Arquitectura hardware-software para acelerar el entrenamiento de SVM.

por

Lázaro Bustio Martínez

Tesis sometida como requisito parcial para obtener el grado de

MAESTRO EN CIENCIAS EN EL ÁREA DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Marzo 2010
Tonantzintla, Puebla

Supervisado por:
Dr. René Cumplido Parra, INAOE
Dra. Claudia Feregrino Uribe, INAOE
Dr. José Hernández Palancar, CENATAV

© INAOE 2010
El Autor otorga al INAOE el permiso de reproducir y distribuir copias en su totalidad o en partes de esta tesis.

Resumen

En este trabajo se detallan las Máquinas de Soporte Vectorial y las bases teóricas sobre las que se sustentan. Se hace una descripción de los principales algoritmos que las implementan y que han mostrado mejores resultados según la bibliografía analizada, explicando sus características y particularidades. Teniendo en cuenta que el entrenamiento de una Máquina de Soporte Vectorial es un problema de complejidad cuadrática respecto a las dimensiones de los datos de entrenamiento, es que se propone el diseño e implementación de una arquitectura hardware-software para acelerar el entrenamiento de SVM mediante procesadores de propósito general cuyas funcionalidades puedan ser expandidas con el uso de coprocesadores acoplados. En el procesador central se ejecutarán las funciones de control y secuenciales mientras que en el coprocesador se ejecutarán las funciones paralelizables. Con esta arquitectura se puede realizar el entrenamiento de SVM 178.7 veces más rápido que el algoritmo por software que mejor resultados ofrece.

Abstract

In this work we describe Support Vector Machines (SVM) and their theoretical foundation. Also, we describe the most cited algorithms in the literature that implement SVM explaining their characteristics and particularities. Since training SVM has quadratic complexity concerning training data size, we propose a hardware-software architecture to speed up SVM's training phase, using a general purpose processor with enhanced functions using a coprocessor. The algorithm is partitioned so that the general-purpose processor will execute iterative and control sections while the coprocessor will execute parallel ones. Experiments demonstrate that architecture can speed up SVM´s training phase 178.7 times compared to the best known algorithm implemented in software.

Agradecimientos

Deseo agradecer a todas las personas que de una manera u otra han dado su aporte a la feliz culminación de este trabajo. Especialmente deseo hacer extensivos mis agradecimientos a la **Revolución Cubana** quien me formó como el profesional que hoy soy.

Al pueblo de México, que mediante el CONACyT se me otorgó la beca número 288534 con la cual me fue posible estudiar este postgrado.

Por la parte mexicana, quisiera agradecer muy especialmente a los Doctores: René Cumplido, Claudia Feregrino y Ariel Carrasco: por el apoyo, la ayuda y esfuerzo brindados. A los trabajadores de la Coordinación de Ciencias Computacionales, especialmente a Lucy, Jessica y Normita por siempre ayudarnos en cualquier cosa que necesitásemos.

Por la parte cubana, quisiera agradecer muy especialmente a todos los trabajadores del CENATAV: especialmente al Dr. Palancar por sus consejos y regaños (todos muy acertados);

a los miembros de mi equipo de trabajo: a Isneri, a Diana y a Oneysis por ayudarme y apoyarme en mis responsabilidades mientras estoy en México.

A mi familia: a mis tíos Rolo y Tati, (nuevamente y no me canso de decirlo!!!) Por su apoyo incondicional en todo momento. A mi mamá y mi papá por su aliento y las preocupaciones y ocupaciones hacia mí.

Índice General

I

Lista de Figuras

Lista de Tablas

Glosario de Términos G

APIs: Interfaces de programación de aplicaciones o API (del inglés A*pplication Programming Interface*). Son un conjunto de métodos e instrucciones que ofrecen las bibliotecas o librerías de programación para ser utilizadas por otros softwares como una capa de abstracción fuera del ámbito de éstas.

Bitstream: Archivo que describe la configuración de un circuito digital a ser implementado en un FPGA.

Datapath: Conjunto de unidades funcionales dentro de una arquitectura digital que donde se realizan las operaciones y transformaciones sobre los datos.

Diagrama de secuencia: Artefacto de la metodología RUP (*Rational Unified Process*) para el diseño de software que muestra la interacción de un conjunto de objetos (o clases en programación) en una aplicación a través del tiempo.

dotProduct: Nombre de la sección de la arquitectura hardware-software

propuesta que se encarga del cálculo del producto punto en el coprocesador.

Flip-Flop: Dispositivo de memoria capaz de almacenar un nivel lógico.

FPGA: Arreglos de Compuertas Lógicas Programables (*Field Programmable Gate Arrays* o FPGA por sus siglas en inglés). Son dispositivos semiconductores que contienen bloques de lógica cuya interconexión y funcionalidad se puede programar. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

FSMO: Nombre de la arquitectura hardware-software propuesta para acelerar el entrenamiento de una Máquina de Soporte Vectorial empleando el algoritmo *Sequential Minimal Optimization*.

LUT: Acrónimo de *Look-Up Table* (o Tabla de Consultas). Son pequeñas memorias que almacenan operaciones lógicas primarias.

Netlist: Archivo que describe, para una arquitectura digital en un FPGA, cómo estarán interconectados los elementos del circuto.

Perfil de ejecución: Análisis que se realiza con el fin de identificar zonas de conflicto en la implementación de software tales como fugas de memoria, número de llamadas a funciones, tiempo de ejecución de cada función, etc.

Slices: Unidad reprogramable básica en los FPGAs que agrupan LUTs, Flip-Flops y elementos de conexión.

VHDL: Siglas que representa la combinación de *VHSIC* y *HDL*, donde *VHSIC* es el acrónimo de *Very High Speed Integrated Circuit* y *HDL* es a su vez el acrónimo de *Hardware Description Language*. Es un lenguaje definido por el IEEE que se emplea en la descripción de circuitos digitales.

Capítulo

Introducción 1

En la actualidad se genera un enorme volumen de datos en cuestión de segundos, resultado del quehacer de la actividad humana. Ser capaz de manejar esta información es sinónimo de poder y organizarla eficientemente es un reto para los científicos a nivel mundial. Muchas son las técnicas para intentar dotar de un valor cualitativo a esos datos como son la clasificación, el agrupamiento, el descubrimiento o minado de patrones frecuentes, las reglas de asociación, entre otras, enmarcadas todas dentro de diferentes líneas de investigación de lo que se conoce como Minería de Datos y Reconocimiento de Patrones. Estas técnicas tienen por objetivo común extraer conocimiento potencialmente útil y comprensible de esos enormes volúmenes de información.

Hasta la fecha se han desarrollado varios algoritmos de clasificación, los cuales se seleccionan y emplean dependiendo del problema a resolver, pero la mayoría resultan inoperantes cuando se enfrentan a grandes volúmenes de datos[1]: requieren un elevado período de tiempo para devolver los resultados o sencillamente no pueden procesarlos. Contar con un algoritmo capaz de clasificar grandes volúmenes de datos, con un consumo computacional adecuado y en un período de tiempo relativamente corto resultaría sumamente conveniente en la ciencia y la vida cotidianas. Una técnica de aprendizaje automático que está siendo empleada en una gran multitud de áreas, entre las que se destaca la clasificación de datos, con resultados notablemente alentadores son las Máquinas de Soporte Vectorial (*Support Vector Machines* o SVM por sus siglas en inglés). Su aparición a principio de los años ’90 del pasado siglo provocó una explosión de aplicaciones y de profundos análisis teóricos lo cual hizo de SVM en la actualidad una de las herramientas más poderosas en el Aprendizaje Automático Supervisado y el Reconocimiento de Patrones [1]. Las SVM fueron desarrolladas por Vladimir Vapnik sobre la base de un problema de clasificación lineal binario; es decir, donde sólo se trabaja con dos clases y se implementa el Principio Inductivo de Minimización Estructural del Riesgo [2] para obtener buenos resultados de generalización sobre un número limitado de patrones de aprendizaje. El objetivo de SVM es encontrar un hiperplano óptimo de separación que maximice la distancia entre las muestras más cercanas de las diferentes clases. Este hiperplano es inducido a partir de los propios datos de entrenamiento y de las muestras más cercanas entre sí de las distintas clases, las cuales son llamadas *Vectores de Soporte*. A partir de los Vectores de Soporte es estimado no sólo el hiperplano óptimo de separación sino otras magnitudes y objetos teóricos que serán explicados con más detalles más adelante.

[1] Cuando se habla de grandes volúmenes de datos se hace referencias a matrices de cientos de miles de renglones por millones de columnas.

El proceso de clasificación mediante SVM consta de dos pasos: entrenamiento y clasificación, donde en el primero se reconocen los patrones del conjunto de datos de entrenamiento con el fin de crear un modelo que luego será empleado en el segundo paso para la clasificación de datos desconocidos. En SVM, la clasificación presenta complejidad de $O(n^2)$ en tiempo respecto al número de muestras de entrenamiento, como se demuestra en [3], por lo que los problemas que se pueden solucionar con esta técnica se ven limitados. Actualmente existen tres algoritmos fundamentales para el entrenamiento de SVM en software [4]: *Chunking*, propuesto por Vapnik en [5], *Sequential Minimum Optimization* (SMO) propuesto por Platt en [6] y SVM^{Light}[7] (este último es una mejora propuesta al algoritmo planteado en [8] por Osuna *et al.*). SMO resultó ser el de mayor impacto ya que logra mejorar la rapidez del entrenamiento de SVM, consume menos recursos computacionales respecto a los algoritmos anteriores, es más fácilmente programable y no requiere de cálculos complejos para resolver el problema de programación cuadrática que SVM presenta [5] y [7].

Dado que SVM resulta inadecuado para la clasificación de grandes volúmenes de datos debido a los altos tiempos de entrenamiento y costo en recursos computacionales que requiere, es que entran a jugar su papel técnicas que puedan ayudar a mejorar el desempeño de SVM. Tal es el caso de plataformas hardware-software como solución a esta problemática; en especial, procesadores de propósito general que permitan extender sus funcionalidades mediante coprocesadores acoplados; donde en el procesador central se ejecuten los procesos iterativos y/o de control y en el coprocesador se ejecuten aquellos procesos que se puedan paralelizar. Los Arreglos de Compuertas Lógicas Programables (*Field Programmable Gate Arrays* o FPGA por sus siglas en inglés) sirven como plataforma para validar y verificar esas ideas.

Lo explicado anteriormente sugiere que contar con un sistema de clasificación de datos mediante SVM sobre una arquitectura hardware-software podría ofrecer resultados alentadores y superar la velocidad de entrenamiento en al menos un orden

de magnitud respecto a los que han sido obtenidos mediante las técnicas de clasificación por software.

Por otro lado, pocos han sido los intentos de llevar SVM a hardware. Fuera de [9] no se han propuesto arquitecturas que implementen algoritmos para el entrenamiento de SVM en hardware, específicamente sobre un FPGA; reflejando la falta de investigaciones en esta área por lo que continúa siendo un problema abierto a la ciencia.

El propósito de este trabajo es diseñar e implementar una arquitectura hardware-software que sea una opción para el entrenamiento de SVM en un tiempo de al menos un orden de magnitud menor que los alcanzados por software y con iguales resultados en la calidad del entrenamiento.

1.1 Problemática.

Mejorar el desempeño de SVM, específicamente minimizar el tiempo de entrenamiento y el consumo de memoria continúa siendo un problema abierto en Reconocimiento de Patrones. A pesar de que se han desarrollado una serie de algoritmos encaminados a resolver las limitaciones existentes, resultan insuficientes estos esfuerzos cuando se habla de grandes volúmenes de datos y todos presentan en común las mismas limitantes:

1. El tiempo de entrenamiento resulta poco viable para grandes volúmenes de datos [4], [10], [11], [12], [13].
2. El consumo de recursos computacionales hace que no se pueda enfrentar muchos problemas actuales (por ejemplo, predicciones de fluctuaciones bursátiles [14], en problemas de astronomía es muy común clasificar objetos celestiales en datos del orden de los millones de objetos de entrenamiento [15], [16]; o en clasificación de datos en la web [17] por solo citar algunos.)

3. Problemas al enfrentar datos dispersos y/o con muchos vectores de soporte (en el orden de los miles vectores de soporte ya resulta complicado el entrenamiento).

1.2 Objetivos de la tesis.

Del estudio de la literatura resalta que arquitecturas digitales para acelerar la fase de entrenamiento en la clasificación de datos no ha sido muy exploradas. A excepción de [9] no se han propuesto otros trabajos y el grueso de investigaciones está enfocado en el diseño e implementación de arquitecturas digitales específicamente para la clasificación de muestras desconocidas a partir de un modelo de entrenamiento previamente generado. En el proceso de clasificación de datos, la fase lenta corresponde al entrenamiento de la máquina de aprendizaje. La falta de investigaciones en esta área está dada por la alta complejidad de la implementación de arquitecturas que permitan acelerar el entrenamiento, o que la relación costo-beneficio para este tipo de desarrollo no resulta viable.

Este trabajo propone una arquitectura para reducir los tiempos de la fase de entrenamiento sin afectar la calidad, haciendo frente de esta manera a la falta de investigaciones en esta área.

1.2.1 Objetivo general.

Como objetivo general de este trabajo se propone:

- Diseñar e implementar una arquitectura hardware-software eficiente para el entrenamiento de SVM que permita reducir en al menos un orden de magnitud los tiempos de procesamiento respecto a los mejores algoritmos implementados en software reportados en la literatura.

1.2.2 Objetivos específicos.

Para darle cumplimiento al objetivo general propuesto se plantean como objetivos específicos:

1. Identificar en la literatura el algoritmo candidato a ser implantado sobre una arquitectura hardware-software que permita acelerar la fase de entrenamiento de SVM.
2. Diseñar e implementar en hardware las funciones del algoritmo seleccionado que puedan ser ejecutadas en paralelo.
3. Diseñar e implementar en software una aplicación para el entrenamiento de SVM que utilice las funciones paralelizadas en hardware.

1.3 Metodología.

La metodología que se siguió para cumplir los objetivos de la tesis es la siguiente:

1. **Estudio teórico previo sobre SVM**. Durante esta fase se recopiló y estudió la bibliografía relevante sobre teoría de SVM y los diferentes algoritmos que las implementan. Esta fase consta de las siguientes sub-fases:
   a) Estudio de la teoría sobre la que se desarrolla SVM. En esta sub-fase se estudió la bibliografía que explica las bases teóricas sobre las que SVM se sostiene.
   b) Estudio de los principales algoritmos reportados en la literatura para el entrenamiento de SVM y en específico aquellos para la clasificación binaria. En esta sub-fase se hizo un estudio de los algoritmos para el entrenamiento de SVM más citados en la literatura y se identificaron sus ventajas y limitaciones con el fin de seleccionar el(los) algoritmo(s) que más factible(s) de implementar sobre un FPGA.
2. **Implementación de algoritmos de entrenamiento de SVM**. En esta fase se implementaron los algoritmos seleccionados en la fase 1.b. y se

propusieron técnicas para acelerar el entrenamiento de SVM y reducir el consumo de recursos computacionales:

a) Proponer el diseño hardware para el algoritmo seleccionado como resultado de la fase 1.b.

3. **Evaluación**. En esta fase se analizó el impacto de la variante propuesta, se evaluó la calidad del entrenamiento y se compararon los resultados alcanzados con los obtenidos por otros algoritmos existentes. Para la evaluación se seleccionó un conjunto de prueba reconocido y citado en tareas de clasificación.

Las fases de la metodología propuesta se resumen en la Fig. 1.

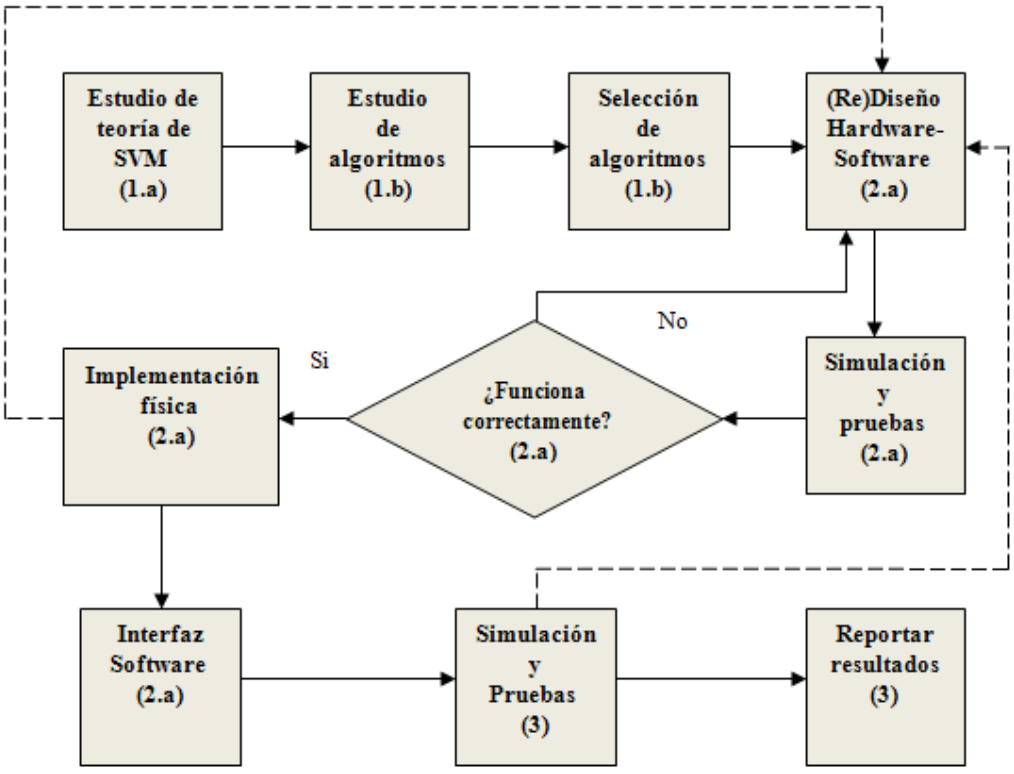


Fig. 1 Metodología a seguir para el cumplimiento de los objetivos de la tesis.

1.4 Estructura de la tesis.

Este trabajo ha sido estructurado en seis secciones. La primera sección corresponde a la introducción, donde se presenta el tema a investigar, se plantean los objetivos que se persiguen y se establece la metodología a seguir. En la segunda sección se explica el Cómputo Reconfigurable y se introducen los FPGAs mostrando sus características, ventajas y desventajas de su uso. En la tercera sección se exponen las bases teóricas de SVM y se describen los algoritmos más relevantes que las implementan, mostrando sus características y haciendo énfasis en sus puntos fuertes y débiles. En la cuarta sección se presenta el análisis y diseño de la arquitectura hardware-software propuesta para acelerar el entrenamiento de SVM. Se detalla la sección de la arquitectura que se ejecuta en software y la sección que se ejecuta en hardware a partir de diagramas y gráficos que explican sus diseños y funcionamientos. En la quinta sección se ofrecen los resultados alcanzados por la arquitectura propuesta y se muestran las comparaciones realizadas con otras implementaciones mostrando los puntos positivos que se alcanzan. En la sexta sección se exponen las conclusiones alcanzadas y los pasos futuros a seguir para esta investigación.

Capítulo

Cómputo Reconfigurable 2

2.1 Enfoques básicos en la implementación de algoritmos.

Existen dos maneras fundamentales para implementar algoritmos en computadoras. El primero es empleando Circuitos Integrados de Aplicación Específica (*Application Specific Integrated Circuits* o ASIC, por sus siglas en inglés). Estos circuitos están diseñados para realizar operaciones puntuales: no están concebidos para lograr la generalidad sino para lograr la especificidad. Están formados por los elementos de hardware indispensables para ejecutar la tarea para la que fueron diseñados y debido a ello es que tienen un alto rendimiento. Sin embargo, al no estar diseñados para la generalidad presentan una flexibilidad nula y esto representa su mayor desventaja. El circuito no pude ser alterado, ni modificado o mejorado después de ser fabricado. Si se requiere alguna modificación del circuito, este debe pasar por un proceso de rediseño y construcción del nuevo diseño.

El segundo enfoque recurre al uso de microprocesadores programados a través de software. Los microprocesadores ejecutan un conjunto de instrucciones para realizar las operaciones que necesita un algoritmo. Al cambiar las instrucciones del software, la funcionalidad del sistema es alterada sin que sea necesario modificar el hardware, lo que permite tener una gran flexibilidad para implementar diversos tipos de algoritmos. Sin embargo, esta flexibilidad afecta al rendimiento en la ejecución de un algoritmo. El microprocesador, antes de poder ejecutar cualquier instrucción, primero debe leerla de la memoria y decodificarla para saber qué acción o acciones debe realizar. En consecuencia se introduce un retraso en la ejecución de cada instrucción. Más aún, si el algoritmo necesita de alguna operación que no existe en el conjunto de instrucciones del microprocesador, ésta debe construirse a partir de las existentes.

El Cómputo Reconfigurable toma las ventajas de ambos enfoques. Mientras trata de mantener un alto rendimiento respecto al tiempo de ejecución del software, también trata de dar la flexibilidad necesaria en hardware que los ASIC no tienen. Los primeros conceptos del Cómputo Reconfigurable se remontan a la década de 1960 [18], pero fue hasta mediados de la década de 1980 cuando el área tomó relevancia gracias a la introducción de los FPGAs. Estos dispositivos ofrecen la flexibilidad que los ASIC no tienen, ya que diversos algoritmos pueden ejecutarse en un FPGA con tan sólo configurarlo cuando se quiera modificar su comportamiento. Esta flexibilidad es análoga a ejecutar diversos algoritmos en software sobre un microprocesador. Otra ventaja importante que tienen las implementaciones de algoritmos en FPGAs, es un aumento considerable en la velocidad de ejecución respecto a implementaciones hechas para software aunque no se alcanzan los mismos índices de aceleración respecto a un ASIC. Esta ventaja se deriva del hecho de que el hardware programable es adaptado a un algoritmo en particular tal como se haría con un ASIC, incluyendo sólo los elementos necesarios para la ejecución de las operaciones que requiere dicho algoritmo.

Las primeras aplicaciones del Cómputo Reconfigurable estuvieron orientadas solamente al procesamiento de señales e imágenes. Recientemente, existen otras

aplicaciones entre las que se incluyen: algoritmos de cifrado, reconocimiento automático de objetivos y compresión de datos sólo por mencionar algunos.

2.2 Plataformas para la implementación de algoritmos.

El Cómputo Reconfigurable está orientado a satisfacer las limitaciones que existen entre los enfoques para la implementación de algoritmos, logrando acercarse al nivel de desempeño que el hardware ofrece sobre el software y al alto nivel de flexibilidad que el software ofrece sobre el hardware. Los dispositivos reconfigurables, incluyendo los FPGAs, contienen un arreglo de elementos computacionales cuyas funciones están determinadas mediante múltiples bits de configuración programables. Estos elementos, llamados bloques lógicos, son conectados entre sí mediante un conjunto de elementos de ruteo que también son configurables. De esta manera, circuitos desarrollados para propósitos generales pueden ser llevados al hardware reconfigurable mediante la obtención y ubicación en los bloques lógicos de las funciones del circuito y mediante la configuración adecuada de los elementos de ruteo para lograr así el circuito deseado.

Los algoritmos de clasificación, cuando se enfrentan a grandes volúmenes de datos y se ejecutan en procesadores de propósitos generales, son costosos en términos del tiempo de procesamiento: se requiere de la velocidad que le imprime el hardware a los algoritmos y procesos que sobre ellos se desarrollan. Cuando se ejecuta un algoritmo de clasificación de datos, se realizan muchas operaciones matemáticas, las cuales no pueden ser programadas eficientemente sin la flexibilidad que ofrecen los procesadores de propósito general. De esta forma se hace necesario lograr un equilibrio entre el hardware y el software para lograr algoritmos más eficientes.

Los FPGAs y el Cómputo Reconfigurable han sido capaces de acelerar una amplia variedad de aplicaciones. Trabajos recientes han mostrado un aumento significativo en la velocidad de sus procesos usando hardware reconfigurable tales como: reconocimiento automático de objetivos, clausuras transitivas de grafos dinámicos,

compresión de datos y algoritmos genéticos para resolver el problema del agente viajero, entre otros [19]. Para el desarrollo de algoritmos en hardware de manera general hay una serie de criterios a tener en cuenta para preferir a los FPGAs, los cuales se ajustan igualmente al desarrollo de algoritmos de clasificación por hardware:

- **Carga de los algoritmos:** El cambio de un algoritmo a otro para la clasificación cuando el diseño se encuentra implantado en la tarjeta se realiza instantáneamente, mientras que en los ASICs implica un rediseño del chip que se desarrolla. En un FPGA esta recarga tarda tan sólo segundos y se puede realizar tantas veces como se requiera.
- **Reconfiguración de algoritmos:** Los algoritmos de clasificación pueden modificarse debido a varias razones y esto no implica un problema para modificar los diseños implantados y que están en uso gracias a que el hardware es reconfigurable.
- **Eficiencia de la arquitectura:** Las arquitecturas hardware pueden ser mucho más eficientes si se diseñan para un conjunto de parámetros específicos. Mientras más específicos sean estos parámetros, más eficiente será la arquitectura. Los FPGAs permiten este tipo de diseños y de optimizaciones de acuerdo a un conjunto de parámetros determinados.
- **Eficiencia de los recursos:** Debido a la reconfiguración del hardware en los FPGAs, un mismo dispositivo puede ser empleado en diversas tareas pagando sólo el costo del tiempo que tarda la reconfiguración del dispositivo.
- **Eficiencia:** Los procesadores de propósito general no traen en su conjunto de operaciones primarias aquellas que se requieren para la clasificación de datos; como por ejemplo, la evaluación de funciones o el cálculo del producto punto por solo citar algunas, por lo que las operaciones matemáticas que se necesitan se obtienen mediante la combinación de estas operaciones primarias. Aunque la frecuencia máxima con que trabajan los

FPGAs es menor que la de los ASICs, queda claro que realizan los mismos procesos de manera más rápida que si se realizan por software.

Las ventajas de las implementaciones por software incluyen facilidad de uso, facilidad de modificaciones, actualizaciones, portabilidad y bajos costos de desarrollo entre otras. Las implementaciones por software ofrecen una velocidad relativamente moderada y altos consumos de potencia comparados con las implementaciones por hardware realizado a la medida.

Las implementaciones mediante ASICs ofrecen a su vez un bajo costo por unidad, pueden lograr mayores velocidades, y pueden obtener menor disipación de potencia. La falta de flexibilidad de los ASICs respecto a las implementaciones por software conduce a altos costos de desarrollo en sentido general. Nuevamente, surgen los FPGAs como la solución a los problemas de ambos enfoques al ser estos dispositivos un punto medio entre ambos como se puede ver en la Fig. 2.

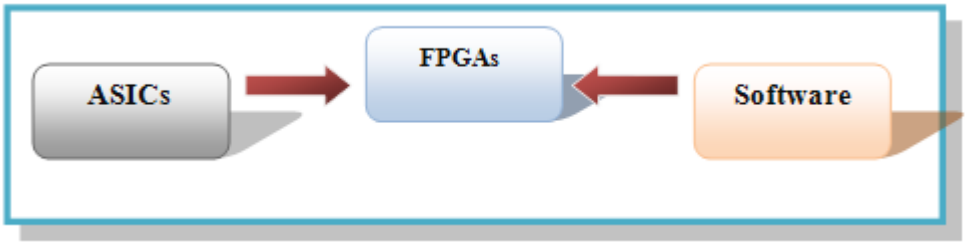


Fig. 2 Ubicación de los FPGAs entre los ASICs y el software.

2.3 FPGA.

Los FPGAs aparecen en 1984 como sucesores de los Dispositivos Complejos Lógicos Programables (*Complex Programmable Logic Device* o CPLDs por sus siglas en inglés) mejorando las deficiencias que éstos presentaban. La arquitectura de un CPLD es más rígida que la de un FPGA y está formado por varias sumas de productos programables cuyos resultados van a parar a un número reducido de flips-flops; mientras que la arquitectura de los FPGAs se basa en un gran número de

bloques que reproducen el comportamiento de las operaciones lógicas básicas. Además de esto, en los FPGAs existe una enorme libertad de interconexión entre dichos bloques permitiendo sobre un mismo dispositivo crear cualquier función. De esta manera, en un FPGA se puede construir desde una simple compuerta AND hasta un microprocesador. Los FPGAs se pueden reprogramar, es decir, los circuitos creados pueden ser "borrados" y sobre el mismo FPGA implantar otro. Esta capacidad permite la creación de circuitos a la medida lo cual trae consigo un bajo consumo de recursos y evita los riesgos tecnológicos que enfrentan los diseñadores de circuitos tradicionales.

Las principales características de los FPGAs son [19]:

- Alta complejidad.
- Bajo costo de desarrollo.
- Fácil depuración de los diseños que se implementan.
- Tolerancia a errores.
- Pocas unidades.
- Tamaño reducido.
- Fiabilidad alta.
- Confidencialidad baja.

Debido a las ventajas que presentan los FPGAs, la mira de los investigadores está dirigida a su uso en aplicaciones donde el poder de procesamiento sea elevado. Aquellos que emplean FPGAs en sus diseños se basan en la premisa de que si se implementan algoritmos rápidos por software sobre un dispositivo FPGA, estos deberían acelerarse notablemente, lo cual no siempre es cierto o la relación costo-beneficio no favorece este tipo de desarrollo.

2.4 Flujo de diseño.

El flujo de diseño que se sigue para implementar algoritmos en hardware se ilustra en la Fig. 4. El paso inicial consiste en separar el algoritmo, para así identificar, a criterio

del diseñador, aquellas partes que puedan ser implementadas eficientemente ya sea en hardware o en software.

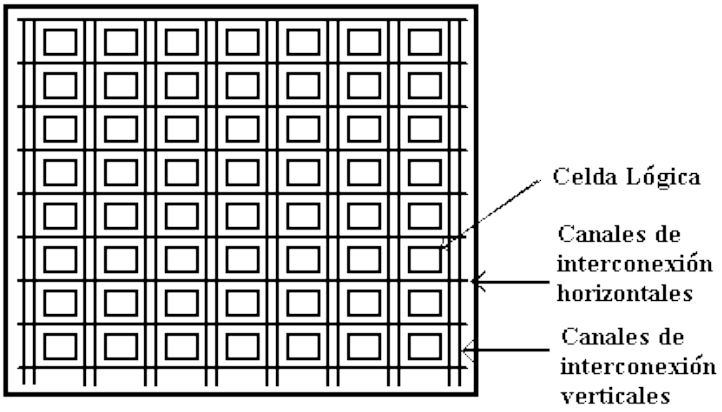


Fig. 3 Aspecto genérico de un FPGA.

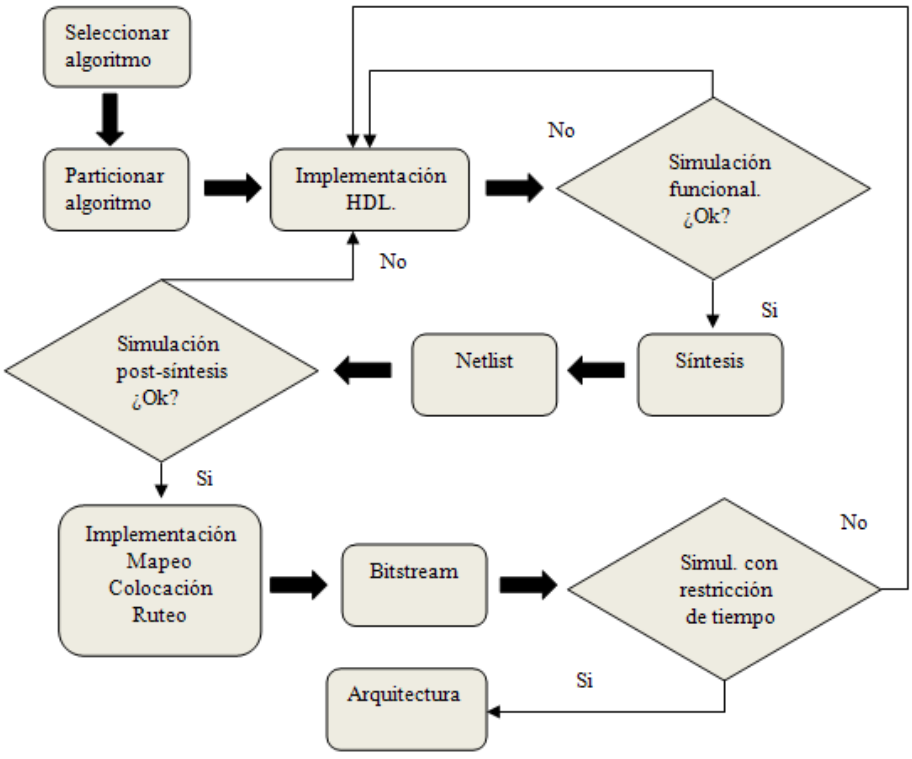


Fig. 4 Flujo del diseño para las arquitecturas digitales sobre FPGAs [20].

La forma en que se implementará en hardware el circuito que ejecutará el algoritmo seleccionado depende del diseñador. Se puede emplear una herramienta de diseño de esquemáticos, donde mediante gráficos se describe el comportamiento del circuito; o emplear un lenguaje de descripción de hardware como VHDL, Verilog o System C, por mencionar algunos, cuyo resultado es el mismo que al emplear los esquemáticos.

Posteriormente, estos diseños son sintetizados (*Síntesis*) donde se lleva del lenguaje HDL a nivel de compuertas lógicas (*Netlist*), obteniéndose de esta manera una representación independiente del hardware a emplear.

El *Netlist* es mapeado (*Mapeo*) hacia los bloques lógicos del hardware específico a emplear. Aquí se verifica que el diseño del circuito cumple con los requerimientos para los que fue creado y con las restricciones del dispositivo hardware sobre el que se ejecutará. En caso de no satisfacer los requerimientos deberá entonces comenzar el proceso con el propósito de corregir errores.

Cuando con la simulación funcional se verifique el buen funcionamiento del circuito entonces se va a la etapa de colocación (*Colocación*), donde se asignan las funciones lógicas obtenidas para el circuito en los bloques particulares del hardware empleado (FPGA en este caso). Automáticamente estos bloques son ruteados (*Ruteo*) por medio de las interconexiones programables para comunicar unos bloques con otros de acuerdo al diseño planteado. Al final del flujo de diseño, se obtiene el archivo de programación (*Bitstream*) con el cual se configurará el FPGA.

Capítulo

Clasificación de datos 3

La clasificación es una técnica que es usada a diario en todas las esferas de la vida cotidiana: se clasifican los frutos buenos de los malos en los mercados; se clasifican los deportistas según sus resultados; se clasifican las noticias en las agencias noticiosas. Todos estos casos tienen en común que el proceso parte de un conjunto de datos sin forma y se asocian según criterios bien establecidos para lograr de esta manera tener bajo una categoría, o clase, aquellos individuos cuyas características se asemejan más. En el Reconocimiento de Patrones, la clasificación de datos es una técnica de vital importancia y a ella se dedica un gran esfuerzo por parte de los investigadores de todo el mundo. Actualmente se espera obtener del proceso de clasificación categorías más refinadas en cuanto a exactitud y procesos más veloces. Existen múltiples algoritmos para la clasificación de datos y a pesar de sus diversas características, todos tienen un punto en común: se tornan no aplicables cuando se intenta clasificar datos masivos.

3.1 Clasificación por Vectores de Soporte.

El problema de la clasificación de datos, visto de manera general, puede ser restringido hasta considerarlo como un problema de clasificación entre dos clases sin que ocurra una pérdida de generalidad. SVM es una técnica diseñada para la clasificación y la regresión, aunque específicamente en la clasificación puede verse como un conjunto de métodos dentro del Aprendizaje Automatizado que toman a los datos de entrada como dos conjuntos de vectores en un espacio *N*-dimensional. Cada conjunto representa una categoría y SVM construye un hiperplano de separación el cual maximiza el margen entre los dos conjuntos (problema de clasificación de dos clases). La función objetivo es optimizada mediante la resolución de un problema de Programación Cuadrática (*Quadratic Programming*, o QP por sus siglas inglés) con restricciones lineales y cuadráticas. Este es un problema computacionalmente intensivo y los requerimientos de memoria crecen con el cuadrado del número de muestras de entrenamiento.

3.1.1 Definición matemática.

El objetivo de SVM es separar los objetos en dos grupos mediante una función que es inducida a partir de los propios objetos. En la Fig. 5 se muestran varios hiperplanos de separación que pueden separar los datos, pero solamente uno maximiza la distancia entre las muestras más cercanas de las distintas clases. Esta distancia máxima es llamada *Margen de Separación* y el hiperplano que maximice la distancia entre él y las muestras más cercanas de ambas clases es llamado *Hiperplano Óptimo de Separación* [21].

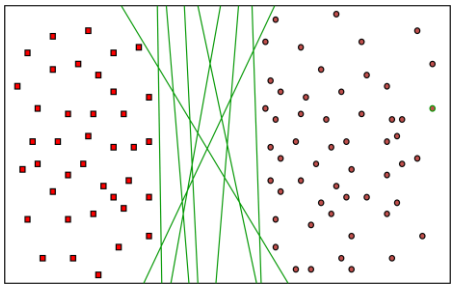


Fig. 5 Posibles hiperplanos separadores para muestras de dos clases.

La clasificación binaria de datos usando SVM presenta tres variantes diferentes:

- Linealmente separable.
- Linealmente no separable.
- Superficie de decisión no lineal.

3.1.1.1 Caso linealmente separable.

La idea básica de la clasificación binaria con SVM es encontrar un hiperplano como superficie de decisión que separe las clases con el mayor margen posible.

La ecuación general del plano en *n*-dimensiones es:

$$\boldsymbol{w} \cdot \boldsymbol{x} = b$$

o lo que es lo mismo:

$$\langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b = 0 \qquad \textbf{(1)}$$

donde $\boldsymbol{x}$ es un vector tamaño $n \times 1$; $\boldsymbol{w}$ es la normal al hiperplano y b es un valor escalar. De todos los puntos al hiperplano, sólo uno tiene la menor distancia S_{min} al origen:

$$S_{min} = \frac{|b|}{\|\boldsymbol{w}\|}$$

Pero existe un error de redundancia en (1) y sin que se pierda generalidad se puede considerar que los hiperplanos canónicos y sus parámetros $\boldsymbol{w}$ y b están restringidos por:

$$min_i|\langle \boldsymbol{w}, \boldsymbol{x_i} \rangle + b| = 1 \quad \textbf{(2)}$$

La idea se ilustra en la Fig. 6 donde se muestra la distancia del punto más cercano a cada posible hiperplano de separación.

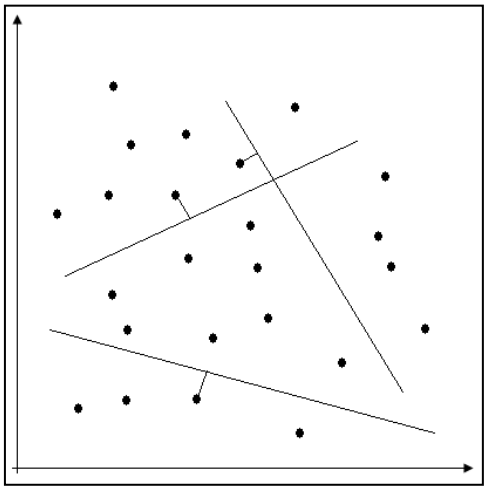


Fig. 6 Distancia más cercana de los datos a cada posible hiperplano de separación.

Sea un problema de clasificación binaria donde la extracción de características se realiza inicialmente, se clasifican los datos de entrenamiento $\boldsymbol{x_i} \in \mathbb{R}^d$ con la etiqueta $y_i \in \{-1; +1\}$ para todos los datos o muestras de entrenamiento $i = 1 \ldots l$ donde l es la cantidad de muestras de entrenamiento y d es la dimensión del problema. Cuando dos clases son potencialmente linealmente separables en $\mathbb{R}^d$, se necesita encontrar un hiperplano separador el cual ofrezca el menor error de generalización entre el número infinito de posibles hiperplanos. Dicho hiperplano es aquel que maximice el margen de separación entre las dos clases y dicho margen es la suma de las distancias del hiperplano a los puntos más cercanos de cada una de las clases.

Lo anteriormente dicho se resume en que dado los patrones de entrenamiento $(\boldsymbol{x_1}, y_1), \cdots, (\boldsymbol{x_l}, y_l)$ donde $\boldsymbol{x_i}$ es un vector *d*-dimensional y l es el número de muestras de entrenamiento, entonces:

$y_i = 1$ Si $\boldsymbol{x_i}$ pertenece a la clase A

$y_i = -1$ Si $\boldsymbol{x_i}$ pertenece a la clase B

Si los datos son linealmente separables, entonces existe un vector *d*-dimensional $\boldsymbol{w}$ y un escalar *b* tales que:

$$\boldsymbol{w} \cdot \boldsymbol{x_i} - b \geq 1 \quad \text{si } y_i = 1; \qquad \boldsymbol{w} \cdot \boldsymbol{x_i} - b \leq 1 \quad \text{si } y_i = -1; \qquad \textbf{(3)}$$

De forma compacta, (2) puede escribirse como

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} - b) \geq 1 \qquad \textbf{(4)}$$

o $-(y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} - b) - 1) \leq 0$ donde el par $(\boldsymbol{w}, b)$ define el hiperplano que separa las dos clases de datos. Con el propósito de hacer cada superficie de decisión $(\boldsymbol{w}, b)$ única, se normaliza la distancia perpendicular desde el origen al hiperplano separador dividiéndola por $\|\boldsymbol{w}\|$, dando la distancia como $\frac{|b|}{\|\boldsymbol{w}\|}$. Como se aprecia en la Fig. 8**,** la distancia perpendicular del origen al hiperplano H_1 $(\boldsymbol{w} \cdot \boldsymbol{x_i} - b = 1)$ es $\frac{|1+b|}{\|\boldsymbol{w}\|}$ y del origen al hiperplano H_2 $(\boldsymbol{w} \cdot \boldsymbol{x_i} - b = -1)$ es $\frac{|b-1|}{\|\boldsymbol{w}\|}$. Aquellos objetos que se encuentran sobre los hiperplanos H_1 y H_2 son llamados *Vectores de Soporte*.

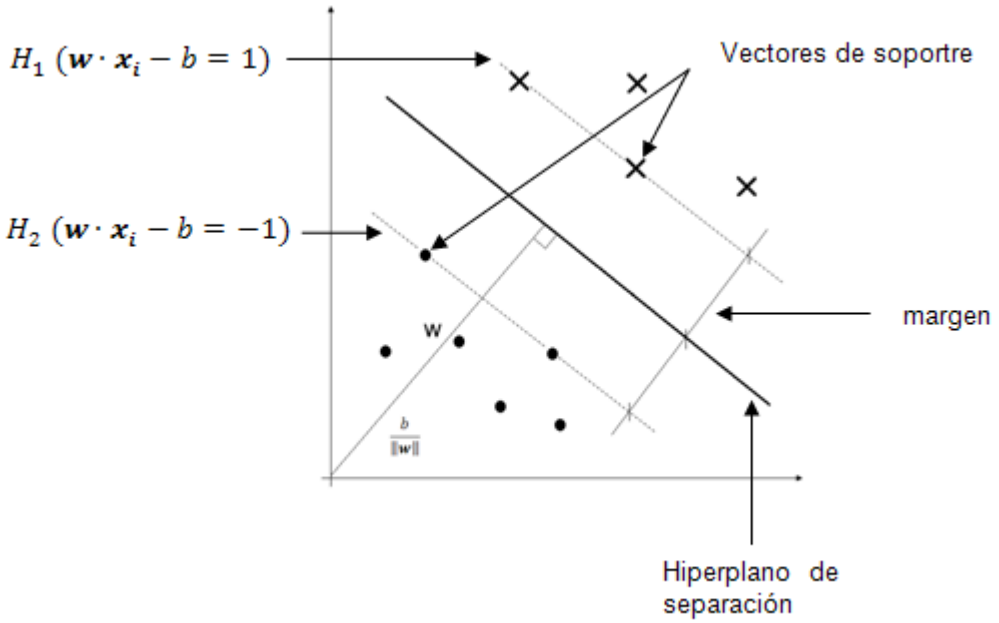


Fig. 7 Plano óptimo de separación y los Vectores de Soporte.

Eliminar cualquiera de los puntos que no se encuentran sobre H_1 y H_2 no afecta el resultado de la clasificación, pero eliminar alguno de los vectores de soporte sí. La distancia entre los dos hiperplanos H_1 y H_2 es $\frac{2}{\|\boldsymbol{w}\|}$, donde $\frac{1}{\|\boldsymbol{w}\|}$ es el margen y determina la capacidad de la máquina de aprendizaje.

El objetivo para lograr una clasificación correcta es maximizar $\frac{2}{\|\boldsymbol{w}\|}$ lo cual es equivalente a minimizar $\frac{\|\boldsymbol{w}\|^2}{2}$. Es este punto se puede formular el problema como:

Minimizar $$f=\frac{\|\boldsymbol{w}\|^2}{2}$$

Sujeto a $$g_i=-(y_i(\boldsymbol{w}\cdot\boldsymbol{x_i}-b)-1)\leq 1,\quad i=1,\cdots,l$$

donde l es el número de datos de entrenamiento.

Este problema puede resolverse usando las técnicas de QP, sin embargo, usar el método de Lagrange [22] para resolver el problema facilita extender el análisis realizado al caso linealmente no separable.

La función Lagrangiana para este problema es:

$$L_p(\boldsymbol{w}, b, \boldsymbol{\Lambda}) = f(\boldsymbol{x}) + \sum_{i=1}^{l} \lambda_i g_i(\boldsymbol{x})$$
$$= \frac{\boldsymbol{w} \cdot \boldsymbol{w}}{2} - \sum_{i=1}^{l} \lambda_i (y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} - b) - 1)$$
$$= \frac{\boldsymbol{w} \cdot \boldsymbol{w}}{2} - \sum_{i=1}^{l} \lambda_i y_i \boldsymbol{w} \cdot \boldsymbol{x_i} + \sum_{i=1}^{l} \lambda_i y_i b + \sum_{i=1}^{l} \lambda_i$$

donde $\Lambda = \{\lambda_1, \lambda_2, \cdots \lambda_l\}$ son los multiplicadores de Lagrange del conjunto de entrenamiento.

Las condiciones de Karush-Kuhn-Tucker (condiciones KKT)[22] para este problema son:

Condición del gradiente:

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{l} \lambda_i y_i \boldsymbol{x_i} = 0 \qquad \textbf{(5)}$$

donde $\frac{\partial L_p}{\partial \boldsymbol{w}} = \left(\frac{\partial L_p}{\partial w_1}, \frac{\partial L_p}{\partial w_2}, \cdots, \frac{\partial L_p}{\partial w_d}, \right)$

$$\frac{\partial L_p}{\partial b} = \sum_{i=1}^{l} \lambda_i y_i = 0 \qquad \textbf{(6)}$$

$$\frac{\partial L_p}{\partial \lambda_i} = g_i(\boldsymbol{x}) = 0 \qquad \textbf{(7)}$$

Condición de ortogonalidad:

$$\lambda_i g_i = -\lambda_i (y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} - b) - 1) = 0, \qquad i = 1, \cdots, l \qquad \textbf{(8)}$$

Condición de viabilidad:

$$-y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} - b) + 1 \leq 0, \qquad i = 1, \cdots, l \tag{9}$$

Condición de no-negatividad:

$$\lambda_i \geq 0, \qquad i = 1, \cdots, l \tag{10}$$

El punto estacionario del Lagrangiano determina las soluciones para el problema de optimización. Sustituyendo las ecuaciones (5) y (6) en la parte derecha de la función Lagrangiana se reduce la función a la forma dual con λ_i como variable dual. El problema después de la sustitución es:

Maximizar

$$L_D = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{l} \lambda_i \lambda_j y_i y_j \boldsymbol{x_i} \cdot \boldsymbol{x_j} \tag{11}$$

Sujeto a:

$$\sum_{i=1}^{l} \lambda_i y_i = 0$$

$$\lambda_i > 0, \qquad i = 1, \ldots, l.$$

Se pueden obtener todos los valores de λ resolviendo la ecuación (11) mediante QP y $\boldsymbol{w}$ puede ser obtenido usando la ecuación (5):

$$\boldsymbol{w} = \sum_{i=1}^{l} \lambda_i y_i \boldsymbol{x_i}$$

Los valores de λ_i que sean mayores que cero para las restricciones dadas corresponden a los vectores de soporte del sistema mientras que los demás no son vectores de soporte.

Se puede calcular el valor de *b* usando la ecuación (8):

$$\lambda_i(y_i(\boldsymbol{w}\cdot\boldsymbol{x_i}-b)-1)=0, \qquad i=1,\dots,l \tag{12}$$

seleccionando el valor de $\boldsymbol{x_i}$ con λ distinto de cero.

La clase de un dato de entrada x es entonces determinada por:

$$clasificación(\boldsymbol{x}) = signo(\boldsymbol{w}\cdot\boldsymbol{x}-b) \tag{13}$$

En la práctica, debido a implementaciones numéricas, el valor de *b* es calculado mediante el promedio de todos los valores de *b* usando la ecuación (12).

3.1.1.2 Caso linealmente no separable.

En la mayoría de los problemas reales el hiperplano de separación lineal no existe, sin embargo; aun así se puede buscar alguno que permita la clasificación introduciendo un conjunto de variables ξ las cuales medirán el grado de violación de las restricciones para el caso linealmente separable.

La formulación del problema sería para este caso:

Minimizar

$$f(\boldsymbol{w},\Xi)=\frac{1}{2}\|\boldsymbol{w}\|^2+C\sum_{i=1}^{l}\xi$$

Sujeto a:

$$y_i(\boldsymbol{w}\cdot\boldsymbol{x_i}-b)\geq 1-\xi_i, \qquad i=1,\dots l$$

$$\xi_i\geq 0, \qquad i=1\dots,l$$

donde $\Xi=(\xi_1,\dots,\xi_l)$ y *C* son parámetros determinados a priori que pueden ser vistos como valores de penalización. Un valor pequeño de *C* maximiza el margen y el hiperplano es menos sensible a los datos anómalos en las muestras de entrenamiento; mientras que un valor grande de *C* minimiza el número de los puntos mal clasificados.

La función Lagrangiana quedaría:

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\lambda_i(y_i(\boldsymbol{x_i}\cdot\boldsymbol{w} - b) - 1 + \xi) - \sum_{i=1}^{l}\mu_i\xi_i$$

donde μ_i son los multiplicadores de Lagrange introducidos por la restricción $\xi_i \geq 0$.

Las condiciones KKT para este problema serían:

Condición del gradiente:

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{l}\lambda_i y_i \boldsymbol{x_i} = 0 \qquad \textbf{(14)}$$

donde $\frac{\partial L_p}{\partial \boldsymbol{w}} = \left(\frac{\partial L_p}{\partial w_1}, \frac{\partial L_p}{\partial w_2}, \cdots, \frac{\partial L_p}{\partial w_d},\right)$

$$\frac{\partial L_p}{\partial b} = \sum_{i=1}^{l}\lambda_i y_i = 0 \qquad \textbf{(15)}$$

$$\frac{\partial L_p}{\partial \xi_i} = C - \lambda_i - \mu_i = 0 \qquad \textbf{(16)}$$

$$\frac{\partial L_p}{\partial \lambda_i} = -(y_i(\boldsymbol{x_i}\cdot\boldsymbol{w} - b) - 1 + \xi_i) \qquad \textbf{(17)}$$

Condición de ortogonalidad:

$$\lambda_i(y_i(\boldsymbol{x_i}\cdot\boldsymbol{w} - b) - 1 + \xi_i) = 0, \qquad i = 1, \cdots, l \qquad \textbf{(18)}$$

Condición de viabilidad:

$$(y_i(\boldsymbol{x_i}\cdot\boldsymbol{w} - b) - 1 + \xi_i) = 0, \qquad i = 1, \cdots, l \qquad \textbf{(19)}$$

Condición de no-negatividad:

$$\xi_i \geq 0, \qquad i = 1, \cdots, l \qquad \textbf{(20)}$$

$$\lambda_i \geq 0, \qquad i = 1, \cdots, l \qquad \textbf{(21)}$$

$$\mu_i \geq 0, \qquad i = 1, \cdots, l \qquad \textbf{(22)}$$

$$\mu_i \xi_i = 0, \qquad i = 1, \cdots, l \qquad \textbf{(23)}$$

Sustituyendo las ecuaciones (14) y (15) en la parte derecha de la función Lagrangiana se obtiene el siguiente problema dual con las mismas variables Lagrangianas que en el caso anterior:

Maximizar

$$L_D = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{l} \lambda_i \lambda_j y_i y_j \boldsymbol{x_i} \cdot \boldsymbol{x_j}$$

Sujeto a:

$$0 \leq \lambda_i \leq C,$$

$$\sum_{i=1}^{l} \lambda_i y_i = 0$$

La solución para $\boldsymbol{w}$ puede ser determinada por la ecuación (14) que describe una de las condiciones de KKT $\boldsymbol{w} = \sum_{i=1}^{l} \lambda_i y_i \boldsymbol{x_i}$

Nuevamente, *b* puede ser encontrado promediando todos los valores de *b* entre todas las muestras de entrenamiento; los cuales pueden ser calculados usando las siguientes condiciones KKT:

$$\lambda_i (y_i (\boldsymbol{w} \cdot \boldsymbol{x_i} - b) - 1 + \xi_i) = 0 \qquad \textbf{(24)}$$

$$(C - \lambda_i)\xi_i = 0 \qquad \textbf{(25)}$$

La ecuación (25) también indica que $\xi_i = 0$ si $\lambda_i < C$. Por lo tanto, b puede ser calculado solamente sobre aquellas muestras que cumplan que:

$$0 < \lambda_i < C.$$

Si $\lambda_i < C$ entonces $\xi_i = 0$ en este caso los vectores de soporte se encuentran a una distancia de $\frac{1}{\|\boldsymbol{w}\|}$ del hiperplano separador (estos vectores son llamados *Vectores de Soporte Marginales*.) Cuando $\lambda_i = C$ entonces los vectores de soporte son puntos mal clasificados si $\xi_i > 1$. Cuando $0 < \xi_i < 1$ los vectores de soporte son clasificados correctamente pero están más cerca que $\frac{1}{\|\boldsymbol{w}\|}$ del hiperplano. Estos son los V*ectores de Soporte Frontera*. Ver la Fig. 9.

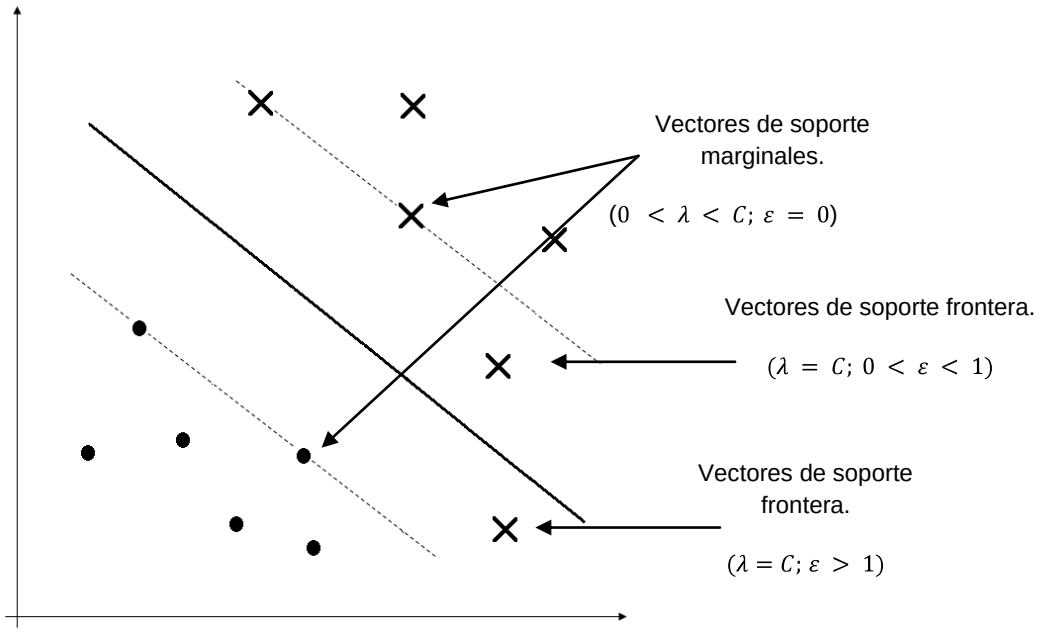


Fig. 8 Vectores de Soporte Marginales y Frontera.

3.1.1.3 Superficie de decisión no lineal.

En la mayoría de los casos en los que se precisa clasificar datos, el hiperplano de separación no es una superficie lineal. Sin embargo, la teoría de SVM puede ser extendida para manejar estos casos. La idea que se sigue es llevar los datos de entrada

$\boldsymbol{x}$ a un espacio de mayor número de dimensiones y allí realizar la separación lineal de los datos donde la separación lineal resulte más evidente. Esto sería:

$$\begin{gathered}\boldsymbol{x} \rightarrow \varphi(\boldsymbol{x}), \\ \boldsymbol{x} = (x_1, x_2, \dots, x_n), \\ \varphi(\boldsymbol{x}) = (\varphi_1(\boldsymbol{x}), \varphi_2(\boldsymbol{x}), \dots, \varphi_n(\boldsymbol{x}), \dots,)\end{gathered}$$

La solución de SVM tiene la misma forma que para el caso linealmente separable, lo que ahora cuenta con la transformación entre los espacios de número de dimensiones diferentes anteriormente explicada.

$$\begin{aligned} clasificación &= signo(\varphi(\boldsymbol{x}) \cdot \boldsymbol{w} - b) \\ &= signo\left(\sum_{i=1}^{l} y_i \lambda_i \varphi(\boldsymbol{x_i}) \cdot \varphi(x) - b\right) \end{aligned} \quad \textbf{(26)}$$

No es necesario conocer la función φ para resolver este problema. Una función K llamada *Función Kernel* que cumpla con la propiedad que:

$$K(x, y) = \varphi(\boldsymbol{x}) \cdot \varphi(y)$$

simplificará la ecuación (26) hasta convertirla en:

$$signo\left(\sum_{i=1}^{l} y_i \lambda_i K(\boldsymbol{x_i}, \boldsymbol{x}) - b\right)$$

Entre las funciones kernels más empleadas están las siguientes:

- Lineal: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \boldsymbol{x_i^T x_j}$
- Polinomial: $K\left(\boldsymbol{x_i}, \boldsymbol{x_j}\right) = (\gamma \boldsymbol{x_i^T x_j} + r)^d, \gamma > 0$
- RBF: $K\left(\boldsymbol{x_i}, \boldsymbol{x_j}\right) = exp\left(-\gamma \left\|\boldsymbol{x_i} - \boldsymbol{x_j}\right\|^2\right), \gamma > 0$
- Sigmoid: $K\left(\boldsymbol{x_i}, \boldsymbol{x_j}\right) = tanh\left(\gamma \boldsymbol{x_i^T x_j} + r\right)$

donde γ, r y d son parámetros del kernel en cuestión [23]

Usando los kernels, la ecuación (11) se puede reescribir como:

$$W(\lambda) = \sum_{i=1}^{l} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j K(\boldsymbol{x_i}, \boldsymbol{x_j}) \lambda_i \lambda_j$$

que en el caso de SVM se reduce a maximizar W sujeto a:

$$0 \leq \lambda_i \leq C \; i = 1, \ldots, l \;\; \text{y} \;\; \sum_{i=1}^{l} y_i \lambda_i = 0$$

el cual es un problema de QP.

La Fig. 9 representa el caso de clasificación donde la superficie de decisión no es lineal, se pueden ver los vectores de soporte y el hiperplano de separación.

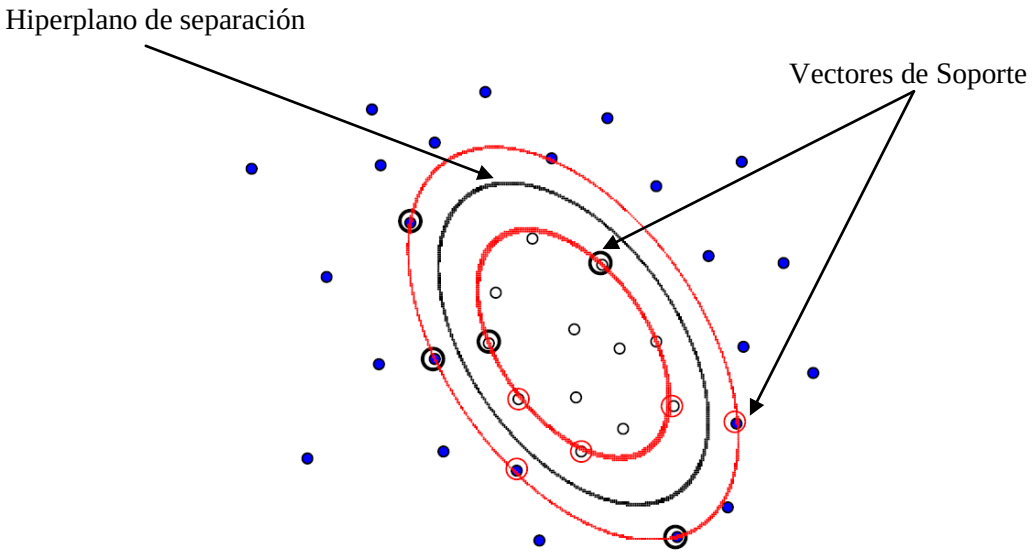


Fig. 9 Superficie de decisión no lineal.

3.1.2 Algoritmos para el entrenamiento de SVM.

Entrenar SVM es equivalente a resolver un problema de QP *N*-dimensional, donde *N* es el número de muestras en la matriz de datos de entrenamiento. Resolver este problema usando las técnicas tradicionales de QP involucra grandes operaciones matriciales, las cuales son altamente costosas en tiempo y en recursos

computacionales; las que en la mayoría de los casos no resultan prácticas y sí extremadamente lentas para problemas grandes. Sólo el caso de un conjunto de datos de entrenamiento de 4000 × 4000 cuyos elementos fuesen de tipo de datos *double* de simple precisión no cabría siquiera en 128 Mbytes de memoria. Debido a esta limitante, y buscando vías para resolverla, se han creado varios algoritmos para solucionar el problema del entrenamiento de SVM.

A grandes rasgos, un pseudo-algoritmo para entrenar SVM en sentido general sería [10]:

1. Seleccionar el parámetro *C* (que representa un compromiso entre minimizar el error de entrenamiento y maximizar del margen de separación entre los elementos de las clases), la función kernel y cualquier otro parámetro requerido por la función kernel empleada.
2. Resolver el problema de QP o la formulación alternativa usando el algoritmo apropiado de QP o Programación Lineal para obtener los vectores de soporte.
3. Obtener el umbral *b* usando los vectores de soporte ya calculados.

3.1.2.1 Algoritmos para el entrenamiento de SVM por software.

Existen varios algoritmos para el entrenamiento de SVM. Por los resultados reportados se destacan básicamente tres [4]:

Chunking: Este algoritmo fue propuesto por Vapnik en [5]. Utiliza la gran dispersidad que poseen los valores obtenidos para los multiplicadores de Lagrange, los cuales varían entre cero y el límite superior dado por *C*. El valor de la forma cuadrática será idéntico para el caso en que se mantienen todas las muestras y para el caso en que se eliminan las filas y columnas de la matriz de entrenamiento que correspondan a multiplicadores de Lagrange con valor cero y algunos de aquellos que violen las condiciones de KKT. Por lo tanto, los grandes problemas de QP pueden descomponerse en una serie de pequeños subproblemas cuyo objetivo es identificar

los multiplicadores de Lagrange con valores iguales a cero para descartarlos. Las dimensiones del problema varían pero son finalmente iguales al número de multiplicadores de Lagrange diferentes de cero. Esta técnica está limitada por el número máximo de vectores de soporte que se pueden manejar y además requiere de optimizadores cuadráticos para resolver la secuencia de subproblemas de optimización [24]. Los subproblemas de optimización que se obtienen con este algoritmo continúan siendo demasiado grandes para poder realizar el entrenamiento de SVM de manera adecuada puesto que requiere de un elevado tiempo y necesita demasiada memoria para enfrentarse a los subproblemas de optimización.

Osuna: Osuna *et al.* en [25] proponen un algoritmo para entrenar las SVMs que hace frente a las limitantes que presenta *Chunking* en cuanto a la selección de los objetos que formarán los subproblemas y el tamaño de estos subproblemas. Este algoritmo, al igual que el anterior, se basa en dos suposiciones: 1) el número de vectores de soporte es pequeño respecto al número total de muestras de entrenamiento; 2) el número total de vectores de soporte no excede a unos cuantos miles (menos de 3000). Se han desarrollado variantes a este algoritmo como es SVM^{Light} [7] para la clasificación y SVM_{Torch} [26] para la regresión.

Para este algoritmo se definen dos conjuntos de índices de para dividir el conjunto de entrenamiento: B y N. El algoritmo se basa en la suposición que se puede definir un conjunto de tamaño fijo B tal que $|B| \leq l$, donde l es el número de muestras de entrenamiento. Además, B debe ser lo suficientemente grande para contener todos los vectores de soporte (o lo que es lo mismo, cuyos valores de $\lambda_i > 0$) pero lo suficientemente pequeño tal que la computadora pueda manejarlo. Bajo estas condiciones, este algoritmo puede ser enunciado como sigue:

1. Arbitrariamente seleccionar $|B|$ muestras del conjunto de entrenamiento.
2. Resolver el problema de optimización definido por las muestras en B.
3. Mientras exista algún $j \in N$ tal que $g(\boldsymbol{x_j})y_j < 1$ donde:

$$g(\boldsymbol{x_j}) = \sum_{i=1}^{l} \lambda_i y_i K(\boldsymbol{x_j}, \boldsymbol{x_i}) + b$$

Reemplazar $\lambda_i = 0, i \in B$ con $\lambda_j = 0$ y resolver el nuevo subproblema.

Como se puede ver, la complejidad de este algoritmo aumenta en correspondencia con la cantidad de muestras de entrenamiento y no es factible para problemas de varias decenas de miles de muestras de entrenamiento (ver la Fig. 10).

El trabajo [25] fue presentado en marzo de 1997 y ya en septiembre aparece una modificación [8] al algoritmo presentado anteriormente que pretendía mejorar sus limitaciones. En este nuevo algoritmo no se tienen en cuenta las suposiciones 1) y 2) enunciadas anteriormente. Igual que en [25], se definen dos conjuntos de índices B y N y presenta igualmente dos puntos medulares. La primera proposición plantea que si se mueve una muestra de B a N esto hace que la función de costo se mantenga inalterable, y la solución es factible en el subproblema. La segunda proposición plantea que mover una muestra que viola la condición de optimalidad de N a B da una mejora estricta en la función de costo cuando el problema es reoptimizado (ver [8] para la demostración).

Tomando como base las proposiciones anteriores se formula el algoritmo de descomposición:

1. Se escogen arbitrariamente $|B|$ muestras del conjunto de entrenamiento.
2. Resolver el subproblema definido por las muestras en B.
3. Mientras exista algún $j \in N$ tal que:

a. $\lambda_j = 0 \; y \; g(\boldsymbol{x_j}) y_j < 1$

b. $\lambda_j = C \; y \; g(\boldsymbol{x_j}) y_j > 1$

c. $0 < \lambda_j < C \; y \; g(\boldsymbol{x_j}) y_j \neq 1$

Reemplazar los $\lambda_i, i \in B$ con λ_j y resolver el nuevo subproblema.

Este algoritmo sugiere que se adicione y se elimine una muestra en cada iteración, lo cual puede ser ineficiente debido a que debería utilizar un paso completo de optimización para obligar a que una muestra obedezca las condiciones de KKT.

SMO: *Sequential Minimal Optimization* o SMO, puede resolver los problemas de QP generados por SVM sin agregar ninguna matriz extra de almacenamiento y sin usar optimizadores numéricos. SMO descompone el problema de QP en pequeños subproblemas bajo determinadas condiciones y emplea el teorema de Osuna *et al.* en [8] para garantizar la convergencia y cada subproblema es resuelto por separado [6].

A diferencia de los otros métodos, SMO selecciona el menor problema de optimización posible en cada paso iteración: dos multiplicadores de Lagrange, ya que los multiplicadores de Lagrange deberán obedecer la restricción de igualdad lineal. De esta forma, en cada iteración SMO selecciona 2 multiplicadores de Lagrange para optimizar conjuntamente, encuentra el valor óptimo para estos dos multiplicadores y actualiza sus variables para reflejar los nuevos valores óptimos. De esta manera se necesitan más iteraciones para lograr la convergencia que los algoritmos anteriores pero los cálculos necesarios para realizar el paso de optimización con 2 multiplicadores de Lagrange es notablemente más rápido que cuando se seleccionan varios multiplicadores.

Básicamente, SMO está compuesto por dos elementos básicos: un método analítico para realizar la optimización con los dos multiplicadores de Lagrange escogidos; y una heurística para seleccionar con cuáles multiplicadores se realizará la optimización.

En la Fig. 10 se muestran los tres algoritmos para el entrenamiento de SVM: *Chunking*, el algoritmo de Osuna y SMO. Cada línea horizontal representa el conjunto de entrenamiento y los cuadros representan los multiplicadores de Lagrange que serán optimizados en cada paso. En *Chunking* el número de muestras a entrenar por paso tiende a crecer. Para el algoritmo de Osuna, un número fijo de muestras

serán optimizadas por cada paso, las cuales serán insertadas y eliminadas del problema en cada paso. Para SMO solamente dos muestras son analíticamente optimizadas en cada paso por lo que cada paso es más rápido [6]

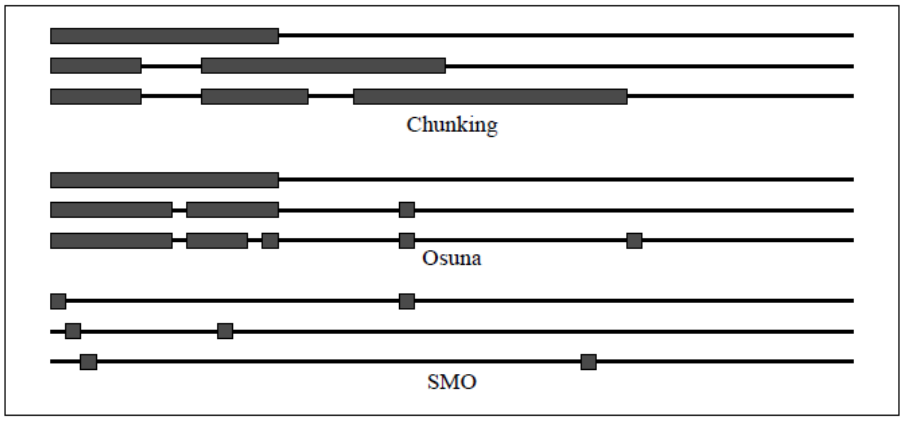


Fig. 10 Conjunto de entrenamiento para Chunking, Osuna y SMO [6].

3.1.2.2 Algoritmos para el entrenamiento de SVM por hardware.

DSVM: Uno de los trabajos más citados sobre FPGAs y SVM fue presentado por Davide Anguita *et al.* en [27]. En este trabajo los autores describen la implementación de kernels digitales para clasificadores, especialmente se refieren al perceptrón. Aquí los autores se basan en una arquitectura digital de punto fijo y ofrecen una variante para el perceptrón llamada *Digital Kernel Perceptron*. Comparan sus resultados con los resultados obtenidos por software para las operaciones de punto flotante y punto fijo obteniendo resultados similares en cuanto a métricas entre los distintos modelos probados y consideraron al diseño por hardware como la mejor opción dado lo reducido del diseño y los resultados obtenidos. En un trabajo posterior [9], los mismos autores proponen una arquitectura digital para el aprendizaje mediante SVM y discuten su implementación en un FPGA. En este trabajo se analiza brevemente los efectos de cuantización en el desempeño del SVM en problemas de clasificación para mostrar su robustez frente a las implementaciones de las operaciones matemáticas de punto fijo. La arquitectura que se propone implementa un algoritmo menos sensible a los errores de cuantización respecto a los algoritmos reportados en la literatura.

La solución que se propone en [9] consta de 2 partes: el algoritmo FIBS y el algoritmo DSVM donde el primero, iterativamente, invoca al segundo para calcular los parámetros de SVM con un valor de *b* fijo y luego, con estos parámetros y mediante un proceso de bisección, se calculan los nuevos valores de *b* hasta que el criterio de convergencia sea alcanzado. Una vez terminado el proceso los vectores de soporte son aquellos tales que

$$SV = \{i: \alpha_i \epsilon (0, C)\}.$$

Este algoritmo utiliza las ventajas del paralelismo en los FPGAs para resolver los problemas de QP que surgen al realizar el aprendizaje por SVM. Para llegar a esta implementación se parte de la idea de diseñar un algoritmo de aprendizaje en dos fases las cuales trabajarían iterativamente. La primera parte se encarga de resolver los problemas de QP para un valor de *b* fijo (en el origen en este caso) mientras que el segundo implementa un procedimiento para actualizar el valor de *b* con el objetivo de obtener iterativamente el valor óptimo. El algoritmo DSVM puede ser visto como un dispositivo al cual se le suministra la matriz kernel *Q*, el umbral fijo *b*, y comenzando con un valor inicial de α^0(multiplicadores de Lagrange), el algoritmo resuelve los problemas de QP dando una solución intermedia α^i. El segmento DSVM es el encargado de calcular los valores de α dados la matriz Kernel, un valor inicial para *b* y un valor inicial para α'. Los multiplicadores de Lagrange se calculan mediante un proceso iterativo cuyas operaciones se realizan de manera concurrente aprovechando las facilidades que ofrecen los FPGA en el paralelismo de instrucciones. Estos valores de α son valores intermedios obtenidos para el *b* intermedio suministrado a DSVM. Con α se calcula en FIBS un *b* más refinado el cual se suministra a DSVM para obtener un α más exacto y este proceso se repite hasta que la condición de convergencia se alcance y se obtenga el *b* óptimo y a partir de él los valores finales para los multiplicadores de Lagrange.

El segmento FIBS se ejecuta de manera secuencial. Tiene dos secciones principales donde primeramente se obtienen los valores de α con valores de b fijos mediante DSVM. En dependencia de la evaluación de:

$$clasificación = signo(y^T \alpha')$$

para los α obtenidos en la fase anterior, se obtienen los valores de los extremos posibles donde se encuentra el valor óptimo de b al cual se llega mediante un proceso de bisección y se repite el proceso nuevamente para el nuevo valor de b calculado.

A pesar de que este algoritmo presenta varias estructuras secuenciales, fue pensado y creado para ejecutarse sobre un FPGA aprovechando las ventajas que estos presentan para paralelizar procesos, ya que las operaciones básicas del algoritmo sí pueden ejecutarse concurrentemente. Este algoritmo se basa en que un kernel Gaussiano mapea los datos a un espacio de características infinito donde el efecto de eliminar algún parámetro de SVM es despreciable. En particular, con esta idea si se hace $b = 0$ se está forzando a que el hiperplano separador pase por el origen del espacio de características y la restricción de igualdad desaparece de la formulación del problema de QP. Esta idea sólo es aplicada a un kernel Gaussiano y no se explica en este artículo cuáles serían los efectos si se empleara otro tipo de kernel.

A continuación se muestra el algoritmo *DSVM*:

Algoritmo 1: **Algoritmo DSVM para un valor de *b* fijo.**

1. input: Q, b, α^0

2. set $\tilde{r} = (r - yb)$; $\eta = \frac{2}{l\, max_{i,j}(q_{i,j})}$

3. set endrun = false; $k = 0$

4. **while not** endrun **do**

5. $g = (-Q\alpha^k + \tilde{r})$

6. $z^{k+1} = \alpha^k + \eta g$

7. $\alpha_i^{k+1} = max\left(0, min\left(z_i^{k+1}, C\right)\right), \forall i = 1, \dots, l$

8. **if** $\forall i\left(\alpha_i^{k+1} - \alpha_i^k\right) \leq \varepsilon$ **then**

9.	endrun = true
10.	**else** $k = k + 1$
11.	**end do**
12.	output: $\alpha' = \alpha^{k+1}$

Algoritmo 2: **Algoritmo FIBS.**

1.	$b_{low} = b = -1;\ \alpha^0 = 0;\ b_{up} = +1; end\ Fibs = false$
2.	$\alpha' = A_b(Q, b, \alpha^0); s = sgn(y^T\alpha');\ \alpha^0 = \alpha'$
3.	**if** $s < 0$
4.	**while** $s < 0$ **do**
5.	$b_{up} = b; b = 2b;\ b_{low} = 2b_{low}$
6.	$\alpha' = A_b(Q, b, \alpha^0); s = sgn(y^T\alpha');\ \alpha^0 = \alpha'$
7.	**end do**
8.	**else if** $s > 0$ **then**
9.	$b = b_{up}$
10.	$\alpha' = A_b(Q, b, \alpha^0); s = sgn(y^T\alpha'); \alpha^0 = \alpha'$
11.	**if** $s > 0$ **then**
12.	**while** $s > 0$ **do**
13.	$b_{low} = b; b = 2b;\ b_{up} = 2b_{up}$
14.	$\alpha' = A_b(Q, b, \alpha^0); s = sgn(y^T\alpha'); \alpha^0 = a'$
15.	**end do**
16.	**else** endFibs = true
17.	**while not** endFibs **do**
18.	$b = \left(\left(b_{low} + b_{up}\right)/2\right)$
19.	$\alpha' = A_b(Q, b, \alpha^0); s = sgn(y^T\alpha'); \alpha^0 = a'$
20.	**if** $s = 0$ **or** $\left(b_{low} - b_{up}\right) \leq \varepsilon_b$ **then** endFibs = true
21.	**else if** $s < 0$ **then** $b_{up} = b$
22.	**else** $b_{low} = b$
23.	**end do**

En [9] se puede ver en detalle el análisis realizado al respecto.

La arquitectura que propone Anguita *et al.* en [9] puede ser vista como un bloque general, *SVMblock*, para el cual sus funcionalidades pueden ser divididas en tres fases básicas:

- **Fase de carga**: Cuando se recibe la señal *start*, (transición de 0 a 1) comienza la carga del vector de clasificación *y* y la matriz de kernel *Q*.
- **Fase de aprendizaje**: Tan pronto como la fase de carga se completa, se comienza la fase de aprendizaje según el algoritmo *FIBS* detallado en [9] y cuando esta fase termina envía una señal de activación para comenzar la fase de salida.
- **Fase de salida**: Permite los resultados del aprendizaje ($b^*, a_1^*, \dots, \alpha_m^*$) a la salida del dispositivo.

Con el fin de comprobar el desempeño de esta implementación los autores realizaron un conjunto de pruebas y se seleccionó la base de datos Sonar [28] y algunos conjuntos de datos derivados de problemas de telecomunicaciones propios de los autores. La base de datos Sonar ha sido ampliamente estudiada y aplicada en la comprobación de diversos algoritmos de aprendizaje como patrón de prueba. Esta base de datos está compuesta por 208 muestras caracterizadas por 60 rasgos cada una usualmente dividida en 104 muestras de entrenamiento y 104 de prueba. Es conocido que un clasificador lineal clasifica erróneamente 23 muestras de entrenamiento, mientras que un RBF-SVM se equivoca en clasificar 6 muestras si se usa un umbral y 8 patrones en caso contrario. Se realizó un experimento empleando un kernel gaussiano con $\sigma^2 = 1\, y\, C = 10$. La solución consistió de 119 vectores de soporte (59 pertenecientes a la clase positiva y 60 pertenecientes a la clase negativa). Posteriormente fueron aplicadas perturbaciones aleatorias de tamaño Δ a los parámetros y luego los errores de clasificación mínimos y máximos fueron medidos para 10^6 pruebas diferentes. A medida que la perturbación fue mayor los errores de clasificación fueron aumentando pero siempre obteniendo resultados semejantes a los

reportados en la literatura para la clasificación de esta base de datos. El algoritmo se implementó sobre un Virtex II de Xilinx.

A modo de resumen; con esta implementación se logra desarrollar un algoritmo para realizar el entrenamiento de SVM que logra minimizar los efectos de cuantización, obtiene resultados semejantes a los algoritmos reportados en la literatura y se mejora el tiempo de convergencia respecto a los algoritmos por software. Lo interesante de esta implementación es que logra poner en un solo chip todo el hardware necesario para el entrenamiento SVM de manera que sólo le sean entregados al dispositivo los parámetros de entrada y este devuelve los resultados: todo el procesamiento se realiza en el FPGA.

3.1.3 Análisis de SMO.

A pesar de que con SMO se obtienen mejores resultados en el desempeño que con otros algoritmos, resulta demasiado complejo en término del número de operaciones a realizar en el entrenamiento de una SVM para enfrentarse a problemas realmente grandes. En [29] se prueba que SMO puede tornarse ineficiente cuando las muestras exceden un determinado número, por lo tanto, ya que SMO es el algoritmo más rápido para entrenar una SVM no habrá manera de enfrentarse a determinados problemas. Cuando se trata de un conjunto de entrenamiento muy grande el proceso será especialmente lento ya que sólo se optimizará una pareja de muestras en cada iteración. Muchas de estas iteraciones serán empleadas en avances en la optimización que aportarán poco o nada a la solución final del problema o deshaciendo cambios previos. Además, si los datos son muy grandes, las estructuras de datos del algoritmo deberían limpiarse cada cierto número de iteraciones, lo cual lejos de ayudar al desarrollo del algoritmo constituiría un limitación.

Pero las dimensiones del conjunto de entrenamiento no son el único problema a enfrentar. Por ejemplo, incrementar el valor del parámetro *C* puede hacer que la clasificación mejore, pero incrementaría dramáticamente el tiempo para llegar a la

solución. En [30] se identifica lo que a juicio de los autores es el punto de ineficiencia de SMO: se calcula la desviación del clasificador en cada iteración, la que frecuentemente no es exacta y por lo tanto causa que la selección de las muestras a optimizar no sea la más adecuada y que además incrementa el tiempo de procesamiento del algoritmo. En ese mismo trabajo se muestra un ejemplo de lo anterior y afirman que su propuesta acelera en 2 veces el desempeño del SMO clásico. Básicamente, la modificación que se hace es emplear dos variables para el umbral *b*: b_{up} y b_{low} y emplean otras restricciones para chequear el cumplimiento de la optimización (ver [30] para mayor referencia). A todo lo antes mencionado puede sumársele el costo que implica el cálculo de la función kernel para datos de grandes dimensiones.

Los algoritmos mencionados anteriormente son variantes para resolver problemas de programación cuadrática que por demás es una tarea lenta que requiere demasiada memoria extra y un profundo conocimiento de análisis numérico. Como se demostró en [6], [31] y [32], SMO resultó ser entre todos los algoritmos analizados el más eficiente y que mejores resultados ofrece. En este punto ya se está en condiciones de ofrecer una descripción de SMO, donde se usará el pseudocódigo original propuesto por Platt en el cual se muestra en el Pseudocódigo 1:

```
target = desired output vector
point = training point matrix

procedure takeStep( i1,i2 )
  if ( i1 == i2 ) return 0
  alph1 = Lagrange multiplier for i1
  y1 = target[i1]
  E1 = SVM output on point[i1] - y1 (check in error cache)
  s = y1 * y2
  Compute L, H via equations (13)[2] and (14)[3]
  if ( L == H )
    return 0
  k11 = kernel( point[i1], point[i1] )
```

[2] En el artículo original de Platt, eq. 13: $L = \max(0, \alpha_2 - \alpha_1)$, $H = min(C, C + \alpha_2 - \alpha_1)$

[3] En el artículo original de Platt, eq. 14: $L = \max(0, \alpha_2 + \alpha_1 - C)$, $H = min(C, \alpha_2 + \alpha_1)$

```
    k12 = kernel( point[i1], point[i2] )
    k22 = kernel( point[i2], point[i2] )
    eta = k11 + k22 – 2 * k12
    if ( eta > 0 )
    {
      a2 = alph2 + y2 * (E1-E2)/eta
      if ( a2 < L ) a2 = L
      else if ( a2 > L ) a2 = H
    }
    else
    {
      Lobj = objective function at a2 = L
      Hobj = objective function at a2 = H
      if ( Lobj < Hobj - eps )
        a2 = L
      else if ( Lobj > Hobj + eps)
        a2 = H
      else
        a2 = alph2
    }
    if ( |a2 - alph2| < eps * (a2 + alph2 + eps))
      return 0
    a1 = alph1 + s * (alph2 – a2)
        Update threshold to reflect change in Lagrange Multipliers
       Update weight vector to reflect change in a1 & a2, if SVM is
       linear
    Update error cache using new Lagrange Multipliers
    Store a1 in the alpha array
    Store a2 in the alpha array
    Return 1
  endprocedure

  procedure examineExample(i2)
    y2 = target[i2]
    alph2 = Lagrange multiplier for i2
        E2 = SVM output on point[i2] – y2 (check in error cache)
    r2 = E2 * y2
        if ((r2 < -tol && alph2 < C) || (r2 > tol && alph2 > 0))
    {
      if ( non-zero & non-C alpha > 1)
      {
            i1 = result of second choice heuristic (section 2.2)
        if takeStep(i1,i2)
          return 1
      }
            Loop over all non-zero and non-C alpha, starting at a
       random point
      {
        i1 = identity of current alpha
        if takeStep( i1,i2 )
          return 1
      }
            Loop over all possible i1, starting at a random point
      {
```

```
      i1 = loop variable
      if takeStep( i1, i2 )
        return 1
    }
  }
  return 0
endprocedure

main routine:
  numChanged  = 0;
  examineAll = 1;
  while ( numChanged > 0 | examineAll)
  {
    numChanged = 0;
    if ( examineAll )
      loop I over all training examples
        numChanged += examineExample(I)
    else
      loop I over examples where alpha is not 0 & not C
        numChanged += examineExample(I)
    if ( examineAll == 1)
      examineAll == 0
    else if (numChanged == 0)
      examineAll = 1
  }
endRoutine
```

Pseudocódigo 1 Algoritmo SMO propuesto por Platt.

Para mayor claridad, se reproduce un sumario del algoritmo SMO:

1. Iterar sobre todas las muestras del conjunto de entrenamiento buscando un λ_1 que viole las condiciones de Karush-Kuhn-Tucker.
   a. Si λ_1 es encontrado, entonces ir al paso **2**.
   b. Si termina de iterar sobre el conjunto de entrenamiento y no se encuentra ningún candidato, entonces iterar sobre el conjunto de muestras cuyos λ son distintos de 0 o *C* (conjunto de las no fronteras) buscando el nuevo λ_1.
   c. Si λ_1 es encontrado, entonces va al paso **2.**
   d. Se alternarán las iteraciones sobre todo el conjunto de entrenamiento y el conjunto de las no fronteras buscando un λ_1 que viole las

condiciones de KKT hasta que todos los λ de las muestras cumplan con la condición de KKT.

e. Termina.

2. Buscar un λ_2 en el conjunto de las no fronteras.

a. Calcular E_1 *y* E_2.

b. Tomar el λ que obtenga el mayor valor de $|E_1 - E_2|$como λ_2.

c. Si las dos muestras son iguales, entonces ir al paso **3**. Si no, calcular los valores de *L* y *H* para λ_2.

d. Si $L = H$ entonces la optimización no puede realizarse. Se desecha este valor de λ_2 y va al paso **3**. De otra forma, se calcula el valor de η.

e. Si η es negativo, entonces abandonar λ_2 e ir a **2**.

f. Si η es positivo, entonces calcular la función objetivo para los puntos *L* y *H* y usar el λ que de mayor resultado para la función objetivo como λ_2

g. Si $\left|\lambda_2^{Nuevo} - \lambda_2^{Viejo}\right|$ es menor que ξ, entonces abandonar λ_2 e ir a **3**. De otra forma, ir a **4**.

3. Iterar sobre el conjunto de las no fronteras comenzando en un punto aleatorio hasta que λ_2 descrito en el paso **2** sea encontrado.

a. Si no es encontrado, entonces iterar sobre todo el conjunto de entrenamiento hasta que un λ_2 que haga el progreso de la optimización sea encontrado.

b. Si no se encuentra ningún λ_2 en esas 2 iteraciones, entonces abandonar λ_1 y regresa a **1** para encontrar un nuevo λ que viole las condiciones de KKT.

4. Calcular el nuevo valor de λ_1

a. Actualizar el valor del umbral *b*, la caché de error y almacenar los nuevos valores de λ_1 y λ_2.

b. Ir a **1**.

Como se puede ver, SMO consta de una jerarquía de heurísticas para seleccionar las dos muestras que se optimizarán y es ahí donde radica la aceleración que este algoritmo le imprime al entrenamiento de SVM. Estas heurísticas consisten en evaluaciones reiterativas de las condiciones de KKT para las muestras candidatas a ser optimizadas.

Del estudio realizado al pseudocódigo de SMO resalta que es un algoritmo básicamente secuencial. Para realizar un paso en la optimización del problema de QP con las muestras candidatas se debió recorrer una jerarquía de heurísticas donde cada paso depende del anterior por lo que no se pude independizar una acción de la otra. Visto de esta manera, es muy difícil llevar este algoritmo, como su autor lo enuncia, a hardware y aprovechar las ventajas del paralelismo. Sin embargo, teniendo en cuenta el enfoque que se persigue con este trabajo, las estructuras de control del algoritmo (la jerarquía de heurísticas) son las candidatas adecuadas a ejecutarse en software (en el procesador) mientras que las funciones que intervienen en la optimización podrían ejecutarse en hardware (en el coprocesador). De esta manera se libera al procesador de los cálculos, reservándolo para la toma de decisiones, mientras que la optimización se realizará en un coprocesador diseñado y construido para este propósito en específico.

Capítulo

Diseño de la arquitectura 4

Como resultado del estudio de la literatura sobre SVM se pudieron identificar tres algoritmos fundamentales para el entrenamiento [4]: tal es el caso de *Chunking* [5], *Sequential Minimum Optimization* (SMO) [6] y SVM^{Light} [7]. De éstos, SMO es el que mejores resultados ofrece y es por ello que se seleccionó como el algoritmo a implementar para lograr la aceleración de una SVM como se establece en el paso 1 de la metodología de trabajo propuesta. Sin embargo, en [29] y [30] se demuestra que para grandes volúmenes de datos, el entrenamiento de SVM mediante SMO requiere de un elevado período de tiempo. Para revertir esta situación la solución estaría en una implementación por hardware, pero esta idea no resulta práctica por el hecho de que SVM requiere de cálculos especializados, los cuales serían muy complejos de realizar puramente en hardware resultando una relación costo-beneficios no adecuada. Por otro lado, el entrenamiento de SVM puede realizarse de una manera relativamente sencilla mediante el empleo de un procesador de propósito general pero no se lograría obtener la aceleración requerida. Por lo tanto y con base en lo

explicado en capítulos anteriores, para lograr una aceleración sustancial en la fase de entrenamiento de SVM junto con facilidad de implementación y confiabilidad en los resultados, las arquitecturas hardware-software resaltan como la variante adecuada a estos requerimientos. Si se analiza el Diagrama 1 y el algoritmo SMO detallado anteriormente, resulta evidente el hecho que el algoritmo SMO presenta una alta dependencia entre sus operaciones y realiza un elevado número de evaluaciones condicionales las que determinan el camino a seguir por el algoritmo. Todo esto atenta contra el desarrollo de una arquitectura que implemente el algoritmo SMO de manera íntegra sobre hardware, dado que la razón principal de su uso para acelerar algoritmos está en el hecho de explotar eficientemente el paralelismo intrínseco del FPGA en aquellas operaciones o algoritmos que puedan ser ejecutados en paralelo. En el caso de SMO, no existe un alto grado de paralelismo en los procesos que desarrolla, pero las funciones más costosas en cuanto al tiempo sí se pueden ejecutar en paralelo. Estas evidencias apuntan a que la variante apropiada para mejorar el desempeño de SMO es una arquitectura hardware-software que implemente las funciones más costosas en cuanto a tiempo de SMO. En software se realizan aquellas funciones que no pueden ser ejecutadas en paralelo y las estructuras de control del algoritmo y el FPGA actuará como un coprocesador donde se ejecutarán las funciones que sí se puedan paralelizar.

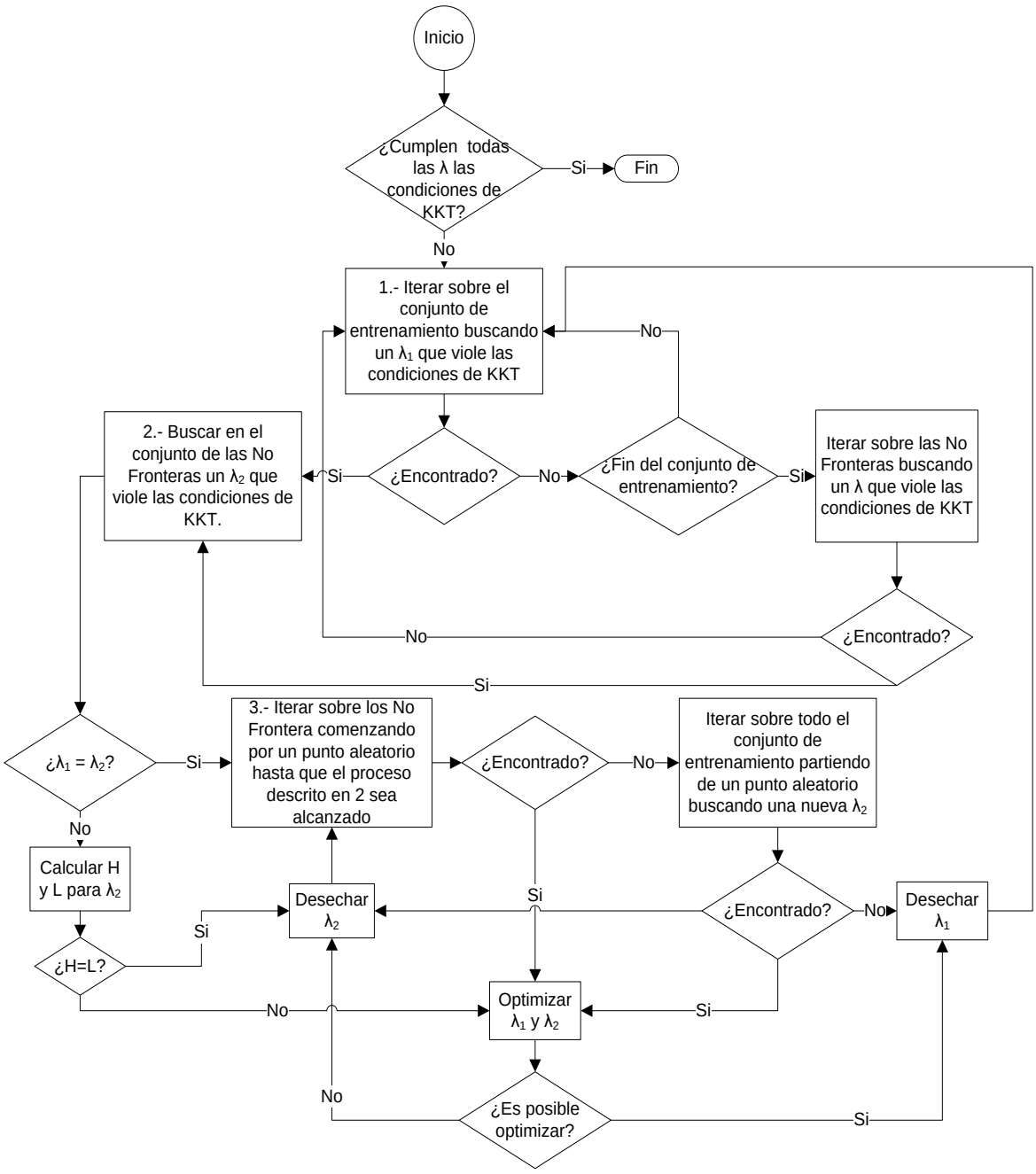


Diagrama 1 Diagrama de flujo de los procesos de SMO.

4.1 Análisis del desempeño de SMO.

En el Diagrama 1 se muestra los procesos que ocurren en SMO para el entrenamiento de datos. Se llevó el pseudocódigo del algoritmo a un diagrama de flujo para representar mejor las operaciones que se ejecutan y ver la dependencia entre ellas. Como se puede ver, primero se busca la primera muestra candidata para efectuar la optimización (λ_1). Si λ_1 no es encontrada se cambia el conjunto de búsqueda y se itera hasta que λ_1 sea encontrada o de lo contrario se termina el entrenamiento. Luego de λ_1 se busca la segunda muestra para la optimización (λ_2) la cual si no es encontrada se cambia el conjunto de búsqueda y de igual manera no se pasa al tercer paso hasta que no se tenga una candidata. Una vez encontrada λ_2 se verifica que $\lambda_1 \neq \lambda_2$ y en caso de que sean distintas se realiza la optimización; mientras que si las muestras encontradas son las mismas se repite el proceso pero esta vez buscando en otro conjunto de búsqueda distinto a los ya explorados, hasta que dos candidatas para la optimización que sea válidas se encuentren y ésta pueda realizarse. Como se puede ver, cada paso en la heurística depende del anterior y están todos estrechamente vinculados.

En la literatura analizada son muy escasos los análisis del desempeño para los algoritmos que realizan el entrenamiento de SVM. Solamente Dey *et al.* en [33] hace un estudio para identificar las secciones responsables de los elevados períodos de tiempo en el entrenamiento de SVM. La exploración estructural de este algoritmo se realiza con el fin de identificar aquellas secciones que sean las responsables de los largos períodos de tiempo empleados en el entrenamiento. Para ello se realizó un estudio del comportamiento del SMO en tiempo de ejecución donde se evidenció el tiempo que tardó en completarse cada una de las funciones del código y la cantidad de llamadas realizadas a dichas funciones. Con este fin se realizó una implementación del algoritmo especificado en el pseudocódigo de SMO enunciado por Platt en [6]. El estudio del comportamiento se realizó en un procesador Intel Core 2 Quad a 2.4GHz de frecuencia y sobre el sistema operativo Windows XP. Se utilizó como perfilador a

AQTime de *AutomatedQA* en su versión 6.3 [34] y como datos de prueba se empleó la base de datos *Adult*, del repositorio UCI [28]. Las características de la base de datos *Adult* se explican más detalladamente en el Capítulo 5. *Adult* ha sido ampliamente usado en la certificación y comprobación de nuevos algoritmos de clasificación: Platt la empleó para demostrar la validez de SMO en [6], [31] y [32]. En este trabajo se usará como referencia comparativa.

El perfil se realizó según las propias especificaciones de la base de datos. Bajo estas condiciones, se perfiló el entrenamiento de cada corpus de la base de datos y los resultados fueron promediados y recopilados en la Tabla 1.

Tabla 1 Resultados del análisis del perfil de ejecución de SMO.

	Kernel Lineal C = 0.05		**Kernel Polinomial C = 0.05, d = 2**		**Kernel RBF C = 0.05, v = 2**	
Función	**Llamadas**	**Tiempo (s)**	**Llamadas**	**Tiempo (s)**	**Llamadas**	**Tiempo (s)**
binSearch	7817	1958	82920	47503	82920	29253
calculateError	15794	727795	7980	2386386	7980	818975
calculateNorm	1	9	NA[4]	NA	NA	NA
dotProduct	14889860	5582598	35987048	26141490	35987048	21174731
examineExample	20523	17838	95515	131428	95515	98130
getb	1	0	1	0	1	0
initializeData	1	8	1	21	1	11
initializeTraining	1	35	1	59	1	45
main	1	983	1	44490	1	1160
myrandom	183	17	81	19	81	9
power	NA	NA	35987048	6013089	35987048	3503052
qsort2	300115	53938	20135710	20662508	20135710	7375240
quicksort	731685	157150	1625385	1527156	1625385	482934
readFile	1	115031	1	305899	1	163125
startLearn	1	38944	1	571525	1	322872
swap	3452739	160179	86264029	11803593	86264029	6131272
takeStep	3851	20226	41334	10099795	41334	3335630
writeModel	1	37699	1	70135	1	54043

[4] NA: No Aplica. Se refiere a un valor que no se puede obtener por alguna razón, ya sea que no se devuelve o que no se calcula, que se obtiene en tiempos extremadamente grandes, etc.

Los resultados obtenidos del perfil realizado a SMO fueron recopilados y representados en las Fig. 11, Fig. 12, Fig. 13 y Fig. 14.

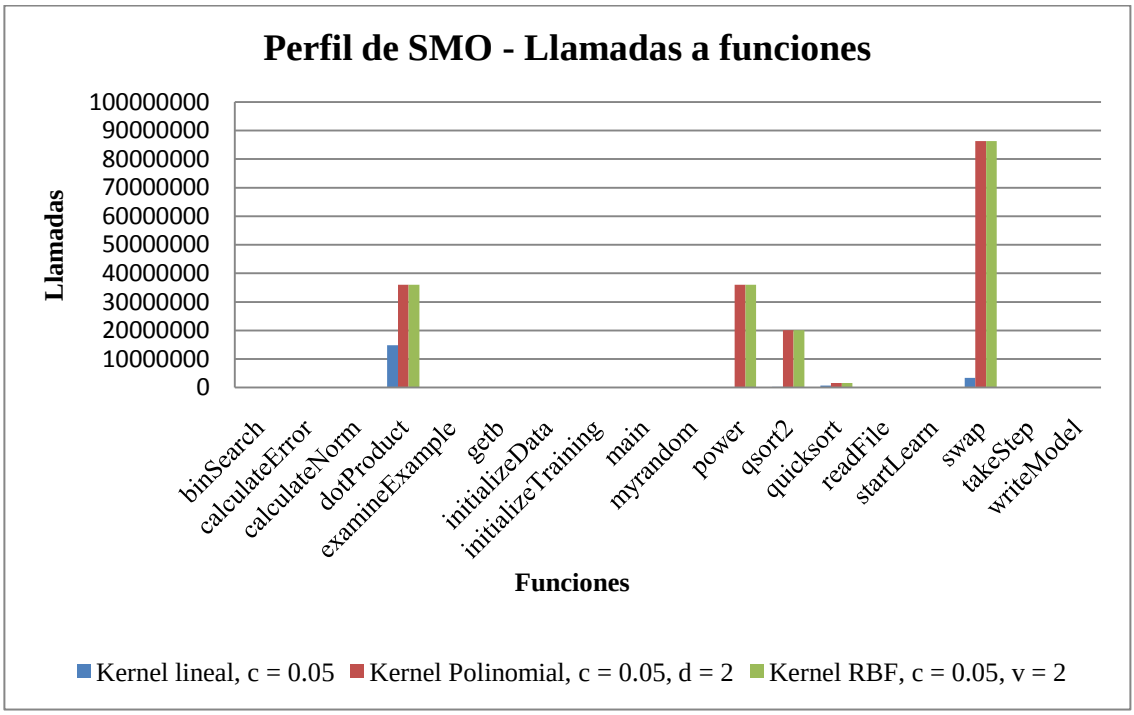


Fig. 11 Resultados del perfil de SMO para llamadas a funciones.

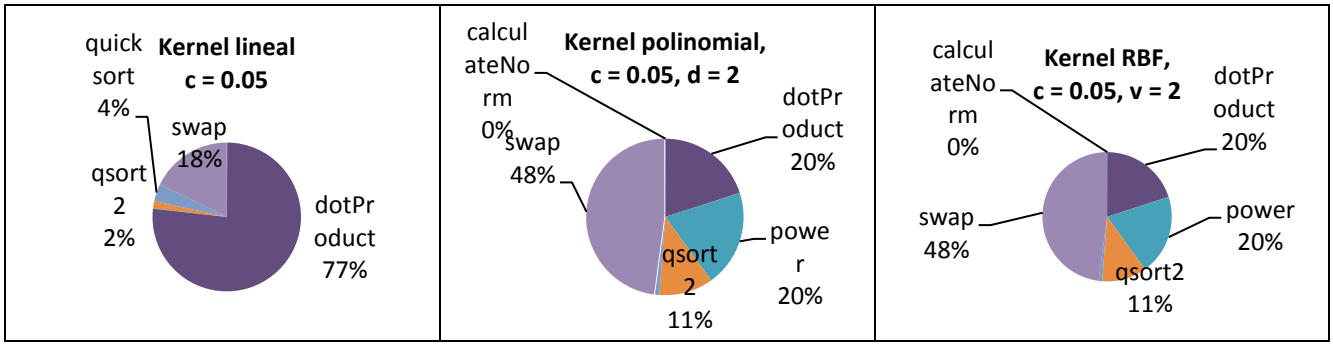


Fig. 12 Resultados del perfil de SMO para llamadas a funciones por cada kernels.

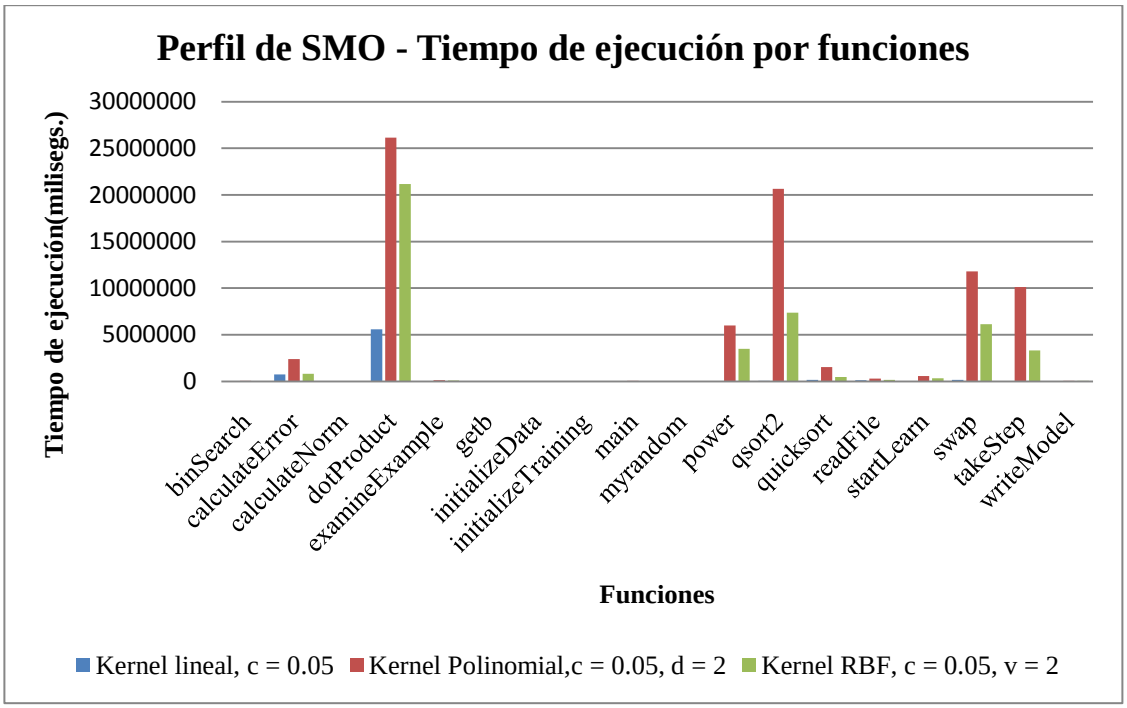


Fig. 13 Resultados del perfil de SMO para el tiempo de ejecución por funciones.

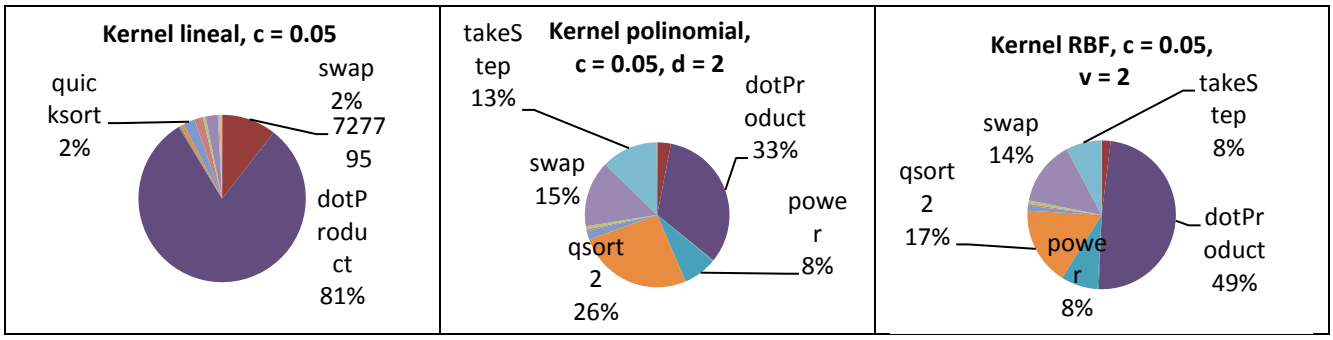


Fig. 14 Resultados del perfil de SMO para el tiempo de ejecución por funciones por cada kernel.

Del análisis de llamadas a funciones se puede observar que para el kernel lineal, el 77% de las llamadas (ver Fig. 11 y Fig. 12) corresponde a la función *dot_product*, mientras que para el resto de los kernels, la mayor cantidad de llamadas, con el 48% del total, se realiza al método *swap* el cual intercambia un elemento *a* por uno *b* en un arreglo mientras que para el kernel polinomial y RBF el método *dot_product* se llama el 20% del total. Esta función *swap* es propia de la implementación de SMO que se analizó y es un artificio para acelerar su ejecución, no está contemplada en el

pseudocódigo de SMO propuesto por Platt en [6]. Este análisis determina la función que más se ejecuta pero dicho resultado carecería de valor al de identificar la(s) función(es) que es (son) responsable(s) del cuello de botella del algoritmo. En las Fig. 13 y Fig. 14 se muestra el análisis del tiempo de ejecución de las funciones del algoritmo. Para el kernel lineal, el 81% del tiempo total de ejecución se emplea en ejecutar la función *dot_product* mientras que para el kernel polinomial y RBF, la misma función tarda el 33% y 49% respectivamente. Para estos dos últimos tipos de kernels, la función *swap* tarda menos del 15%, lo cual resulta ser un valor ligeramente elevado a efectos de identificar cuellos de botella en el algoritmo por lo que se tendrá en cuenta posteriormente. Algo similar ocurre con la función *qsort2*, la cual se tarda el 26% del tiempo total de ejecución para el kernel polinomial y el 17% del tiempo para el RBF. En el análisis del resto de las funciones, éstas no son relevantes en cuanto a tiempo o llamadas comparadas contra *dot_product.*

Del estudio anterior queda claro que las funciones que provocan el cuello de botella son *dot_product* y *qsort2*. La primera se encarga de calcular el producto punto de dos vectores de entrada mientras que la segunda ordena los elementos acotados entre un valor *i* por la izquierda y un valor *j* por la derecha de un arreglo, de tal manera que *array[i] < array[j]*. Estas dos funciones son las responsables de los elevados tiempos de procesamiento: la función *dot_product* contiene una multiplicación clásica de vectores que se puede ejecutar perfectamente en paralelo al igual que *qsort2*, pero a diferencia de esta última, *dot_product* calcula los valores del producto punto, los cuales no son requeridos de inmediato una vez invocada la función, lo que no ocurre con el resultado de *qsort2*, que sí es necesario para determinar por cuál rama de la jerarquía de heurísticas continuar en la determinación de las muestras a optimizar. Además, la función *qsort2* no está en el algoritmo SMO planteado por Platt en [6] sino que se utiliza en esta implementación de SMO para facilidad del programador. Dados estos hechos, lo que se lograse ganar en velocidad al ejecutar el método *qsort2* en hardware se perdería en la comunicación con el procesador. Por lo tanto, como el producto punto resulta ser siempre la función que más tiempo emplea SMO en

ejecutar, por ser parte original de SMO y por estar presente en todos los kernels es que resulta ser la función candidata para ejecutarse por hardware. La Fig. 11, Fig. 12, Fig. 13 y Fig. 14 respaldan esta conclusión.

Los resultados obtenidos con este perfil son perfectamente consistentes con los resultados alcanzados por Dey *et al.* en [33], el cual realizó un estudio semejante para SMO como primer paso de su investigación. El conjunto de prueba que se utilizó fue creado específicamente para este trabajo y tenía 2000 muestras y los resultados alcanzados se pueden ver en la Fig. 15.

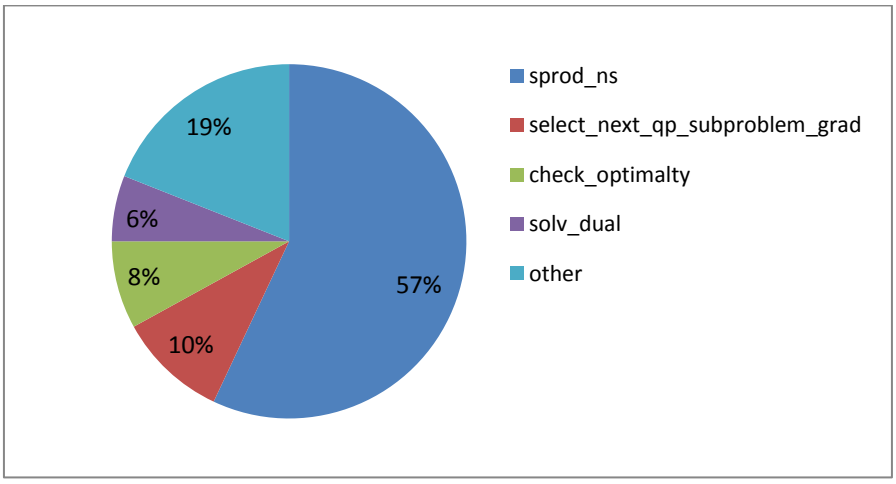


Fig. 15 Resultados obtenidos por Dey *et* al. en [33] para el análisis del perfil de SVM.

De este análisis se puede ver que más del 50% del tiempo total de ejecución es invertido en el procedimiento *sprod_ns*. Luego de analizar *sprod_ns* se comprobó que este procedimiento es el encargado de realizar el cálculo del producto punto: contiene operaciones de multiplicación y acumulación (MAC) en punto flotante de doble precisión dentro de un ciclo. Esta función es la responsable de los altos tiempos de procesamiento en el algoritmo.

De aquí se puede concluir que llevando a hardware la ejecución del cálculo del producto punto y dejar la ejecución de las estructuras de control del algoritmo en

software es posible que se pueda lograr una aceleración importante en el entrenamiento de SVM mediante el algoritmo SMO.

4.2 Descripción de la arquitectura del sistema.

Después de estudiar el algoritmo SMO, se llegó a la conclusión de que la mejor variante para acelerar el entrenamiento de SVM con SMO es mediante el diseño e implementación de una arquitectura hardware-software. De esta manera, en cada parte de la arquitectura se ejecutaran los procesos que mejores resultados puedan ofrecer aprovechándose de las ventajas de la plataforma sobre las que se ejecuten.

Para llevar a cabo este trabajo se tuvieron en cuenta una serie de consideraciones. El formato del archivo de entrenamiento se definió igual al que emplea LibSVM [35], logrando de esta manera compatibilidad entre los archivos de entrada para poder realizar comparaciones de resultados posteriormente. LibSVM es una librería especializada en la clasificación por vectores de soporte, regresión por vectores de soporte y estimación de distribuciones. Implementa el algoritmo propuesto por Fan *et. al.* en [36] que es una evolución de SMO y ha sido implementado en una amplia gama de lenguajes de programación demostrando su efectividad (ver [35] para mayor detalle). Los archivos de entrenamiento deben cumplir con formato presentado en la Fig. 16.

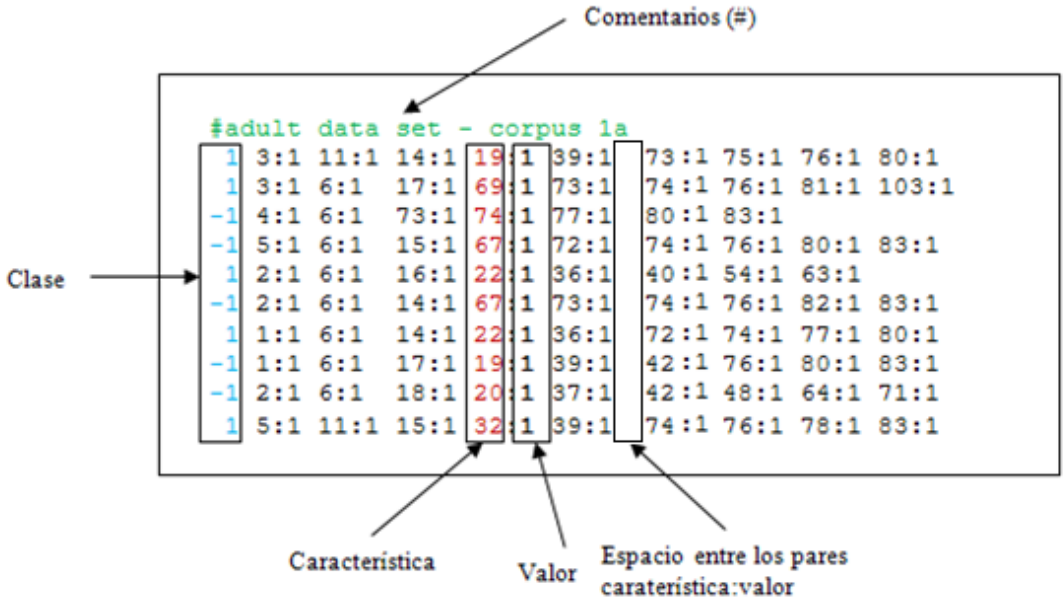


Fig. 16 Formato del archivo de entrenamiento.

Las líneas que se quieran establecer como comentarios deberán comenzar con el símbolo "*#*". Pueden existir tantos comentarios como se requiera. A continuación de los comentarios, debe seguirle las muestras de entrenamiento, las que deben comenzar con el valor de la clase a la que pertenece la muestra en cuestión. Estas clases podrán ser *1* ó *-1*. Después del valor de la clase, continúan los pares *característica:valor*, aquí no es necesario establecer todos y cada uno de estos pares, sino solamente aquellos cuyo valor sea distinto de cero. Visto en forma de tabla, el par *característica:valor* no es más que el valor de una muestra en una columna. Las muestras pueden tener diferentes longitudes y número de pares *característica:valor*. Pueden existir tantas muestras como sea necesario y el fichero no debe terminar con algún símbolo preestablecido. En esta tesis, las bases de datos que se analizarán para esta arquitectura serán binarias, es decir; sus columnas sólo contendrán valores 1 ó 0.

El archivo con el modelo generado igualmente tiene el mismo formato que el generado por LibSVM. Esto es:

- Línea primera: Valor del kernel empleado seguido del símbolo "#", indicando un comentario.
- Línea segunda: Número máximo de características del archivo de entrenamiento seguido por el símbolo "#" y un comentario.
- Línea tercera: Vector de pesos seguido por el símbolo "#" y un comentario. Esta línea se escribirá si se emplea el kernel lineal, en caso contrario no aparecerá.
- Línea cuarta: Valor del umbral del entrenamiento (parámetro *b*) seguido por el símbolo "#" y un comentario.
- Línea quinta: Valor del parámetro *C* seguido por el símbolo "#" y un comentario.
- Línea sexta: Número de vectores de soporte seguido por el símbolo "#" y un comentario.
- Línea séptima hasta el fin del archivo: para cada línea, el resultado de la multiplicación del valor de la clase por λ del vector de soporte seguido por los pares *característica:valor* del vector de soporte en cuestión.

En sentido general, la idea que se propone se muestra en la Fig. 17.

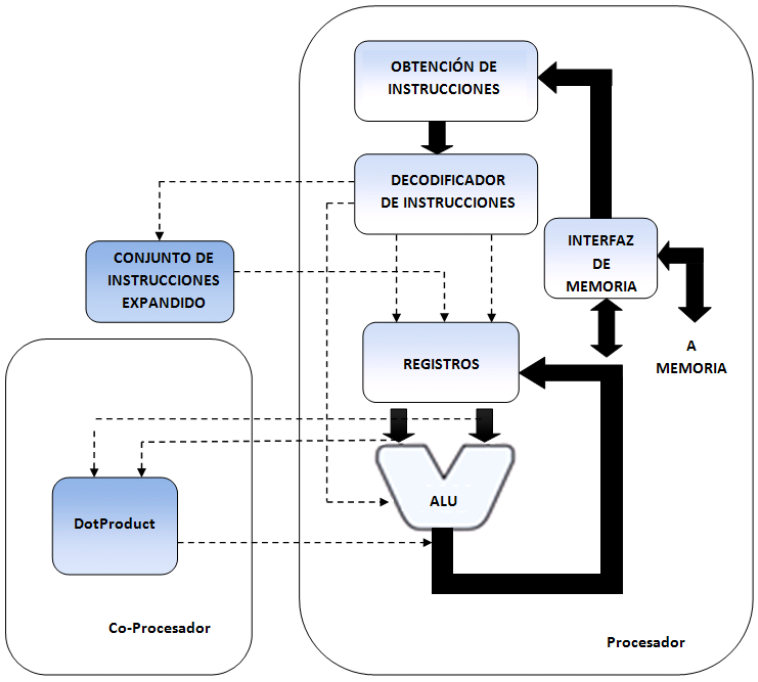


Fig. 17 Estructura general de la arquitectura propuesta.

La arquitectura estará formada por un procesador de propósito general que permita ampliar su funcionamiento mediante un conjunto de instrucciones extendido y un coprocesador donde las estructuras de control de datos del algoritmo se ejecuten en el procesador central y el cálculo del producto punto se realice en el coprocesador. La comunicación entre el procesador y el coprocesador se realizaría por un bus de expansión creado para estos efectos en lugar de emplear el bus PCI. El software en el procesador será el responsable de leer el archivo de entrenamiento e inicializar y configurar las estructuras de datos, permitiendo además recibir los parámetros para el funcionamiento del algoritmo. De esta manera, cuando el entrenamiento comienza, el procesador ejecuta los mecanismos de control y en el coprocesador las funciones que sean costosas en cuanto a tiempo y recursos computacionales. Para validar esta idea se recurrió a los FPGAs ya que los procesadores comerciales, como los de Intel, AMD, etc.; no permiten este tipo de desarrollos.

4.2.1 Sección a ejecutarse en software.

Para definir mejor el funcionamiento de cada módulo de la arquitectura, la sección que se ejecutará en el procesador tendrá las siguientes responsabilidades:

1. Cargar el archivo de entrenamiento para SVM y permitir la entrada de los parámetros del entrenamiento.
2. Ejecutar el algoritmo y que éste seleccione el conjunto de heurísticas correspondientes para el procesamiento del fichero de entrenamiento analizado.
3. Realizar los cálculos correspondientes en cada sección de la heurística seleccionada.
4. Entregar los vectores con los índices de las muestras al coprocesador para que éste pueda realizar el producto punto cuando sea requerido.
5. Obtener del coprocesador el resultado del cálculo del producto punto.
6. Generar el archivo con el modelo del entrenamiento.

El módulo que se ejecutará en el coprocesador tendrá las siguientes responsabilidades:

7. Permitir en sus memorias el almacenamiento de la matriz de entrenamiento.
8. Devolver las muestras de entrenamiento a partir de los índices de entrada.
9. Calcular el producto punto de las muestras de entrenamiento.

Para darle cumplimiento a estos requerimientos, la aplicación que implementa la arquitectura hardware-software se estructuró como muestra la Fig. 18.

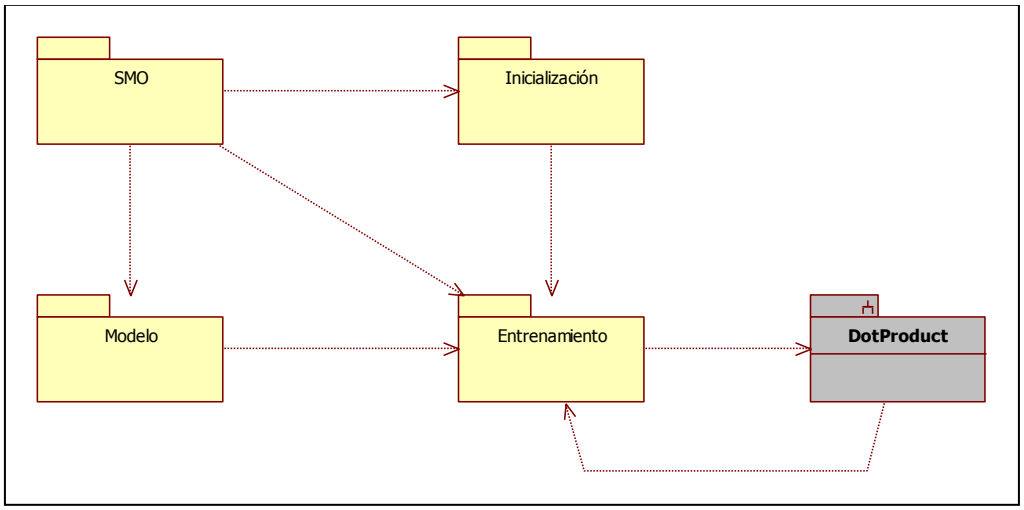


Fig. 18 Organización de los módulos de la arquitectura para acelerar el entrenamiento de SVM mediante SMO. En gris la sección que se ejecutará en el coprocesador.

SMO: Es el módulo principal del software, de donde se hacen todas las llamadas a los demás módulos y donde reside la interfaz de usuario que permite la selección de los parámetros para el funcionamiento del algoritmo. Es el encargado de dar cumplimiento a los requerimientos 1 y 2.

Inicialización: Este módulo es llamado por el módulo **SMO** y le corresponde la localización, inicialización y configuración del dispositivo hardware además de la inicialización de las variables y reservación de los espacios de memoria para el funcionamiento del software. Es el encargado de cumplir los requerimientos 1 y 7.

Entrenamiento: Aquí se realizan todos los procesos propios del algoritmo SMO. Se lee el archivo de entrenamiento y se crean las estructuras de datos necesarias para poder procesarlo. Según los parámetros de entrada y la naturaleza de los datos de entrada, en este módulo se encuentran los mecanismos para establecer la jerarquía de heurísticas necesarias para el entrenamiento. Le corresponde cumplir los requerimientos 2, 3, 4 y 5 además de permitir la comunicación desde y hacia el coprocesador.

DotProduct: Corresponde a la implementación en el coprocesador para el cálculo del producto punto, correspondiente a la sección más costosa dentro del algoritmo SMO. Este módulo responde a los requerimientos 7, 8 y 9.

Modelo: Este módulo es el encargado de generar el modelo del entrenamiento para ser empleado luego en la clasificación. Responde al punto 6 de los requerimientos.

Los procesos, flujos de datos y acciones en la arquitectura propuesta se muestran en el diagrama de secuencia representado en la Fig. 19.

En el diagrama se muestra la secuencia de acciones a realizar para entrenar SVM mediante SMO. El usuario indica la ejecución de SMO. Para ello accede a la interfaz del programa suministrando los parámetros del entrenamiento. Se inicializan las estructuras de datos y comienza el entrenamiento. En esta fase se calculan los productos puntos, se generan números aleatorios y se realizan búsquedas. Todas estas acciones se realizan para efectuar la optimización, y en el momento de terminar el entrenamiento se crea el archivo con el modelo de entrenamiento terminándose así la ejecución del software.

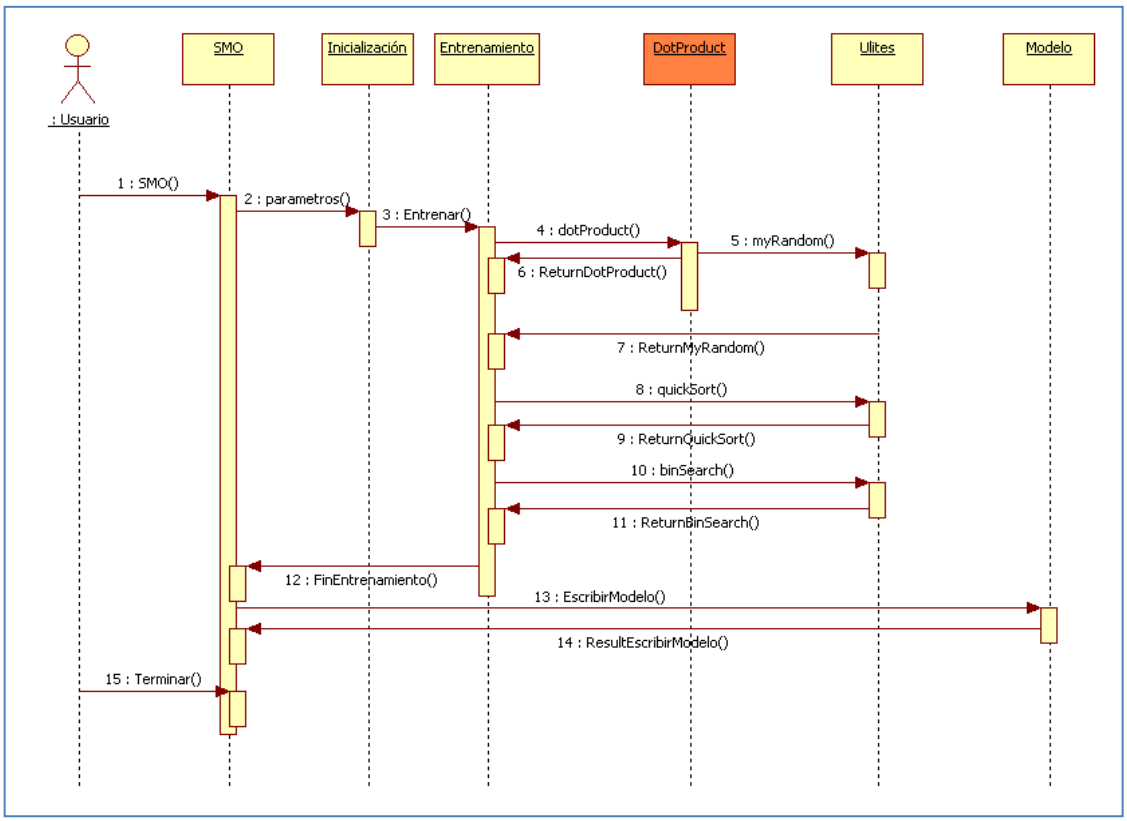


Fig. 19 Diagrama de secuencia para el entrenamiento de SVM mediante SMO.

4.2.2 Sección a ejecutarse en hardware.

Para el diseño e implementación de la sección de la aplicación que se ejecutará en hardware se propone la representación de los datos de entrenamiento como una matriz sin emplear ningún método de compresión o de representación y se requiere que el archivo de entrenamiento sea binario; es decir, que los valores de sus características sean 1 ó 0. De no ser así, deberá en un paso previo describirse los datos mediante este tipo de representación para que puedan ser empleados por la arquitectura. Debido a que el cálculo del producto punto es un proceso que se repite un elevado número de veces y que además los valores para dicho cálculo en el entrenamiento no varían, la estrategia adecuada para no incurrir en el retraso exagerado sería mapear la base de datos dentro del coprocesador. Este enfoque permite realizar el entrenamiento de matrices con un número fijo de renglones y columnas. En [37], [13] y [38] ofrecen variantes para entrenar matrices de datos que

no puedan ser procesadas con los algoritmos tradicionales debido al costo computacional y a los tiempos de entrenamiento que se incurrirían. Lo que se propone a grandes rasgos es descomponer la matriz de entrenamiento en varias sub-matrices con menor número de muestras que la original y realizar el entrenamiento por separados de estas sub-matrices de manera paralela y luego llegar a la solución global a partir de las soluciones particulares. Este enfoque se puede aplicar con la arquitectura propuesta aprovechándose las potencialidades de los FPGAs en la reconfiguración de circuitos, pagando solamente el costo del tiempo de reconfiguración del dispositivo y de la lógica de control que sería necesario implementar.

El cálculo del producto punto para esta representación de datos se puede expresar como:

$$dotProduct = \sum_{j=1}^{d} \boldsymbol{x_j} \cdot \boldsymbol{y_j}$$

donde $\boldsymbol{x_j}$ y $\boldsymbol{y_j}$ son los vectores de entrenamiento a los que se les desea calcular el producto punto y d es el número de componentes en los vectores.

La arquitectura digital que implementa esta expresión matemática consta de 5 bloques básicos:

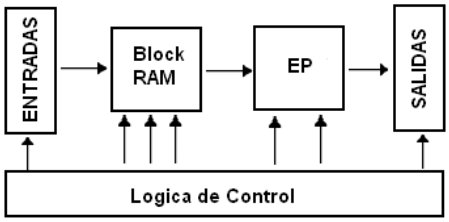


Fig. 20 Elementos principales de la arquitectura dotProduct.

donde:

ENTRADAS: Señales de control, registros y de datos que es necesario suministrarle a la arquitectura para su funcionamiento.

Block RAM: Bloque de memorias RAM que contienen la matriz de entrenamiento. Cada renglón del bloque de memorias corresponde con una muestra de entrenamiento, y cada columna corresponde con una característica.

EP: Elemento procesador que se encarga de calcular el producto punto de dos vectores.

SALIDAS: Elementos de salida de la arquitectura con el resultado del producto punto.

LÓGICA DE CONTROL: Todos aquellos elementos que permiten el flujo y control de datos dentro de la arquitectura.

Mediante las ENTRADAS se obtienen los índices de los vectores que tendrán parte en el cálculo del producto punto y además, también se utiliza en el mapeo de la matriz de entrenamiento en el Bloque RAM de la arquitectura. A continuación ya se dispondrá de la matriz de entrenamiento a partir de la cual se obtendrán las dos muestras de entrenamiento cuyos índices son indicados por ENTRADAS. Estos vectores llegan a Elemento Procesador (EP) donde se realiza el cálculo del producto punto y luego, el resultado se devuelve a través de SALIDAS.

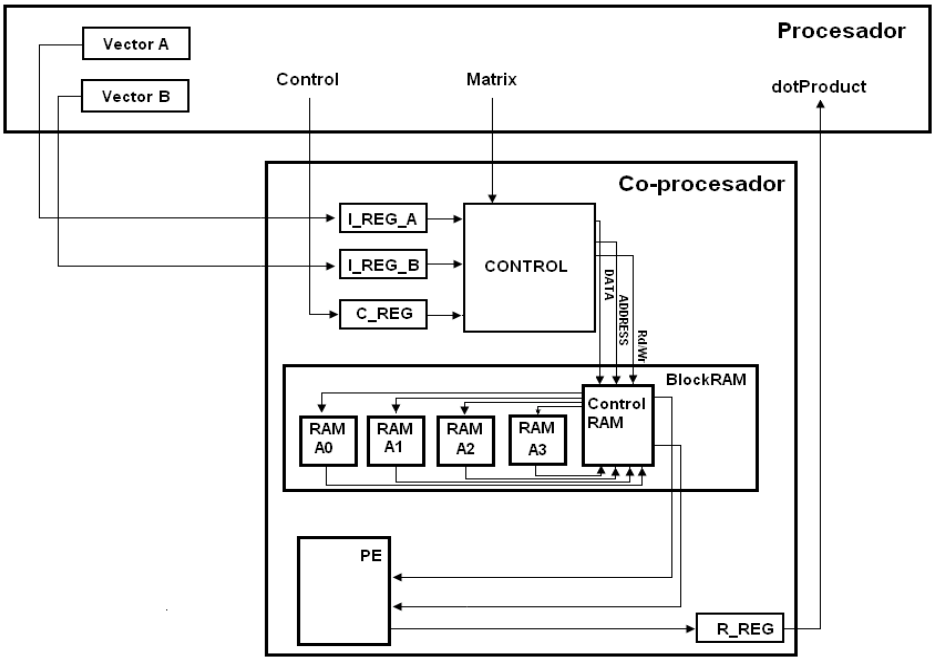


Fig. 21 Estructura general de la arquitectura propuesta.

Con el fin de estructurar mejor los datos y manejar adecuadamente la memoria del coprocesador, se creó el siguiente mapa de memoria donde residirá la matriz de entrenamiento:

Tabla 2 Mapa de memoria de la arquitectura *dotProduct*.

Elemento	Dirección		Descripción
	Inicio	Fin	
RAM_A0	0x2000	0x2FFF	RAM que almacena los 32 bits más significativos (31-0) de la palabra a almacenar.
RAM_A1	0x3000	0x3FFF	RAM que almacena los bits de 63 a 32 de la palabra a almacenar.
RAM_A2	0X4000	0X4FFF	RAM que almacena los bits de 97 a 64 de la palabra a almacenar.
RAM_A3	0x5000	0x5FFF	RAM que almacena los 32 bits menos significativos palabra a almacenar (127-98).
C_REG	0x6800	0x6800	Registro de control de la arquitectura.
I_REG_A	0x6801	0x6801	Registro que almacena el valor del primer vector a intervenir en el producto punto.
I_REG_B	0x6802	0x6802	Registro que almacena el valor del segundo vector a intervenir en el producto punto.
R_REG	0x6803	0x6803	Registro que almacena el resultado del producto punto cuanto esté disponible.

A modo de complemento, la tabla anterior se puede representar gráficamente como:

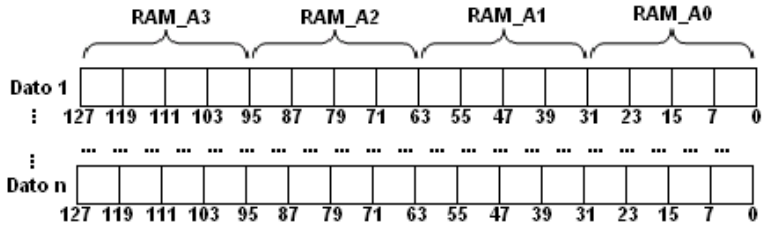


Fig. 22 Representación de los datos de 128 bits en el bloque de RAM.

donde los 32 bits más significativos se mapean en la RAM A0, los próximos 32 bits menos significativos se mapean en la RAM A1 y así sucesivamente para cada una de las muestras de entrenamiento. El bloque de RAM de la arquitectura permite mapear hasta 4096 palabras por lo que se necesitarían 12 bits para direccionar cada una de las muestras de entrenamiento. Los registros I_REG_A y I_REG_B almacenarán las direcciones de las muestras de entrenamiento que participarán en el cálculo del producto punto. Estos registros son de 16 bits lo que permite direccionar las 4096 palabras que contiene el bloque de RAM y aún queda espacio por si se desea ampliar la cantidad de palabras que se almacenarán en las RAM. C_REG controla la arquitectura indicando cuándo comenzar a cargar los datos a la RAM, cuándo leer de la RAM y cuándo comenzar a calcular el producto punto. En la Fig. 23 se muestra su comportamiento:

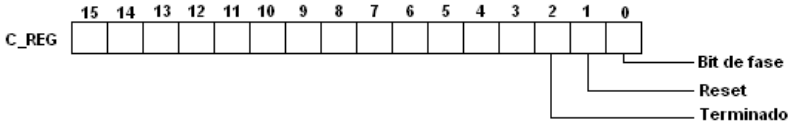


Fig. 23 Especificación del registro de control C_REG.

- **Bit de fase:** indica si la arquitectura se encuentra en la fase de carga (toma valor 0) o en la fase de procesamiento (toma valor 1). Más adelante se explicará en qué consisten estas fases.
- **Reset:** Cuando se activa este bit, los registros de direcciones toman valor cero, se establece la fase de carga y el EP se encuentra listo para comenzar a procesar un nuevo dato. Es activo en 1.
- **Terminado:** Bit que indica el fin del procesamiento por EP. Activo en 1.

De manera general la arquitectura propuesta funciona en dos fases. En la primera fase, o de inicialización y carga, se inicia el dispositivo y se mapea en los bloques de RAM la matriz con los datos de entrenamiento. En la segunda fase, o fase de procesamiento, la arquitectura recibe los índices de las muestras de entrenamiento con las que se obtendrá el producto punto y ejecuta el cálculo.

La fase de inicialización y carga se activa al comenzar la ejecución del programa. Para ello se establece el bit de fase de C_REG en cero. Cuando esto ocurre, el control de la arquitectura inhabilita I_REG_B y conecta el bus de direcciones del bloque de RAM a I_REG_A para indicar la posición en que se escribirá el valor que contiene *matrix* (ver la Fig. 21 para mayores referencias). De esta manera se garantiza la inicialización del bloque de RAM. Una vez terminada esta fase, se mantendrá en este estado mientras el bit de fase permanezca en cero; cuando se active en 1, el control de la arquitectura desconecta *matrix*, conecta I_REG_B al bloque de RAM y se entregan las muestras de entrenamiento indicadas por I_REG_A y I_REG_B al EP que se encarga de realizar el producto punto. Una vez que se alcanza el resultado, es almacenado en R_REG y se activa el bit Terminado de C_REG y ya la arquitectura se encuentra lista para procesar nuevos datos.

El EP es el que lleva a cabo el cálculo del producto punto de dos muestras de entrenamiento dadas. Para las condiciones de las matrices de entrenamiento que se manejan, las muestras de entrenamiento serán vectores cuyas componentes tendrán valores 1’s ó 0’s. Realizar el producto punto de este tipo de vectores se reduce a

aplicar una operación lógica AND a los dos vectores de entrada y posteriormente contar la cantidad de 1´s que tiene el vector resultante:

101011100010101110
AND 111011101110101000
10**1**0**111**000**1**0**1**000
8 <= resultado del producto punto.

De esta manera, la arquitectura del elemento procesador se puede ver en la Fig. 24.

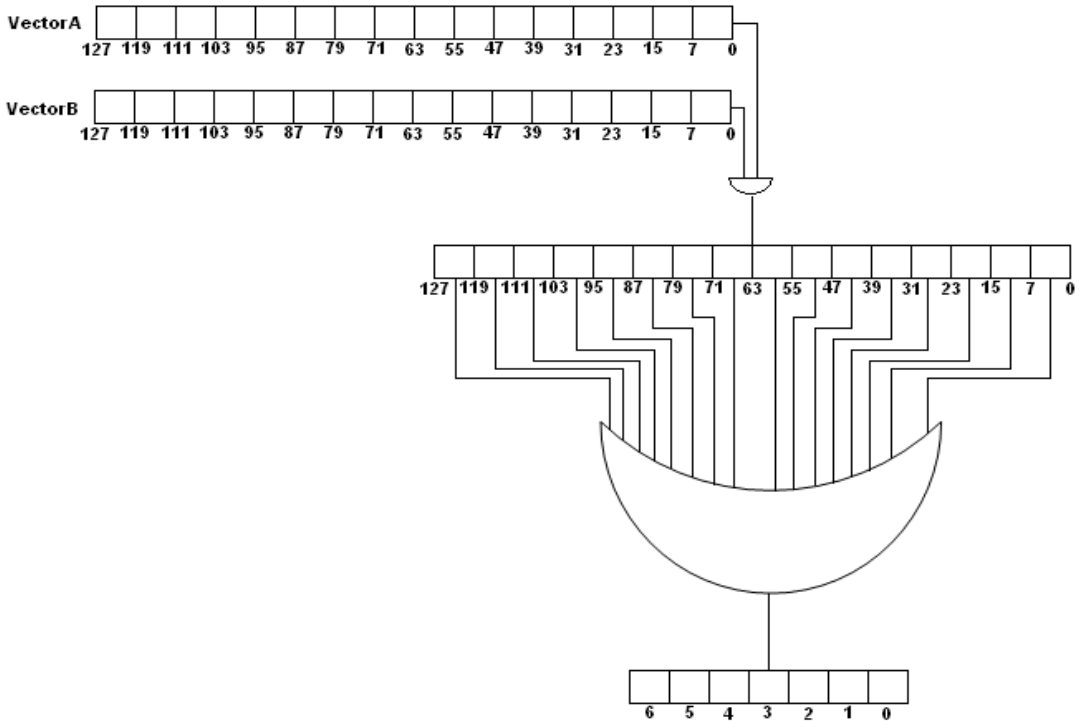


Fig. 24 Estructura general del Elemento Procesador.

Hay que destacar que el EP es capaz de realizar el cálculo en tres ciclos de reloj. De esta manera, el tiempo de procesamiento estará dado por:

$$t = 3 \times v$$

donde v es el número de productos puntos que se deben realizar. El proceso de mapeo de la matriz de entrenamiento hacia la RAM tardará tantos ciclos de reloj como

muestras tenga la matriz de entrenamiento aunque se debe destacar que este mapeo se realiza una sola vez al inicio de la aplicación. De esta manera se puede ver que el mismo cálculo del producto punto en software se descompone en un conjunto de corrimiento de valores y de comparaciones, mientras que con la arquitectura propuesta se obtiene el mismo resultado con un arreglo de compuertas AND y OR en cascada las que son más eficientes desde el punto de vista de hardware.

4.3 Arquitectura de validación.

Para probar el funcionamiento correcto de la arquitectura propuesta se empleó un FPGA sobre el que se implementó el coprocesador siguiendo la idea que se plantea en la arquitectura original. En la Fig. 25 se muestra la estructura general de la arquitectura de validación:

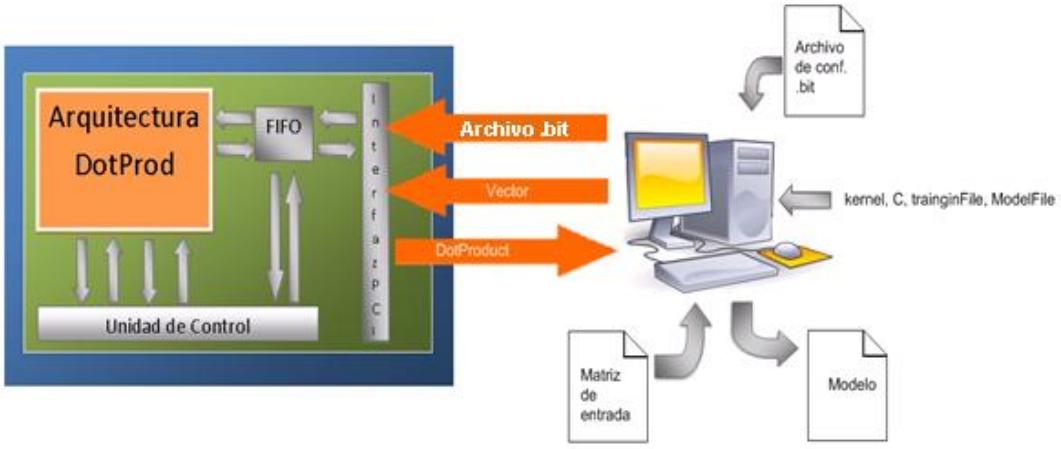


Fig. 25 Arquitectura general para acelerar el entrenamiento de SVM mediante SMO.

El funcionamiento a seguir por la arquitectura de validación es idéntico al propuesto para el coprocesador acoplado al procesador central. De esta manera el software resultante se ejecutará desde la consola de Windows, en el procesador central y la estructura del comando de invocación será:

```
smo.exe [opciones] <fichero de entrenamiento> <fichero modelo>
```

Las opciones que se le indican al software son:

- **`-t`**: tipo de kernel a emplear.
  - **`0`**: kernel lineal.
  - **`1`**: kernel Polinomial.
  - **`2`**: kernel RBF.
- **`-c`**: Parámetro *C*. Valor real que corresponde con el consenso entre la especialización y la generalización de la SVM.
- **`-d`**: Grado del polinomio si el kernel seleccionado fue polinomial.
- **`-v`**: Varianza si el kernel seleccionado fue el RBF.

Para la programación de la arquitectura de validación (arquitectura *dotProduct*) se empleó el lenguaje VHDL y la suite de desarrollo ISE 9.2 de Xilinx, con ModelSIM SE 6.5 como simulador e implementada en la tarjeta XtremeDSP Virtex IV. La aplicación software fue desarrollada con Microsoft Visual C++ 6.0 y como lenguaje de programación se utilizó ANSI C. Con el FPGA empleado, el fabricante distribuye las APIs necesarias para establecer la comunicación y el flujo de datos entre el FPGA y las aplicaciones en software que lo manejen.

Capítulo

5 Experimentos y resultados alcanzados

Luego de estudiar las bases teóricas de SVM se seleccionó el algoritmo que mejores resultados presenta en el entrenamiento en cuanto al uso recursos computacionales y tiempo de ejecución. De este algoritmo se realizó un estudio del desempeño identificando las secciones que provocan los altos tiempos de entrenamiento. Se logró determinar que el cálculo del producto punto ocupa entre el 33% (cuando se realiza el entrenamiento empleando un kernel polinomial) y el 81% (cuando se emplea un kernel lineal) y se validaron estos resultados con otros estudios reportados en la literatura [33]. Tomando como base estos hechos se diseñó e implementó una arquitectura hardware-software para acelerar la fase de entrenamiento de SVM donde las estructuras de control y de toma de decisiones se realizan en software mientras el cálculo del producto punto se realiza en hardware. Para demostrar la validez de la idea se empleó una tarjeta XtremeDSP Virtex IV sobre la que se implementó el módulo que calcula el producto punto mientras que en C se implementó el módulo de software. Para realizar los experimentos se empleó el

conjunto de datos *Adult* [28], específicamente el empleado por Platt en [6]. Por las características de *Adult*, se emplea mayormente en tareas de clasificación de datos y contiene valores categóricos y enteros en sus atributos. Presenta 14 atributos o características de los cuales 8 son categóricos y 6 son continuos; y 48842 instancias o muestras con valores faltantes. Las 6 características con valores continuos fueron representadas en forma de matriz binaria y se obtuvieron en total 123 características de valores binarios dispersos. Las 48842 instancias en el artículo de Platt fueron separadas en 9 archivos de entrenamiento de tamaños variables los cuales se describen en la Tabla 3. En este trabajo se emplea este conjunto de datos para comparar los resultados obtenidos con las especificaciones del propio conjunto de datos, con los resultados obtenidos por Platt y para reportar los resultados alcanzados frente a un conjunto de datos que es de los más empleados en tareas de clasificación.

Tabla 3 Descripción de la base de datos *Adult*.

Nombre	Archivo	# Muestras	# Características
Adult-1	a1a	1605	123
Adult-2	a2a	2265	123
Adult-3	a3a	3185	123
Adult-4	a4a	4781	123
Adult-5	a5a	6414	123
Adult-6	a6a	11221	123
Adult-7	a7a	16101	123
Adult-8	a8a	22697	123
Adult-9	a9a	32562	123

El segundo conjunto de entrenamiento empleado es un conjunto artificial creado para este trabajo. Consta de 12 muestras de entrenamiento con 14 características cuyos valores son valores 0 ó 1 dispersos y fue creado con el objetivo de validar los resultados alcanzados con LibSVM. Al ser un conjunto pequeño resulta fácil realizar un seguimiento de la ejecución de su entrenamiento con el fin de determinar patrones de comportamiento característicos de SMO.

5.1 Perfil del producto punto.

El cálculo del producto punto es la sección responsable de los tiempos elevados de entrenamiento de SMO. Para identificar la magnitud de este problema, se aisló esta operación y se realizaron varias pruebas para determinar el tiempo que tarda en completarse dicho cálculo para vectores de longitudes variables y valores binarios aleatorios. De esta manera, se realizó la implementación en C del cálculo clásico del producto punto, el cual está dado por:

$$dotProduct = \sum_{j=1}^{d} \boldsymbol{x_j} \cdot \boldsymbol{y_j}$$

La implementación de esta operación en software se puede ver en el Código 1:

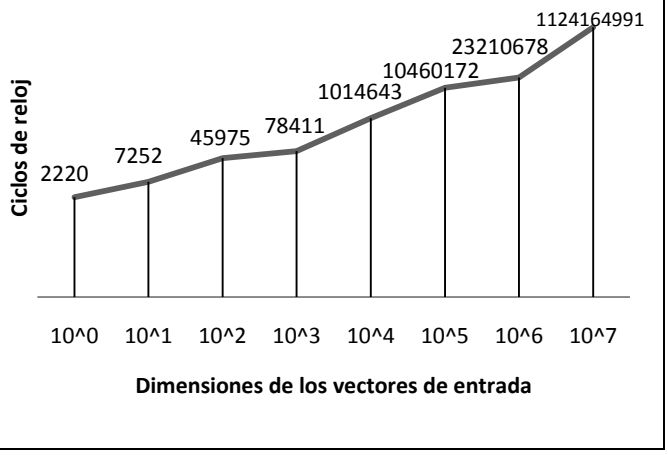
```
int dotProduct(int *vectA, int *vectB, int lenght )
{
    int dot = 0;
    int i = 0;

    for ( i = 0; i < lenght; i++)
    {
          dot += vectA[i] * vectB[i];
    }

    return dot;
}
```

Código 1 Implementación clásica del producto punto.

Se realizaron 8 experimentos con el fin de identificar el tiempo empleado en calcular el producto punto en un procesador Pentium 4 a 3GHz de frecuencia y los resultados se muestran en la Tabla 4.

Tabla 4 Perfil del cálculo del producto punto para varias longitudes de vectores de entrada por software.

No	Tamaño del vector	Tiempo empleado (C.C.[5])
1	1	2220
2	10	7252
3	10^2	45957
4	10^3	78411
5	10^4	1014643
6	10^5	10460172
7	10^6	23210678
8	10^7	1124164991



De estos resultados resalta que los ciclos de reloj requeridos para el cálculo del producto punto crecen directamente proporcionales al tamaño de los vectores de entrada. Debido a que la convergencia de SMO depende de que todas las muestras del conjunto de entrenamiento cumplan las condiciones de KKT y no se pueda saber a priori en qué momento se logra la convergencia, en la Tabla 1 (Capítulo 4) se puede notar que el número de veces que se repite el producto punto en el proceso de entrenamiento de una SVM puede estar en el orden de las decenas de miles hasta millones, por lo que el entrenamiento de SVM para bases de datos grandes sería demasiado lento.

En hardware, para calcular el producto punto, el tiempo requerido es independiente del tamaño de los vectores de entrada. La arquitectura *dotProduct* puede trabajar con vectores de hasta 128 componentes. Este número se seleccionó debido a que corresponde con un valor lo suficientemente grande para contener a la mayoría de las bases de datos del repositorio UCI y a la vez lo suficientemente pequeño como para seguir la ejecución del entrenamiento por software con relativa facilidad. A efectos de esta arquitectura, el tiempo empleado en el cálculo del producto punto de vectores de

[5] C.C.: Abreviaturas en inglés de *Clock Cycles*, o Ciclos de Reloj.

hasta 128 componentes es independiente del tamaño de los vectores de entrada. En la Fig. 26 se muestra el *datapath* a seguir en el cálculo del producto punto:

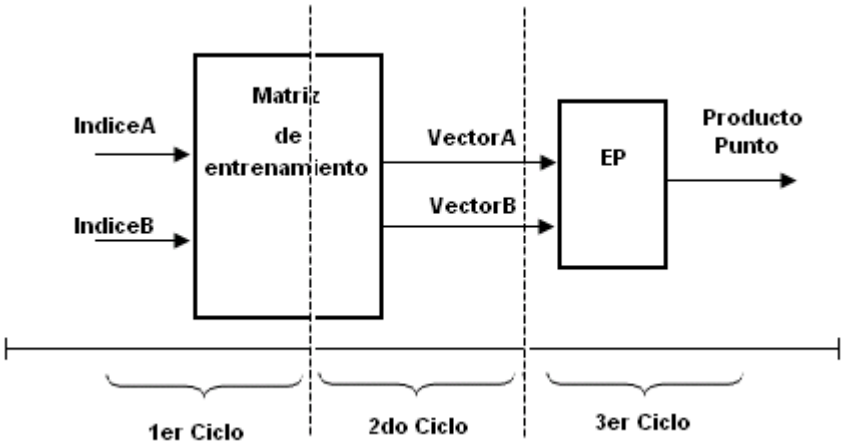


Fig. 26 *Datapath* del cálculo del producto punto.

Como se muestra en la Fig. 26, con la arquitectura propuesta calcula el producto punto de vectores de 128 componentes en 3 ciclos reloj. Para poder procesar vectores mayores se tendrían que ampliar la memoria y el EP tanto como se requiera o como lo permita el área del FPGA empleado; o conectar varios bloques que contendrían la arquitectura como se muestra en la Fig. 27. Otra manera para enfrentar esta limitante sería realizar un proceso iterativo de reuso de recursos explotando una de las principales ventajas de los FPGAs de manera tal que sobre el mismo hardware se pudiera implementar varias arquitecturas en dependencia de la fase de procesamiento en que se encuentre la realización del entrenamiento. Esta variante queda fuera del alcance de este trabajo. En la Tabla 5 se resumen los tiempos que se pueden alcanzar en el cálculo del producto punto para vectores de diferentes longitudes.

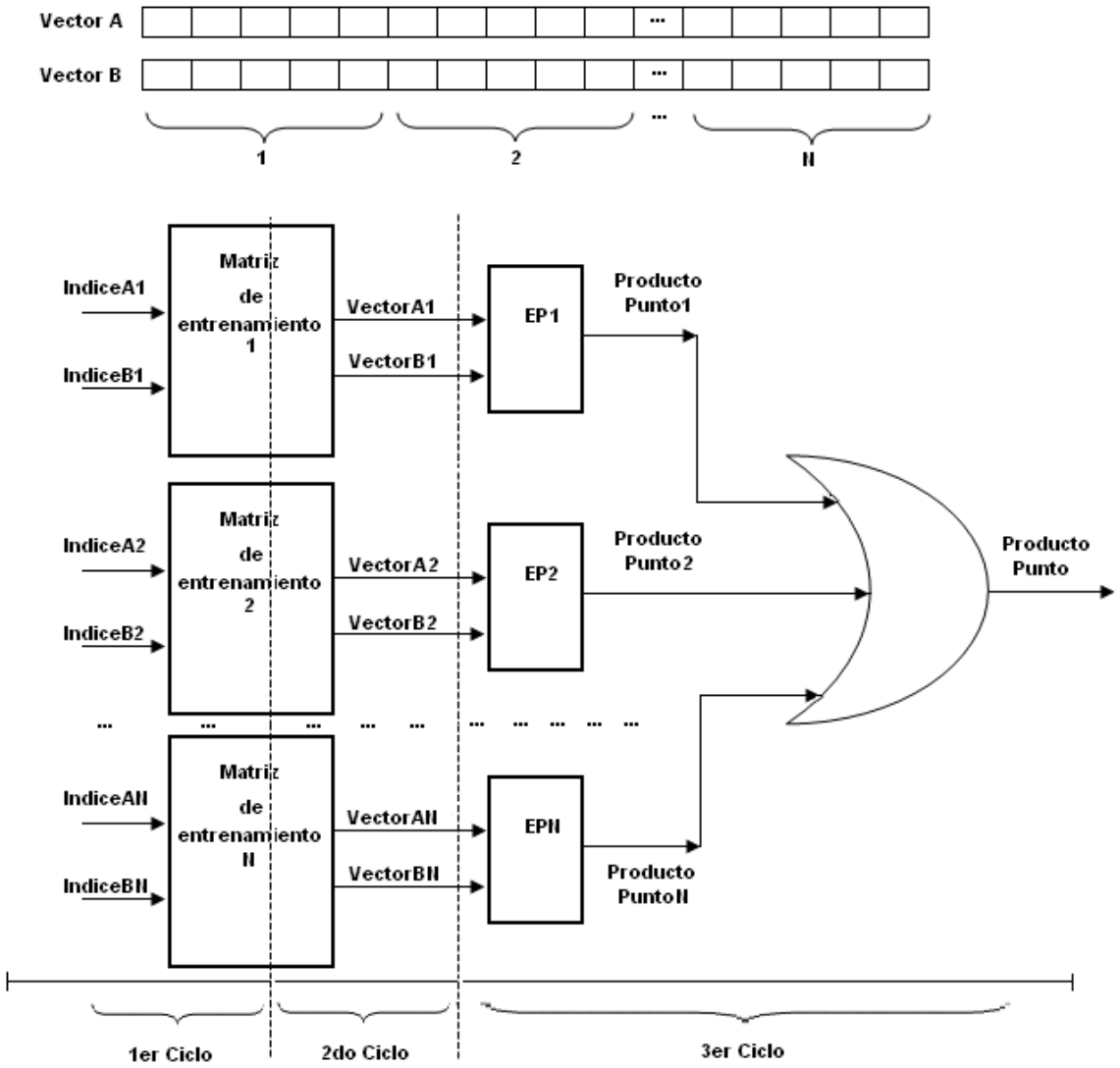
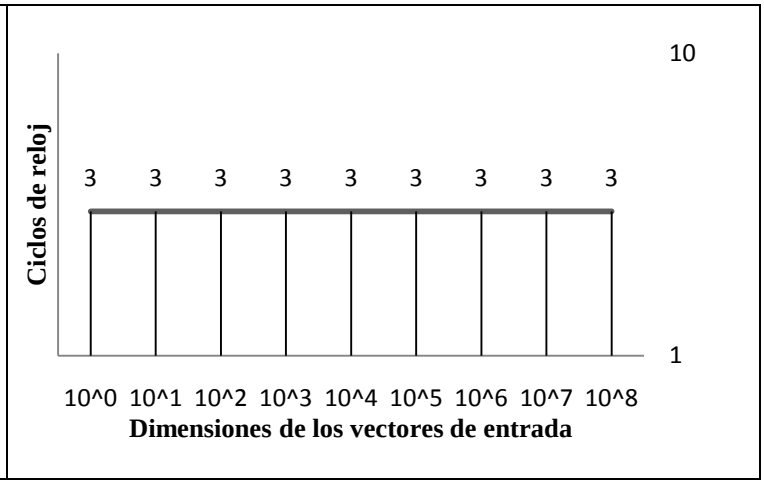


Fig. 27 Arquitectura para ampliar el cálculo del producto punto.

Tabla 5 Resultados del perfil para el cálculo del producto punto para varias longitudes de vectores de entrada por hardware.

No	Tamaño del vector	Tiempo empleado (C.C.)
1	1	3
2	10	3
3	10^2	3
4	10^3	3
5	10^4	3
6	10^5	3
7	10^6	3
8	10^7	3



De esta manera, el producto punto de vectores de 128 bits empleando la arquitectura *dotProduct* tardaría 3 ciclos de reloj mientras que el mismo cálculo por software estaría entre los 45957 y 78411 ciclos de reloj.

5.2 Entrenamiento de SMO.

Para demostrar la validez de la arquitectura propuesta se realizaron experimentos con el conjunto de datos artificial y con el conjunto de datos *Adult*. El objetivo de llevar a cabo los experimentos con el conjunto artificial está en la comparación de los resultados alcanzados por la implementación de la arquitectura en un software de aplicación (FSMO) con los resultados que ofrece LibSVM. De esta manera se verifica que la implementación que se propone entrena correctamente una SVM. Los experimentos desarrollados con el conjunto de datos *Adult* se realizaron para comparar los resultados obtenidos contra los reportados en [6], para medir los tiempos de ejecución y verificar que se cumplen los objetivos.

Por otro lado, con el fin de demostrar la viabilidad de llevar a hardware las secciones responsables de los cuellos de botella de algoritmos por software, específicamente de SMO para SVM, y luego permitir la comunicación y flujo de datos entre la PC y el FPGA es que se seleccionó la tarjeta XtremeDSP Virtex IV,

específicamente esta tarjeta cuenta con el dispositivo XC4SX35-10FF668 de Xilinx [39]. Este FPGA está diseñado para ofrecer una plataforma de desarrollo completa para aplicaciones de análisis de señales donde se requiera la velocidad que el hardware le imprime a sus procesos y además está completamente soportado por System Generator para el diseño y cosimulación de circuitos. Con el FPGA, el fabricante distribuye además un conjunto de APIs y núcleos hardware que permiten la comunicación de control y datos vía bus PCI con la PC para el rápido y fácil desarrollo de aplicaciones hardware-software. Estas dos características hacen de esta tarjeta la adecuada para la validación y prueba de la arquitectura hardware-software que se propone.

5.2.1 Experimentos con el conjunto artificial.

Se empleó a LibSVM, SMO y FSMO para entrenar el conjunto artificial con un kernel lineal y C = 4.5 y para demostrar la efectividad de la arquitectura propuesta frente a estas 2 implementaciones. Los resultados alcanzados se pueden ver en la Tabla 6.

Tabla 6 Resultados del entrenamiento de SVM para un kernel lineal con C = 4.5.

	Umbral *b*	**Vector *w***	**SV no fronteras**	**SV fronteras**	**Total SV**	**Iteraciones**
LibSVM	0.762556	NA	10	0	10	53
SMO(soft)	0.763204	4.215593	11	0	11	123
FSMO(hard)	0.763204	4.215593	11	0	11	123

Del experimento se puede ver que se alcanzan resultados idénticos en las implementaciones de SMO por software y por hardware lo que demuestra la exactitud de los procesos desarrollados por esta implementación. Sin embargo, no se obtienen exactamente los mismos resultados cuando se compara con LibSVM. La convergencia se logra con LibSVM en sólo 53 iteraciones mientras que con FSMO se necesitan 123. El motivo de estas diferencias está en que la LibSVM no implementa exactamente el algoritmo SMO, sino una evolución del mismo [36]. Otro factor que se debe tener en cuenta a la hora de analizar la convergencia es el valor seleccionado

para *épsilon*: en FSMO se empleó la *épsilon* de la PC. En la Tabla 7 se muestra la desviación que se obtiene entre FSMO y LibSVM.

Tabla 7 Desviación de los resultados de FSMO respecto a LibSVM.

Corpus	Umbral *b*		Diferencia	Porcentaje
	FSMO	LibSVM		
conjunto artificial	0.763204	0.762556	0.000648	0.084

La desviación que se obtiene no resulta significativa pero sí trae consigo posibles diferencias en la clasificación de nuevas muestras. El número de vectores de soporte obtenidos con LibSVM es menor que los que se obtienen con FSMO y ésta es la causa de las posibles diferentes clasificaciones, lo cual está dado porque los vectores de soporte son obtenidos en dependencia de la orientación del hiperplano de separación y es el umbral *b* quien refleja esta orientación. Al obtenerse diferentes valores de *b* la inclinación del hiperplano separador no será la misma y esto traerá consigo la determinación como vectores de soporte de muestras que no lo son y la no determinación de muestras que sí lo son. Los valores diferentes de *b* están determinados por el número de iteraciones, la *épsilon* de la PC y de optimizaciones en la programación y estructuras de datos empleadas.

Debido a que FSMO y LibSVM implementan algoritmos diferentes se puede decir que un 0.084% de desviación entre los valores del umbral *b* es un resultado correcto. Por otra parte, las pruebas realizadas a la arquitectura durante su implementación demostraron que los resultados del cálculo del producto punto por hardware es exactamente igual al obtenido por software. Esto demuestra que las diferencias obtenidas en el entrenamiento por FSMO y LibSVM están dadas netamente en la implementación del módulo que se ejecutará en software.

5.2.2 Experimentos con el conjunto *Adult.*

Una vez que se demostró que la implementación del producto punto por hardware y por software ofrecen los mismos resultados, y que el entrenamiento con ambas técnicas es igual, se experimenta entonces los resultados prácticos y de tiempo. Con

este fin se seleccionó el conjunto *Adult* el que además fue empleado por Platt en [6] para demostrar la validez de SMO.

Debido a problemas de disponibilidad de área en el FPGA seleccionado, solamente se podrán procesar matrices de 4096 objetos de 128 características. Se empleó la tarjeta XtremeDSP Virtex IV debido a las ventajas que presenta en cuanto a facilidad de desarrollo y de comunicación con la PC para de mostrar la validez de la idea que se propone. En caso de que se necesite procesar datos de mayores dimensiones que permitidos, basta con implantar la arquitectura en un dispositivo de mayores prestaciones o implementar un método incremental donde se reuse la arquitectura para resolver problemas de mayor tamaño.

De los 9 corpus que presenta el conjunto de datos *Adult*, solamente *Adult-1*, *Adult-2* y *Adult-3* tienen dimensiones que pueden ser manejados por la arquitectura *dotProduct*.

Tabla 8 Resultados del entrenamiento de *Adult* con el software SMO para el kernel lineal y $C = 0.05$.

Corpus	Objs.	Iter.	Tiempo de entrenamiento		Umbral b	S.V. no fronteras	S.V. fronteras	Total S.V.
			segs	C.C.				
Adult-1	1605	3474	8	NA	0.887279	48	631	679
Adult-2	2265	4968	16	NA	1.129381	50	929	979
Adult-3	3185	5850	29	NA	1.178716	58	1212	1270

Tabla 9 Resultados del entrenamiento de *Adult* con el software FSMO para el kernel lineal y $C = 0.05$.

Corpus	Objs	Iter.	Tiempo de entrenamiento		Umbral b	S.V. no fronteras	S.V. fronteras	Total S.V.
			segs	C.C.(10^{12})				
Adult-1	1605	3474	364	1.08974	0.887279	48	631	679
Adult-2	2265	4968	746	2.23287	1.129381	50	929	979
Adult-3	3185	5850	1218	3.62842	1.178716	58	1212	1270

Tabla 10 Resultados del entrenamiento de *Adult* reportados por Platt en [6] para el kernel lineal y $C = 0.05$.

Corpus	Objs	Iter.	Tiempo de entrenamiento		Umbral b	S.V. no fronteras	S.V. fronteras	Total S.V.
			segs	C.C.				
Adult-1	1605	3230	0.4	NA	0.88499	42	633	675
Adult-2	2265	4635	0.9	NA	1.12781	47	930	977
Adult-3	3185	6950	1.8	NA	1.17302	57	1210	1267

En la Tabla 8, Tabla 9 y Tabla 10 se muestran los resultados alcanzados por las implementaciones de SMO por software, SMO acelerado por hardware y SMO reportado por Platt. Igual que el caso del conjunto de datos artificial ocurre una desviación entre los datos obtenidos entre los resultados alcanzados por Platt y los que se obtienen mediante la implementación del algoritmo. Las causas de esta desviación son las mismas que las explicadas para el caso del conjunto artificial. Por otro lado, para hacer las comparaciones se utilizaron los resultados reportados en [6] y en este trabajo no se da ningún detalle de implementación más allá de la herramienta de programación y el compilador empleado. Por lo tanto, no se puede llegar a conclusiones con estas bases sobre las optimizaciones al software para obtener los resultados que Platt presenta. En la Tabla 11 se muestran las desviaciones que se incurren con cada uno de los corpus empleados y los valores alcanzados por Platt.

Tabla 11 Desviaciones del valor del umbral b obtenidos entre FSMO y el SMO reportado por Platt.

Corpus	**Umbral *b***		**Diferencia**	**Porcentaje**
	FSMO	**SMO**		
Adult-1	0.887279	0.88499	0.002	0.257
Adult-2	1.129381	1.12781	0.0015	0.139
Adult-3	1.178716	1.17302	0.005	0.483

Como se pude ver de la Tabla 11, la desviación que se incurre es de menos del 0.5% en el peor de los casos.

De manera general, la arquitectura hardware-software FSMO entrena adecuadamente una SVM en cuanto a los resultados de entrenamiento alcanzados. Sin embargo, los tiempos de procesamiento son muy diferentes entre las diferentes implementaciones. Los tiempos de entrenamiento se representan en la Fig. 28.

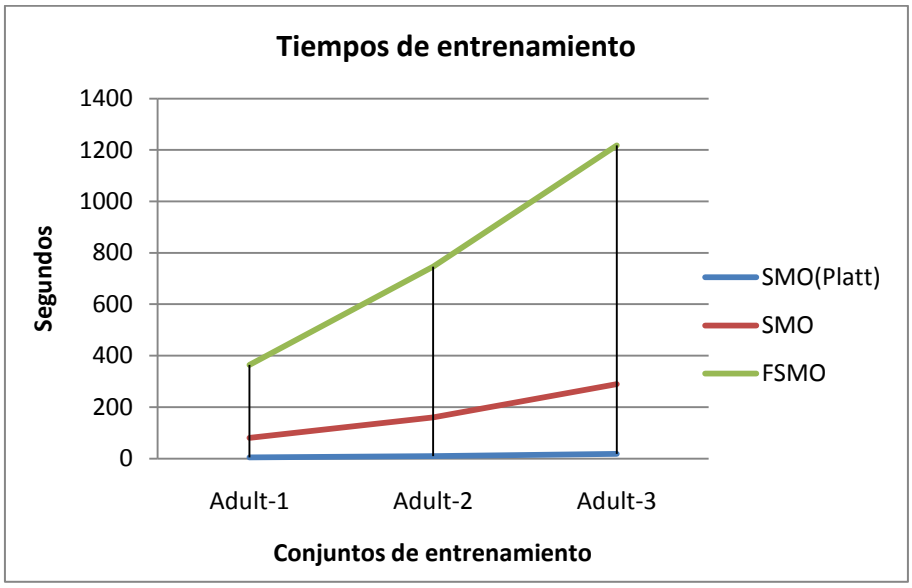


Fig. 28 Tiempos de entrenamiento alcanzados con las 3 implementaciones de SMO analizadas.

De la Fig. 28 se puede observar que la variante por hardware es la más lenta de todas las implementaciones. Con el fin de explicar este resultado se realizó el perfil de ejecución de FSMO para identificar la función que provoca el cuello de botella y los resultados alcanzados se muestran en la tabla Tabla 12.

Los resultados mostrados en la Fig. 28, la Tabla 12 y la Fig. 29 demuestran que para FSMO las funciones que ocupan la mayor parte del tiempo de entrenamiento y llamadas son las funciones *DIME_DataWriteSingle*, *DIME_DataReadSingle* y *DIME_AddressWriteSingle*, con un 87.8% del total de las llamadas y un 92% del tiempo de ejecución. Estas funciones son las encargadas de la comunicación desde la PC hacia el FPGA y viceversa. La función que calcula el producto punto ocupa solamente el 3.7% de las llamadas de ejecución y el 8% del tiempo empleado en el entrenamiento. Para la implementación de FSMO el cuello de botella en la ejecución está dado en la transferencia de datos entre la PC y el FPGA, no en las heurísticas que desarrolla el algoritmo ni en el cálculo del producto punto que fue acelerado en hardware por lo que la arquitectura propuesta cumple los objetivos planteados.

Función	Llamadas	Tiempo (segs)	%
a1a			
DIME_DataWriteSingle	44702348	294323.41	53.5
DIME_DataReadSingle	14889860	96012.35	17.4
DIME_AddressWriteSingle	59592208	93152.287	16.9
main	1	38766.471	7
dotProduct	14889860	20358.494	3.7
calculateError	15794	3004.881	0.5
DIME_ConfigDevice	2	2357.425	0.4
DIME_OpenCard	1	1385.202	0.3
swap	3452739	319.959	0.1
a2a			
DIME_DataWriteSingle	90845555	521369.549	56
DIME_DataReadSingle	30270929	168200.842	18.1
DIME_AddressWriteSingle	121116484	148695.316	16
main	1	76827.731	8.2
dotProduct	30270929	8672.286	0.9
DIME_ConfigDevice	2	2447.819	0.3
calculateError	22389	1615.836	0.2
DIME_OpenCard	1	1390.409	0.1
swap	7295713	850.254	0.1
a3a			
DIME_DataWriteSingle	150649928	846571.687	59.1
DIME_DataReadSingle	50205720	273438.5	19.1
DIME_AddressWriteSingle	200855648	237931.522	16.6
main	1	54232.587	3.8
dotProduct	50205720	10334.834	0.7
DIME_OpenCard	1	3563.33	0.2
DIME_ConfigDevice	2	2366.933	0.2
calculateError	28381	2044.602	0.1

Tabla 12 Análisis del perfil de FSMO para *Adult*, con kernel lineal y C = 0.05.

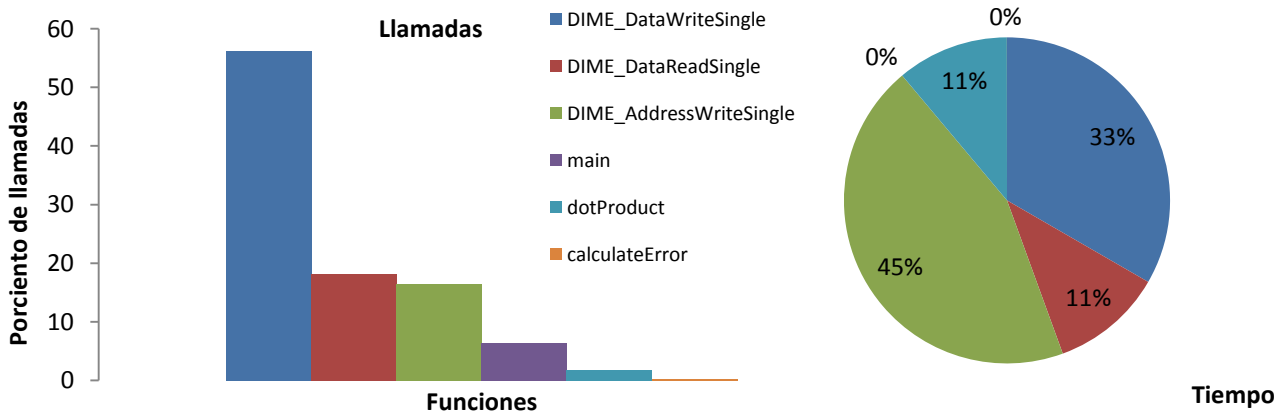


Fig. 29 Perfil de llamadas promedio y de tiempo promedio para *Adult*, con kernel lineal y C = 0.05.

5.3 Síntesis de la arquitectura.

En la arquitectura *dotProduct* la mayor parte de la lógica está implementada en el control de las memorias RAM que componen el Bloque RAM. Estas memorias están conectadas entre sí para permitir almacenar los vectores de entrada de 128 bits y permitir fácilmente la ampliación con el fin de poder procesar datos más grandes.

La Tabla 13 muestra el resultado de la síntesis de la arquitectura.

Tabla 13 Resultados de la síntesis de la arquitectura dotProduct para el FPGA XC4SX35 de Xilinx obtenido mediante la herramienta Xilinx ISE.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	308	30720	1%
Number of 4 input LUTs	3352	30720	10%
Logic Distribution			
Number of occupied Slices	2102	15360	13%
Number of Slices containing unrelated logic	0	2102	0%
Total Number of 4 input LUTs	3560	30720	11%
Number of bonded IOBs	44	448	9%
Number of BUFG/BUFGCTRLs	2	32	6%
Number of FIFO16/RAMB16s	64	192	33%
Number of DCM_ADVs	1	8	12%
Total equivalent gate count for design	4229711		
Additional JTAG gate count for IOBs	2112		

De la tabla anterior se puede ver que el recurso crítico en esta implementación es el uso de las RAMs. La arquitectura propuesta que requiere del 33% del total de RAMs por lo que se puede ampliar para manejar matrices de entrenamiento hasta 3 veces mayores. También se pueden procesar matrices de entrenamiento aún mayores si se emplean memorias externas pero en este caso el recurso crítico serían las LUTs. En la Tabla 13 se puede ver que el uso actual de las LUTs es del 10% del total, por lo que la arquitectura propuesta se puede extender hasta 10 veces más.

5.4 Discusión de los resultados.

Luego de implementar el cálculo del producto punto en hardware y desarrollar el software que realiza la comunicación entre el FPGA y la PC se logró entrenar una SVM mediante la implementación por hardware de los procesos que son computacionalmente intensos en el algoritmo SMO. Con esta implementación se logró obtener resultados de entrenamiento semejantes en al menos un 99% a los alcanzados con LibSVM para un conjunto de entrenamiento artificial creado para demostrar la validez de la arquitectura propuesta.

Se demostró experimentalmente que FSMO es una implementación válida según los resultados del entrenamiento obtenidos. Los experimentos se realizaron con el conjunto *Adult* con el fin de comparar los resultados alcanzados con el trabajo de Platt en [6], y se demostró una semejanza de más del 99%. No obstante, para otras bases de datos el patrón de comportamiento sería el mismo ya que el entrenamiento de SMO depende de las heurísticas a seguir y dichas heurísticas dependen de los resultados del producto punto el cual es implementado por la arquitectura *DotProduct* de la misma manera siempre, ofreciendo resultados sin posibilidad de error. De esta forma, para cualquier experimento realizado con *FSMO* sobre datos linealmente separables el patrón de comportamiento será el mismo que el presentado por *Adult* y se obtendrá una calidad de entrenamiento del 99%.

Los tiempos de convergencia alcanzados fueron superiores con FSMO pero se debe destacar que LibSVM es un software altamente optimizado y depurado lo cual redunda en un menor tiempo de procesamiento. El valor del umbral alcanzado con FSMO difiere del obtenido con LibSVM en un 0.084%. Los resultados de entrenamiento comparados con [6] es de menos del 0.5% de diferencia en el peor de los casos. En [6], Platt no hace referencias ni explica detalles de implementación de su algoritmo que serían imprescindibles para realizar un análisis sobre las diferencias entre sus resultados y los obtenidos en una implementación de SMO a partir del pseudo-código presentado en este mismo artículo.

Al determinar las funciones que hacen que los tiempos de entrenamiento sean extremadamente grandes se pudo sustentar la idea de que el FPGA seleccionado, a pesar de las ventajas que presenta respecto a otros dispositivos, no resulta la solución adecuada para la arquitectura que se presenta.

Con la arquitectura propuesta se pudo demostrar que:

- Entrena correctamente la SVM.
- Garantiza la convergencia del algoritmo.
- Las estructuras de control y las heurísticas de SMO funcionan correctamente.
- El cálculo del producto punto se realiza correctamente y 178.7 veces más rápido que si se realizara en software.

Como se planteó, la idea a seguir es utilizar un procesador al que se le pueda acoplar un coprocesador de manera tal que en el software que se ejecute en el procesador estén las estructuras de control de SMO y en el coprocesador se encontrará la arquitectura para acelerar el producto punto. De esta forma no se haría uso del bus PCI para la comunicación entre el procesador y coprocesador haciendo que la comunicación sea más rápida. Para desarrollar esta variante es necesario emplear tecnologías diseñada para soportar dicha característica.

Luego de la síntesis de la arquitectura *dotProduct* se determinó que la arquitectura podrá funcionar a una frecuencia máxima de 35MHz. Teniendo en cuenta que el cálculo del producto punto mediante la arquitectura *dotProduct* se realiza en 3 ciclos de reloj entonces la arquitectura para SMO acelerada por hardware podrá ejecutar 11666666 cálculos de productos punto por segundo para vectores de entrada de 128 bits. Sin embargo, el cálculo del producto punto en un procesador Intel Pentium IV corriendo a 3GHz requiere 45957 ciclos de reloj, por lo que se podrán procesar 65278 cálculos de productos punto por segundo para vectores de entrada de 128 bits. Con esto se demuestra que la arquitectura *dotProduct* es 178.7 veces más rápida que su

implementación por software. Si se implementa la arquitectura *dotProduct* completamente en hardware en un coprocesador acoplado al procesador central se podrá lograr, según los resultados alcanzados, una aceleración de alrededor de 178.7 veces como mínimo respecto a su implementación puramente en software.

Capítulo

Conclusiones y trabajo futuro 6

En este capítulo se describen las conclusiones alcanzadas con esta investigación además de las extensiones y modificaciones futuras de que podría ser objeto.

6.1 Revisión de objetivos.

El objetivo principal que se persigue con este trabajo es el diseño e implementación de una arquitectura hardware-software para acelerar la fase de entrenamiento de una SVM de manera tal que su desempeño sea superior en al menos un orden de magnitud al de los algoritmos existentes por hardware.

Se considera que este objetivo fue alcanzado ya que:

1. Se realizó un estudio de la literatura existente donde se identificó entre varios candidatos el algoritmo que mejor resultados ofrece en el entrenamiento de

SVM y de él, se identificó la función responsable del cuello de botella (cálculo del producto punto con un 81% de consumo de tiempo de ejecución).

2. Se implementó en software el algoritmo SMO a partir del pseudocódigo ofrecido por el autor.
3. Se diseñó e implementó una arquitectura hardware para acelerar el cálculo del producto punto.
4. Se diseñó e implementó en software el algoritmo SMO donde las estructuras de control y las funciones secuenciales se ejecutan en software; mientras que en hardware se realiza el cálculo del producto punto.

De esta manera se diseñó e implementó una arquitectura hardware-software para acelerar el entrenamiento de SVM, alcanzando tiempos de procesamiento 178.7 veces más rápido que por software.

6.2 Conclusiones.

Las SVM son una novedosa y reciente técnica del Aprendizaje Automatizado que están arrojando un elevado número de aplicaciones prácticas y desarrollos teóricos, los cuales han demostrado su robustez y eficiencia. Sin embargo, el tiempo de entrenamiento de una SVM en ocasiones no resulta viable y por lo tanto se buscan soluciones a esta problemática. Luego de terminar esta investigación concluye que:

- Los problemas de clasificación que el algoritmo SMO puede enfrentar se ve limitado por el número de muestras de entrenamiento que puede manejar.
- Implementar SMO de manera íntegra en hardware no resulta una opción viable porque este algoritmo se basa en una jerarquía de heurísticas la cual es un conjunto de evaluaciones condicionales que resultan muy costosas en términos del número necesario de compuertas lógicas equivalentes para su implementación.
- Los cálculos matemáticos necesarios en la fase de optimización de candidatos resulta muy complicado de implementar en hardware o la relación costo-beneficio resulta inadecuada.

- SMO es un algoritmo esencialmente secuencial lo cual hace muy difícil su implementación de manera íntegra en hardware.
- La función más costosa en términos de llamadas y tiempo de ejecución es del 77% y el 81% del total respectivamente.
- El cálculo del producto punto es la función responsable del cuello de botella en el entrenamiento de SVM mediante SMO.
- A pesar de que SMO es un algoritmo secuencial, la función responsable del cuello de botella es fácilmente implementada en paralelo.
- El diseño de una arquitectura hardware – software que realice el entrenamiento de SVM de tal manera que las funciones secuenciales y las estructuras de control se ejecuten en software mientras que el cálculo del producto punto se realice en hardware puede acelerar el entrenamiento de SVM.
- Es posible calcular el producto punto hasta 178.7 veces más rápido con la arquitectura propuesta que en un procesador Intel Pentium 4 que corre a una frecuencia de 3GHz.

Por todo lo anterior se puede concluir que el diseño e implementación de una arquitectura hardware-software para el entrenamiento de SVM empleando el algoritmo SMO donde en software se implementen los procesos secuenciales y de control del algoritmo y en hardware se implemente el cálculo del producto punto es una solución válida para acelerar la fase de entrenamiento de SVM ya que se logra una aceleración de dos ordenes de magnitud respecto al algoritmo original.

6.3 Trabajos futuros.

Como trabajo futuro se propone continuar la implementación de la arquitectura completamente sobre un FPGA mediante el empleo de un procesador por software. Para ello se propone evaluar los FPGAs que provee Altera y su soft-processor NIOS II. Este procesador es conocido por la flexibilidad en su diseño, permitiendo la

modificación de su ALU y de su conjunto de instrucciones, por lo que podría resultar el candidato adecuado para implementar la arquitectura propuesta. Se agregaría a su conjunto de instrucciones aquellas que sean necesarias para la comunicación con el coprocesador y para la ejecución de SMO. De esta manera no se emplearía el bus PCI para la comunicación entre el procesador y el FPGA, se encontraría toda la arquitectura embebida dentro del mismo dispositivo hardware y se evitaría el cuello de botella provocado por las funciones que realizan la comunicación entre la PC y el FPGA.

A grandes rasgos, la idea que se propone con el uso del soft-procesor NIOS II se muestra en la Fig. 30.

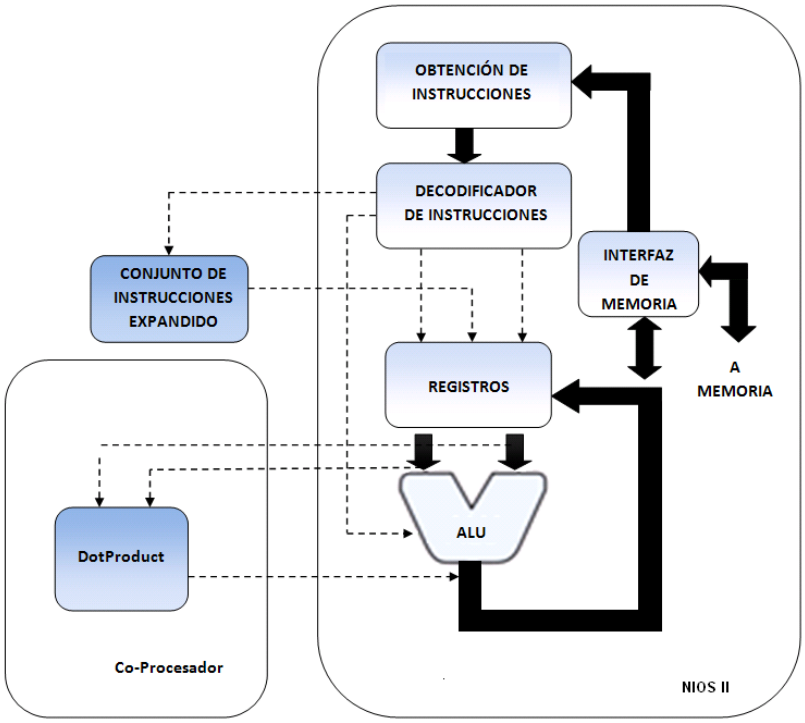


Fig. 30 Estructura de SMO acelerado por hardware sobre NIOS II.

Como se puede ver, se trata de la misma idea que ya se había presentado sólo que esta vez se propone desarrollarla sobre un procesador de propósito general por software, el cual permite ampliar su conjunto de instrucciones tal como se había

recomendado y que además se implementa sobre FPGAs, permitiendo de esta manera un diseño más práctico y flexible de la arquitectura.

Referencias R

1. Vapnik, V., *Statistical Learning Theory*. 1998: Wiley-Interscience.
2. Vapnik, V., *The Nature of Statistical Learning Theory (Information Science and Statistics)*. 1999: Springer.
3. Burges, C., *A tutorial on support vector machines for pattern recognition.* Data Mining and Knowledge Discovery, 1998. **2**: p. 121-167.
4. Wang, G., *A Survey on Training Algorithms for Support Vector Machine Classifiers.* Networked Computing and Advanced Information Management, International Conference on, 2008. **1**: p. 123-128.
5. Vapnik, V., *Estimation of Dependences Based on Empirical Data: Empirical Inference Science (Information Science and Statistics)*. 2006: Springer.
6. Platt, J., *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Technical Report. 1998, MST-TR-98-14: Microsoft Research.

7. Joachims, T., *Making large-scale SVM learning practical.* Advances in kernel methods: support vector learning, 1999: p. 169-184.

8. Osuna, E., R. Freund, and F. Girosi, *An improved training algorithm for support vector machines.* Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop, 1997: p. 276-285.

9. Anguita, D., A. Boni, and S. Ridella, *A digital architecture for support vector machines: theory, algorithm, and FPGA implementation.* Neural Networks, IEEE Transactions on, 2003. **14**(5): p. 993-1009.

10. Ahmad, A.R., et al., *The Comparative Study of SVM Tools for Data Classification.* IMAGE2003, 2008: p. 57-63.

11. Kotsiantis, S., I. Zaharakis, and P. Pintelas, *Machine learning: a review of classification and combining techniques.* Artificial Intelligence Review, 2006. **26**(3): p. 159-190.

12. Jian-Pei Zhang, Z.-W.L., Jing Yang, *A parallel SVM training algorithm on large-scale classification problems.* Proceedings of International Conference on Machine Learning and Cybernetics, 2005. **3**: p. 1637- 1641.

13. Graf, H., et al., *Parallel support vector machines: The cascade svm.* In Advances in Neural Information Processing Systems, 2005. **17**: p. 521-528.

14. Thi, N., *GA-SVM Based Framework for Time Series Forecasting.* International Conference on Natural Computation, 2009. **1**: p. 493-498.

15. Franco-Arcega, A., et al., *A New Incremental Algorithm for Induction of Multivariate Decision Trees for Large Datasets*, in *Proceedings of the 9th International Conference on Intelligent Data Engineering and Automated Learning*. 2008, Springer-Verlag: Daejeon, South Korea. p. 282 - 289

16. Romano, R.A., C.R. Aragon, and C. Ding, *Supernova Recognition Using Support Vector Machines*, in *Proceedings of the 5th International Conference on Machine Learning and Applications*. 2006, IEEE Computer Society. p. 77-82

17. Medina-Pagola, J.E., *Estado del Arte del Web Mining.* Reporte Técnico. 2007, RT_001: Centro de Aplicaciones de Tecnologías de Avanzada (CENATAV).

18. Estrin, G., et al., *Parallel Processing in a Restructurable Computer System.* Electronic Computers, IEEE Transactions on, 1963. **EC-12**(6): p. 747-755.

19. Boemo Scalvinoni, E. *Estado del Arte de la Tecnologia FPGA*. 2005; Available from: http://utic.inti.gov.ar/publicaciones/cuadernilloUE/CT_Microelectronica17_FPGA.pdf.

20. Xilinx. *FPGA Design Flow Overview*. 2010 [cited 2010 jan, 25]; Available from: http://www.xilinx.com/itp/xilinx82/help/iseguide/html/ise_fpga_design_flow_overview.htm.

21. Byun, H., S.-W. Lee, and A. Verri, *Applications of Support Vector Machines for Pattern Recognition: A Survey*, in *Pattern Recognition with Support Vector Machines*. 2002, Springer Berlin Heidelberg. p. 571-591.

22. Ecker, J. and M. Kupferschmid, *Introduction to Operations Research*. 1998: John Wiley & Sons Inc. 528.

23. Weisstein, E.W. *"Riemann-Lebesgue Lemma"*. [MathWorld--A Wolfram Web Resource] 2010 [cited 2010 January, 07]; Available from: http://mathworld.wolfram.com/Riemann-LebesgueLemma.html.

24. Muller, K.R., et al., *An introduction to kernel-based learning algorithm.* IEEE Transactions., 2001. **12**(Neural Networks): p. 181-202.

25. Osuna, E., R. Freund, and F. Girosi, *Training Support Vector Machines: an Application to Face Detection. Computer Vision and Pattern Recognition*, 1997: p. 130-136.

26. Collobert, R. and S. Bengio, *SVMTorch: support vector machines for large-scale regression problems.* Journal of Machine Learning, 2001. **1**: p. 143-160.

27. Anguita, D., A. Boni, and S. Ridella, *Digital kernel perceptron.* Electronics Letters, 2002. **38**(10): p. 445-446.

28. Asuncion, A. and D.J. Newman, *{UCI} Machine Learning Repository.* University of California, Irvine, School of Information and Computer Sciences, 2007.

29. Etin, S. and U. Elias, *Parallelizing SMO for solving SVMs*. 2000, Technion - Israel Institute of Technology. p. 49.

30. Keerthi, S., et al., *Improvements to Platt's SMO Algorithm for SVM Classifier Design.* Neural Computation, 2001. **13**(3): p. 637-649.

31. Platt, J.C., *Fast training of support vector machines using sequential minimal optimization.* Advances in kernel methods: support vector learning., 1999: p. 185-208.

32. Platt, J., *Using Analytic QP and Sparseness to Speed Training of Support Vector Machines.* Neural Information Processing Systems 11, 1999. **11**: p. 557-563.

33. Dey, S., et al., *Embedded Support Vector Machine : Architectural Enhancements and Evaluation.* VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on, 2007: p. 685-690.

34. AutomatedQA. *AutomatedQA´s AQTime*. 2009 [cited 2009 november 5]; Available from: http://www.automatedqa.com/.

35. Chang, C. and C. Lin. *{LIBSVM}: a library for support vector machines*. 2001; Available from: http://www.csie.ntu.edu.tw/~cjlin/libsvm.

36. Fan R.-E., Chen P.-H., and L. C.-J., *Working set selection using second order information for training SVM.* . Journal of Machine Learning Research 2005. **6**: p. 1889 - 1918.

37. Collobert, R., S. Bengio, and Y. Bengio, *A parallel mixture of SVMs for very large scale problems.* Neural Comput., 2002. **14**(5): p. 1105-1114.

38. Wen, Y.-M. and B.-L. Lu, *A Cascade Method for Reducing Training Time and the Number of Support Vectors*, in *Advances in Neural Networks - ISNN 2004*. 2004. p. 480-486.

39. Xilinx. *XtremeDSP Development Kit — Virtex-4 Edition*. 2010 [cited 2009 october, 02]; Available from: http://www.xilinx.com/products/devkits/DO-DI-DSP-DK4-UNI-G.htm.