{h}_n\\ \hat{\sigma}^2_{fn}\end{bmatrix}\begin{bmatrix}\mathbf{h}_n\\ \hat{\sigma}^2_{fn}\end{bmatrix}^\top, \tag{27c}$$

where $\mathbf{h}_n = \boldsymbol{\Sigma}_{n-1}\mathbf{K}_{n-1}^{-1}\mathbf{k}_n$, and $\hat{\sigma}^2_{fn}$ and $\hat{\sigma}^2_{yn}$ are the predictive variances of the latent function and the new output, respectively, calculated at the new input. Details can be found in [18].

The update equations (27) are formulated in terms of the predictive mean and covariance, $\boldsymbol{\mu}_n$ and $\boldsymbol{\Sigma}_n$, which allows us to interpret them directly in terms of the underlying GP. They can be reformulated in terms of $\boldsymbol{\alpha}_n$ and a corresponding matrix, as shown in [3]. Specifically, the relationship between $\boldsymbol{\mu}_n$ and $\boldsymbol{\alpha}_n$ is (see [18])

$$\boldsymbol{\alpha}_n = \mathbf{K}_n^{-1}\boldsymbol{\mu}_n \tag{28a}$$

$$= \dot{\mathbf{K}}_n^{-1}\mathbf{y}_n \tag{28b}$$

Interestingly, while standard KRLS obtains $\boldsymbol{\alpha}_n$ based on the *noisy* observations $\mathbf{y}_n$ through Eq. (28b), the GP-based formulation shows that $\boldsymbol{\alpha}_n$ can be obtained equivalently using the values of the *noiseless* function evaluated at the inputs $\boldsymbol{\mu}_n$, through Eq. (28a).

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 [9]. This allows, for instance, to establish confidence intervals when predicting new outputs. And, more importantly in adaptive contexts, it allows to explicitly handle the uncertainty about all learned data. In the sequel, we will review a recursive algorithm that exploits this knowledge to perform tracking.

3.5 KRLS Tracker algorithm

In [18], a KRLS Tracker (KRLS-T) algorithm was devised that explicitly handles uncertainty about the data, based on the above discussed probabilistic Bayesian framework. While in stationary environments it operates identically to the earlier proposed Sparse Online GP algorithm (SOGP) from [3], it includes a *forgetting mechanism* that enables it to handle non-stationary scenarios as well.

In non-stationary scenarios, adaptive online algorithms should be capable of tracking the changes of the observed model. This is possible by weighting past data less heavily than more recent data. A quite radical example of forgetting is provided by the SW-KRLS algorithm, which either assigns full validity to the data (those in its window), or discards them entirely.

KRLS-T includes a framework that permits several forms of forgetting. We focus on the forgetting strategy called "back to the prior" (B2P), in which the mean and covariance are replaced through

$$\boldsymbol{\mu} \leftarrow \sqrt{\lambda}\boldsymbol{\mu} \tag{29a}$$

$$\boldsymbol{\Sigma} \leftarrow \lambda\boldsymbol{\Sigma} + (1-\lambda)\mathbf{K}. \tag{29b}$$

As shown in [18], 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 [18]).

Equations (29) may seem like an *ad-hoc* step to enable forgetting. However, it can be shown that the whole learning procedure —including the mentioned controlled forgetting step— corresponds exactly to a principled non-stationary scheme within the GP framework, as described in [20]. It is sufficient to consider an augmented input space that includes the time stamp t of each sample and define a *spatio-temporal* covariance function:

$$\kappa_{\mathrm{st}}([t \;\; \mathbf{x}^\top]^\top, [t' \;\; \mathbf{x}'^\top]^\top) = \kappa_{\mathrm{t}}(t,t')\kappa_{\mathrm{s}}(\mathbf{x},\mathbf{x}'), \tag{30}$$

where $\kappa_{\mathrm{s}}(\mathbf{x},\mathbf{x}')$ is the already-known spatial covariance function and $\kappa_{\mathrm{t}}(t,t')$ is a temporal covariance function giving more weight to samples that are closer in time. Inference on this augmented model effectively accounts for non-stationarity in $f(\cdot)$ and recent samples have more impact in predictions for the current time instant. It is fairly simple to include this augmented model in the online learning process described in the previous section. When the temporal covariance is set to $k_{\mathrm{t}}(t,t') = \lambda^{\frac{|t-t'|}{2}}$, $\lambda \in (0,1]$, inference in the augmented spatio-temporal GP model is exactly equivalent to using (29) after each update (27).

This equivalence has interesting consequences. Most importantly, it implies that the optimal hyperparameters for the recursive problem can be determined by performing standard GP hyperparameter estimation techniques, such as Type-II Maximum Likelihood estimation, on the equivalent spatio-temporal batch problem. This is an important accomplishment in kernel adaptive filtering theory, as it allows to determine the hyperparameters in a principled manner, including kernel parameters, the noise level and the forgetting factor λ. See [20] for further details.

The KRLS-T algorithm is summarized in Alg. 3. Its first step in each iteration consists in applying a forgetting strategy, which takes away some of the weight of older information. KRLS-T

Algorithm 3 KRLS Tracker (KRLS-T) algorithm.

Initialize variables as empty or zero.
for $n = 1, 2, \ldots$ **do**
  Forget: replace $\boldsymbol{\mu}_{n-1}$ and $\boldsymbol{\Sigma}_{n-1}$ through Eqs. 29
  Observe input $\mathbf{x}_n$.
  Calculate predictive mean: $\hat{y}_n = \mathbf{k}_n \mathbf{K}_{n-1}^{-1} \boldsymbol{\mu}_{n-1}$
  Calculate predictive variance $\hat{\sigma}_{yn}^2$.
  Observe true output y_n.
  Compute $\boldsymbol{\mu}_n$, $\boldsymbol{\Sigma}_n$, $\mathbf{K}_n^{-1}$.
  Add basis $\mathbf{x}_n$ to the dictionary.
  if number of bases in the dictionary $> M$ **then**
    Determine the least relevant basis, $\mathbf{u}_m$.
    Remove basis $\mathbf{u}_m$ from $\boldsymbol{\mu}_n$, $\boldsymbol{\Sigma}_n$, $\mathbf{K}_n^{-1}$
    Remove basis $\mathbf{u}_m$ from the dictionary.
  end if
end for

accepts every datum into its dictionary (as long as this does not render $\mathbf{K}_n^{-1}$ rank-deficient), and at the end of each iteration it prunes the least relevant basis, after projecting its information onto the remaining bases. For pruning, it uses the least a posteriori SE (see Table 1). Additional details can be found in [18].

4 Stochastic gradient descent with kernels

Up till this point we have reviewed several online algorithms that recursively estimate the batch solution to the kernel ridge regression problem. These algorithms have quadratic complexity in terms of the number of data that they store in their dictionary, $\mathcal{O}(m^2)$, which may be excessive in certain scenarios. It is possible to obtain algorithms with lower complexity, typically $\mathcal{O}(m)$, by performing approximations to the optimal recursive updates, as we will describe here.

The starting point is, again, the kernel ridge regression problem, which we repeat for convenience:

$$\min_{\phi(\mathbf{w})} J = \|\mathbf{y} - \phi(\mathbf{X})\phi(\mathbf{w})\|^2 + c\|\phi(\mathbf{w})\|^2.$$

Earlier, we dealt with techniques that focus on the batch solution to this problem. The same solution can be obtained through an iterative procedure, called the steepest-descent method [13]. It consists in iteratively applying the rule

$$\phi(\mathbf{w}) \leftarrow \phi(\mathbf{w}) - \frac{\eta}{2} \frac{\partial J}{\partial \phi(\mathbf{w})}, \tag{31}$$

where η represents a learning rate. After replacing the derivative in Eq. (31) by its instantaneous estimate, and, omitting regularization momentarily, we obtain a low-cost online algorithm with the following stochastic gradient descent update rule

$$\begin{aligned} \phi(\mathbf{w}_n) &= \phi(\mathbf{w}_{n-1}) + \eta \left(y_n \phi(\mathbf{x}_n) - \phi(\mathbf{x}_n)\phi(\mathbf{w}_{n-1})^\top \phi(\mathbf{x}_n) \right) \\ &= \phi(\mathbf{w}_{n-1}) + \eta e_n \phi(\mathbf{x}_n) \end{aligned}$$

By relying on the Representer Theorem [14], $\phi(\mathbf{w}_n)$ and $\phi(\mathbf{w}_{n-1})$ are expressed as linear combinations of the transformed data, yielding

$$\sum_{i=1}^{n} \alpha_n(i)\phi(\mathbf{x}_i) = \sum_{i=1}^{n-1} \alpha_{n-1}(i)\phi(\mathbf{x}_i) + \eta e_n \phi(\mathbf{x}_n), \tag{32}$$

where $\alpha_n(i)$ denotes the i-th element of the vector $\boldsymbol{\alpha}_n$. If regularization is not omitted, the update rule reads

$$\sum_{i=1}^{n} \alpha_n(i)\phi(\mathbf{x}_i) = (1-\eta c)\sum_{i=1}^{n-1} \alpha_{n-1}(i)\phi(\mathbf{x}_i) + \eta e_n \phi(\mathbf{x}_n) \tag{33}$$

The update (32) is the core equation used to derive kernel least mean square (KLMS) algorithms. In essence, these algorithms are kernelized versions of the classical least-mean-squares (LMS) algorithm [13]. Similar to the previously discussed online kernel algorithms, KLMS algorithms usually also build an online dictionary, but since their core update is of linear complexity in term of the number of points in the dictionary, $\mathcal{O}(M)$, their dictionary update should not exceed this complexity.

KLMS algorithms possess the interesting property that their learning rule also provides them with a tracking mechanism, at no additional cost. KRLS algorithms, on the other hand, require their standard design to be specifically extended in order to obtain this property, as we discussed. In what follows we will discuss the mechanics of the three most popular KLMS algorithms, highlighting their similarities and differences.

Several forms exist to obtain the new weights $\boldsymbol{\alpha}_n$ in such a way that Eq. (32) holds. The simplest form to obtain the weights at step n consists in maintaining the previous weights, i.e. $\alpha_n(i) = \alpha_{n-1}(i)$, for $i = 1, \ldots, n-1$ and adding a new weight $\alpha_n(n)$ that accounts for the term $\eta e_n \phi(\mathbf{x}_n)$. This is the update mechanism behind NORMA [6] and Q-KLMS [2].

4.1 Naive Online regularized Risk Minimization Algorithm

Naive Online regularized Risk Minimization Algorithm (NORMA) is a family of stochastic-gradient online kernel-based algorithms [6]. It includes regularization and thus uses Eq. (33) as it basic update. We will focus on its standard form for regression with a squared loss function. By concentrating all the novelty in the new coefficients α_n, the update for NORMA at time step n reads

$$\boldsymbol{\alpha}_n = \begin{bmatrix} (1-\eta c)\boldsymbol{\alpha}_{n-1} \\ \eta e_n \end{bmatrix}, \tag{34}$$

Note that the coefficients shrink as n grows. Therefore, after a certain amount of iterations, the oldest coefficient can be discarded without affecting the solution's quality. Hence, a sliding-window dictionary mechanism is obtained that prevents the functional representation from growing too large during online operation.

4.2 Quantized KLMS

A second algorithm that concentrates all the novelty in one coefficient is quantized KLMS (Q-KLMS) [2]. The main characteristic of Q-KLMS is that it slows down its growth by constructing a dictionary through a *quantization* process. For each new datum, Q-KLMS uses the coherence criterion from [11] to check whether or not to add the datum to the dictionary. Q-KLMS was

proposed based on a specific version of the coherence criterion that uses the Euclidean distance, though any kernel could be used in its criterion. Note that the calculation of the coherence criterion has linear cost in terms of the current dictionary size, M, making it especially useful for KLMS algorithms.

If the coherence condition is not fulfilled, $\mu \leq \mu_0$, Q-KLMS adds the new datum to its dictionary and updates the coefficients as follows

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ \eta e_n \end{bmatrix}. \tag{35}$$

If the coherence condition is fulfilled, $\mu > \mu_0$, Q-KLMS only updates the coefficient of the dictionary element that is closest to the new datum. If we denote the index of the closest dictionary element as j, the update rule for this case reads

$$\alpha_n(j) = \left[\alpha_{n-1}(j) + \eta e_n\right], \tag{36}$$

Note that Q-KLMS does not include regularization. As shown in [7], it is member of a class of KLMS algorithms that possess a self-regularizing property.

4.3 Kernel Normalized LMS

Instead of concentrating all the novelty in one coefficient, a different approach is followed in [11]. Specifically, the new coefficient vector $\boldsymbol{\alpha}_n$ is obtained by projecting the previous vector, $\boldsymbol{\alpha}_{n-1}$, onto the line defined by $\mathbf{k}_n^\top \boldsymbol{\alpha} - y_n = 0$. As shown in [11], this yields the following normalized KLMS update

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ 0 \end{bmatrix} + \frac{\eta e_n}{\epsilon + \|\mathbf{k}_n\|^2} \begin{bmatrix} \mathbf{k}_n \\ k_{nn} \end{bmatrix}, \tag{37}$$

when the dictionary is to be expanded, where ϵ is a regularization constant. This algorithm, denoted kernel normalized LMS (KNLMS) [11], uses the online sparsification based on the coherence criterion to determine whether or not to include new data. In particular, when the new datum does not meet the coherence condition, KNLMS updates its coefficients without increasing the order, following the rule

$$\boldsymbol{\alpha}_n = \boldsymbol{\alpha}_{n-1} + \frac{\eta e_n}{\epsilon + \|\mathbf{k}_n\|^2} \mathbf{k}_n. \tag{38}$$

5 Performance comparisons

The regression performance of online kernel methods has been studied in several ways. The standard manner to compare the performance is to analyze their learning curves, which depict their regression error over time. This, however, requires to choose the parameters for each algorithm that are optimal in some sense. We will analyze some learning curves first, on a standard data set, and then we will show how a more global comparison can be obtained that encompasses multiple parameter configurations for each algorithm. Unless stated otherwise, we will use a Gaussian kernel of the form

$$\kappa(\mathbf{x}, \mathbf{x}') = \sigma_0^2 \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\ell^2}\right), \tag{39}$$

in which σ_0^2 is the signal power and ℓ is the length scale.

The results from this section can be reproduced by the code included in an open-source toolbox, available at http://sourceforge.net/projects/kafbox/, which we have developed specifically for this purpose. It contains implementations of the most popular kernel adaptive filtering algorithms.

5.1 Online regression on the KIN40K data set

In the first experiment we train the online algorithms to perform regression of the KIN40K data set[2]. This data set is obtained from the forward kinematics of an 8-link all-revolute robot arm, and it represents a stationary regression problem. It contains 40000 examples, each consisting of an 8-dimensional input vector and a scalar output. We randomly select 10000 data points for training and used the remaining 30000 points for testing the regression.

For all algorithms we use an anisotropic Gaussian kernel in which the hyperparameters were determined offline by standard GP regression. In particular, the noise-to-signal ratio was $\sigma_n^2/\sigma_0^2 = 0.0021$. Each algorithm performs a single run over the data. The performance is measured as the normalized mean-square error (NMSE) on the test data set at different points throughout the training run.

The results are displayed in Fig. 2. The algorithm-specific parameters were set as follows: ALD-KRLS uses $\nu = 0.1$, KRLS-T uses $M = 500$, SW-KRLS uses $M = 500$, Q-KLMS uses $\eta = 0.5$ and $\epsilon_u = 1$, and NORMA uses $\eta = 0.5$ and $M = 1500$. Note that apart from the method mentioned in [20] and the typical cross-validation, parameters for these algorithms are typically determined by heuristics.

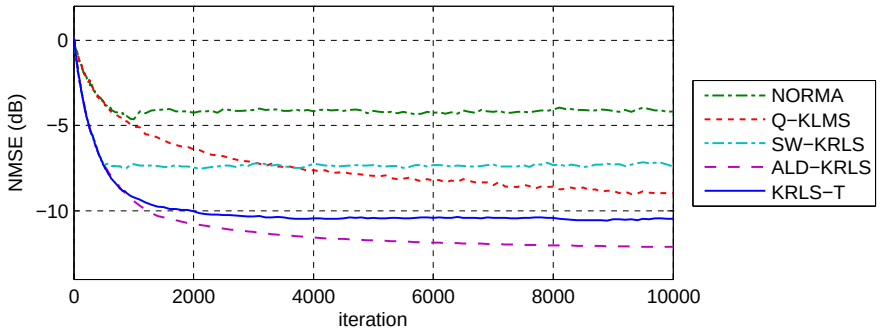

Figure 2: Learning curves on the KIN40K data set, with specific parameters per algorithm.

While Fig. 2 shows some interesting results, one may wonder if it is possible to improve the performance of a specific algorithm by tweaking its parameters. Indeed, algorithms that use a budget parameter M to determine their maximum dictionary size will typically obtain lower NMSE values if the budget is raised. Nevertheless, this will increase their computational complexity. A similar phenomenon is observed for all algorithms. Algorithms that use a threshold to determine their budget, such as ALD-KRLS and Q-KLMS, obtain a better steady-state NMSE at the cost of a higher complexity. There is thus a trade-off between the cost of an algorithm, in terms of computation and memory required, and its performance, in terms of the error it obtains and how

[2] Available at http://www.cs.toronto.edu/~delve/data/datasets.html.

fast it converges to its steady-state. In the following, we will analyze these trade-offs instead of the learning curves, as they may provide us with a more global picture of the performance of the algorithms.

5.2 Cost-versus-performance trade-offs

The computational cost of an algorithm is often measured as the CPU time. Nevertheless, this measure depends on the machine and the particular implementation of the algorithm. For a fairer comparison, we use a count of the floating point operations (FLOPS) instead, which are measured explicitly by the toolbox used for the experiments. We also measure the used memory, in terms of the number of bytes necessary to store the variables. In the results, we report the maximum FLOPS and maximum bytes per iteration, as these are the values that impose limits on the hardware.

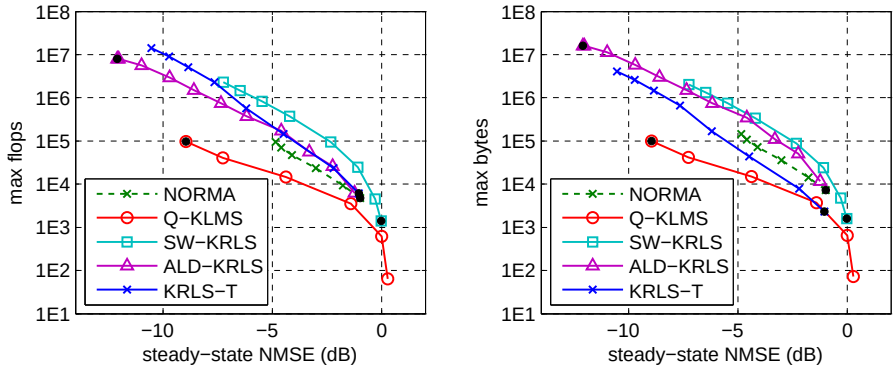

Figure 3: Maximum FLOPS used per iteration (left) and maximum bytes used per iteration (right), in the online regression experiment on the KIN40K data set, for different parameter configurations and different algorithms. Parameter values are represented in Table 2 and the black dots represent the first configuration, for each algorithm.

Fig. 3 illustrates the trade-offs obtained by each algorithm. The markers represent the results obtained for different sets of algorithm parameters. Each algorithm has one budget-related parameter, which also determines the computational and memory complexities. We fix every other parameter to a value close to its optimum and vary the budget parameter over a wide range. The full parameter values are displayed in Table 2. The steady-state NMSE is measured as the average NMSE over the last 1000 iterations.

The best-performing algorithm configurations are located to the left in both plots of Fig. 3, corresponding to low steady-state errors, and to the bottom, corresponding to low algorithmic complexities. The black dots show the first configuration of each algorithm, and typically they also represent an initial parameter setting beyond which it is difficult to move, for instance due to numerical limits.

By leaving the NMSE results out of Fig. 3, we obtain the plot of Fig. 4. It shows that for all algorithms there is an approximately linear relationship between the number of bytes stored in memory and the number of FLOPS required per iteration. Notice the logarithmic scales used. Algorithms that lie below the diagonal are more efficient with computation, i.e. when using the same amount of memory they require less computations, for a given NMSE. Algorithms that lie above

Table 2: Parameters used in the KIN40K online regression.

Method	Fixed parameter	Varying parameters
NORMA	$\eta = 0.5$	$\tau = 100, 200, 500, 1000, 1500, 2000$
Q-KLMS	$\eta = 0.5$	$\epsilon_u = 1, 1.2, 1.5, 2, 3, 5, 10, 12, 15$
SW-KRLS	—	$M = 10, 20, 50, 100, 200, 300, 400, 500$
ALD-KRLS	—	$\nu = .1, .2, .3, .4, .5, .6, .7, .8, .9, .95, .99$
KRLS-T	$\lambda = 1$	$M = 10, 20, 50, 100, 200, 300, 400, 500$

the diagonal are more efficient with memory, i.e. when performing the same amount of computation they require less memory, for a given NMSE.

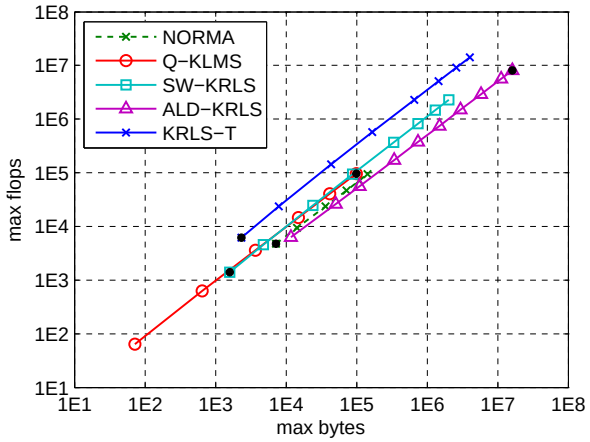

Figure 4: FLOPS per iteration versus bytes stored, in the KIN40K experiment.

5.3 Empirical convergence analysis

Apart from the steady-state error, an important measure for online and adaptive algorithms is the speed at which they converge to this error, called the *convergence rate*. Some theoretical convergence analyses have been carried out on specific algorithms, for instance the KNLMS algorithm in [8]. Here, in line with the previous experiments, we will perform an empirical convergence analysis that compares several algorithms.

In order to estimate the convergence rate, we measure the number of iterations it takes an algorithm to get within 1 dB of its steady-state error. The trade-off between this measure and the steady-state NMSE is shown in Fig. 5. Again, results that are most to the left or to the bottom of the plot represent the most interesting algorithm configurations, as they reach the lowest steady-state error or have the fastest convergence, respectively.

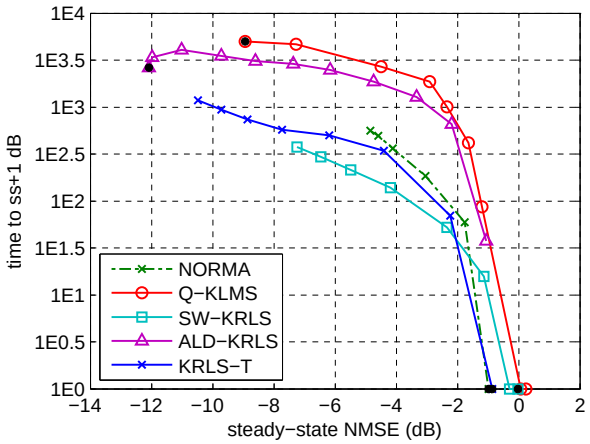

Figure 5: Trade-off between steady-state error and convergence rate in the KIN40K experiment.

5.4 Experiments with real-world data

Fig. 6 shows the convergence results obtained on two real-world data sets. The first data set is obtained by measuring the response of a wireless communication channel. The online algorithm requires to learn the nonlinear channel response and track its changes in time. The second data set is a recording of a patient's body surface during robotic radiosurgery. The online algorithm is used to predict the position of several markers on the body as the patient moves, so that the robot can use this prediction to compensate for its mechanical delay in positioning itself. More detailed descriptions of both experiments can be found in [19]. The parameters used in both experiments can be found in Tables 3 and 4, respectively.

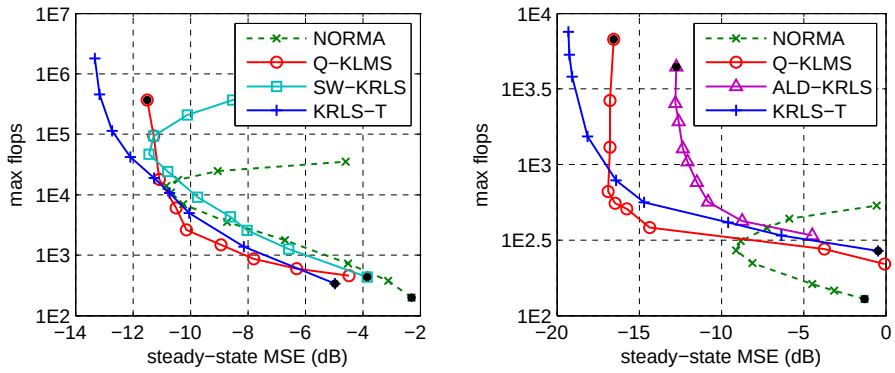

Figure 6: Results obtained for identifying the wireless communication channel (left) in the identification problem and for predicting a patient's position in the prediction problem.

In the first case, as the solution is to be implemented for real-time operation on a compact, low-power device, the maximum amount of FLOPS is typically fixed and the algorithm that obtains the lowest error under this condition is chosen for implementation. In the second case there is usually

no such restriction on complexity, as large surgical robots can be equipped with sufficient resources. Instead, the main requirement is to maintain the prediction error as low as possible.

Table 3: Parameters used in the online channel identification.

Method	Fixed	Varying parameters
NORMA	$\eta = 0.5$	$\tau = 5, 10, 20, 50, 100, 200, 310, 400, 500, 700, 1000$
Q-KLMS	$\eta = 0.6$	$\epsilon_u = 0.1, 1, 2, 3, 4, 5, 6, 7, 8$
SW-KRLS	—	$M = 5, 10, 15, 20, 30, 50, 70, 100, 150, 200, 300$
KRLS-T	$\lambda = 0.995$	$M = 2, 5, 10, 15, 20, 30, 50, 100, 200$

Table 4: Parameters used in the motion prediction.

Method	Fixed	Varying parameters
NORMA	$\eta = 0.99$	$\tau = 3, 4, 5, 10, 15, 20, 30, 40, 60$
Q-KLMS	$\eta = 0.99$	$\epsilon_u = .2, .5, 1, 2, 2.5, 2.75, 4, 6, 7$
ALD-KRLS	—	$\nu = .0001, .001, .003, .01, .02, .05, .1, .3, .5$
KRLS-T	$\lambda = 0.999$	$M = 3, 4, 5, 7, 10, 20, 50, 70, 100$

6 Further reading

We have given an overview of one decade of research in the field of online regression with kernels. The standard approach followed in this field, which consists in constructing kernel-based versions of classical adaptive filtering algorithms such as LMS and RLS, has produced several efficient state-of-the-art algorithms. Some other, related classes of algorithms that we have not discussed here are kernel affine projection algorithms (KAPA) [11, 7], which occupy the middle ground between KLMS and KRLS, and projection-based subgradient methods [17].

Several new directions are also being explored to improve the learning capabilities of these algorithms. A major focus of new algorithms is on the automatic learning of hyperparameters. Some algorithms pursue this by performing stochastic natural gradient descent in an online manner, in order to approximately maximize the marginal likelihood [10]. Other algorithms follow approaches inspired by the recent advances in neural networks, and they focus on online multi-kernel learning, where the parameter learning consists in assigning weights to several different kernels that are applied in parallel [23]. Many approaches in this direction seek sparse solutions by performing L_1-norm based learning.

References

[1] Nachman Aronszajn. Theory of reproducing kernels. *Transactions of the American mathematical society*, 68(3):337–404, 1950.

[2] Badong Chen, Songlin Zhao, Pingping Zhu, and José C. Príncipe. Quantized kernel least mean square algorithm. *IEEE Transactions on Neural Networks and Learning Systems*, 23(1):22–32, January 2012.

[3] Lehel Csató and Manfred Opper. Sparse online Gaussian processes. *Neural Computation*, 14(3):641–668, 2002.

[4] Bas J. De Kruif and Theo J. A. De Vries. Pruning error minimization in least squares support vector machines. *IEEE Transactions on Neural Networks*, 14(3):696–702, 2003.

[5] Yaakov Engel, Shie Mannor, and Ron Meir. The kernel recursive least squares algorithm. *IEEE Transactions on Signal Processing*, 52(8):2275–2285, August 2004.

[6] Jyrki Kivinen, Alexander J. Smola, and Robert C. Williamson. Online learning with kernels. *IEEE Transactions on Signal Processing*, 52(8):2165–2176, August 2004.

[7] Weifeng Liu, José C. Príncipe, and Simon Haykin. *Kernel Adaptive Filtering: A Comprehensive Introduction*. Wiley, 2010.

[8] Wemerson D. Parreira, Jose Carlos M. Bermudez, Cédric Richard, and Jean-Yves Tourneret. Stochastic behavior analysis of the Gaussian kernel least-mean-square algorithm. *IEEE Transactions on Signal Processing*, 60(5):2208–2222, 2012.

[9] Fernando Pérez-Cruz, Steven Van Vaerenbergh, Juan José Murillo-Fuentes, Miguel Lázaro-Gredilla, and Ignacio Santamaría. Gaussian processes for nonlinear signal processing: An overview of recent advances. *IEEE Signal Processing Magazine*, 30:40–50, July 2013.

[10] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[11] Cédric Richard, José Carlos M Bermudez, and Paul Honeine. Online prediction of time series data with kernels. *IEEE Transactions on Signal Processing*, 57(3):1058–1067, March 2009.

[12] Craig Saunders, Alexander Gammerman, and Volodya Vovk. Ridge regression learning algorithm in dual variables. In *Proceedings of the 15th International Conference on Machine Learning (ICML)*, pages 515–521, Madison, WI, USA, July 1998.

[13] Ali H. Sayed. *Fundamentals of Adaptive Filtering*. Wiley-IEEE Press, 2003.

[14] Bernhard Schölkopf, Ralf Herbrich, and Alexander J. Smola. A generalized representer theorem. In *Computational learning theory*, pages 416–426. Springer, 2001.

[15] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels*. The MIT Press, Cambridge, MA, USA, 2002.

[16] Johan A.K. Suykens, Jos De Brabanter, Lukas Lukas, and Joos Vandewalle. Weighted least squares support vector machines: robustness and sparse approximation. *Neurocomputing*, 48(1):85–105, 2002.

[17] Sergios Theodoridis, Konstantinos Slavakis, and Isao Yamada. Adaptive learning in a world of projections. *IEEE Signal Processing Magazine*, 28(1):97 –123, January 2011.

[18] Steven Van Vaerenbergh, Miguel Lázaro-Gredilla, and Ignacio Santamaría. Kernel recursive least-squares tracker for time-varying regression. *IEEE Transactions on Neural Networks and Learning Systems*, 23(8):1313–1326, August 2012.

[19] Steven Van Vaerenbergh and Ignacio Santamaría. A comparative study of kernel adaptive filtering algorithms. In *2013 IEEE Digital Signal Processing (DSP) Workshop and IEEE Signal Processing Education (SPE)*, 2013.

[20] Steven Van Vaerenbergh, Ignacio Santamaría, and Miguel Lázaro-Gredilla. Estimation of the forgetting factor in kernel recursive least squares. In *2012 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, September 2012.

[21] Steven Van Vaerenbergh, Ignacio Santamaría, Weifeng Liu, and José C. Príncipe. Fixed-budget kernel recursive least-squares. In *2010 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Dallas, USA, April 2010.

[22] Steven Van Vaerenbergh, Javier Vía, and Ignacio Santamaría. A sliding-window kernel RLS algorithm and its application to nonlinear channel identification. In *2006 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 789–792, Toulouse, France, May 2006.

[23] Masahiro Yukawa. Multikernel adaptive filtering. *IEEE Transactions on Signal Processing*, 60(9):4672–4682, 2012.