ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

Algoritmos de Filtrado Kernel Adaptativo Multi-Output
(Multi-Output Kernel Adaptive Filtering Algorithms)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Diego Cuevas Fernández

Septiembre - 2020

Índice general

Agradecimientos **1**

Resumen **2**

Abstract **3**

1. Introducción **4**
- 1.1. Descripción del problema 4
- 1.2. Planteamiento 6
- 1.3. Estructura del trabajo 7

2. Filtrado adaptativo **8**
- 2.1. Filtrado adaptativo en procesado de señal 8
- 2.2. Filtrado adaptativo lineal 9
  - 2.2.1. Algoritmo LMS 10
- 2.3. Filtrado adaptativo kernel 11
  - 2.3.1. Métodos kernel 12
  - 2.3.2. Algoritmo KLMS 13
  - 2.3.3. Limitando la complejidad del KLMS: Técnicas de *sparsification* 16
  - 2.3.4. Algoritmo QKLMS 17
  - 2.3.5. *Random Fourier Features* 18
  - 2.3.6. Algoritmo RFF-KLMS 21
  - 2.3.7. KAFBOX 21

3. Filtrado kernel adaptativo *Multi-Output* **23**
- 3.1. Planteamiento del problema *Multi-Output* 23
- 3.2. Técnicas de concatenación 24
  - 3.2.1. Algoritmo LMS-MO 26
  - 3.2.2. Algoritmo KLMS-MO 26
  - 3.2.3. Algoritmo QKLMS-MO 27
  - 3.2.4. Algoritmo RFF-KLMS-MO 27
- 3.3. Arquitectura de predicción *Multi-Output* 28

4. Análisis de resultados I: Base de datos artificial **30**
- 4.1. Generación de la base de datos artificial 30
- 4.2. Resultados 32
  - 4.2.1. MSE 34
  - 4.2.2. QKLMS-MO y RFF-KLMS-MO 38

5. Análisis de resultados II: Base de datos real 46
5.1. *Occupancy Detection Dataset* 46
5.2. Resultados 47
5.2.1. MSE 49
5.2.2. Tiempo de entrenamiento 49
5.2.3. FLOPS 49
5.2.4. Almacenamiento requerido 51

6. Conclusiones 52

A. Códigos MATLAB 53
A.1. Algoritmos SO 53
A.1.1. LMS 53
A.1.2. KLMS 53
A.1.3. QKLMS 54
A.1.4. RFF-KLMS 54
A.2. KAFBOX 55
A.3. Algoritmos MO 57
A.3.1. LMS-MO 57
A.3.2. KLMS-MO 57
A.3.3. QKLMS-MO 58
A.3.4. RFF-KLMS-MO 58
A.4. Generación de la base de datos artificial 59
A.5. Complejidad computacional de los algoritmos 60
A.5.1. Desglose de operaciones QKLMS-MO 60
A.5.2. Desglose de operaciones RFF-KLMS-MO 61

Referencias 61

Índice de figuras

2.1. Sistema desconocido con entrada $\mathbf{x}_n$ y salida d_n en el instante de tiempo n. 10
2.2. Filtro adaptativo lineal para identificación de sistemas. 10
2.3. Filtro adaptativo kernel para identificación de sistemas no lineales. 12
2.4. Métodos kernel aplicados a un problema de clasificación binaria. 13
2.5. Construcción siempre creciente del diccionario (arriba a la izquierda), en el que el diccionario contiene n elementos en la iteración n, y mediante técnicas de *sparsification* por construcción (arriba a la derecha) y por "poda"(abajo), que ralentizan su crecimiento. 17
2.6. Cada componente del mapa de características $z_\omega(\mathbf{x})$ proyecta $\mathbf{x}$ sobre una dirección aleatoria ω dibujada a partir de la transformada de Fourier $p(\omega)$ de $\kappa(\Delta)$, y envuelve esta línea sobre el círculo unitario en $\mathcal{R}^2$. El mapeado $\psi_\omega(\mathbf{x}) = \cos\left(\omega^T\mathbf{x} + b\right)$ adicionalmente rota este círculo una cantidad aleatoria b y proyecta los puntos en el intervalo $[0, 1]$. 20
2.7. Vista cualitativa de la aproximación del producto interno del kernel gaussiano mediante *Random Fourier Features*. La curva azul muestra la función gaussiana exacta y el diagrama de *sparsification* rojo muestra la aproximación realizada para distintos valores de D. Claramente, la varianza en la aproximación decrece a medida que se incrementa D. 20

3.1. Filtrado adaptativo componente a componente en un sistema MIMO de M entradas y M salidas. 25
3.2. Filtrado adaptativo MO en un sistema MIMO de M entradas y M salidas. 25
3.3. Arquitectura de filtrado adaptativo LMS-MO + KLMS. 29

4.1. Generación de la base de datos artificial. 31
4.2. Series temporales de entrada en una simulación del Escenario 1. . 33
4.3. Series temporales de salida en una simulación del Escenario 1. . . 33
4.4. Curvas de aprendizaje para el Escenario 1. 35
4.5. Curvas de aprendizaje para el Escenario 2. 35
4.6. Curvas de aprendizaje para el Escenario 3. 36
4.7. Curvas de aprendizaje para el Escenario 4. 36

4.8. Curvas de aprendizaje de los algoritmos QKLMS-MO y RFF-KLMS-MO sobre series temporales de 50000 muestras para el Escenario 1. 37
4.9. Curvas de aprendizaje del algoritmo QKLMS-MO para distintos valores del parámetro ϵ_u. 39
4.10. Curvas de aprendizaje del algoritmo RFF-KLMS-MO para distintos valores del parámetro D. 39
4.11. Tiempo empleado en el entrenamiento de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. 40
4.12. FLOPS por iteración de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. 42
4.13. FLOPS por iteración frente al MSE de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. 43
4.14. Almacenamiento máximo requerido por los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. 44
4.15. Almacenamiento máximo requerido frente al MSE de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. 45

5.1. *Setup* empleado para la adquisición de los datos del *Occupancy Detection Dataset*. En él, se muestran los sensores de luz, CO_2 y DHT22 (temperatura y humedad), el sistema radio ZigBee, el microcontrolador y la cámara digital controlada por una Raspberry Pi. 47
5.2. Valores de CO_2 y luz (entradas del sistema). 48
5.3. Valores de temperatura y humedad (salidas del sistema). 48
5.4. Predicciones realizadas por el algoritmo QKLMS-MO sobre la base de datos real. 50
5.5. Predicciones realizadas por el algoritmo RFF-KLMS-MO sobre la base de datos real. 50
5.6. Curvas de aprendizaje de los algoritmos QKLMS-MO y RFF-KLMS-MO sobre la base de datos real. 51

Índice de tablas

4.1. Escenarios contemplados para la generación de la base de datos artificial. 32
4.2. Valores óptimos de la anchura del kernel σ_k en los distintos escenarios. 34
4.3. Valores del resto de hiperparámetros de los distintos algoritmos de filtrado adaptativo aplicados sobre la base de datos artificial. . 34
4.4. Valores de los hiperparámetros de los algoritmos QKLMS-MO y RFF-KLMS-MO utilizados para analizar la influencia de m y D sobre el MSE. 38
4.5. Tiempo empleado en el entrenamiento de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros ϵ_u y D. 40
4.6. Número de FLOPS por operación para el procesador x86 de Intel. 41
4.7. FLOPS por iteración de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. . . . 41
4.8. Almacenamiento máximo requerido por los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D. 44

5.1. Valores del resto de hiperparámetros de los algoritmos QKLMS-MO y RFF-KLMS-MO usados sobre la base de datos real. 49

Agradecimientos

Dos años después y en medio de una pandemia, aquí estoy de nuevo terminando un trabajo que marca el final de otra etapa de mi vida. Por ello, me gustaría agradecer a todas aquellas personas que lo han hecho posible.

En primer lugar, me gustaría agradecer a Jesús Pérez por ofrecerme allá por 2017 la posibilidad de empezar a trabajar en el ámbito del *Machine Learning*, resultando ser una de mis grandes pasiones hoy en día.

También quiero agradecer a mi tutor Ignacio Santamaría la confianza depositada en mí para la realización de este nuevo proyecto, el cual no sabíamos muy bien al principio hacia dónde nos llevaría pero que ha resultado ser muy satisfactorio. Tampoco me quiero olvidar de Steven Van Vaerenbergh, quien me ha ayudado enormemente a la hora de desarrollar los códigos, a pesar de estar trabajando en otro departamento este año. También me gustaría agradecer a Jesús Ibáñez por su predisposición a ayudarme con una parte del proyecto que finalmente, debido a las circunstancias, no se pudo realizar.

Por otro lado, quiero agradecer especialmente a mis amigos por preocuparse por mí, apoyarme en los malos ratos y sobre todo por hacer que mi vida sea mejor con ellos a mi lado. Aunque no os lo diga a menudo, muchísimas gracias por todo.

Por último, me gustaría agradecer a mi familia por su apoyo incondicional, su paciencia y los buenos ratos que compartimos. Sin vuestra ayuda, nada de lo que he conseguido hasta hoy hubiese sido posible.

Resumen

El aprendizaje supervisado tradicional es una herramienta de gran utilidad capaz de ayudar a la hora de tomar decisiones rápidas y precisas en tareas de regresión y predicción. El objetivo de este tipo de aprendizaje es aprender una función que mapee desde el espacio de instancias de entrada al espacio de salida, donde la salida puede ser una etiqueta o un valor escalar. No obstante, la complejidad creciente de las tareas y los problemas hacen necesaria la creación de nuevos modelos de *Machine Learning* que realicen una toma de decisiones más sofisticada, de tal forma que sean capaces de resolver varios problemas de este tipo al mismo tiempo, lo que se conoce con el nombre de *Multitask Learning*.

Por ello, este proyecto se centrará en el desarrollo de nuevos modelos de *Multitask Learning* y, más concretamente, de algoritmos de filtrado adaptativo kernel de tipo *Multi-Output*, que consigan aprovechar la correlación existente entre las distintas series temporales para mejorar los resultados obtenidos. Además, se pretende que su complejidad sea mucho menor que la de otros modelos desarrollados hasta el momento, como pueden ser los procesos Gaussianos (GPs).

Para evaluar las prestaciones de los métodos propuestos, se utiliza en primer lugar una base de datos generados mediante simulación, sobre la que se tiene un cierto control de sus parámetros, para realizar un completo análisis de las prestaciones de todos los algoritmos desarrollados y, a continuación, se seleccionan aquellos dos que han presentado unas mejores prestaciones para su aplicación sobre una base de datos real.

Los resultados obtenidos para todos los algoritmos desarrollados son bastante buenos, destacando especialmente los de las versiones *Multi-Output* del KLMS con limitación del tamaño del diccionario mediante el empleo del criterio de distancia Euclídea (QKLMS-MO) y con aproximaciones de dimensiones finitas de las funciones kernel empleadas (RFF-KLMS-MO). Estos dos algoritmos obtienen unos resultados considerablemente buenos en términos del error cuadrático medio (del orden de 10^{-1}), resultando bastante mejor el RFF-KLMS-MO en términos de complejidad computacional y tiempo empleado en el entrenamiento.

Abstract

Traditional supervised learning is a very useful tool that helps in making fast and accurate decisions for regression and prediction tasks. The main goal of this type of learning is to learn a function that maps from the input feature space to the output space, where the output is either a single label or a single value. However, the increasing complexity of tasks and problems urges the need of new *Machine Learning* models that have more sophisticated decision making mechanisms so they can solve several problems simultaneously, which is known as *Multitask Learning*.

Thus, this project will focus on the development of new *Multitask Learning* models and, more precisely, of multi-output kernel adaptive filtering algorithms, that take advantage of the existing correlation between time series to improve their results. Besides, its computational complexity is intended to be much lower than that of other models that have been developed up to now, such as Gaussian Processes (GPs).

In order to evaluate their performance, a simulated and parameter-controlled dataset is first used to carry out a complete analysis of all algorithms capabilities. Then, a real dataset is used to evaluate the performance of the two best-performing algorithms.

The results obtained for all algorithms are valid, especially those obtained by the *Multi-Output* versions of the KLMS with dictionary size limitations based on the Euclidean distance criterion (QKLMS-MO) and with finite-dimensional approximations of the kernel functions (RFF-KLMS-MO). These two algorithms provide very good results in terms of mean squared error (around 10^{-1}), proving RFF-KLMS-MO to be a better choice in terms of computational complexity and training time.

Capítulo 1

Introducción

1.1. Descripción del problema

El aprendizaje supervisado tradicional, visto como uno de los paradigmas de *Machine Learning* más adoptados por máquinas y aplicaciones del mundo real, es capaz de ayudar a la hora de tomar decisiones rápidas y precisas en tareas de regresión y predicción. El objetivo del aprendizaje supervisado tradicional es aprender una función que mapee desde el espacio de instancias de entrada al espacio de salida, donde la salida puede ser una etiqueta simple (para tareas de predicción) o un valor simple (para tareas de regresión). Por tanto, este tipo de aprendizaje se suele utilizar para resolver cuestiones que tengan respuestas sencillas, como preguntas de verdadero-falso o de valores simples. Por ejemplo, se puede emplear un modelo tradicional de clasificación binaria para detectar si un correo electrónico entrante es *spam* o no. Por otro lado, se puede utilizar un modelo tradicional de regresión para predecir el consumo de energía diario basándonos en la temperatura, la velocidad del viento o la humedad.

No obstante, con la demanda de los investigadores y las empresas industriales por una complejidad creciente de las tareas y los problemas, existe una urgente necesidad de crear nuevos modelos de *Machine Learning* que puedan beneficiarse de una toma de decisiones más sofisticada [1].

Hoy en día, muchas aplicaciones modernas de *Machine Learning* requieren resolver varios problemas de toma de decisiones o de predicción y, en numerosas ocasiones, la clave para obtener mejores resultados y hacer frente a la falta de datos consiste en explotar las dependencias entre dichos problemas.

En las redes de sensores, por ejemplo, los datos no observados (*missing entries*) de ciertos sensores pueden predecirse explotando su correlación con las señales adquiridas por otros sensores próximos [2]. En geoestadística, la predicción de la concentración de metales altamente contaminantes, cuya medida es muy cara de realizar, se puede hacer utilizando variables sobremuestreadas como un *proxy* [3]. En el ámbito de la computación gráfica, un tema recurrente es la animación y simulación del movimiento humanoide físicamente plausible. Dado un conjunto de poses que delinean un movimiento particular (por ejemplo, caminar), se requiere completar la secuencia rellenando los *frames* faltantes con poses de aspecto natural. Esto se puede hacer debido a que el movimiento humano exhibe un alto grado de correlación. Por ejemplo, si se considera la forma

de andar, cuando se mueve la pierna derecha hacia delante, inconscientemente se prepara la pierna izquierda (que está tocando en ese momento el suelo) para empezar a moverse tan pronto como la pierna derecha se pose en el suelo. Al mismo tiempo, las manos se mueven de manera síncrona con las piernas. Por ello, se pueden aprovechar todas estas correlaciones implícitas para predecir nuevas poses y generar nuevas secuencias de caminar de aspecto natural [4]. Por último, en el campo de la categorización de textos, es posible asignar un documento a múltiples temas o que este tenga múltiples etiquetas [5].

En todos los ejemplos que se han mencionado, el enfoque más simple ignora la potencial correlación entre las diferentes componentes del problema y emplea modelos que realizan predicciones de manera individual para cada salida. Sin embargo, estos ejemplos sugieren utilizar un enfoque diferente a través de una predicción conjunta que explote la interacción entre las diferentes componentes para conseguir mejorar las predicciones individuales. En el ámbito del *Machine Learning*, este tipo de modelado se conoce a menudo con el nombre de aprendizaje multitarea o *Multitask Learning*. Como se ha dicho, la idea clave de este tipo de aprendizaje consiste en que la información compartida entre las diferentes tareas se utilice para conseguir unas mejores prestaciones en comparación con el aprendizaje de dichas tareas de manera individual [6].

Por este motivo, en este proyecto se pretenden desarrollar nuevos modelos de *Multitask Learning* y, más concretamente, de filtrado adaptativo kernel de tipo *Multi-Output*. Este tipo de filtros son hoy en día uno de los temas centrales en el ámbito del procesado de señal y son ampliamente utilizados en entornos no estacionarios debido a que son capaces de adaptar su función de transferencia en función de los parámetros cambiantes del sistema que genera los datos de entrada. Además, presentan características como una alta precisión, simplicidad algorítmica, robustez y baja latencia, lo que hace que sean considerados una herramienta esencial en los sistemas de aprendizaje inteligente. Además, la definición que hacen de una medida de la información instantánea de las observaciones permite que los filtros adaptativos de tipo kernel sean capaces de seleccionar activamente los datos de entrenamiento en escenarios de aprendizaje *online*.

A pesar de todo ello, hoy en día estos algoritmos se implementan en su versión *Single-Output* y las adaptaciones *Multi-Output* realizadas hasta el momento, como pueden ser los procesos Gaussianos o *Gaussian Processes* (GPs), presentan una complejidad computacional (cúbica con el número de salidas y de muestras utilizadas) y de almacenamiento (cuadrática con el número de salidas y de muestras utilizadas) bastante elevadas [7]. Por ello, en este trabajo se buscará un enfoque mucho más simple, basado en algoritmos tipo *Kernel Least Mean Square* o KLMS, que permita reducir la complejidad de los algoritmos y seguir obteniendo al mismo tiempo buenos resultados con ellos.

Para ello, se desarrollarán nuevos algoritmos siguiendo el estilo utilizado en la *toolbox* de MATLAB denominada KAFBOX, desarrollada por S. Van Vaerenbergh, que implementa algoritmos de filtrado adaptativo *Single-Output* tanto lineal como de tipo kernel. De esta manera, los nuevos algoritmos desarrollados en este trabajo podrán ser incorporados en futuras versiones de dicha *toolbox*.

Para probar los distintos algoritmos, se utilizará en primer lugar una base de datos artificial, generada con datos simulados, sobre la que se disponga de cierto control sobre algunos de sus parámetros, como el grado de no linealidad de las series temporales o la correlación existente entre ellas. Esta base de datos

se utilizará como punto de partida para realizar un completo análisis de todos los algoritmos. A continuación, se utilizará una base de datos reales pública para probar sobre ella aquellos algoritmos que hayan obtenido unos mejores resultados sobre la base de datos artificial.

Por lo tanto, el objetivo principal del proyecto consiste en desarrollar nuevos algoritmos de filtrado adaptativo kernel de tipo *Multi-Output* que tengan una menor complejidad computacional y menores requerimientos de almacenamiento que los existentes hasta el momento.

En los siguientes apartados, se explican con mayor detalle las distintas fases de las que consta el proyecto, así como la estructura del documento y el contenido de cada uno de los capítulos que lo forman.

1.2. Planteamiento

En esta sección, se expone brevemente cómo se desarrollará el proyecto y las distintas fases de las que se compone.

- **Fase 1** : en primer lugar, se desarrollarán nuevos algoritmos de filtrado kernel adaptativo *Multi-Output*, utilizando para ello dos estrategias distintas: (i) desarrollar de versiones *Multi-Output* de algoritmos KLMS *Single-Output* basadas en la concatenación de los datos de entrada y (ii) desarrollar una nueva estructura de predicción combinando filtros adaptativos lineales y no lineales. Finalmente, se proponen dos algoritmos como "ganadores" : el QKLMS-MO, de mayor complejidad computacional y mayor tiempo de entrenamiento pero menor error cuadrático medio, y el RFF-KLMS-MO, computacionalmente más simple, con tiempos de entrenamiento menores y mayor error cuadrático medio.

  Todos estos algoritmos de filtrado adaptativo *Multi-Output* propuestos en el proyecto se implementarán tomando como punto de partida los algoritmos de filtrado adaptativo *Single-Output* de la *toolbox* KAFBOX.

- **Fase 2**: la siguiente fase del proyecto consistirá en la creación de una base de datos artificial, sobre la que se tenga control de algunos de sus parámetros, como la correlación entre las series temporales de entrada y su no linealidad. Sobre esta base de datos se probarán los distintos algoritmos desarrollados anteriormente, prestando especial atención al error cuadrático medio (MSE) como figura de mérito principal. Para ello, se presentarán las gráficas que muestran la evolución del MSE a lo largo de las distintas iteraciones de entrenamiento de todos los algoritmos para cuatro escenarios distintos.

- **Fase 3**: en esta fase, se analizarán con mayor detalle los algoritmos "ganadores" , realizando una comparativa en términos del tiempo empleado en el entrenamiento, número de operaciones de punto flotante (*Floating Point Operations* o FLOPS) realizadas en cada iteración y almacenamiento requerido por cada algoritmo.

- **Fase 4**: en la última fase del proyecto, se probarán los algoritmos "ganadores" sobre una base de datos real (*Occupancy Detection Dataset* [8]), realizándose una comparación exhaustiva en términos de MSE, tiempo

empleado en el entrenamiento, número de FLOPS en cada iteración y almacenamiento requerido por cada uno de los algoritmos.

1.3. Estructura del trabajo

La memoria del proyecto se encuentra dividida en seis capítulos y un anexo. A continuación, se explica brevemente el contenido de cada uno de ellos:

- **Capítulo 1**: en este primer capítulo, se introducen el problema del filtrado kernel adaptativo *Multi-Output* y las soluciones propuestas para ello en este trabajo. Además, se explica brevemente cómo se ha llevado a cabo el proyecto y las fases de las que consta.
- **Capítulo 2**: en este capítulo, se explica el funcionamiento de los diversos algoritmos de filtrado adaptativo *Single-Output* utilizados, sirviendo como punto de partida para el desarrollo de sus versiones *Multi-Output* en el siguiente capítulo. También se presenta brevemente la *toolbox* KAFBOX, que implementa todos estos algoritmos en MATLAB.
- **Capítulo 3**: en este capítulo, se plantea el problema de predicción *Multi-Output* y se explican las distintas aproximaciones al mismo llevadas a cabo en este proyecto.
- **Capítulo 4**: en este capítulo, se analizan las prestaciones de los distintos algoritmos de filtrado adaptativo *Multi-Output*, así como de la arquitectura desarrollada, aplicados a una base de datos artificial sobre la que se tiene control de ciertos parámetros.
- **Capítulo 5**: en este capítulo, se analizan las prestaciones de los algoritmos de filtrado adaptativo QKLMS-MO y RFF-KLMS-MO aplicados sobre la base de datos real *Occupancy Detection*.
- **Capítulo 6**: en el último capítulo, se presentan algunas conclusiones extraídas a partir del trabajo realizado.
- **Apéndice A**: en esta parte de la memoria, se recogen todos los códigos de MATLAB realizados, así como algunos pertenecientes a la *toolbox* KAFBOX.

Capítulo 2

Filtrado adaptativo

En este capítulo, se explicará el funcionamiento de los diversos algoritmos de filtrado adaptativo *Single-Output* utilizados en este proyecto, que servirán como punto de partida para el desarrollo de sus versiones *Multi-Output*. Además, se presentará brevemente la *toolbox* KAFBOX, que implementa todos estos algoritmos en MATLAB.

En la sección 2.1 se explicará brevemente en qué consiste un filtro adaptativo y algunas de sus aplicaciones en el ámbito del procesado de señal. En la sección 2.2, se explicará el filtrado adaptativo lineal y el algoritmo LMS. En la sección 2.3 se explicará el filtrado adaptativo no lineal por medio de *kernels* y los algoritmos KLMS, QKLMS y RFF-KLMS.

2.1. Filtrado adaptativo en procesado de señal

El filtrado adaptativo es un problema clásico en el ámbito del procesado de señal [9]. En general, un filtro adaptativo consiste en una estructura de filtro a la que se le dota de un algoritmo adaptativo para poder modificar y ajustar su función de transferencia, típicamente mediante el empleo de una señal de error.

Los filtros adaptativos son ampliamente utilizados en entornos no estacionarios debido a que son capaces de adaptar su función de transferencia para seguir la evolución de los parámetros cambiantes del sistema que genera los datos de entrada [10][11]. Actualmente, este tipo de filtros se han convertido en imprescindibles y están presentes en numerosas aplicaciones de procesado digital de señal actuales, principalmente debido al incremento que se ha producido en la potencia de cálculo de los ordenadores y a la necesidad de procesar los datos de manera secuencial (*online*). Los filtros adaptativos se usan rutinariamente en la actualidad en cualquier aplicación de comunicación para realizar la ecualización de un canal de comunicaciones, el *beamforming* en un *array* de antenas o la cancelación de ecos. Además, también se suelen utilizar en otras áreas del procesado de señal, como pueden ser el procesado de imagen o el tratamiento de señales médicas.

Aplicando los principios del filtrado adaptativo lineal al espacio de características kernel, se obtienen diversos algoritmos potentes de filtrado adaptativo no lineal. La necesidad urgente de este tipo de algoritmos adaptativos no lineales en ciertas aplicaciones de comunicaciones y herramientas de recomendación web

para la transmisión de bases de datos ha incrementado el interés en ciertas áreas del procesado de señal, como por ejemplo el aprendizaje activo y secuencial. En este campo, la introducción de los filtros adaptativos de tipo kernel es notable. Los algoritmos de aprendizaje *online* secuenciales y adaptativos son considerados una herramienta esencial en el procesado de señal y en los sistemas de aprendizaje inteligente. Esto se debe principalmente a que estos algoritmos presentan características como la precisión, la simplicidad algorítmica, la robustez, una baja latencia y una implementación rápida y eficiente. Además, mediante la definición de una medida de la información instantánea en las observaciones, los filtros adaptativos de tipo kernel son capaces de seleccionar activamente los datos de entrenamiento en escenarios de aprendizaje *online*. Este mecanismo de aprendizaje activo proporciona un marco bien fundamentado para el descubrimiento del conocimiento, la supresión de redundancias y la detección de anomalías.

Los métodos de filtrado kernel proveen, por tanto, de un marco excelente para lidiar con la enorme variedad de algoritmos y aplicaciones existentes. Los métodos kernel no son solo útiles en muchas de las aplicaciones tradicionales del procesado digital de señal, como pueden ser el reconocimiento de patrones o el procesado del lenguaje, de audio o de vídeo. Hoy en día, los métodos kernel también son uno de los principales candidatos para aplicaciones emergentes, como por ejemplo la creación de interfaces cerebro-ordenador, el procesado de imágenes satelitales, el modelado de mercados de valores, el diseño de antenas y de redes de comunicaciones, la fusión de datos multimodales y su procesado, el reconocimiento de comportamientos y emociones a partir del lenguaje o de vídeos, las aplicaciones de control, predicción y análisis espectral, y el aprendizaje en entornos complejos como las redes sociales [12].

2.2. Filtrado adaptativo lineal

Para la descripción básica de los algoritmos de filtrado adaptativo se considera en esta sección un problema de identificación, que consiste en modelar un sistema desconocido y posiblemente variante en el tiempo, observando para ello las entradas y salidas del mismo a lo largo de un tiempo. Se denotará a la entrada del sistema en el instante n como $\mathbf{x}_n$ y a su salida como d_n (véase la Fig. 2.1). Se asume que la señal de entrada $\mathbf{x}_n$ tiene media cero y se representa normalmente mediante un vector de *delays* temporales con L *taps* de la señal x_n en el instante de tiempo n como $\mathbf{x}_n = [x_n, x_{n-1}, ..., x_{n-L+1}]^T$, donde $\mathbf{x}^T$ denota la transpuesta del vector $\mathbf{x}$.

En la Fig. 2.2, se muestra el diagrama de un filtro adaptativo lineal. La entrada al filtro adaptativo en el instante de tiempo n es $\mathbf{x}_n$ y su salida y_n se obtiene por medio de la siguiente operación lineal:

$$y_n = \mathbf{w}_n^H \mathbf{x}_n \tag{2.1}$$

donde H representa el operador hermítico.

El filtrado adaptativo lineal sigue el marco del aprendizaje *online*, que consiste en dos pasos básicos que se repiten en cada instante de tiempo n. En primer lugar, el algoritmo *online* recibe una nueva observación $\mathbf{x}_n$, para la cual calcula la salida estimada y_n en base a su estimación actual de $\mathbf{w}_n$. A continuación, el algoritmo observa la salida deseada d_n (también conocida como "símbolo" en

Figura 2.1: Sistema desconocido con entrada $\mathbf{x}_n$ y salida d_n en el instante de tiempo n.

Fuente: Adaptive Kernel Learning for Signal Processing [9]

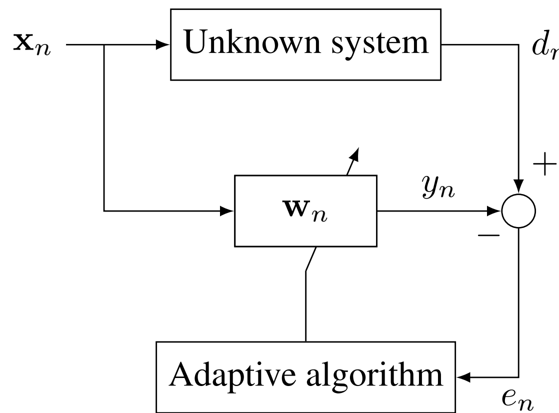


Figura 2.2: Filtro adaptativo lineal para identificación de sistemas.

Fuente: Adaptive Kernel Learning for Signal Processing [9]

comunicaciones), lo que permite calcular el error de estimación $e_n = d_n - y_n$ y actualizar la estima de $\mathbf{w}_n$ en consecuencia.

2.2.1. Algoritmo LMS

Las clásicas técnicas adaptativas de optimización tienen sus orígenes en el planteamiento teórico denominado como algoritmo de descenso por gradiente [13]. El algoritmo más conocido de entre los algoritmos de descenso de gradiente estocástico es el *Least Mean Square* (LMS), que minimiza una estima instantánea del error cuadrático medio definido como la esperanza matemática del cuadrado de la señal de error $J_n = \mathbb{E}\left[|e_n|^2\right]$. Como este error es una función del vector $\mathbf{w}_n$, la idea del algoritmo es modificar este vector hacia la dirección de máximo decrecimiento de J_n. Esta dirección es justo la opuesta a su gradiente $\nabla_{\mathbf{w}} J_n$. Asumiendo que las señales son complejas y estacionarias, la esperanza matemática del error viene dada por

$$
\begin{aligned}
\mathbb{E}\left[|e_n|^2\right] &= \mathbb{E}\left[|d_n - \mathbf{w}_n^H \mathbf{x}_n|^2\right] = \mathbb{E}\left[|d_n|^2 + \mathbf{w}_n^H \mathbf{x}_n \mathbf{x}_n^H \mathbf{w}_n - 2\mathbf{w}_n^H \mathbf{x}_n d_n^*\right] \\
&= \sigma_d^2 + \mathbf{w}_n^H \mathbf{R}_{xx} \mathbf{w}_n - 2\mathbf{w}_n^H \boldsymbol{\rho}_{xd}
\end{aligned}
\tag{2.2}
$$

donde $\mathbf{R}_{xx}$ representa la matriz de autocorrelación de la señal $\mathbf{x}_n$, $\boldsymbol{\rho}_{xd}$ es el vector de correlación cruzada entre la señal $\mathbf{x}$ y la salida deseada del filtro d_n y σ_d^2 es la varianza de la salida del sistema deseada.

El gradiente de 2.2 con respecto al vector $\mathbf{w}_n$ se puede expresar como

$$
\nabla_{\mathbf{w}} J_n = 2\mathbf{R}_{xx}\mathbf{w}_n - 2\boldsymbol{\rho}_{xd} \tag{2.3}
$$

En consecuencia, la regla de adaptación basada en descenso por gradiente se convierte en

$$
\mathbf{w}_{n+1} = \mathbf{w}_n - \mu \nabla_{\mathbf{w}} J_n \tag{2.4}
$$

donde μ representa el tamaño del paso (*step size*) o tasa de aprendizaje (*learning rate*) del algoritmo.

El algoritmo *Least Mean Squares* (LMS), introducido en 1960 por Bernard Widrow [11], es un método muy sencillo y elegante de entrenar un sistema adaptativo lineal para minimizar el error cuadrático medio (MSE). Este método aproxima el gradiente $\nabla_{\mathbf{w}} J_n$ por medio de una estimación instantánea. A partir de la Ec. 2.3, dicha aproximación se puede escribir como

$$
\nabla_{\mathbf{w}} J_n \approx 2\mathbf{x}_n \mathbf{x}_n^H \mathbf{w}_n - 2\mathbf{x}_n d_n^* \tag{2.5}
$$

Usando esta aproximación en la Ec. 2.4, se llega a la conocida regla de actualización de descenso estocástico por gradiente, que es el núcleo del algoritmo LMS:

$$
\begin{aligned}
\mathbf{w}_{n+1} &= \mathbf{w}_n - \mu \mathbf{x}_n \left(\mathbf{x}_n^H \mathbf{w}_n - d_n^*\right) \\
&= \mathbf{w}_n + \mu \mathbf{x}_n e_n^*
\end{aligned}
\tag{2.6}
$$

Este procedimiento de optimización también es la base para el ajuste de estructuras de filtros no lineales como las redes neuronales [14] y algunos de los filtros adaptativos basados en kernels que se explicarán más adelante. El código de MATLAB para cada iteración del entrenamiento con un nuevo par de datos $(\mathbf{x}, d)$ implementado en la *toolbox* KAFBOX utilizada en este proyecto se muestra en el Script A.1.1 del Anexo.

2.3. Filtrado adaptativo kernel

El problema del filtrado no lineal y la adaptación *online* de los pesos del modelo fue introducido por primera vez por las redes neuronales en los años 90 [15] [16]. A lo largo de la última década, se ha depositado un gran interés en desarrollar versiones no lineales de filtros adaptativos por medio de kernels [17]. El objetivo era desarrollar máquinas capaces de aprender a lo largo del tiempo en entornos cambiantes y que tuviesen al mismo tiempo las buenas características de convexidad, convergencia y razonable complejidad computacional, algo que no se había conseguido implementar con éxito en las redes neuronales.

El filtrado adaptativo kernel pretende formular los filtros adaptativos lineales clásicos en espacios de Hilbert con kernel de reproducción (RKHS o *Reproducing Kernel Hilbert Spaces*), de tal forma que se resuelva en el espacio transformado un problema convexo de mínimos cuadrados. Varios filtros adaptativos kernel básicos se obtienen aplicando un filtro adaptativo lineal directamente sobre los datos transformados, como se puede observar en la Fig. 2.3. En consecuencia, resulta necesario reformular las operaciones basadas en el producto escalar en términos de "evaluaciones kernel". Los algoritmos resultantes típicamente consisten en expresiones algebraicamente sencillas, a pesar de que cuentan con potentes capacidades de filtrado no lineal. No obstante, el diseño de este tipo de métodos kernel *online* requiere lidiar con algunos de los retos que típicamente surgen cuando se hace uso de los kernels, como por ejemplo los problemas de *overfitting* y de complejidad computacional.

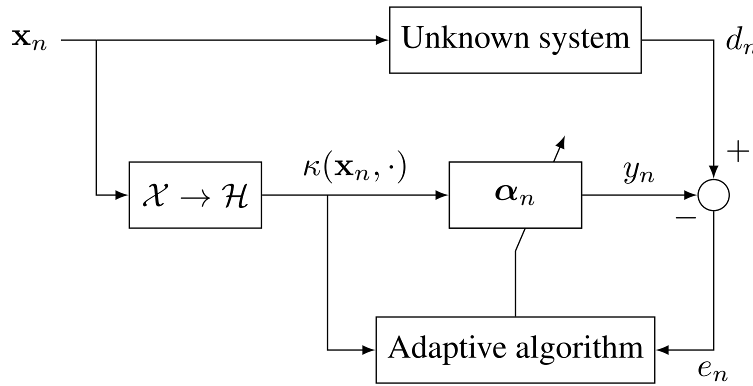


Figura 2.3: Filtro adaptativo kernel para identificación de sistemas no lineales.

Fuente: Adaptive Kernel Learning for Signal Processing [9]

2.3.1. Métodos kernel

El secreto del éxito de numerosos algoritmos de *Machine Learning* se basa en la búsqueda de un espacio de características efectivo y adecuado para el problema que se desea resolver. Numerosas aplicaciones aplican una etapa previa de reducción de la dimensionalidad, en la que se aplican, por ejemplo, los algoritmos PCA (*Principal Component Analysis*) o LDA (*Latent Dirichlet Allocation*).

$$\mathbf{x}_i \in \mathbb{R}^d \to \mathbf{z}_i \in \mathbb{R}^r, \quad r < d \tag{2.7}$$

En cambio, los métodos kernel utilizados en el filtrado adaptativo no lineal siguen una aproximación distinta en la que se realiza (de manera implícita) una expansión de la dimensionalidad.

$$\mathbf{x}_i \in \mathbb{R}^d \to \Phi(\mathbf{x}_i) \in \mathbb{R}^r, \quad r \gg d \tag{2.8}$$

La ventaja que proporciona el ir a un espacio de dimensión más alta es que se consiguen soluciones lineales en dichos espacios de características, que se convierten en soluciones no lineales en el espacio de entrada. Esto se puede ver en la Fig. 2.4, donde se ha considerado un problema de clasificación binaria en $\mathbb{R}^2$ y en el que los datos transformados pertenecen a $\mathbb{R}^3$.

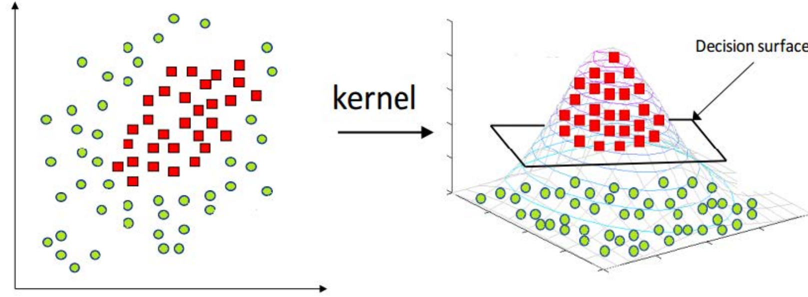


Figura 2.4: Métodos kernel aplicados a un problema de clasificación binaria.

Fuente: Métodos kernel para clasificación [18]

Además, habitualmente no resulta necesario conocer explícitamente el mapeado $\Phi(\mathbf{x})$, sino que basta con conocer la función kernel asociada

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) \tag{2.9}$$

, la cual se encarga de realizar el mapeado y el cómputo de los productos escalares al mismo tiempo. Esto es lo que se conoce comúnmente con el nombre de *kernel trick*.

2.3.2. Algoritmo KLMS

Los primeros acercamientos al filtrado adaptativo kernel trajeron consigo la introducción de una versión kernel de las célebres redes ADALINE [19] (similares al perceptrón pero utilizando la regla de aprendizaje del algoritmo LMS), aunque este método no era *online*. También se propuso en [20] un algoritmo que aplicaba el descenso estocástico por gradiente en RKHS. Este algoritmo es popularmente conocido como NORMA (*Naive Online Regularized Risk Minimization Algorithm*) aprende de manera *online* y puede demostrarse que es equivalente a una versión kernel del algoritmo *Leaky* LMS, el cual es una versión regularizada del LMS convencional.

2.3.2.1. Derivación del KLMS

El algoritmo *Kernel Least Mean Squares* (KLMS) [21] no es más que la *kernelización* del algoritmo LMS estándar. En esencia, una función no lineal $\boldsymbol{\phi}(\cdot)$ mapea los datos $\mathbf{x}_n$ desde el espacio de la entrada a $\boldsymbol{\phi}(\mathbf{x}_n)$ en el espacio de características. Se asume que $\mathbf{w}_{\mathcal{H}}$ es el vector de pesos en este nuevo espacio, de tal manera que la salida del filtro es $y_n = \mathbf{w}_{\mathcal{H},n}^T \mathbf{x}_n$, donde $\mathbf{w}_{\mathcal{H},n}$ representa la estima de $\mathbf{w}_{\mathcal{H}}$ en el instante de tiempo n. Además, cabe destacar que para la

derivación de los métodos kernel nos centraremos en el caso de señales reales, ya que las aplicaciones presentadas en este proyecto trabajan con señales de este tipo. Sabiendo la respuesta deseada del filtro d_n, se pretende minimizar el error cuadrático $J_{\mathbf{w}_{\mathcal{H},n}}$ con respecto a $\mathbf{w}_{\mathcal{H}}$. De una forma similar a la Ec. 2.6, la regla de actualización de descenso estocástico por gradiente viene ahora dada por

$$\mathbf{w}_{\mathcal{H},n} = \mathbf{w}_{\mathcal{H},n-1} + \eta e_n \phi\left(\mathbf{x}_n\right) \tag{2.10}$$

Inicializando $\mathbf{w}_{\mathcal{H},0} = 0$ (y por lo tanto $e_0 = d_0 = 0$), la solución después de n iteraciones se puede expresar de forma cerrada como

$$\mathbf{w}_{\mathcal{H},n} = \eta \sum_{i=1}^{n} e_i \ \phi\left(\mathbf{x}_i\right) \tag{2.11}$$

Explotando el denominado *kernel trick*, se puede obtener la función de predicción

$$y_* = \eta \sum_{i=1}^{n} e_i \ \phi\left(\mathbf{x}_i\right) \phi\left(\mathbf{x}_*\right) = \eta \sum_{i=1}^{n} e_i \ \kappa\left(\mathbf{x}_i, \mathbf{x}_*\right) \tag{2.12}$$

donde $\mathbf{x}_*$ representa un nuevo dato de entrada y $\kappa\left(\cdot,\cdot\right)$ es la función kernel. En este proyecto se ha utilizado el comúnmente usado kernel Gaussiano $\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma_k^2}}$, donde σ_k representa la anchura del kernel (*kernel width*).

Como se ha podido observar, en el algoritmo KLMS no se utilizan explícitamente los pesos $\mathbf{w}_{\mathcal{H},n}$ del filtro no lineal. Además, debido a que la salida en el momento actual solo viene determinada por los datos de entrada previos y todos los errores anteriores, esta se puede calcular fácilmente en el espacio de la entrada. Estas muestras de error son similares a los términos de innovación en el filtrado adaptativo lineal [14], ya que añaden nueva información para mejorar la estimación de la salida. Cada nueva muestra de entrada da lugar a una salida y, por tanto, a un correspondiente error, que no se modifica posteriormente y se incorpora para la estimación de la siguiente salida. Este cálculo recursivo hace que el algoritmo KLMS sea especialmente útil para el procesado de señal no lineal *online* (adaptativo).

En [21] se demuestra que el algoritmo KLMS está bien planteado en un RKHS sin necesidad de introducir un término de regularización extra en el caso de que los datos de entrenamiento sean finitos, ya que se fuerza que la solución siempre pertenezca al subespacio abarcado por los datos de entrada. La ausencia de este término de regularización explícito trae consigo dos importantes ventajas. En primer lugar, el algoritmo KLMS tiene una implementación mucho más sencilla que NORMA, ya que las ecuaciones de actualización son versiones kernel directas de las ecuaciones lineales originales del LMS. En segundo lugar, el algoritmo KLMS potencialmente puede proporcionar mejores resultados, ya que la regularización sesga la solución óptima. En particular, se demostró que un paso lo suficientemente pequeño puede proporcionar un mecanismo suficiente de auto-regularización. Es más, como el espacio abarcado por las muestras mapeadas es posiblemente de infinitas dimensiones, la proyección del error de la señal deseada d_n podría ser muy pequeña, de acuerdo al teorema de Cover [22]. Por contra, la velocidad de convergencia y el desajuste con respecto a la solución óptima también dependen del paso o *step size* escogido. Como consecuencia, estas prestaciones entran en conflicto con la capacidad de generalización.

2.3.2.2. Desafíos en la implementación y formulación dual

Otra desventaja importante del algoritmo KLMS se hace evidente cuando se analiza la Ec. 2.12. Para hacer una predicción, el algoritmo necesita almacenar todos los errores e_i previos y todos los datos de entrada $\mathbf{x}_i$ procesados hasta el momento (para $i = 1, 2, ..., n$). En escenarios de aprendizaje *online* donde se reciben continuamente datos, el tamaño del filtro KLMS crecerá indefinidamente, lo que supone un desafío para su implementación. Este aspecto se hace incluso más evidente si se expresa la actualización de los pesos dada por la Ec. 2.10 con una formulación de filtrado más estándar, apoyándose en el teorema de representación (*Representer Theorem*) [23]. Este teorema establece que la solución $\mathbf{w}_{\mathcal{H},n}$ se puede expresar como una combinación lineal de los datos de entrada transformados:

$$\mathbf{w}_{\mathcal{H},n} = \sum_{i=1}^{n} \alpha_i \ \phi\left(\mathbf{x}_i\right) \tag{2.13}$$

Esto permite que la función de predicción se pueda reescribir como la conocida expansión kernel:

$$y_* = \sum_{i=1}^{n} \alpha_i \ \kappa\left(\mathbf{x}_i, \mathbf{x}_*\right) \tag{2.14}$$

Los coeficientes de la expansión α_i se denominan variables duales y la reformulación del problema de filtrado en términos de los α_i se conoce con el nombre de formulación dual. La regla de actualización de la Ec. 2.10 se convierte ahora en

$$\sum_{i=1}^{n} \alpha_i \ \phi\left(\mathbf{x}_i\right) = \sum_{i=1}^{n-1} \alpha_i \ \phi\left(\mathbf{x}_i\right) + \eta e_n \phi\left(\mathbf{x}_n\right) \tag{2.15}$$

y tras multiplicar a ambos lados por el nuevo dato $\phi\left(\mathbf{x}_n\right)$ y adoptando una notación vectorial, se obtiene

$$\boldsymbol{\alpha}_n^T \mathbf{k}_n = \boldsymbol{\alpha}_{n-1}^T \mathbf{k}_{n-1} + \eta e_n k_{n,n} \tag{2.16}$$

donde $\boldsymbol{\alpha}_n = \left[\alpha_1, \alpha_2, \ldots, \alpha_n\right]^T$, el vector $\mathbf{k}_n$ contiene los kernels de los n datos y el dato más nuevo, $\mathbf{k}_n = \left[\kappa\left(\mathbf{x}_1, \mathbf{x}_n\right), \kappa\left(\mathbf{x}_2, \mathbf{x}_n\right), \ldots, \kappa\left(\mathbf{x}_n, \mathbf{x}_n\right)\right]^T$, y $k_{n,n} = \kappa\left(\mathbf{x}_n, \mathbf{x}_n\right)$. El algoritmo KLMS resuelve esta relación actualizando el vector $\boldsymbol{\alpha}_n$ como

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ \eta e_n \end{bmatrix} \tag{2.17}$$

El código de MATLAB para realizar una iteración completa de entrenamiento para un nuevo par de datos $(\mathbf{x}, d)$ se muestra en el Script A.1.2 del Anexo.

La regla de actualización de la Ec. 2.17 enfatiza la naturaleza creciente del filtro KLMS, lo que imposibilita su implementación directa en la práctica. Para poder diseñar un algoritmo KLMS práctico, el número de términos en la expansión kernel de la Ec. 2.14 debería estar de alguna manera limitado. Esto se puede conseguir implementando alguna técnica para limitar la complejidad del filtro, conocidas como técnicas de *sparsification*, cuyo objetivo es identificar los

términos de la expansión kernel que se pueden omitir sin degradar la solución. En el siguiente apartado se comentarán las técnicas de este tipo utilizadas en el proyecto.

Por último, cabe destacar que la complejidad computacional y de memoria del algoritmo KLMS son lineales en ambos casos en términos del número de datos que almacena, $\mathcal{O}(n)$. La complejidad del algoritmo LMS también es lineal, aunque no en términos del número de datos sino en los de la dimensión de estos datos, $\mathcal{O}(L)$.

2.3.3. Limitando la complejidad del KLMS: Técnicas de *sparsification*

Tal y como se ha comentado, la idea detrás de los métodos de *sparsification* es construir un diccionario disperso de bases que representen lo suficientemente bien los datos no presentes en el mismo. Como regla general en teoría del aprendizaje, es deseable diseñar una red con el menor número posible de elementos de procesado. El empleo de estas técnicas de *sparsification* reduce la complejidad en términos de computación y de memoria, y generalmente brinda una mejor capacidad de generalización para los nuevos datos que se procesen [24] [25]. En el contexto de los métodos kernel, la *sparsification* busca identificar las bases de la expansión kernel $y_* = \sum_{i=1}^{n} \alpha_i \ \kappa(\mathbf{x}_i, \mathbf{x}_*)$ que pueden ser descartadas sin incurrir en una pérdida de rendimiento significativa.

La *sparsification online* se realiza típicamente empezando con un diccionario vacío, $\mathcal{D}_0 = \emptyset$, y, en cada iteración, se añade el dato de entrada $\mathbf{x}_i$ si cumple con un determinado criterio de *sparsification*. Se denota al diccionario en el instante de tiempo $n-1$ como $\mathcal{D}_{n-1} = \{\mathbf{c}_i\}_{i=1}^{m_{n-1}}$, donde $\mathbf{c}_i$ es el centro almacenado i, tomado de los datos de entrada $\mathbf{x}$ recibidos hasta ese momento, y m_{n-1} representa la cardinalidad del diccionario en dicho instante. Cuando se recibe un nuevo par entrada-salida $(\mathbf{x}_n, d_n)$, se toma la decisión de si se debe añadir $\mathbf{x}_n$ como centro al diccionario o no. Si se cumple el criterio de *sparsification*, entonces $\mathbf{x}_n$ se añade al diccionario, $\mathcal{D}_n = \mathcal{D}_{n-1} \cup \{\mathbf{x}_n\}$. Si el criterio no se cumple, el diccionario se mantiene como está, $\mathcal{D}_n = \mathcal{D}_{n-1}$, para preservar su complejidad. Este tipo de técnicas se denominan de *sparsification* por construcción y son las utilizadas por el algoritmo QKLMS, el cual se explicará en el siguiente apartado.

Si no se permite que el diccionario crezca más allá de un tamaño máximo especificado m, debido por ejemplo a limitaciones en hardware o en tiempo de ejecución, resulta necesario descartar las bases a partir de un cierto punto. A este proceso se le conoce con el nombre de "poda" o *pruning* y es otro método de *sparsification* utilizado en filtrado adaptativo. En el caso de este proyecto, se implementará el algoritmo KLMS haciendo uso de esta técnica de *sparsification* y limitando el tamaño máximo del diccionario a $m = 1000 \ muestras$.

En la Fig. 2.5 se ilustran tres procesos de construcción del diccionario, cada uno utilizando una técnica de *sparsification* diferente. En ella, cada línea horizontal representa la presencia de un centro en el diccionario. Para cualquier iteración dada, los elementos en el diccionario vienen indicados por las líneas horizontales presentes en dicha iteración.

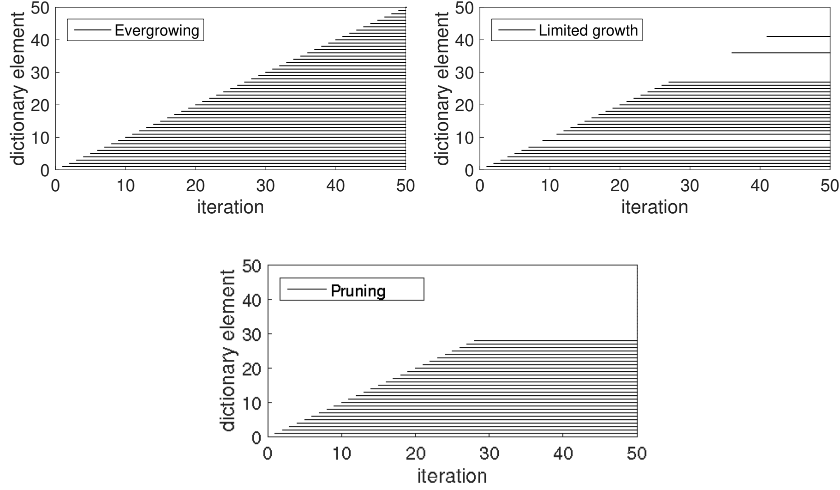


Figura 2.5: Construcción siempre creciente del diccionario (arriba a la izquierda), en el que el diccionario contiene n elementos en la iteración n, y mediante técnicas de *sparsification* por construcción (arriba a la derecha) y por “poda”(abajo), que ralentizan su crecimiento.

Fuente: Adaptive Kernel Learning for Signal Processing [9]

2.3.4. Algoritmo QKLMS

Una de las técnicas que consiguen la *sparsification* por construcción es mediante el empleo del denominado criterio de coherencia. Se trata de un criterio sencillo que comprueba si el dato recién llegado es lo suficientemente informativo y, por tanto, si se debe incluir en el diccionario.

Dado el diccionario $\mathcal{D}_{n-1}$ en la iteración $n-1$ y el dato recién llegado $\mathbf{x}_n$, el criterio de coherencia para incluir el nuevo dato es

$$\max_{j \in \mathcal{D}_{n-1}} |\kappa(\mathbf{x}_n, \mathbf{c}_j)| < \mu_0 \tag{2.18}$$

En esencia, el criterio de coherencia comprueba la similitud, medida por la función kernel, entre el nuevo dato y el elemento del diccionario más similar a este. Si esta similitud se encuentra por debajo de un cierto umbral μ_0, el dato se inserta en el diccionario. Cuanto más alto sea el valor escogido de este umbral μ_0, más datos se incluirán en el diccionario. Como se puede ver, se trata de un criterio efectivo que posee una complejidad computacional lineal en cada iteración, ya que solo requiere calcular m funciones kernel, haciéndolo especialmente útil para algoritmos de tipo KLMS.

En el año 2012, se propuso un algoritmo KLMS que utiliza una versión del criterio de coherencia para seleccionar los datos que se deben incluir en el diccionario. Este nuevo algoritmo recibe el nombre de KLMS cuantificado (QKLMS) [26].

En este caso, el criterio utilizado es

$$\min_{j \in \mathcal{D}_{n-1}} ||\mathbf{x}_n - \mathbf{c}_j|| > \epsilon_u \tag{2.19}$$

que en esencia es equivalente al criterio de coherencia con un kernel basado en distancia Euclídea. En este caso, cuanto más pequeño sea el umbral ϵ_u elegido, mayor número de datos se incluirán en el diccionario.

En este algoritmo, cuando el criterio de *sparsification* decide incluir un nuevo dato en el diccionario, se actualizan los coeficientes de la siguiente manera:

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ \eta e_n \end{bmatrix} \tag{2.20}$$

Es decir, los coeficientes α_i se actualizan de la misma manera que se hacía en el algoritmo KLMS convencional (véase la Ec. 2.17).

En caso de que el dato no cumpla el criterio de distancia Euclídea 2.19, este no es incluido en el diccionario. En su lugar, se recupera el elemento del diccionario más cercano y se actualiza su correspondiente coeficiente como

$$\alpha_{n,j} = \alpha_{n-1,j} + \eta e_n \tag{2.21}$$

donde j es el índice del diccionario del elemento más cercano. A pesar de que conceptualmente es muy sencillo, este algoritmo obtiene buenos resultados en aquellas aplicaciones en las que la capacidad de almacenamiento es limitada.

El código de MATLAB para realizar una iteración completa de entrenamiento del algoritmo QKLMS para un nuevo par de datos $(\mathbf{x}, d)$ se muestra en el Script A.1.3 del Anexo.

2.3.5. *Random Fourier Features*

El uso de las distintas técnicas de *sparsification* ha conseguido mejorar el algoritmo KLMS original y ha permitido que se pueda implementar de una manera práctica. Sin embargo, estos métodos siguen presentando ciertas desventajas. En primer lugar, estas técnicas no consiguen solucionar totalmente el problema del almacenamiento creciente. Estos algoritmos consiguen frenar la tasa de crecimiento lineal del algoritmo KLMS y hacen que esta sea sublineal. No obstante, todavía presentan una complejidad creciente con el número de iteraciones. Por ello, para aplicaciones en tiempo real que requieran hacer uso de estos algoritmos de aprendizaje durante largos periodos de tiempo en condiciones no estacionarias, estos algoritmos pueden llegar a crecer descontrolados. En segundo lugar, las técnicas de *sparsification* utilizadas introducen una carga computacional adicional al algoritmo en cada iteración, lo que hace que su tiempo de ejecución sea superior al del algoritmo original.

Para sortear los inconvenientes que presentan las soluciones basadas en técnicas de *sparsification* como el algoritmo QKLMS, se decidió tomar otro enfoque a la hora de abordar el problema de la naturaleza creciente del KLMS. En lugar de intentar restringir el número de vectores de entrada que se incorporan al diccionario y frenar así la tasa de crecimiento de la expansión kernel, se decidió abordar el problema desde la raíz.

Si se observa la Ec. 2.14 del algoritmo KLMS, se puede apreciar que el problema de crecimiento surge asociado al teorema de representación. El uso de este teorema es imprescindible, ya que la representación como vector de pesos de la

función de predicción no resulta nada práctica en espacios funcionales debido a su infinita dimensionalidad. Por ello, diversos autores propusieron utilizar aproximaciones de dimensiones finitas de los pesos de la función. En otras palabras, propusieron proyectar los vectores de entrada en un espacio de dimensiones finitas apropiado en el que los productos escalares sean aproximaciones cercanas a los productos escalares infinito dimensionales originales (que son simplemente evaluaciones kernel). En resumen, lo que pretendían conseguir de esta forma fue un mapeado de dimensiones finitas $\Psi : \mathbb{R}^d \rightarrow \mathbb{R}^D$ (siendo $d = L$) tal que

$$\langle \mathbf{\Psi}(\mathbf{x}_i), \mathbf{\Psi}(\mathbf{x}_j) \rangle \approx \kappa(\mathbf{x}_i, \mathbf{x}_j) \tag{2.22}$$

Estos vectores $\mathbf{\Psi}(\mathbf{x})$ se obtienen usando mapas de características aleatorizados. La aleatorización o *randomization* es un principio relativamente antiguo utilizado en el ámbito del aprendizaje automático. Por ejemplo, la utilización de redes aleatorias de funciones no lineales en problemas de regresión ha resultado bastante satisfactoria [27].

Los mapas de características aleatorizados más utilizados son los denominados *RFF* o *Random Fourier Features*. La idea subyacente se basa en el teorema de Bochner [28], resultado fundamental en el análisis armónico que establece que cualquier función kernel κ que sea invariante a la traslación ($\kappa(\mathbf{x}_i, \mathbf{x}_j) = \kappa(\mathbf{x}_i - \mathbf{x}_j)$) y que esté normalizada ($\kappa(\mathbf{x}_i - \mathbf{x}_j) \leq \kappa(0) = 1$) es una transformada de Fourier de una medida de probabilidad definida de manera única en $\mathbb{R}^d$.

Es decir, el teorema de Bochner garantiza que la transformada de Fourier de un kernel apropiadamente escalado e invariante a la traslación es una densidad de probabilidad. Definiendo $z_\omega(x) = e^{j\omega^T \mathbf{x}}$, se obtiene

$$\kappa(\mathbf{x}_i - \mathbf{x}_j) = \int_{\mathcal{R}^d} p(\omega)\ e^{j\omega^T(\mathbf{x}_i - \mathbf{x}_j)}\ d\omega = \mathbb{E}_\omega\left[z_\omega(\mathbf{x}_i)\, z_\omega(\mathbf{x}_j)^*\right] \tag{2.23}$$

Por lo tanto, $z_\omega(\mathbf{x}_i)\, z_\omega(\mathbf{x}_j)^*$ es un estimador insesgado de $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ cuando ω sigue la distribución p. Para reducir la varianza de esta estima, se toma la media muestral de D puntos $z_\omega(\cdot)$ escogidos aleatoriamente. Por tanto, el producto interno D-dimensional $\frac{1}{D}\sum_{k=1}^{D} z_{\omega k}(\mathbf{x}_i)\, z_{\omega k}(\mathbf{x}_j)$ es una aproximación de baja varianza de la función kernel $\kappa(\mathbf{x}_i, \mathbf{x}_j)$.

En general, las características $z_\omega(\mathbf{x})$ son complejas. Sin embargo, explotando la propiedad de simetría del kernel κ, se puede expresar utilizando bases de cosenos con valores reales. Por consiguiente, el mapeado

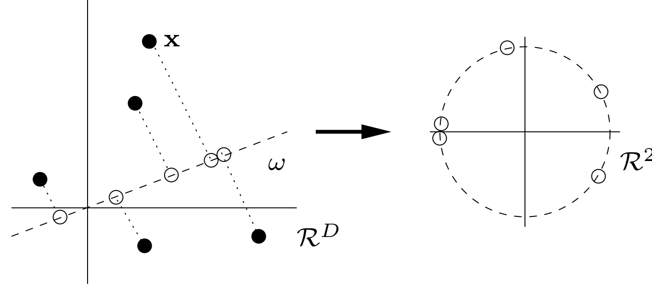
$$\psi_\omega(\mathbf{x}) = \sqrt{2}\cos\left(\omega^T \mathbf{x} + b\right) \tag{2.24}$$

donde ω sigue la distribución p y b sigue una distribución uniforme en $[0, 2\pi]$, también satisface la condición $\mathbb{E}_\omega[\psi_\omega(\mathbf{x}_i)\, \psi_\omega(\mathbf{x}_j)] = \kappa(\mathbf{x}_i - \mathbf{x}_j)$.

El vector $\mathbf{\Psi}(\mathbf{x}) = [\psi_{\omega_1}(\mathbf{x}), \psi_{\omega_2}(\mathbf{x}), \ldots, \psi_{\omega_D}(\mathbf{x})]$ es por tanto una característica aleatoria de Fourier o *Random Fourier Feature* (RFF) D-dimensional del vector de entrada $\mathbf{x}$. Este mapeado satisface la aproximación de la Ec. 2.22.

Para realizar la aproximación del kernel gaussiano de anchura σ_k utilizado en este proyecto es necesario que los términos $\omega_i,\ i = 1, \ldots, D$ sigan una distribución normal $\mathcal{N}\left(0, \dfrac{\mathbf{I}_d}{\sigma_k^2}\right)$ (véase la Fig. 2.6).

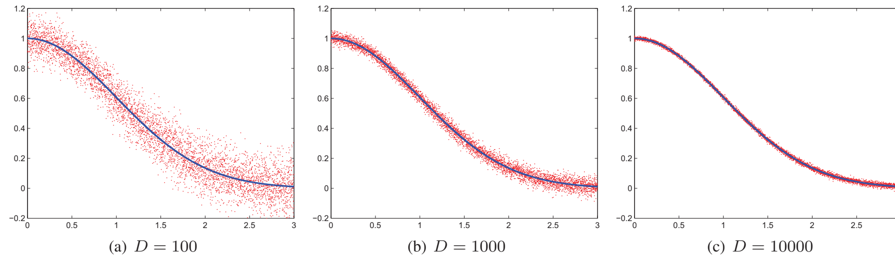
La calidad de la aproximación realizada viene determinada por la dimensionalidad del mapeado D, que es un parámetro definido por el usuario. En la Fig. 2.7, se muestra una vista cualitativa de la aproximación de $\kappa(\mathbf{x}, 0)$, utilizando el producto interno D-dimensional $\langle \boldsymbol{\Psi}(\mathbf{x}), \boldsymbol{\Psi}(\mathbf{0}) \rangle$ descrito anteriormente, para diferentes valores de D.



Kernel Name	$k(\Delta)$	$p(\omega)$
Gaussian	$e^{-\frac{\|\Delta\|_2^2}{2}}$	$(2\pi)^{-\frac{D}{2}} e^{-\frac{\|\omega\|_2^2}{2}}$

Figura 2.6: Cada componente del mapa de características $z_\omega(\mathbf{x})$ proyecta $\mathbf{x}$ sobre una dirección aleatoria ω dibujada a partir de la transformada de Fourier $p(\omega)$ de $\kappa(\Delta)$, y envuelve esta línea sobre el círculo unitario en $\mathcal{R}^2$. El mapeado $\psi_\omega(\mathbf{x}) = \cos\left(\omega^T \mathbf{x} + b\right)$ adicionalmente rota este círculo una cantidad aleatoria b y proyecta los puntos en el intervalo $[0, 1]$.

Fuente: Random Features for Large-Scale Kernel Machines [29]



(a) $D = 100$ (b) $D = 1000$ (c) $D = 10000$

Figura 2.7: Vista cualitativa de la aproximación del producto interno del kernel gaussiano mediante *Random Fourier Features*. La curva azul muestra la función gaussiana exacta y el diagrama de *sparsification* rojo muestra la aproximación realizada para distintos valores de D. Claramente, la varianza en la aproximación decrece a medida que se incrementa D.

Fuente: Online Learning with Kernels [30]

2.3.6. Algoritmo RFF-KLMS

Gracias a esto, en 2012 se propuso un nuevo algoritmo denominado *Random Fourier Features* KLMS (RFF-KLMS) [30], que hacía uso de las aproximaciones de características aleatorias de Fourier de los kernels para realizar el descenso estocástico por gradiente en el que se basa el algoritmo KLMS.

En este nuevo algoritmo, en vez de utilizar las evaluaciones kernel exactas para realizar el aprendizaje se usa el espacio de las *Random Fourier Features* para mapear explícitamente los vectores de datos de entrada. Debido a que este espacio es de dimensión finita, ahora es posible trabajar directamente con los pesos del filtro en este espacio. De esta forma, las ecuaciones de actualización del KLMS en este nuevo espacio RFF se convierten exactamente en las mismas que las del algoritmo LMS.

El pseudocódigo del algoritmo RFF-KLMS se muestra a continuación:

Algoritmo 2.1: RFF-KLMS

Input: Datos de entrenamiento secuenciales $\{\mathbf{x}_n, d_n\}_{n=1}^N$, anchura del kernel gaussiano σ_k, tamaño del paso μ, dimensión RFF D

1 Extraer i.i.d. $\{\omega_i\}_{i=1}^D$ de $\mathcal{N}\left(0, I_d/\sigma_k^2\right)$, donde d es la dimensión del espacio de entrada
2 Extraer i.i.d. $\{b_i\}_{i=1}^D$ de $U\left(0, 2\pi\right)$
3 Inicializar $\mathbf{\Omega}_1 = 0 \in \mathcal{R}^D$
4 **for** $n = 1 : N$
5 Calcular el vector RFF $\mathbf{\Psi}\left(\mathbf{x}_n\right) = \left[\psi_{\omega_1}\left(\mathbf{x}_n\right), \ldots, \psi_{\omega_D}\left(\mathbf{x}_n\right)\right]$, donde cada $\psi_{\omega_i}\left(\mathbf{x}_n\right) = \cos\left(\omega_i^T \mathbf{x}_n + b_i\right)$
6 Calcular la predicción $y_n = \langle \mathbf{\Omega}_n, \mathbf{\Psi}\left(\mathbf{x}_n\right)\rangle$
7 Calcular el error $e_n = d_n - y_n$
8 Actualizar $\mathbf{\Omega}_{n+1} = \mathbf{\Omega}_n + \mu e_n \mathbf{\Psi}\left(\mathbf{x}_n\right)$
9 **end**

El código de MATLAB para realizar una iteración completa de entrenamiento del algoritmo RFF-KLMS para un nuevo par de datos $(\mathbf{x}, d)$ se muestra en el Script A.1.4 del Anexo.

2.3.7. KAFBOX

La *Kernel Adaptive Filtering Toolbox* (`https://github.com/steven2358/kafbox/tree/dev`) [31], desarrollada por Steven Van Vaerenbergh, es una *toolbox* de MATLAB que permite evaluar y comparar distintos algoritmos de filtrado adaptativo kernel (KAF). Esta *toolbox* incluye una larga lista de algoritmos que han ido apareciendo en la literatura, así como herramientas adicionales para la estimación de los hiperparámetros y para la optimización y prototipado de los algoritmos.

Los algoritmos de filtrado adaptativo kernel de la *toolbox* KAFBOX se implementan como objetos usando la sintaxis `classdef`. Como todos los algoritmos KAF son métodos *online*, cada uno de ellos incluye dos operaciones básicas:

- **Evaluar**: obtener la salida del filtro, dada una nueva entrada $\mathbf{x}_n$. Esta operación viene implementada en el método `evaluate`.

- **Entrenar**: actualizar los parámetros del filtro, dado un nuevo par de datos de entrada $(\mathbf{x}_n, d_n)$. Esta operación viene implementada en el método `train`.

Como ejemplo, en el Script A.2 del Anexo se muestra el código del algoritmo KLMS. La definición del objeto contiene dos conjuntos de propiedades: una para los hiperparámetros (líneas 14-19) y otra para las variables que aprenderá a lo largo del entrenamiento (líneas 21-24). El primer método que aparece (líneas 27-35) es el constructor del objeto, que copia la configuración de hiperparámetros especificada. El segundo método (líneas 37-44) es la función `evaluate`, que realiza la operación $y_* = \sum_{i=1}^{n} \alpha_i \, \kappa\left(\mathbf{x}_i, \mathbf{x}_*\right)$. Además, incluye una sentencia `if` para comprobar si el algoritmo ya ha realizado al menos una iteración del entrenamiento. En caso contrario, se devolverán valores 0 como predicción. Finalmente, el método `train` implementa una única iteración del algoritmo de aprendizaje *online*. Este método típicamente se encarga también de la inicialización del algoritmo, de tal manera que las funciones que operan sobre un determinado objeto KAF no se tengan que preocupar de inicializar las distintas variables. Para los algoritmos utilizados en este proyecto, la iteración de entrenamiento ocupa unas pocas líneas de código MATLAB.

Por último, cabe destacar que para la realización de este trabajo se utilizará la versión 2.2 de la *toolbox* KAFBOX.

Capítulo 3

Filtrado kernel adaptativo *Multi-Output*

En este capítulo, se planteará el problema de predicción *Multi-Output* y se explicarán las distintas aproximaciones al mismo llevadas a cabo en este proyecto.

En la sección 3.1 se planteará el nuevo escenario *Multi-Output*. En la sección 3.2 se explicarán las técnicas de concatenación y los algoritmos desarrollados en este proyecto (LMS-MO, KLMS-MO, QKLMS-MO y RFF-KLMS-MO). Por último, en la sección 3.3 se definirá una arquitectura de algoritmos de filtrado adaptativo diseñada para intentar obtener mejores resultados que con la aplicación de los mismos por separado.

3.1. Planteamiento del problema *Multi-Output*

Todos los algoritmos que se han explicado hasta el momento tienen como objetivo predecir una única salida, es decir son *Single-Output*. No obstante, en este trabajo se pretende convertir estos algoritmos en *Multi-Output* y poder así predecir varias salidas al mismo tiempo. A estos nuevos algoritmos se los denominará con el sufijo MO (*Multi-Output*), mientras que los *Single-Output* tendrán asociado el sufijo SO.

Formalmente, el problema de predicción *Single-Output* presentado en el capítulo anterior requiere estimar una única salida a partir de una entrada $\mathbf{x}_*$ dada. Para ello, es necesario construir un estimador $f_*(\mathbf{x}_*)$ sobre la base de un conjunto de entrenamiento formado por N parejas de datos entrada-salida $\mathcal{S} = (\mathbf{X}, \mathbf{Y}) = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$. El espacio de entrada $\mathcal{X}$ es, por tanto, un espacio de vectores mientras que el espacio de salida es un espacio de escalares. En cambio, en el aprendizaje *Multi-Output*, el espacio de salida también es un espacio de vectores y el estimador $\mathbf{f}$ es una función de valores vectoriales. En realidad, también se puede ver esta situación como un problema en el que hay que resolver M problemas *Single-Output*, donde cada problema viene descrito por una de las componentes $f_1, \ldots, f_M$ de $\mathbf{f}$.

Para solucionar este tipo de problemas, la idea clave es trabajar bajo la suposición de que los distintos problemas están relacionados de alguna manera. La idea, por tanto, consiste en explotar la relación existente entre dichos proble-

mas para obtener mejores resultados que en el caso de resolver cada problema de manera individual.

Hoy en día, existen diversas técnicas para solucionar este tipo de problemas de manera eficiente, como pueden ser los procesos Gaussianos o *Gaussian Processes* (GPs) aplicados a funciones de valores vectoriales [6]. Un GP es un proceso estocástico que presenta la característica de que cualquier número finito de variables aleatorias tomadas de una realización del mismo siguen una distribución Gaussiana conjunta. Los GPs se suelen utilizar como distribuciones de probabilidad *a priori* para las funciones de estimación f. Por tanto, desde un punto de vista bayesiano, los GPs especifican las creencias previas que se tienen sobre las propiedades de las funciones que se desean modelar. Estas creencias se van actualizando a medida que se van recibiendo nuevos datos por medio de una función de probabilidad, que relaciona las creencias previas con las observaciones actuales. Esto da lugar a una distribución de probabilidad actualizada, la distribución *a posteriori*, que se utiliza para ir realizando la predicción de cada una de las salidas.

No obstante, los GPs presentan una complejidad computacional cúbica con el número de salidas y de muestras utilizadas y de almacenamiento cuadrática con el número de salidas y de muestras utilizadas. Por ello y ya que apenas existen en la actualidad técnicas de complejidad lineal, en este trabajo se busca un enfoque mucho más simple, basado en versiones *Multi-Output* de algoritmos tipo *Kernel Least Mean Square* o KLMS, que permita reducir la complejidad de los algoritmos y seguir obteniendo al mismo tiempo buenos resultados con ellos.

3.2. Técnicas de concatenación

Como primera aproximación, en un sistema *Multiple-Input Multiple-Output* (MIMO) de M entradas y M salidas se podría emparejar cada entrada con una salida y aplicar los distintos algoritmos *Single-Output* propuestos a cada par de series temporales (véase la Fig. 3.1).

A pesar de ser una solución válida, esta no es óptima, ya que no tiene en cuenta las correlaciones existentes entre las series temporales de entrada, lo que consigue importantes mejoras en las prestaciones de los distintos algoritmos. Por ello, en este trabajo se propone una forma sencilla pero efectiva de convertir los algoritmos propuestos anteriormente en *Multi-Output* aprovechando esta correlación: la concatenación de los datos de entrada.

El método propuesto consiste, por tanto, en concatenar en cada instante temporal las L muestras de cada una de las M series temporales de entrada, formando así un vector de entrada de dimensiones $(L \cdot M) \times 1$ (véase la Fig. 3.2).

Además, para obtener a la salida la predicción de las M componentes y no solo de una de ellas, resulta necesario redimensionar los pesos del filtro (que pasarán de ser un vector a ser una matriz), así como el resto de variables de cada algoritmo de filtrado adaptativo. Estos cambios se explicarán con mayor detalle para cada algoritmo en los siguientes apartados.

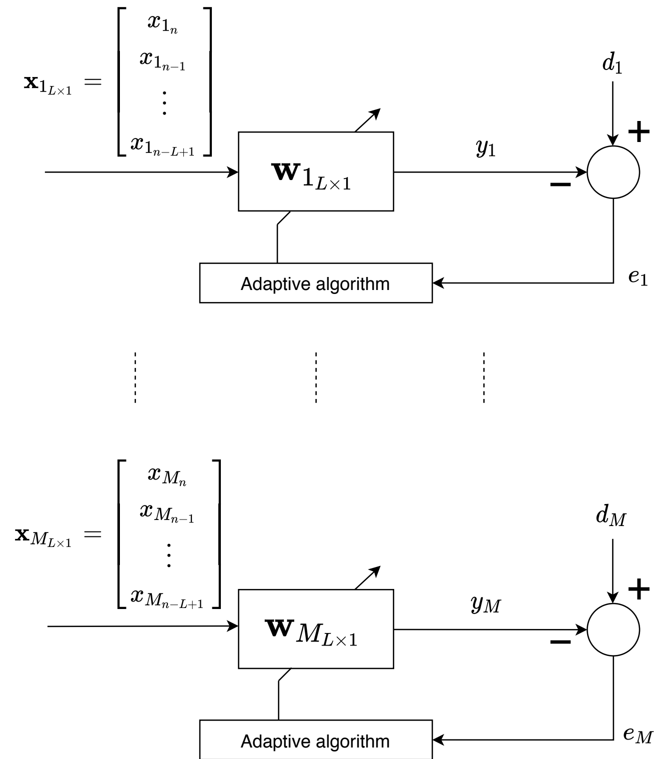


Figura 3.1: Filtrado adaptativo componente a componente en un sistema MIMO de M entradas y M salidas.

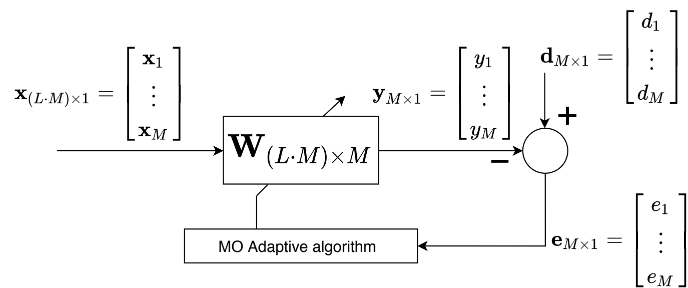


Figura 3.2: Filtrado adaptativo MO en un sistema MIMO de M entradas y M salidas.

3.2.1. Algoritmo LMS-MO

Para adaptar el algoritmo LMS al caso *Multi-Output*, únicamente es necesario redimensionar el tamaño de los pesos del filtro adaptativo, convirtiéndose ahora en una matriz W de dimensiones $(L \cdot M) \times M$. Este método puede ser interpretado de manera equivalente como un LMS-MIMO.

De esta forma, al recibir un nuevo dato de entrada $\mathbf{x}$ de tamaño $(L \cdot M) \times 1$ se obtiene su salida de igual manera que con el algoritmo LMS *Single-Output* (véase la Ec. 2.1):

$$\mathbf{y}_n = \mathbf{W}_n^H \mathbf{x}_n \tag{3.1}$$

Como se puede ver, la salida obtenida es ahora de dimensión $M \times 1$. A continuación, se calcula el error cometido:

$$\mathbf{e}_n = \mathbf{d}_n - \mathbf{y}_n \tag{3.2}$$

Por último, se actualizan los pesos del filtro adaptativo de acuerdo a la regla de actualización del algoritmo LMS (véase la Ec. 2.6), que ahora se convierte en

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \mu \mathbf{x}_n \mathbf{e}_n^H \tag{3.3}$$

siendo W de dimensión $(L \cdot M) \times M$, $\mathbf{x}$ de dimensión $(L \cdot M) \times 1$ y $\mathbf{e}$ de dimensión $M \times 1$.

El código de MATLAB para cada iteración del entrenamiento con un nuevo par de datos $(\mathbf{x}, \mathbf{d})$ se muestra en el Script A.3.1 del Anexo. En este caso, se ha decidido implementar el algoritmo considerando que el vector con los datos de entrada es un vector fila $(1 \times (L \cdot M))$, en vez de un vector columna $((L \cdot M) \times 1)$ tal y como se ha especificado en el desarrollo previo.

3.2.2. Algoritmo KLMS-MO

En el caso del filtrado adaptativo no lineal, el redimensionamiento es muy similar al llevado a cabo para el algoritmo LMS. En este caso, es necesario cambiar el tamaño del vector de los coeficientes α_i, convirtiéndose ahora en una matriz $\mathbf{A}$ que, al finalizar la iteración n, tiene unas dimensiones $n \times M$.

De esta manera, se obtiene la predicción de las M salidas de forma análoga a la del algoritmo KLMS:

$$\mathbf{y}_n = \mathbf{A}_n^T \mathbf{k}_n \tag{3.4}$$

siendo

$$\mathbf{A}_n = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1M} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \dots & \alpha_{nM} \end{bmatrix} \tag{3.5}$$

y

$$\mathbf{k}_n = \begin{bmatrix} \kappa\left(\mathbf{x}_1, \mathbf{x}_n\right) \\ \kappa\left(\mathbf{x}_2, \mathbf{x}_n\right) \\ \vdots \\ \kappa\left(\mathbf{x}_n, \mathbf{x}_n\right) \end{bmatrix} \tag{3.6}$$

A continuación, se calcula el error cometido de acuerdo a la Ec. 3.2 y, por último, se añade el vector que contiene los datos de entrada $\mathbf{x}_n$ al diccionario y se actualiza la matriz $\mathbf{A}$ como:

$$\mathbf{A}_n = \begin{bmatrix} \mathbf{A}_{n-1} \\ \eta \mathbf{e}_n^T \end{bmatrix} \tag{3.7}$$

Como se puede ver, la Ec. 3.7 se corresponde con la Ec. 2.14 del algoritmo KLMS en forma matricial.

El código de MATLAB para cada iteración del entrenamiento con un nuevo par de datos $(\mathbf{x}, \mathbf{d})$ se muestra en el Script A.3.2 del Anexo. Al igual que con el algoritmo LMS-MO, se ha decidido implementar el KLMS-MO considerando que el vector con los datos de entrada es un vector fila en vez de un vector columna.

3.2.3. Algoritmo QKLMS-MO

En el caso del algoritmo QKLMS-MO, el redimensionamiento es el mismo que en el caso del KLMS-MO. La única novedad existente se produce en el caso de que el nuevo dato no cumpla el criterio de coherencia (Ec. 2.19) y, por tanto, no se incluya en el diccionario.

En el algoritmo QKLMS, esto conlleva la recuperación del elemento del diccionario más cercano y la actualización de su correspondiente coeficiente (véase la Ec. 2.21). En el caso del algoritmo QKLMS-MO, cada elemento del diccionario lleva asociado M coeficientes (uno por cada serie temporal, es decir, una fila completa de la matriz A), por lo que la actualización se hará de la siguiente manera

$$\mathbf{A}_{n,j} = \mathbf{A}_{n-1,j} + \eta \mathbf{e}_n^T \tag{3.8}$$

siendo j el índice del diccionario del elemento más cercano.

El código de MATLAB para realizar una iteración completa de entrenamiento del algoritmo QKLMS-MO para un nuevo par de datos $(\mathbf{x}, d)$, considerando el vector $\mathbf{x}$ como un vector fila, se muestra en el Script A.3.3 del Anexo.

3.2.4. Algoritmo RFF-KLMS-MO

Para adaptar el algoritmo RFF-KLMS a su versión *Multi-Output* es necesario redimensionar los siguientes parámetros:

- $\boldsymbol{\omega}_{L\times D} \rightarrow \boldsymbol{\omega}_{(L\cdot M)\times D}$
- $\boldsymbol{\Omega}_{1\times D} \rightarrow \boldsymbol{\Omega}_{M\times D}$

Una vez hecho esto, cada vez que llega un nuevo par de datos de entrada $(\mathbf{x}, d)$ se procede de manera análoga a como se haría con el algoritmo RFF-KLMS *Single-Output*. Para ello, en primer lugar se calcula el vector de características aleatorias

$$\mathbf{\Psi}(\mathbf{x}_n) = \begin{bmatrix} \psi_{\omega_1}(\mathbf{x}_n) & \dots & \psi_{\omega_D}(\mathbf{x}_n) \end{bmatrix} \tag{3.9}$$

donde cada

$$\psi_{\omega_i}(\mathbf{x}_n) = \cos\left(\omega_i^T \mathbf{x}_n + b_i\right) \tag{3.10}$$

En segundo lugar, se calcula la predicción

$$\mathbf{y}_n = \langle \mathbf{\Omega}_n, \mathbf{\Psi}(\mathbf{x}_n) \rangle = \mathbf{\Omega}_n \mathbf{\Psi}^H(\mathbf{x}_n) \tag{3.11}$$

Como se puede ver, la salida obtenida es ahora de dimensión $M \times 1$. A continuación, se calcula el error cometido de acuerdo a la Ec. 3.2 y, por último, se actualiza la matriz $\mathbf{\Omega}$ como

$$\mathbf{\Omega}_{n+1} = \mathbf{\Omega}_n + \mu \mathbf{e}_n \mathbf{\Psi}(\mathbf{x}_n) \tag{3.12}$$

El código de MATLAB para realizar una iteración completa de entrenamiento del algoritmo RFF-KLMS-MO para un nuevo par de datos $(\mathbf{x}, d)$, considerando el vector $\mathbf{x}$ como un vector fila, se muestra en el Script A.3.4 del Anexo.

3.3. Arquitectura de predicción *Multi-Output*

Como última opción, para intentar obtener mejores resultados que con los algoritmos MO, se ha decidido diseñar una arquitectura de filtrado adaptativo que combine las dos estrategias propuestas en la sección anterior: los algoritmos MO y los algoritmos de filtrado adaptativo *Single-Output* aplicados a cada par entrada-salida.

En la Fig. 3.3, se muestra un esquema de la arquitectura diseñada en este proyecto. Para obtener la predicción, la arquitectura diseñada aplica en primer lugar el algoritmo LMS-MO para extraer la parte lineal de la salida deseada $\mathbf{d}$ ($\mathbf{y}_l$), aprovechando para ello la posible correlación de las series temporales de entrada. A continuación, utiliza la salida obtenida $\mathbf{y_l}$ para predecir la parte no lineal de $\mathbf{d}$ ($\mathbf{y}_{nl}$) haciendo uso del algoritmo KLMS aplicado a cada par entrada-salida. Una vez que se dispone de ambas, la predicción final de la arquitectura viene dada por

$$\mathbf{y} = \mathbf{y_l} + \mathbf{y_{nl}} \tag{3.13}$$

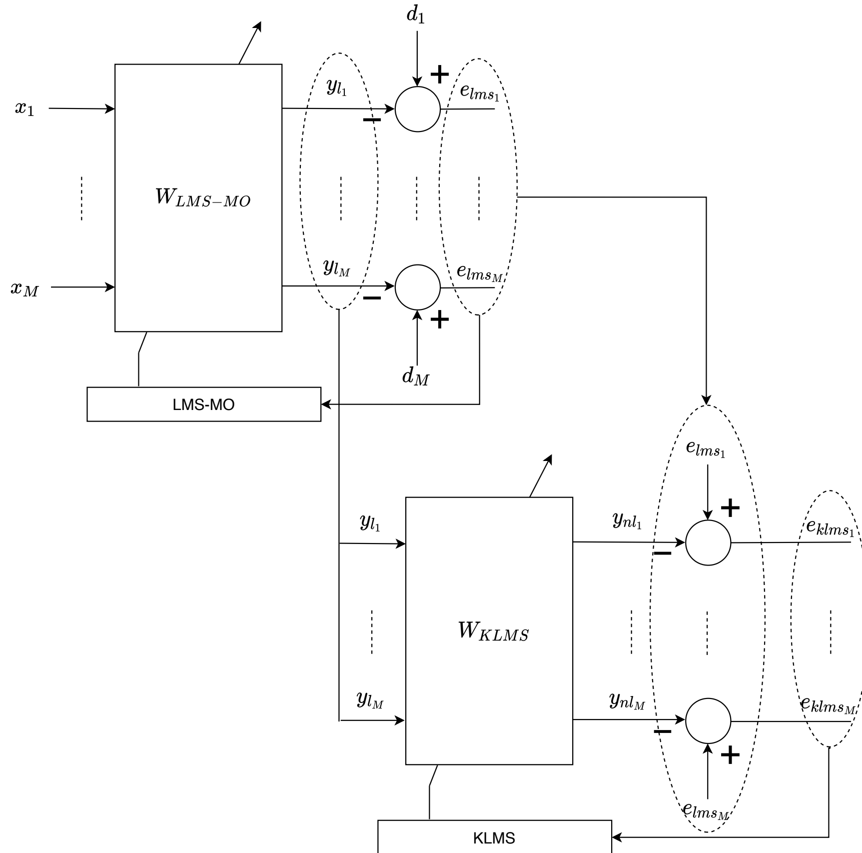
Una vez obtenidas las predicciones finales, es necesario entrenar la arquitectura. Para ello, en primer lugar se calcula el error cometido por el algoritmo LMS-MO como

$$\mathbf{e_{lms}} = \mathbf{d} - \mathbf{y_l} \tag{3.14}$$

Con este error $\mathbf{e_{lms}}$ se actualizan los pesos del filtro LMS-MO. A continuación, se calcula el error cometido por el algoritmo KLMS (para todos los pares entrada-salida) como

$$\mathbf{e_{klms}} = \mathbf{e_{lms}} - \mathbf{y_{nl}} \tag{3.15}$$

Con este error $\mathbf{e_{klms}}$ se actualizan los coeficientes α_i del algoritmo KLMS.



$$\begin{bmatrix} y_1 \\ \vdots \\ y_M \end{bmatrix} = \mathbf{y} = \mathbf{y_l} + \mathbf{y_{nl}} = \begin{bmatrix} y_{l_1} \\ \vdots \\ y_{l_M} \end{bmatrix} + \begin{bmatrix} y_{nl_1} \\ \vdots \\ y_{nl_M} \end{bmatrix}$$

Figura 3.3: Arquitectura de filtrado adaptativo LMS-MO + KLMS.

Capítulo 4

Análisis de resultados I: Base de datos artificial

En este capítulo, se analizarán las prestaciones de los distintos algoritmos de filtrado adaptativo *Multi-Output*, así como de la arquitectura desarrollada, aplicados a una base de datos artificial sobre la que se tiene control de ciertos parámetros.

En la sección 4.1 se explicará como se generan los datos de la base artificial creada. En la sección 4.2 se mostrará la convergencia del error cuadrático medio (MSE) a lo largo de todo el entrenamiento para los distintos algoritmos MO y de la arquitectura desarrollada. Además, se compararán sus resultados con los obtenidos por los algoritmos *Single-Output* aplicados a cada par entrada-salida. Una vez hecho esto, se seleccionarán aquellos algoritmos que presenten un menor MSE y se analizarán sus prestaciones con mayor profundidad, realizando para ello una comparativa del tiempo empleado en el entrenamiento, el número de FLOPS por iteración y el almacenamiento requerido por cada uno de ellos.

4.1. Generación de la base de datos artificial

Como primer paso para la evaluación de las prestaciones de los distintos algoritmos de filtrado adaptativo desarrollados, se ha generado una base de datos artificial sobre la que se tenga control de sus parámetros. En este caso, dichos parámetros controlables serán, tal y como se explicará un poco más adelante, el coeficiente de la no linealidad cuadrática γ y el coeficiente de correlación entre las series temporales de entrada ρ.

Las series temporales de la base de datos artificiales se generan de acuerdo al esquema que se muestra en la Fig. 4.1. En primer lugar, se generan M series temporales de N muestras temporales cada una, las cuales siguen una distribución normal estándar ($x_i \sim \mathcal{N}(0,1)\ ;\ i = 1, \ldots, M$). A continuación, se introduce una cierta correlación temporal entre las muestras de cada serie y las series se mezclan entre sí de tal forma que su matriz de covarianza sea

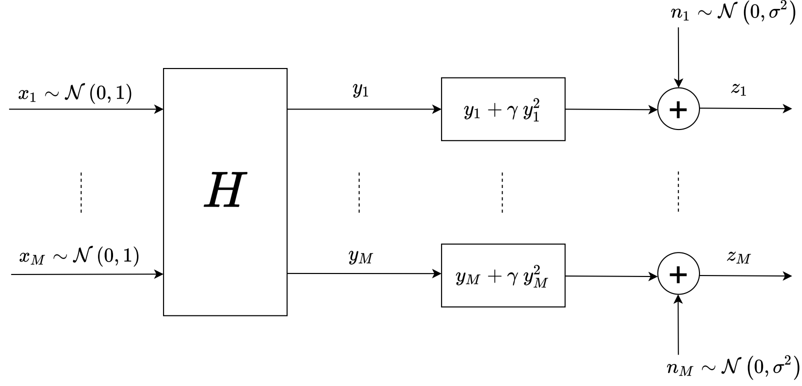


Figura 4.1: Generación de la base de datos artificial.

Fuente: Elaboración propia

$$\mathbf{C}_{[M\times M]} = \begin{bmatrix} 1 & \rho & \cdots & \rho \\ \rho & 1 & \cdots & \rho \\ \vdots & \vdots & \ddots & \vdots \\ \rho & \rho & \cdots & 1 \end{bmatrix} \tag{4.1}$$

siendo ρ el coeficiente de correlación entre dos series temporales distintas cualesquiera.

Para conseguir esto, se realiza el producto de Kronecker entre el filtro $\mathbf{h}$ que introduce la correlación temporal entre las muestras y la matriz de covarianza $\mathbf{C}$ deseada:

$$\mathbf{H} = \mathbf{C} \otimes \mathbf{h} \tag{4.2}$$

En este caso, al ser el filtro $\mathbf{h}$ de tamaño $1 \times L$ y la matriz de covarianza de tamaño $M \times M$, la matriz de mezcla $\mathbf{H}$ tiene unas dimensiones de $M \times (L \times M)$.

Una vez se dispone de la matriz de mezcla $\mathbf{H}$, esta se multiplica en cada instante temporal por el vector que contiene la muestra actual y las $L-1$ muestras anteriores de cada serie temporal concatenadas:

$$\begin{bmatrix} y_{1_n} \\ y_{2_n} \\ \vdots \\ y_{M_n} \end{bmatrix} = \mathbf{y_n} = \mathbf{H} \cdot \mathbf{x_n} = \mathbf{H} \cdot \begin{bmatrix} x_{1_n} \\ x_{1_{n-1}} \\ \vdots \\ x_{1_{n-L+1}} \\ \vdots \\ x_{M_n} \\ x_{M_{n-1}} \\ \vdots \\ x_{M_{n-L+1}} \end{bmatrix} \tag{4.3}$$

Por último, una vez generadas todas las muestras temporales de las series $\mathbf{y}$, estas se hacen pasar una a una por un sistema que las añade una distorsión cuadrática con coeficiente γ y un ruido blanco Gaussiano de varianza σ^2:

$$z_{i_n} = y_{i_n} + \gamma \cdot y_{i_n}^2 + n_{i_n} \quad ; \quad i = 1, \ldots, M \tag{4.4}$$

La implementación en MATLAB de la función encargada de generar los datos de esta base artificial se muestra en el Script A.4 del Anexo.

Una vez que se dispone de esta función de MATLAB, se generan los datos con los que posteriormente se analizarán las prestaciones de los algoritmos de filtrado adaptativo implementados. Para ello, se decide generar datos con distintas combinaciones de valores de γ y ρ. Los distintos escenarios considerados se recogen en la Tabla 4.1.

	γ	ρ
Escenario 1	0.15	0.3
Escenario 2	0.3	0.3
Escenario 3	0.15	0.6
Escenario 4	0.3	0.6

Tabla 4.1: Escenarios contemplados para la generación de la base de datos artificial.

Para los cuatro escenarios, se generan $M = 3$ series temporales, el número de muestras temporales de cada serie es $N = 4000$ muestras, el *time-embedding* es de $L = 2$ muestras, la varianza de ruido es $\sigma^2 = 0{,}01$ y el filtro utilizado para conseguir la correlación temporal entre las muestras es $\mathbf{h} = \begin{bmatrix} 1 & 0{,}6 \end{bmatrix}$.

Además, para poder obtener unas curvas de aprendizaje más suaves a la hora de probar los algoritmos, las series temporales de cada escenario se han generado aleatoriamente 1000 veces para poder promediar los resultados. A modo de ejemplo, en las Figs. 4.2 y 4.3 se muestran las series temporales de entrada y de salida de una de las simulaciones del Escenario 1.

4.2. Resultados

Una vez creada la base de datos artificial, se prueban sobre ella los distintos algoritmos desarrollados previamente y se analizan sus prestaciones. Para ello, en primer lugar se determinan los hiperparámetros óptimos de los distintos algoritmos. Para obtener la anchura del kernel σ_k a utilizar en los algoritmos de filtrado adaptativo no lineal se hace uso de la herramienta `kafbox_parameter_estimation` de la *toolbox* KAFBOX, basada en la *toolbox* GPML de Rasmussen y Williams [32]. Para realizar esta optimización se utilizan las 100 primeras muestras de una de las series temporales generadas para cada escenario. Cabe destacar también que el kernel usado en todos estos experimentos es de tipo gaussiano. El valor óptimo de σ_k obtenido por el procedimiento de optimización para cada escenario se muestra en la Tabla 4.2.

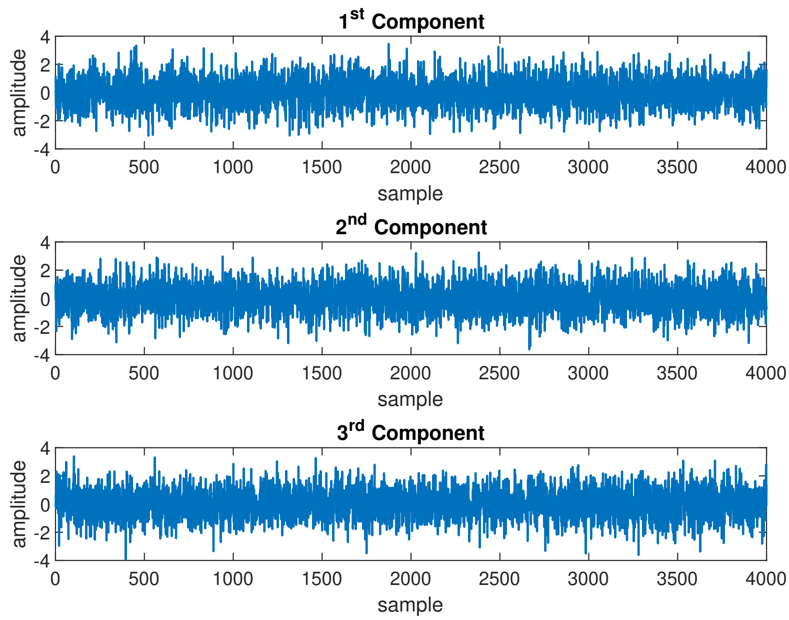


Figura 4.2: Series temporales de entrada en una simulación del Escenario 1.

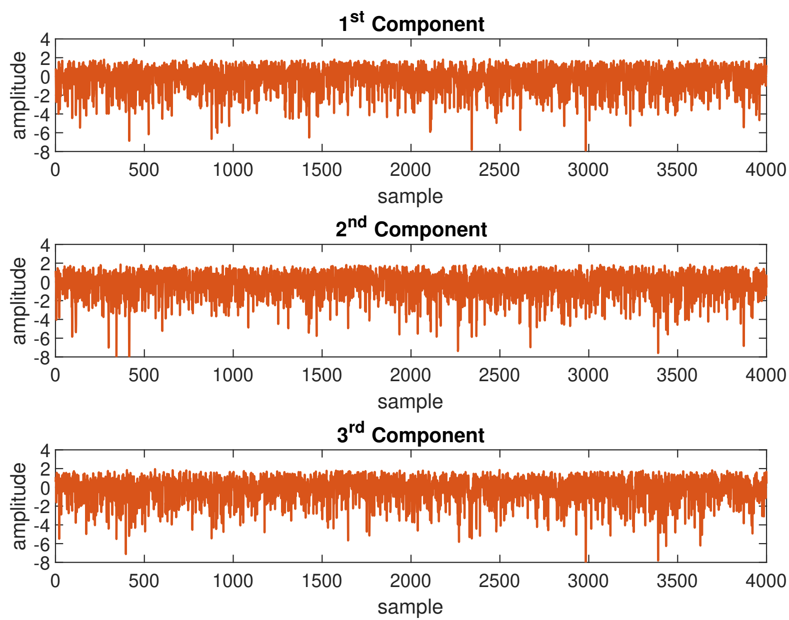


Figura 4.3: Series temporales de salida en una simulación del Escenario 1.

	$\boldsymbol{\sigma_{k_{opt}}}$
Escenario 1	3.17
Escenario 2	2.23
Escenario 3	2.20
Escenario 4	1.67

Tabla 4.2: Valores óptimos de la anchura del kernel σ_k en los distintos escenarios.

Además del valor óptimo de la anchura del kernel σ_k, los filtros adaptativos utilizados tienen otros parámetros relacionados con el tamaño del diccionario m o la tasa de aprendizaje cuyos valores no pueden ser elegidos de manera automática. Los valores seleccionados para estos parámetros (iguales en todos los escenarios) se muestran en la Tabla 4.3. Cabe destacar que la arquitectura de filtrado adaptativo diseñada utiliza los mismos parámetros en sus algoritmos LMS-MO y KLMS que los recogidos en dicha tabla.

Algoritmo	**Parámetros**
LMS // LMS-MO	$\mu = 0{,}02$
KLMS // KLMS-MO	$\eta = 0{,}4$, $m = 1000$
QKLMS // QKLMS-MO	$\eta = 0{,}4$, $\epsilon_u = 0{,}1//1{,}22$
RFF-KLMS // RFF-KLMS-MO	$\mu = 0{,}4$, $D = 1000$

Tabla 4.3: Valores del resto de hiperparámetros de los distintos algoritmos de filtrado adaptativo aplicados sobre la base de datos artificial.

Cabe destacar que las distintas tasas de aprendizaje (μ o η) se han seleccionado de tal forma que se minimice el MSE y el resto de parámetros relacionados con el tamaño del diccionario se han ajustado de tal forma que el diccionario tenga un tamaño final de aproximadamente 1000 datos.

4.2.1. MSE

Una vez seleccionados los valores de todos los hiperparámetros, se entrenan para los cuatro escenarios todos los algoritmos MO desarrollados, así como sus versiones *Single-Output* aplicadas a cada par entrada-salida. También se entrena la arquitectura de filtrado adaptativo diseñada previamente. Para analizar las prestaciones de los algoritmos, se sacan las curvas que muestran la evolución del error cuadrático medio promediado sobre las $M = 3$ series temporales a lo largo de todo el entrenamiento. Este MSE se puede expresar en cada instante temporal n como

$$MSE_{mean_n} = \frac{1}{M} \sum_{i=1}^{M} (d_{i_n} - y_{i_n})^2 \tag{4.5}$$

Las curvas de aprendizaje de cada algoritmo para cada uno de los escenarios considerados se muestran en las Figs. 4.4, 4.5, 4.6 y 4.7. Como se puede ver, los algoritmos MO desarrollados obtienen valores de MSE mucho más bajos que sus correspondientes versiones $Single-Output$ aplicadas a cada par entrada-salida, llegando a ser en algunos casos casi un orden de magnitud inferior.

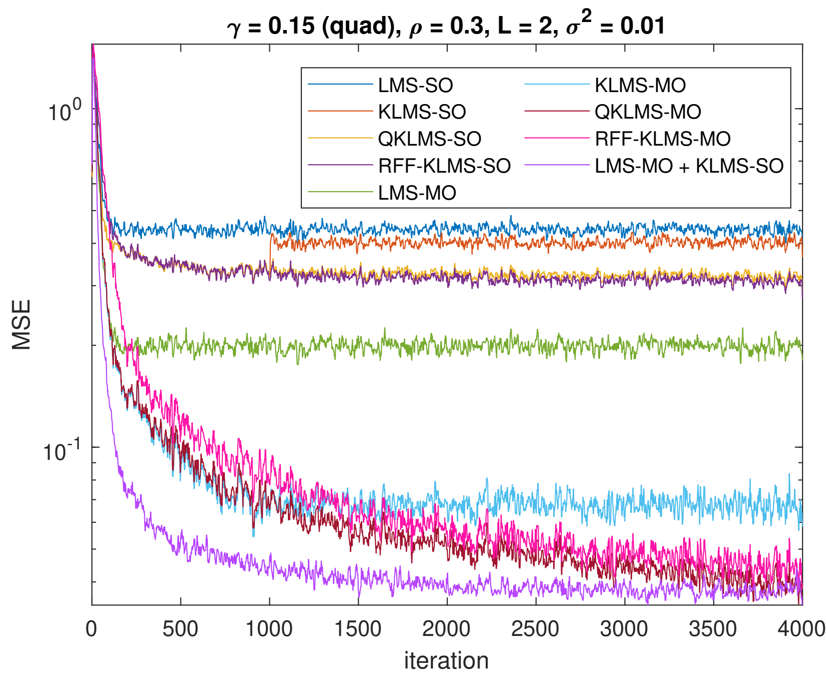


Figura 4.4: Curvas de aprendizaje para el Escenario 1.

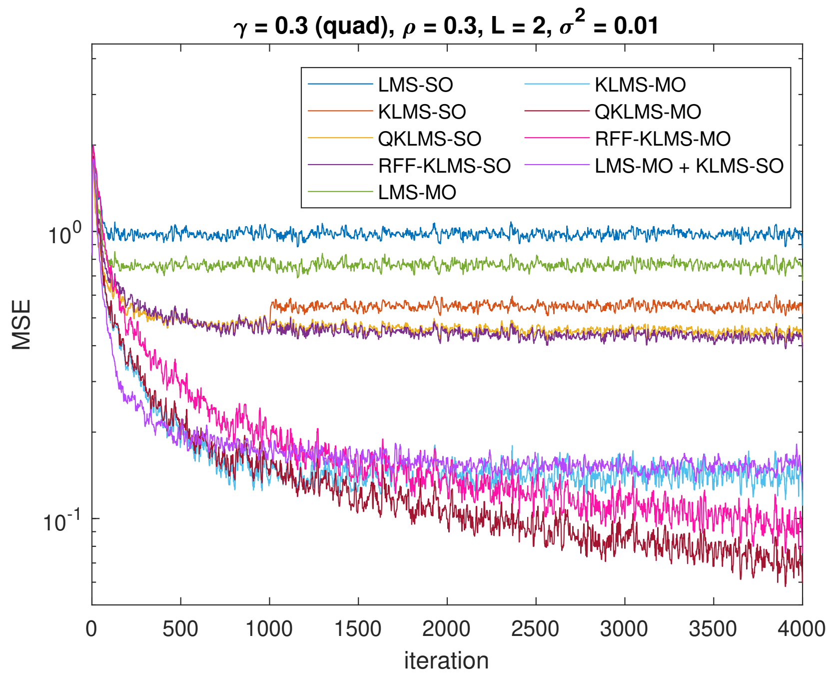


Figura 4.5: Curvas de aprendizaje para el Escenario 2.

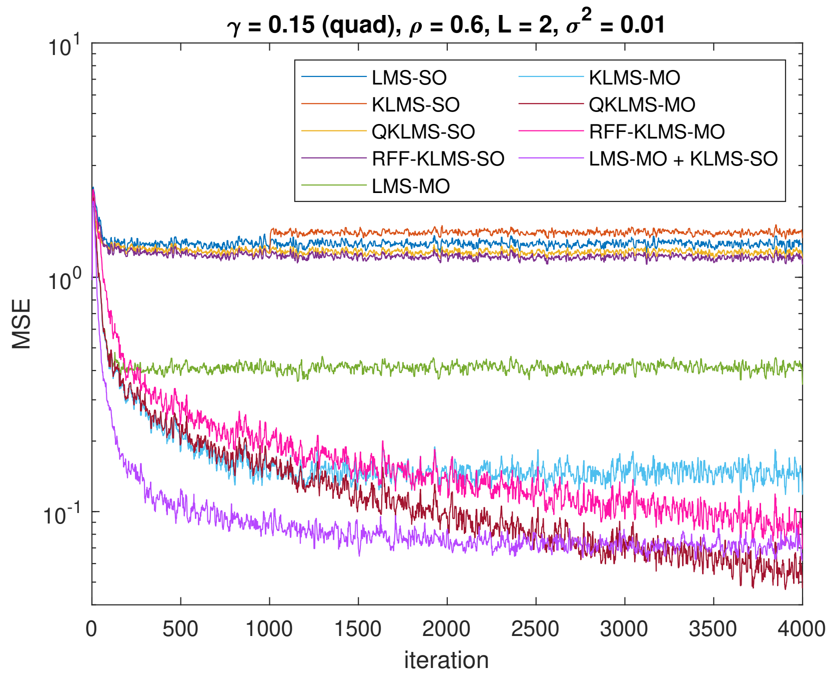


Figura 4.6: Curvas de aprendizaje para el Escenario 3.

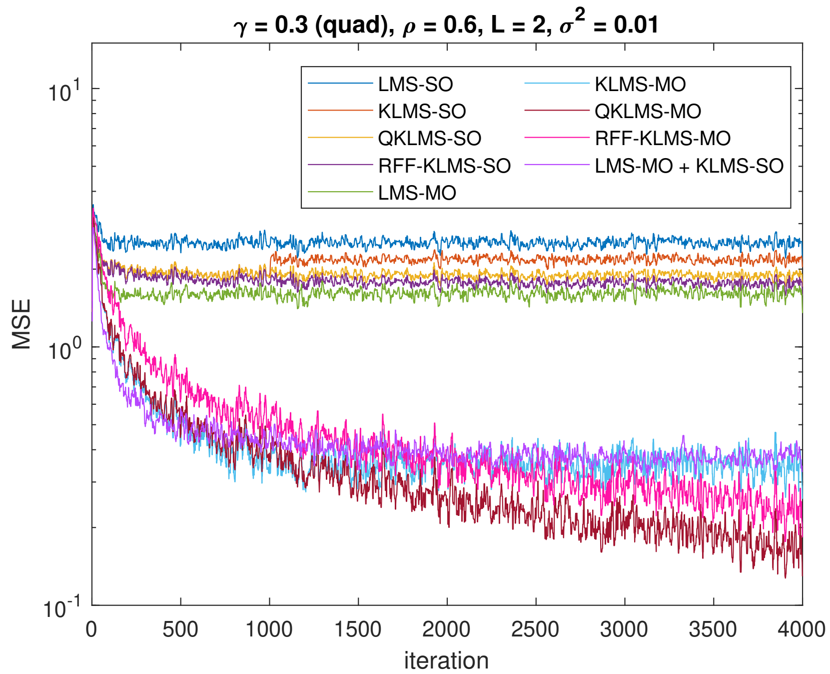


Figura 4.7: Curvas de aprendizaje para el Escenario 4.

Esta mejora se hace aún más evidente en los Escenarios 2 y 4, donde la correlación existente entre las series temporales es mayor.

También se puede observar que, al incrementar el coeficiente de la no linealidad γ, los algoritmos de filtrado adaptativo lineal empeoran enormemente sus prestaciones, tal y como cabía esperar. Dentro de los algoritmos de filtrado adaptativo no lineal, los que menor MSE obtienen son los algoritmos QKLMS-MO y RFF-KLMS-MO. Además, y a diferencia del resto de algoritmos, su curva de MSE no se estabiliza en un determinado valor y continúa bajando a lo largo de todo el entrenamiento, por lo que si se continuase entrenando con más muestras seguiría bajando aún más. Para comprobar esto, se generaron 50000 muestras adicionales de cada serie temporal en el Escenario 1 y se aplicaron nuevamente estos algoritmos. Las nuevas curvas de aprendizaje se muestran en la Fig. 4.8, donde se aprecia que el MSE continúa descendiendo hasta la iteración 20000 para el QKLMS-MO y 40000 para el RFF-KLMS-MO.

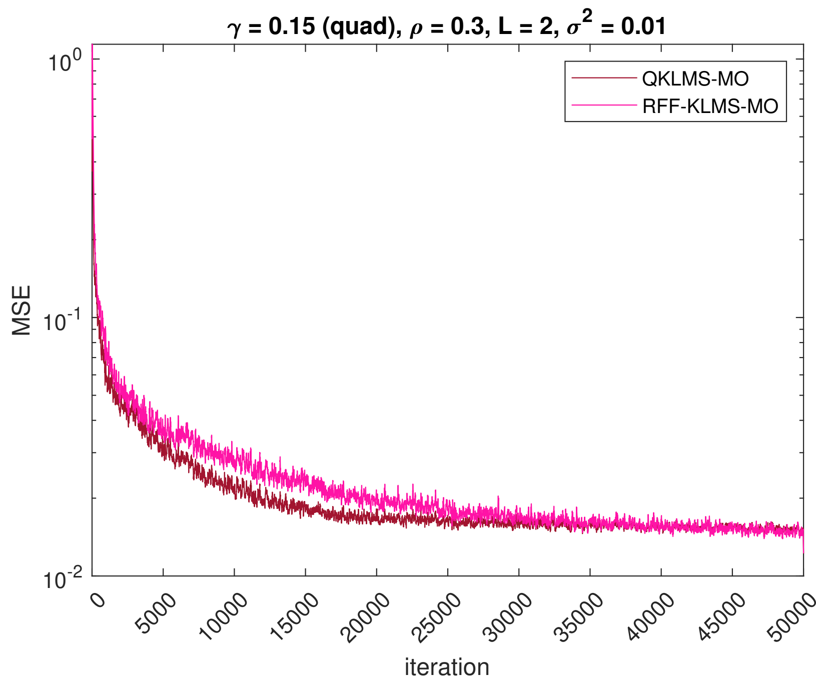


Figura 4.8: Curvas de aprendizaje de los algoritmos QKLMS-MO y RFF-KLMS-MO sobre series temporales de 50000 muestras para el Escenario 1.

Por último, se aprecia que la arquitectura desarrollada ofrece unas prestaciones similares a las de los algoritmos QKLMS-MO y RFF-KLMS-MO en los escenarios donde el coeficiente de la no linealidad γ es bajo y algo peor que estos en el resto de casos.

Por lo tanto, tras todos estos experimentos se puede determinar que los mejores algoritmos en términos de MSE son el QKLMS-MO y el RFF-KLMS-MO. En la próxima sección, se analizarán más en detalle sus prestaciones.

4.2.2. QKLMS-MO y RFF-KLMS-MO

Como se ha comprobado en el apartado anterior, los mejores algoritmos en términos de MSE son el QKLMS-MO y RFF-KLMS-MO. Por ello, se decide realizar un análisis más exhaustivo para estos dos algoritmos, observando la influencia del tamaño del diccionario m (QKLMS-MO) o del parámetro D (RFF-KLMS-MO) en las curvas de MSE, midiendo los tiempos empleados en el entrenamiento, el número de FLOPS por iteración y el almacenamiento requerido por cada uno de ellos. Para todas estas simulaciones, se han utilizado los datos del Escenario 1.

4.2.2.1. Influencia de m y D sobre el MSE

En primer lugar, se analiza la influencia de los parámetros m y D en las curvas de MSE de los algoritmos QKLMS-MO y RFF-KLMS-MO. Para ello, se varian sus valores de acuerdo a la Tabla 4.4 y se van obteniendo las distintas curvas de MSE.

Algoritmo	Parámetros fijos	Parámetros variables
QKLMS-MO	$\eta = 0{,}4$ $\sigma_k = 3{,}17$	$\epsilon_u \in \{2{,}39, 1{,}89, 1{,}54, 1{,}03, 0{,}10\}$
RFF-KLMS-MO	$\mu = 0{,}4$	$D \in \{100, 250, 500, 1500, 4000\}$

Tabla 4.4: Valores de los hiperparámetros de los algoritmos QKLMS-MO y RFF-KLMS-MO utilizados para analizar la influencia de m y D sobre el MSE.

Cabe destacar que los valores del parámetro ϵ_u se han elegido de tal manera que el tamaño final del diccionario sea $m \in \{100, 500, 1000, 2000, 4000\}$ y poder así comparar ambos algoritmos entre sí.

Las distintas curvas obtenidas para ambos algoritmos se muestran en las Figs. 4.9 y 4.10 y han sido obtenidas promediando 1000 simulaciones.

Como se puede ver, a medida que ϵ_u decrece (es decir, m aumenta) o que D aumenta, el MSE es menor. Esta diferencia del valor del MSE llega a ser de casi un 35 % entre el menor y el mayor valor de los parámetros analizados. Por otra parte, también se observa que la diferencia existente entre las curvas morada y verde de cada algoritmo es casi inexistente, por lo que no aporta casi nada aumentar el tamaño del diccionario (m) o la dimensión del espacio de características (D) más allá de las 1500 muestras.

Por último, también se puede ver que ambos algoritmos tienen unas prestaciones similares en términos de MSE, siendo mínimamente mejor en este aspecto el algoritmo QKLMS-MO.

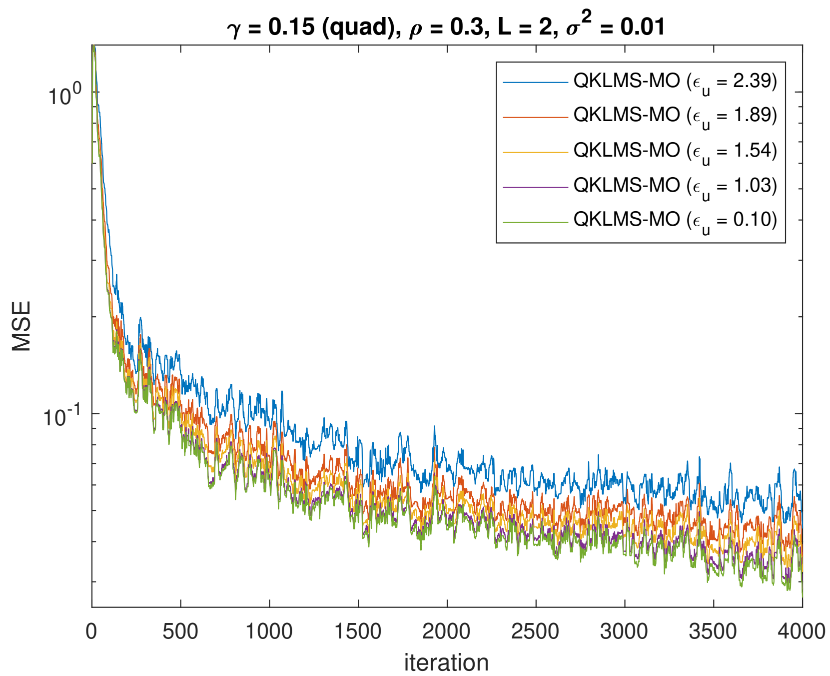


Figura 4.9: Curvas de aprendizaje del algoritmo QKLMS-MO para distintos valores del parámetro ϵ_u.

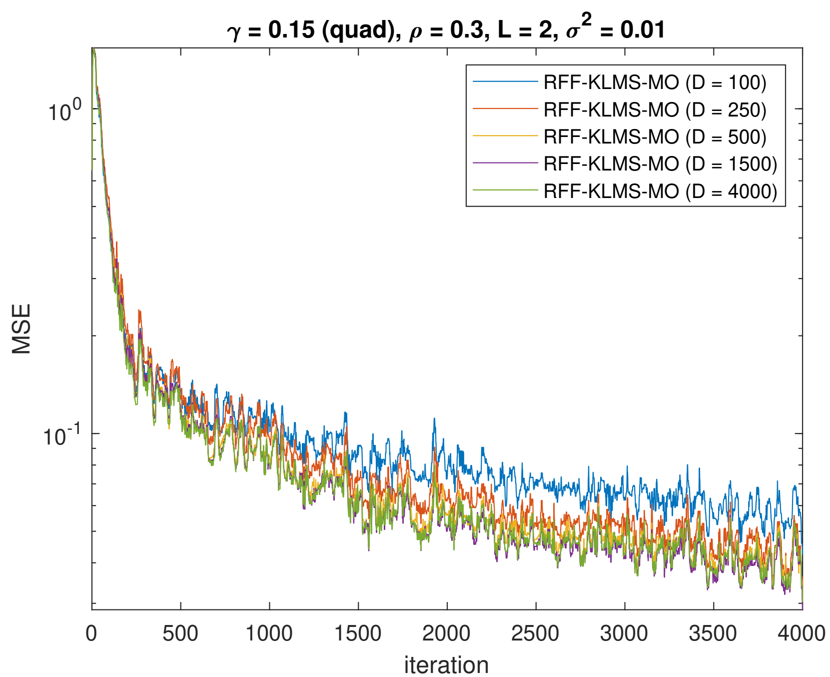


Figura 4.10: Curvas de aprendizaje del algoritmo RFF-KLMS-MO para distintos valores del parámetro D.

4.2.2.2. Tiempo de entrenamiento

Una vez obtenidas las curvas de MSE, se lleva a cabo un análisis del tiempo de entrenamiento de ambos algoritmos para los mismos valores de los parámetros ϵ_u y D considerados anteriormente (Tabla 4.4).

Los tiempos obtenidos para cada algoritmo y cada valor de los parámetros ϵ_u (o m equivalente) y D se muestran en la Tabla 4.5 y en la Fig. 4.11. Al igual que en las anteriores simulaciones, se ha entrenado cada algoritmo 1000 veces, se ha medido el tiempo que emplea cada una de las veces y se ha obtenido su media. Las simulaciones se han realizado en un ordenador ASUS con un microprocesador Intel Core i7-7500U trabajando a una frecuencia de 2.7 GHz.

ϵ_u // D	$t_{QKLMS-MO}$ (ms)	$t_{RFF-KLMS-MO}$ (ms)
2.39 // 100	90.6	32.5
1.89 // 250	102.1	49.4
1.54 // 500	139.6	66.5
1.03 // 1500	201.9	145.4
0.10 // 4000	392.7	383.3

Tabla 4.5: Tiempo empleado en el entrenamiento de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros ϵ_u y D.

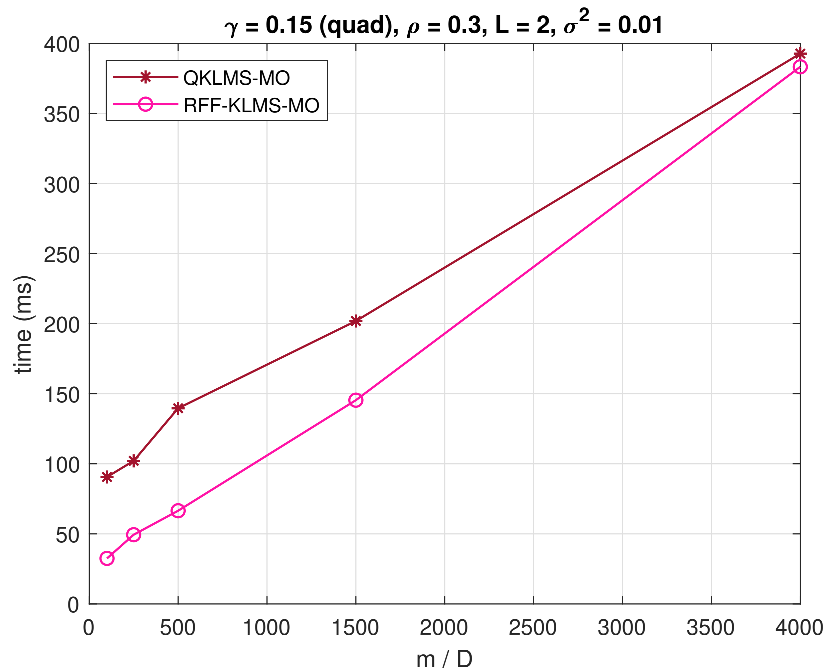


Figura 4.11: Tiempo empleado en el entrenamiento de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

Como se puede observar, el algoritmo RFF-KLMS-MO es significativamente más rápido a la hora de entrenar que el algoritmo QKLMS-MO, llegando a realizar el entrenamiento casi en la mitad de tiempo que este último para los valores más bajos de m o D. Por ello, resulta idóneo para aplicaciones de tiempo real que requieran altas velocidades de computación ya que, en caso contrario, podrían no llegar a procesarse ciertos datos.

4.2.2.3. FLOPS

A continuación, se van a comparar ambos algoritmos en términos de su complejidad computacional. Para ello, se va a calcular el número máximo de operaciones de punto flotante (FLOPS) realizadas por cada algoritmo a lo largo del entrenamiento. Para ello, se considera que se utiliza el procesador x86 de Intel, cuyos valores de FLOPS para cada operación son los que se muestran en la Tabla 4.6.

Operación	**FLOPS**
Suma	1
Multiplicación	1
División	8
Exponencial	20
Coseno	20

Tabla 4.6: Número de FLOPS por operación para el procesador x86 de Intel.

Para el caso del kernel Gaussiano, el microprocesador realiza $m \cdot (2 \cdot L \cdot M - 1)$ sumas, $m \cdot (L \cdot M + 1)$ y m exponenciales para resolverlo, siendo m el tamaño final del diccionario, L el *time-embedding* y M el número de series temporales de entrada o de salida.

El desglose de las operaciones realizadas en cada iteración por el algoritmo QKLMS-MO se muestra sobre el Script A.5.1 del Anexo. Para ello, se ha tomado el caso en el que se realizan más operaciones, que es cuando no se decide incluir un cierto dato en el diccionario y es necesario actualizar el coeficiente α_i del elemento más cercano a este. El desglose de las operaciones realizadas por el algoritmo RFF-KLMS-MO se muestra sobre el Script A.5.2 del Anexo.

Una vez que se conoce el número de operaciones de cada tipo, estas se multiplican por el número de operaciones de punto flotante que conlleva cada una de acuerdo a la Tabla 4.6 para obtener el número máximo de FLOPS por iteración para cada algoritmo. Los resultados obtenidos se muestran en la Tabla 4.7 y en la Fig. 4.12.

m // **D**	$\mathbf{FLOPS_{QKLMS-MO}}$	$\mathbf{FLOPS_{RFF-KLMS-MO}}$
100	6106	4425
250	15256	11025
500	30506	22025
1500	91506	66025
4000	244006	176025

Tabla 4.7: FLOPS por iteración de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

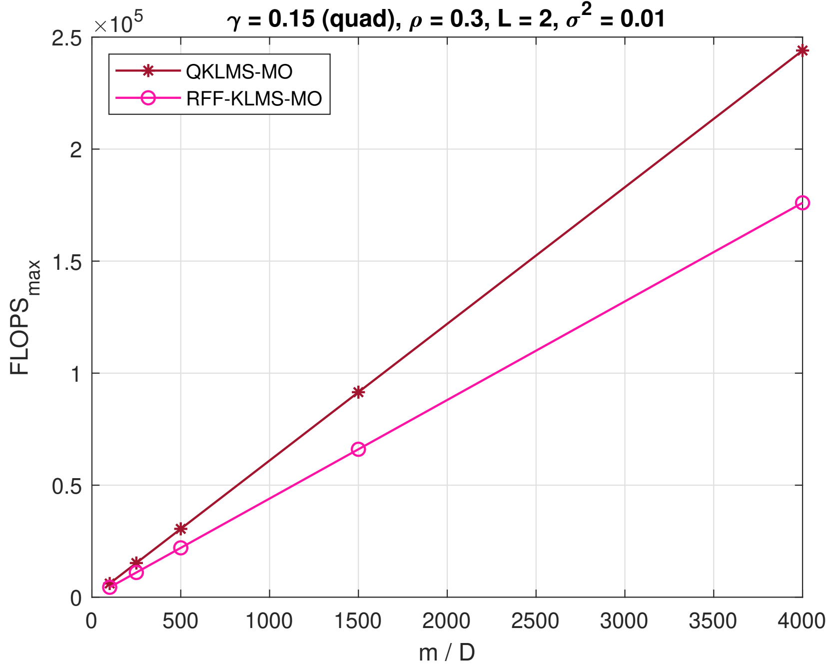


Figura 4.12: FLOPS por iteración de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

Como se puede comprobar, el número de FLOPS por iteración es bastante menor en el algoritmo RFF-KLMS-MO, siendo más significativa la diferencia entre ambos a medida que m o D crece.

Por último, se decide representar los FLOPS de cada algoritmo respecto al MSE obtenido al final del entrenamiento, lo que da una idea de la capacidad computacional requerida para obtener un determinado valor de MSE. Si, por ejemplo, se trabaja en escenarios con restricciones en la complejidad computacional, se deberá seleccionar aquel algoritmo que mejor opere bajo esta restricción determinando qué curva está más a la izquierda para el número de FLOPS disponibles. De la misma manera, si se fija un MSE máximo permitido, se pueden obtener los FLOPS requeridos por cada algoritmo. Estas curvas se presentan en la Fig. 4.13.

Como se puede ver, si la complejidad computacional disponible es bastante limitada, resulta más interesante utilizar el algoritmo RFF-KLMS-MO. En cambio, en el resto de situaciones es preferible utilizar el algoritmo QKLMS-MO, ya que proporciona un mejor MSE para la misma cantidad de FLOPS.

4.2.2.4. Almacenamiento

Por último, se va a analizar el almacenamiento requerido por ambos algoritmos de filtrado adaptativo. En el caso del algoritmo QKLMS-MO, se requiere almacenar el diccionario y los coeficientes α_i. Por ello, teniendo en cuenta que cada dato ocupa 8 *bytes*, el almacenamiento requerido por el algoritmo QKLMS-MO es de

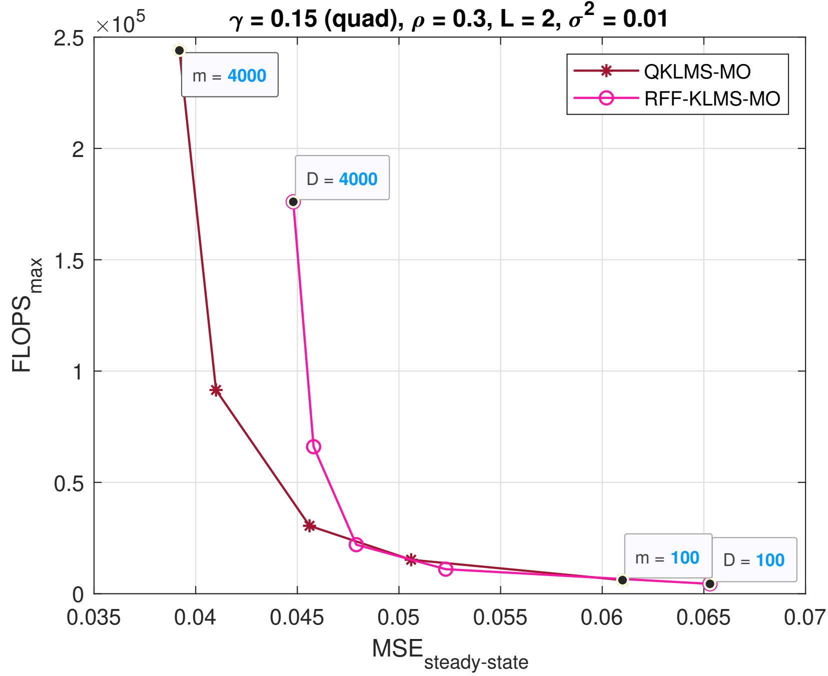


Figura 4.13: FLOPS por iteración frente al MSE de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

$$bytes_{QKLMS-MO} = 8 \cdot \left(\underbrace{m \cdot (L \cdot M)}_{kaf.dict} + \underbrace{m \cdot M}_{kaf.alpha} \right) \tag{4.6}$$

Por otro lado, en el algoritmo RFF-KLMS-MO se requiere almacenar las variables ω_i, b_i y $\boldsymbol{\Omega}$. Por ello, el almacenamiento requerido por el algoritmo RFF-KLMS-MO es de

$$bytes_{RFF-KLMS-MO} = 8 \cdot \left(\underbrace{D \cdot (L \cdot M)}_{kaf.omega} + \underbrace{D}_{kaf.b} + \underbrace{D \cdot M}_{kaf.Omega} \right) \tag{4.7}$$

Los resultados obtenidos para ambos algoritmos se muestran en la Tabla 4.8 y en la Fig. 4.14.

Como se puede apreciar, el almacenamiento requerido por el QKLMS-MO es ligeramente menor que el necesitado por el RFF-KLMS-MO, haciéndose algo más notable esta diferencia a medida que crecen m y D.

m // D	$\text{bytes}_{\text{QKLMS-MO}}$	$\text{bytes}_{\text{RFF-KLMS-MO}}$
100	7200	8000
250	18000	20000
500	36000	40000
1500	108000	120000
4000	288000	320000

Tabla 4.8: Almacenamiento máximo requerido por los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

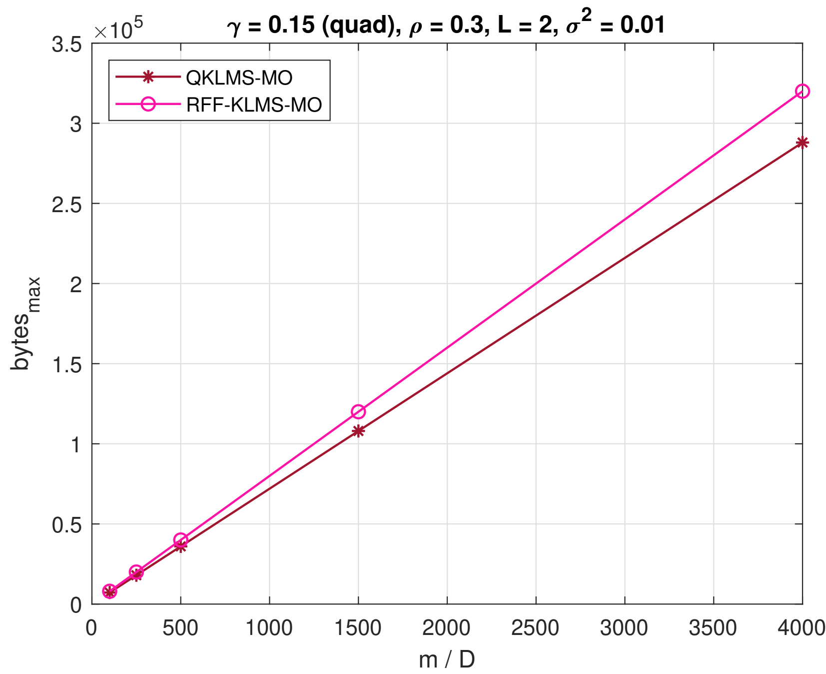


Figura 4.14: Almacenamiento máximo requerido por los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

Por último, se decide representar el almacenamiento requerido por cada algoritmo respecto al MSE obtenido al final del entrenamiento, lo que da una idea de la capacidad de almacenamiento requerida para obtener un determinado valor de MSE. Estas curvas se presentan en la Fig. 4.15.

Como se puede ver, en términos de la memoria utilizada parece que siempre resulta algo ventajoso usar el algoritmo QKLMS-MO.

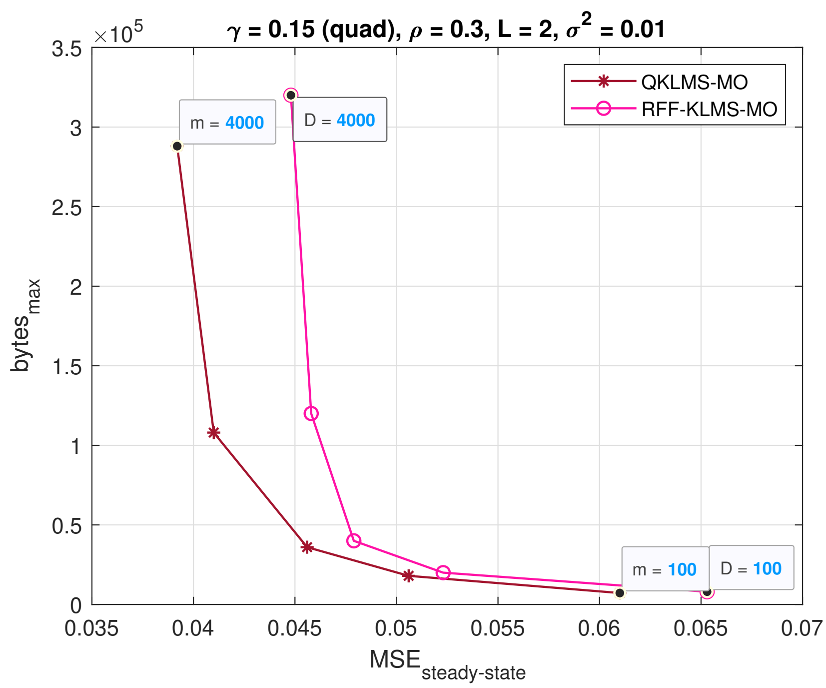


Figura 4.15: Almacenamiento máximo requerido frente al MSE de los algoritmos QKLMS-MO y RFF-KLMS-MO para distintos valores de los parámetros m y D.

Capítulo 5

Análisis de resultados II: Base de datos real

En este capítulo, se analizarán las prestaciones de los algoritmos de filtrado adaptativo QKLMS-MO y RFF-KLMS-MO cuando estos aplicados a la base de datos real *Occupancy Detection*.

En la sección 5.1 se explicará brevemente la estructura de la base de datos *Occupancy Detection* y las series temporales que se usarán en el proyecto. En la sección 5.2 se hará una comparativa de los dos algoritmos analizados, mostrando sus curvas de evolución del MSE a lo largo del entrenamiento, el tiempo empleado para ello, el número de FLOPS por iteración y el almacenamiento requerido por cada uno de ellos.

5.1. *Occupancy Detection Dataset*

Una vez evaluadas las prestaciones de los distintos algoritmos de filtrado adaptativo desarrollados sobre la base de datos artificial, se procede a analizar aquellos dos que obtuvieron un menor MSE sobre una base de datos real. Para ello, se decide utilizar la base de datos denominada *Occupancy Detection* [8], creada por Candanedo y Feldheim en 2016.

Esta base de datos fue generada para probar distintos modelos de clasificación como *Random Forest* (RF), *Gradient Boosting Machines* (GBM), *Linear Discriminant Analysis* (LDA) y *Classification and Regression Trees* (CART) y poder detectar con ellos si una oficina estaba ocupada o no a partir de las medidas realizadas de la luz, la temperatura, la humedad y el CO_2.

Las distintas medidas realizadas se llevaron a cabo en una oficina de dimensiones 5,85 $m \times$ 3,50 $m \times$ 3,53 m. Para adquirir los datos, se utilizó un microcontrolador al que se conectó un sistema radio ZigBee para transmitir la información a una estación de grabación. Además, se disponía de una cámara digital usada para determinar si la oficina estaba ocupada o no. Esta disposición puede verse en la Fig. 5.1.

Para este proyecto, se decide utilizar los valores medidos de la luz y el CO_2 para predecir los valores de temperatura y humedad en la oficina. Para ello, se hace uso del *dataset* denominado `datatraining`, que contiene medidas de las cuatro magnitudes mencionadas anteriormente realizadas durante casi 6 días.

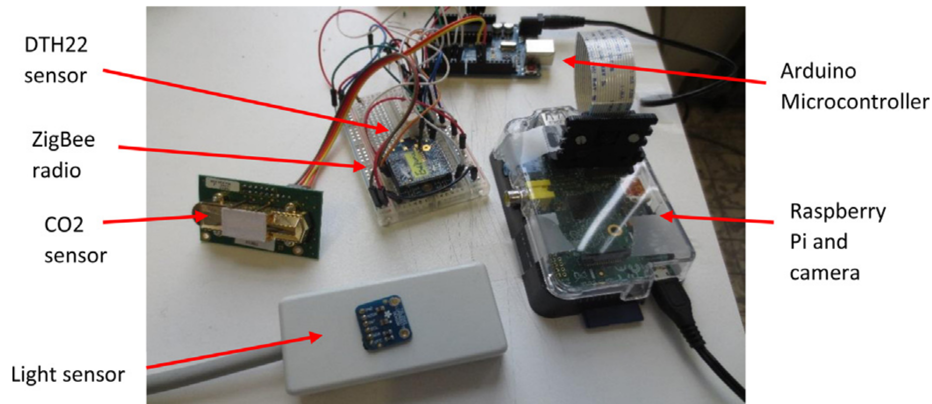


Figura 5.1: *Setup* empleado para la adquisición de los datos del *Occupancy Detection Dataset*. En él, se muestran los sensores de luz, CO_2 y DHT22 (temperatura y humedad), el sistema radio ZigBee, el microcontrolador y la cámara digital controlada por una Raspberry Pi.

Fuente: Accurate occupancy detection of an office room [8]

Estas medidas se realizaron cada minuto entre las 17:51 del 4 de febrero de 2015 y las 09:33 del 10 de febrero de 2015, por lo que este *dataset* consta de unas 8143 muestras temporales de cada serie. Estas series temporales se muestran en las Figs. 5.2 y 5.3.

En este caso, a las series temporales utilizadas se les ha aplicado un procesado previo, consistente en una normalización estándar, para obtener mejores resultados en los distintos experimentos. La fórmula empleada e esta normalización estándar es

$$x_{norm_n} = \frac{x_n - \mu}{\sigma} \tag{5.1}$$

donde x_n es la muestra n de una serie temporal de la entrada del sistema, μ es la media de todas las muestras de la serie temporal y σ su desviación estándar.

Para llevar a cabo el análisis de las prestaciones del QKLMS-MO y del RFF-KLMS-MO, se dispone por tanto de $M = 2$ series temporales de entrada y otras tantas de salida. Cada una de ellas dispone de 8143 muestras, de las cuales $N_{hyper} = 750$ muestras se utilizarán para estimar la anchura del kernel σ_k óptima y las restantes $N = 7393$ se utilizarán para entrenar los algoritmos. Por último, el *time-embedding* utilizado es de $L = 2$.

5.2. Resultados

Una vez explicada la estructura de la base de datos real, se prueban sobre ella los algoritmos QKLMS-MO y RFF-KLMS-MO y se analizan sus prestaciones.

Para ello, en primer lugar se determinan los hiperparámetros óptimos de ambos algoritmos. Para obtener la anchura del kernel a utilizar en estos dos algoritmos se hace uso nuevamente de la herramienta `kafbox_parameter_estimation`

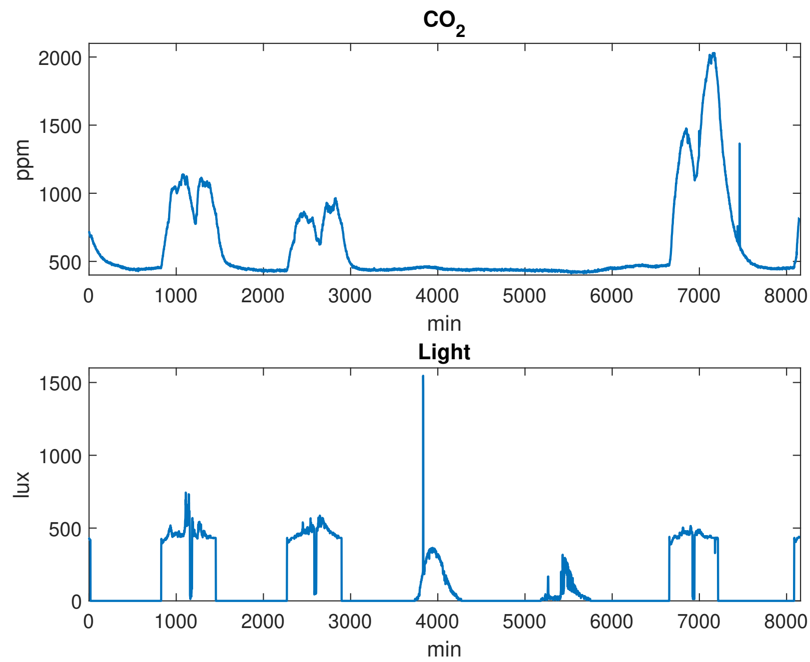


Figura 5.2: Valores de CO_2 y luz (entradas del sistema).

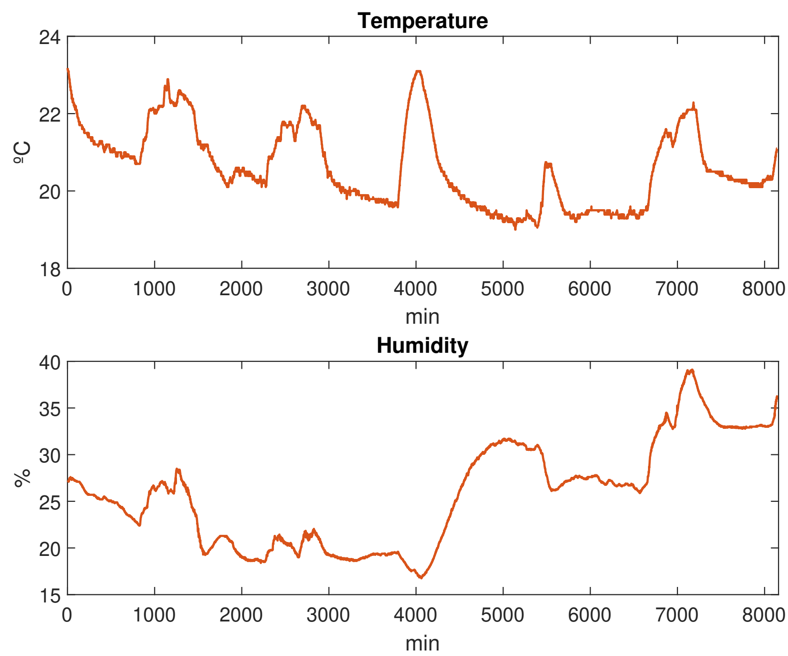


Figura 5.3: Valores de temperatura y humedad (salidas del sistema).

de la *toolbox* KAFBOX. Para llevar a cabo esta optimización del parámetro σ_k se utilizan las 750 primeras muestras de las series de entrada al sistema, tal y como se ha comentado al final de la anterior sección. Al igual que en los experimentos realizados sobre la base de datos artificial, el kernel usado en este caso también es de tipo Gaussiano. Tras realizar el procedimiento de optimización, el valor óptimo de la anchura del kernel obtenido es $\sigma_k = 14{,}75$.

El resto de parámetros de los filtros adaptativos utilizados se seleccionan manualmente. Los valores escogidos para estos parámetros se muestran en la Tabla 5.1. El parámetro ϵ_u se ha seleccionado de tal manera que el tamaño final del diccionario sea $m = 1000$.

Algoritmo	**Parámetros**
QKLMS-MO	$\eta = 0{,}8$, $\epsilon_u = 5{,}8 \times 10^{-2}$
RFF-KLMS-MO	$\mu = 0{,}8$, $D = 1000$

Tabla 5.1: Valores del resto de hiperparámetros de los algoritmos QKLMS-MO y RFF-KLMS-MO usados sobre la base de datos real.

5.2.1. MSE

Una vez escogidos los valores de todos los hiperparámetros, se lleva a cabo el entrenamiento de los dos algoritmos. Las predicciones realizadas por ambos algoritmos, así como las series temporales originales, pueden verse en las Figs. 5.4 y 5.5.

Para analizar las prestaciones de ambos algoritmos, se sacan las curvas que muestran la evolución del MSE promediado sobre las $M = 2$ series temporales a lo largo de todo el entrenamiento. Estas se muestran en la Fig. 5.6.

Como se puede observar, el MSE obtenido por ambos algoritmos es siempre inferior a 10^{-3}, siendo algo inferior en el caso del algoritmo QKLMS-MO, tal y como cabía esperar. Concretamente, el algoritmo QKLMS-MO obtiene un $MSE_{medio} = 6{,}17 \times 10^{-4}$ y el algoritmo RFF-KLMS-MO un $MSE_{medio} = 7{,}40 \times 10^{-4}$.

5.2.2. Tiempo de entrenamiento

Una vez obtenido el MSE, se analiza el tiempo requerido por cada algoritmo para completar el entrenamiento. Al igual que cuando se llevo a cabo este análisis sobre la base de datos artificial, los algoritmos se corren en un ordenador ASUS con microprocesador Intel Core i7-7500U trabajando a una frecuencia de 2.7 GHz. En este caso, el algoritmo QKLMS-MO requiere un tiempo $t_{QKLMS-MO} = 309{,}4\ ms$ y el algoritmo RFF-KLMS-MO un tiempo $t_{RFF-KLMS-MO} = 180{,}7\ ms$. Como se puede apreciar, estos resultados concuerdan con los obtenidos para la base de datos artificial.

5.2.3. FLOPS

A continuación, se analiza el número de operaciones de punto flotante por iteración que realiza cada algoritmo. Aplicando los valores de la Tabla 4.6 y el desglose de operaciones de cada algoritmo realizado en los Scripts 3.2 y 3.3 se

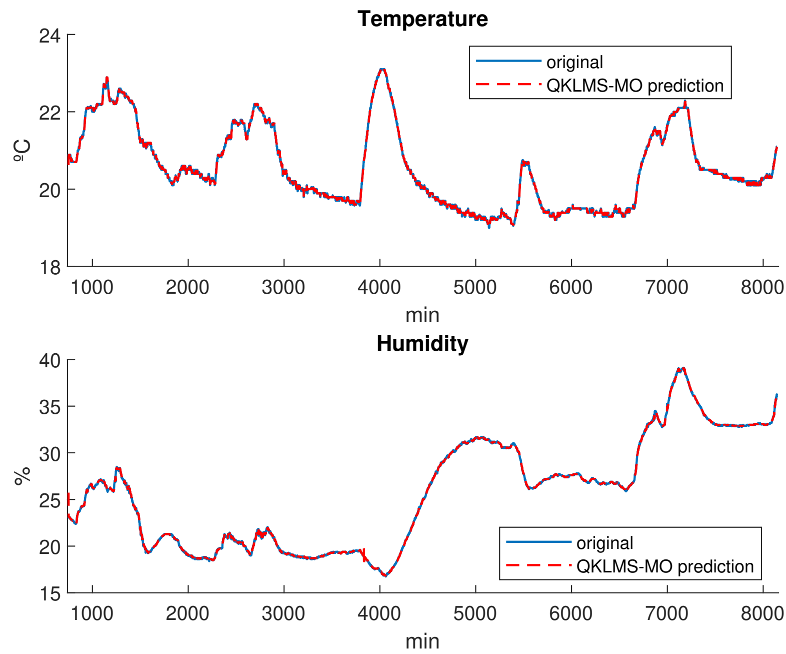


Figura 5.4: Predicciones realizadas por el algoritmo QKLMS-MO sobre la base de datos real.

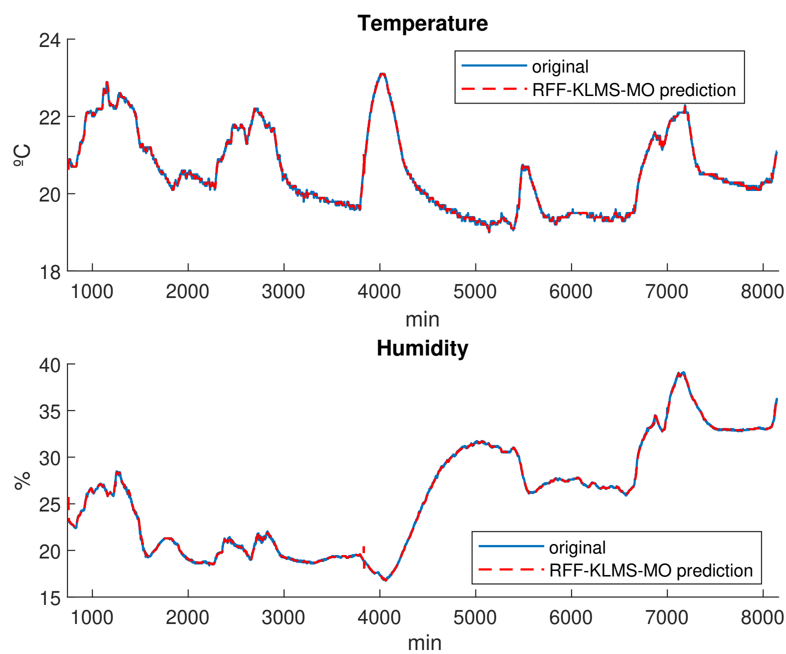


Figura 5.5: Predicciones realizadas por el algoritmo RFF-KLMS-MO sobre la base de datos real.

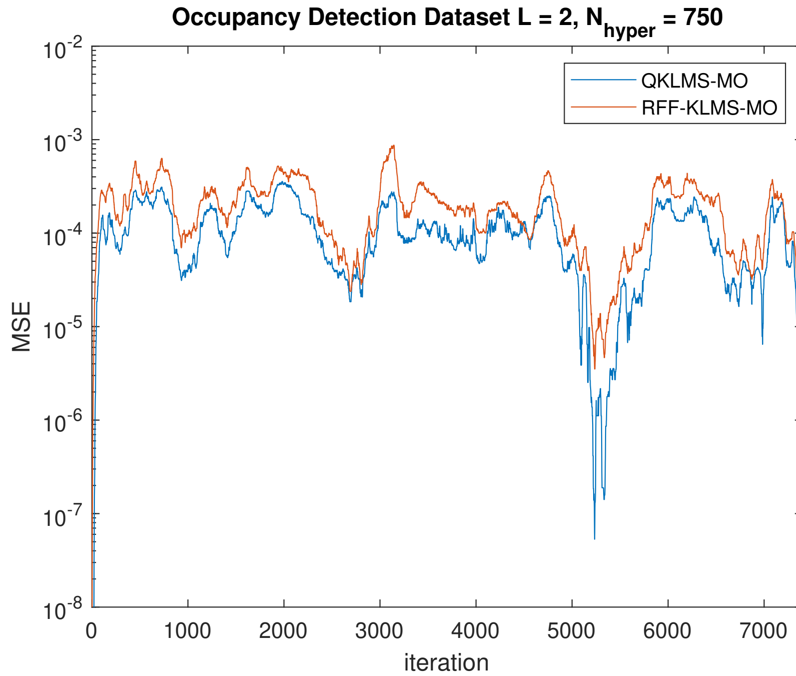


Figura 5.6: Curvas de aprendizaje de los algoritmos QKLMS-MO y RFF-KLMS-MO sobre la base de datos real.

obtiene que el QKLMS-MO realiza $FLOPS_{QKLMS-MO} = 47004$ y el RFF-KLMS-MO $FLOPS_{RFF-KLMS-MO} = 36017$.

5.2.4. Almacenamiento requerido

Por último, se analiza el almacenamiento máximo requerido por cada algoritmo. Utilizando las Ecs. 4.6 y 4.7, se obtiene que el algoritmo QKLMS-MO requiere un espacio de almacenamiento de $bytes_{QKLMS-MO} = 48000$ y el algoritmo RFF-KLMS-MO $bytes_{RFF-KLMS-MO} = 56000$.

Capítulo 6

Conclusiones

Tras la realización de todos los experimentos con los distintos algoritmos de filtrado adaptativo propuestos sobre las dos bases de datos utilizadas (artificial y real), se han extraído una serie de conclusiones que se presentan a continuación:

- En primer lugar, los algoritmos MO desarrollados ofrecen unas muy buenas prestaciones en términos de MSE en comparación con sus versiones *Single-Output*, llegando a ser esta diferencia en algunos casos de casi un orden de magnitud. Además, esta mejora es aún más evidente cuando la no linealidad de las series temporales a predecir es mayor, por lo que estos algoritmos resultan de gran utilidad cuando se quieren llevar a cabo tareas de predicción en escenarios que introducen mucha distorsión no lineal.

- En segundo lugar, los algoritmos QKLMS-MO y RFF-KLMS-MO, a pesar de tener una estructura e implementación muy sencillas, proporcionan mejores prestaciones que la arquitectura de filtrado adaptativo diseñada. De hecho, la arquitectura solo mejora mínimamente el MSE obtenido por estos otros algoritmos en el Escenario 1 (véase la sección 4.2) y su coste computacional es mucho mayor, al tener que correr dos algoritmos de filtrado adaptativo en lugar de uno solo.

- En último lugar, se ha comprobado que el algoritmo RFF-KLMS-MO es un algoritmo de filtrado adaptativo *Multi-Output* especialmente interesante en el ámbito de las aplicaciones de tiempo real y/o en situaciones en las que la capacidad de cómputo disponible sea limitada, debido a sus pequeños tiempos de entrenamiento y su baja complejidad computacional en comparación con el QKLMS-MO, tal y como se ha comprobado en las secciones 4.2.2 y 5.2. Además, en términos de MSE obtiene unos resultados muy parecidos al QKLMS-MO y únicamente requiere de un poco más de espacio de almacenamiento.

Apéndice A

Códigos MATLAB

A.1. Algoritmos SO

A.1.1. LMS

```
1 y = x' * w; % evaluate filter output
2 err = d - y; % instantaneous error
3 w = w + mu * x * err'; % update filter coefficients
```

Script A.1: Iteración de entrenamiento del algoritmo LMS para un nuevo par de datos.

A.1.2. KLMS

```
1 k = kernel(kaf.dict,x,kerneltype,kernelpar); % kernels
      between dictionary and x
2 y = k' * kaf.alpha; % evaluate function output
3 err = d - y; % instantaneous error
4
5 kaf.dict = [kaf.dict; x]; % add base to dictionary
6 kaf.alpha = [kaf.alpha; kaf.eta*err]; % add new
    coefficient
```

Script A.2: Iteración de entrenamiento del algoritmo KLMS para un nuevo par de datos.

A.1.3. QKLMS

```
k = kernel(kaf.dict,x,kerneltype,kernelpar); % kernels
     between dictionary and x
y = k' * kaf.alpha; % evaluate function output
err = d - y; % instantaneous error
m = size(kaf.dict,1);
if m==0
    d2 = kaf.epsu^2 + 1; % force addition of initial
        base
else
    [d2,j] = min(sum((kaf.dict - repmat(x,m,1)).^2,2))
        ; % find distance to closest dictionary element
end

if d2 <= epsu^2
    alpha(j) = alpha(j) + kaf.eta*err; % reduced
        coefficient update
else
    kaf.dict = [kaf.dict; x]; % add base to dictionary
    kaf.alpha = [kaf.alpha; kaf.eta*err]; % add new
        coefficient
end
```

Script A.3: Iteración de entrenamiento del algoritmo QKLMS para un nuevo par de datos.

A.1.4. RFF-KLMS

```
if ~numel(kaf.omega) % initialize
    rng('default');
    rng(kaf.seed);
    kaf.omega = 1/kaf.kernelpar*randn(kaf.D,size(x,2))
        ;
    kaf.b = 2*pi*rand(kaf.D,1);
    kaf.Omega = zeros(kaf.D,1);
end

Psi = cos(kaf.omega*x' + kaf.b); % compute RFF vector
y = kaf.Omega'*Psi/kaf.D; % evaluate filter output
err = d - y; % instantaneous error
kaf.Omega = kaf.Omega + kaf.mu*err*Psi; % update
    filter coefficients
```

Script A.4: Iteración de entrenamiento del algoritmo RFF-KLMS para un nuevo par de datos.

A.2. KAFBOX

```
% 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 < kernel_adaptive_filter

    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
```

```
39                 k = kernel(kaf.dict,x,kaf.kerneltype,
                       kaf.kernelpar);
40                 y_est = k'*kaf.alpha;
41             else
42                 y_est = zeros(size(x,1),1); % zeros if
                        not initialized
43             end
44         end
45
46         function train(kaf,x,y) % train the algorithm
47             if (size(kaf.dict,1)<kaf.M), % avoid
                   infinite growth
48                 y_est = kaf.evaluate(x);
49                 err = y - y_est; % instantaneous error
50                 kaf.dict = [kaf.dict; x]; % add base
                       to dictionary
51                 kaf.alpha = [kaf.alpha; kaf.eta*err];
                       % add new coefficient
52             end
53         end
54
55     end
56 end
```

Script A.5: Código de MATLAB de la *toolbox* KAFBOX para la clase de objetos del algoritmo KLMS.

A.3. Algoritmos MO

A.3.1. LMS-MO

```
y = x_concat * obj.w;
err = d - y; % instantaneous error
obj.w = obj.w + obj.mu * x_concat' * err; % update
    filter coefficients
```

Script A.6: Iteración de entrenamiento del algoritmo LMS-MO para un nuevo par de datos.

A.3.2. KLMS-MO

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

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

Script A.7: Iteración de entrenamiento del algoritmo KLMS-MO para un nuevo par de datos.

A.3.3. QKLMS-MO

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

m = size(kaf.dict,1);
if m==0
    d2 = kaf.epsu^2 + 1; % force addition of initial base
else
    [d2,j] = min(sum((kaf.dict - repmat(x_concat,m,1)).^2,2)); % find distance to closest dictionary element
end

if d2 <= epsu^2
    alpha(j,:) = alpha(j,:) + kaf.eta*err; % reduced coefficient update
else
    kaf.dict = [kaf.dict; x_concat]; % add base to dictionary
    kaf.alpha = [kaf.alpha; kaf.eta*err]; % add new coefficient
end
```

Script A.8: Iteración de entrenamiento del algoritmo QKLMS-MO para un nuevo par de datos.

A.3.4. RFF-KLMS-MO

```
if ~numel(kaf.omega) % initialize
    rng('default');
    rng(kaf.seed);
    kaf.omega = 1/kaf.kernelpar*randn(kaf.D,size(x_concat,2));
    kaf.b = 2*pi*rand(kaf.D,1);
    kaf.Omega = zeros(kaf.D,M);
end

Psi = cos(kaf.omega*x_concat' + kaf.b); % compute RFF vector
y = Psi'*kaf.Omega/kaf.D; % evaluate filter output
err = d - y; % instantaneous error
kaf.Omega = kaf.Omega + kaf.mu*Psi*err; % update filter coefficients
```

Script A.9: Iteración de entrenamiento del algoritmo RFF-KLMS-MO para un nuevo par de datos.

A.4. Generación de la base de datos artificial

```
1  function [X,Z,H] = generate_seqs_quad(num_seqs, N, L,
      gamma, ro, filter_, noise_var)
2      %GENERATE_SEQS_QUAD
3      %This function generates NUM_SEQS non-linear
          correlated sequences
4      %(with correlation coefficient RO and quadratic
          coefficient GAMMA) of
5      %length N, using a time-embedding of L samples.
6
7      xt = randn(N+L-1,num_seqs); % temporal series
8      X = xt(L:end,:);
9
10     Y = zeros(num_seqs,N); % Y matrix for adaptive
          filtering algorithms (num_seqs x N)
11
12     diagonal = (1-ro) * ones(1,num_seqs);
13     C = ro * ones(num_seqs) + diag(diagonal); %
          Covariance matrix (num_seqs x num_seqs)
14     H = kron(C,filter_); % Mixing matrix (num_seqs x
          num_seqs*L)
15
16     for i = 1:N
17         x_aux = xt(i:L+i-1,:);
18         x = x_aux(:); % Column vector containing L
              concatenated samples of each sequence
19         y = H * x; % Output vector (num_seqs x 1)
20         Y(:,i) = y;
21     end
22
23     Y = Y';
24     Z = Y - gamma * Y.^2 + sqrt(noise_var)*randn(N,
          num_seqs); % non-linear function
25 end
```

Script A.10: Función de MATLAB para la generación de la base de datos artificial.

A.5. Complejidad computacional de los algoritmos

A.5.1. Desglose de operaciones QKLMS-MO

```
k = kernel(kaf.dict,x_concat,kerneltype,kernelpar);
% kernel: m

y = k * kaf.alpha;
% mult: m * M
% sum: (m-1) * M

err = d - y;
% sum: M

m = size(kaf.dict,1);
if m==0
    d2 = kaf.epsu^2 + 1;
else
    [d2,j] = min(sum((kaf.dict - repmat(x_concat,m,1))
        .^2,2));
    % mult: m * L * M
    % sum: (2*L*M - 1) * M
end

if d2 <= epsu^2
    alpha(j,:) = alpha(j,:) + kaf.eta*err;
    % mult: M
    % sum: M
else
    kaf.dict = [kaf.dict; x_concat];
    kaf.alpha = [kaf.alpha; kaf.eta*err];
end
```

Script A.11: Desglose de las operaciones realizadas en cada iteración del algoritmo QKLMS-MO.

A.5.2. Desglose de operaciones RFF-KLMS-MO

```
if ~numel(kaf.omega)
    rng('default');
    rng(kaf.seed);
    kaf.omega = 1/kaf.kernelpar*randn(kaf.D,size(
        x_concat,2));
    kaf.b = 2*pi*rand(kaf.D,1);
    kaf.Omega = zeros(kaf.D,M);
end

Psi = cos(kaf.omega*x_concat' + kaf.b);
% mult: D * L * M
% sum: D * (L*M - 1) + D
% exp: D

y = Psi'*kaf.Omega/kaf.D;
% mult: D * M
% sum: (D-1) * M
% div: M

err = d - y;
% sum: M

kaf.Omega = kaf.Omega + kaf.mu*Psi*err;
% mult: D * M + 1
% sum: D * M
```

Script A.12: Desglose de las operaciones realizadas en cada iteración del algoritmo RFF-KLMS-MO.

Referencias

[1] D. Xu, Y. Shi, I. W. Tsang, Y. Ong, C. Gong, and X. Shen, “Survey on multi-output learning,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 7, pp. 2409–2429, 2020. 4

[2] M. A. Osborne, S. J. Roberts, A. Rogers, S. D. Ramchurn, and N. R. Jennings, “Towards real-time information processing of sensor network data using computationally efficient multi-output gaussian processes,” in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pp. 109–120, 2008. 4

[3] P. Goovaerts, *Geostatistics for Natural Resource Evaluation*, vol. 42. 1997. 4

[4] J. M. Wang, D. J. Fleet, and A. Hertzmann, “Gaussian process dynamical models for human motion,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 283–298, 2008. 5

[5] H. Kazawa, T. Izumitani, H. Taira, and E. Maeda, “Maximal margin labeling for multi-topic text categorization.,” vol. 17, 2004. 5

[6] M. Álvarez, L. Rosasco, and N. Lawrence, “Kernels for vector-valued functions: A review,” *Foundations and Trends® in Machine Learning*, vol. 4, 2011. 5, 24

[7] M. Álvarez and N. Lawrence, “Computationally efficient convolved multiple output gaussian processes,” *Journal of Machine Learning Research*, vol. 12, pp. 1459–1500, 2011. 5

[8] L. M. Candanedo and V. Feldheim, “Accurate occupancy detection of an office room from light, temperature, humidity and CO_2 measurements using statistical learning models,” *Energy and Buildings*, vol. 112, pp. 28–39, Enero 2016. 6, 46, 47

[9] S. Van Vaerenbergh, *Adaptive Kernel Learning for Signal Processing*, ch. 9. Wiley, 2018. 8, 10, 12, 17

[10] M. H. Hayes, *Statistical digital signal processing and modeling.* New York: John Wiley & Sons, 1996. 8

[11] B. Widrow, J. McCool, and M. Ball, “The complex lms algorithm,” *Proceedings of the IEEE*, vol. 63, no. 4, pp. 719–720, 1975. 8, 11

[12] J. L. Rojo-Alvarez, M. Martinez-Ramon, J. Munoz Mari, and G. Camps-Valls, *Adaptive Kernel Learning for Signal Processing*, ch. 1. Wiley, 2018. 9

[13] A. H. Sayed, *Fundamentals of adaptive filtering*. New York: IEEE Press Wiley-Interscience, 2003. OCLC: ocm52287219. 10

[14] S. S. Haykin, *Adaptive filter theory*. Upper Saddle River, N.J: Prentice Hall, 4th ed., 2002. 11, 14

[15] G. Dorffner, "Neural networks for time series processing," 1996. 11

[16] K. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990. 11

[17] J. C. Principe, W. Liu, and S. S. Haykin, *Kernel adaptive filtering: a comprehensive introduction*. Adaptive and learning systems for signal processing, communication, and control, Hoboken, N.J: John Wiley, 2010. 11

[18] S. Van Vaerenbergh and I. Santamaría, "Métodos kernel para clasificación." Apuntes de la asignatura: Aprendizaje Automático II (Máster Data Science). Universidad de Cantabria. 13

[19] T.-T. Frieß and R. F. Harrison, "A kernel based adaline," in *ESANN*, 1999. 13

[20] J. Kivinen, A. J. Smola, and R. C. Williamson, "Online learning with kernels," *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2165–2176, 2004. 13

[21] 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, 2008. 13, 14

[22] S. S. Haykin, *Neural networks: a comprehensive foundation*. Upper Saddle River, N.J: Prentice Hall, 2nd ed., 1999. 14

[23] B. Schölkopf, R. Herbrich, and A. J. Smola, "A Generalized Representer Theorem," in *Computational Learning Theory* (G. Goos, J. Hartmanis, J. van Leeuwen, D. Helmbold, and B. Williamson, eds.), vol. 2111, pp. 416–426, Springer Berlin Heidelberg, 2001. 15

[24] J. Platt, "A resource-allocating network for function interpolation," *Neural Computation*, vol. 3, no. 2, p. 213–225, 1991. 16

[25] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Berlin, Heidelberg: Springer-Verlag, 1995. 16

[26] B. Chen, S. Zhao, P. Zhu, and J. C. Principe, "Quantized kernel least mean square algorithm," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 1, pp. 22–32, 2012. 17

[27] A. Rahimi and B. Recht, "Uniform approximation of functions with random bases," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, pp. 555–561, 2008. 19

[28] M. Reed and B. Simon, *Fourier Analysis, Self-Adjointness*. No. II in Methods of Modern Mathematical Physics, Acad. Press, 2007. 19

[29] A. Rahimi and B. Recht, "Random Features for Large-Scale Kernel Machines," in *Advances in Neural Information Processing Systems*, vol. 20, 2007. 20

[30] A. Singh, N. Ahuja, and P. Moulin, "Online learning with kernels: Overcoming the growing sum problem," in *2012 IEEE International Workshop on Machine Learning for Signal Processing*, pp. 1–6, 2012. 20, 21

[31] S. Van Vaerenbergh and I. Santamaría, "A comparative study of kernel adaptive filtering algorithms," in *2013 IEEE Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE)*, pp. 181–186, Aug. 2013. Software available at `https://github.com/steven2358/kafbox/`. 21

[32] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*. Adaptive computation and machine learning, Cambridge, Mass: MIT Press, 2006. 32