Universidade Federal do Rio Grande do Norte
Centro de Tecnologia
Programa de Pós-Graduação em Engenharia Elétrica e de Computação

Proposta de Implementação em FPGA de Máquina de Vetores de Suporte (SVM) utilizando Otimização Sequencial Mínima (SMO)

Daniel Holanda Noronha

Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (Área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Número de ordem PPgEEC: M504
Natal, RN, Novembro de 2017

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Noronha, Daniel Holanda.

Proposta de implementação em FPGA de máquina de vetores de suporte (SVM) utilizando otimização sequencial mínima (SMO) / Daniel Holanda Noronha. - Natal, RN, 2017.

66 f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Programa de Pós-graduação em Engenharia Elétrica e Computação. Natal, RN, 2017.

Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes.

1. Inteligência artificial - Dissertação. 2. FPGA - Dissertação. 3. SVM - Dissertação. I. Fernandes, Marcelo Augusto Costa. II. Título.

RN/UF/BCZM CDU 004.8

Proposta de Implementação em FPGA de Máquina de Vetores de Suporte (SVM) utilizando Otimização Sequencial Mínima (SMO)

Daniel Holanda Noronha

Dissertação de Mestrado apresentada em 20 de Novembro de 2017. Banca examinadora composta pelos seguintes membros:

Prof. Dr. Marcelo Augusto Costa Fernandes (Orientador) ..... DCA/UFRN

Prof. Dr. Adrião Duarte Doria Neto ..... DCA/UFRN

Prof. Dr. José Alberto Nicolau de Oliveira ..... DEE/UFFN

Prof. Dr. Antonio Carlos Schneider Beck Filho (Avaliador Externo) .. UFRGS

Agradecimentos

Aos meus pais, Cleófanes Noronha e Gardênia Holanda, que sempre me deram amor, apoio e todas as melhores condições possíveis para que meu tempo de estudo sempre fosse de qualidade.

À minha avó, Maria do Céu, que sempre fez tudo ao seu alcance para que eu fosse uma pessoa tranquila e despreocupada.

Ao meu irmão, Gabriel Holanda, pelo companheirismo e pelas boas risadas.

À minha amada companheira, Giulia Natalini, que esteve comigo nos momentos mais difíceis e sempre me apoiou incondicionalmente, transformando qualquer momento em um momento bom e que valesse a pena.

À UFRN e a todos os meus professores do Departamento de Engenharia Computação e Automação, por compartilharem seus conhecimentos comigo e com meus demais colegas.

Ao meu orientador, Marcelo Fernandes, por me mostrar que estou só no começo e que muitos novos e belos desafios ainda virão.

Aos meus colegas, por estarem comigo nos estudos, se esforçando e dando seu máximo por anos, mesmo nos fins de semana e feriados.

À Deus, por ter colocado todas essas pessoas maravilhosas em minha vida.

"If you are completely comfortable with your life,
and feeling absolutely no anxiety,
then you are probably coasting,
and not really growing."
– Mardy Grothe

Resumo

A importância do uso de FPGAs como aceleradores vem crescendo fortemente nos últimos anos. Companhias como Amazon e Microsoft estão incorporando FPGAs em seus data centers, objetivando especialmente acelerar algoritmos em suas ferramentas de busca. No centro dessas aplicações estão algoritmos de aprendizado de máquina, como é o caso da Máquina de Vetor de Suporte (SVM). Entretanto, para que essas aplicações obtenham a aceleração desejada, o uso eficiente dos recursos das FPGAs é necessário. O projeto possui como objetivo a implementação paralela em hardware tanto da fase *feed-forward* de uma Máquina de Vetores de Suporte (SVM) quanto de sua fase de treinamento. A fase *feed-forward* (inferência) é implementada utilizando o kernel polinomial e de maneira totalmente paralela, visando obter a máxima aceleração possível ao custo de uma maior utilização da área disponível. Além disso, a implementação proposta para a inferência é capaz de computar tanto a classificação quanto a regressão utilizando o mesmo hardware. Já o treinamento é feito utilizando Otimização Sequencial Mínima (SMO), possibilitando a resolução da complexa otimização da SVM através de passos simples. A implementação da SMO também é feita de modo extremamente paralelo, fazendo uso de técnicas para aceleração como a cache do erro. Ademais, o Kernel Amigável ao Hardware (HFK) é utilizado para diminuir a área utilizada pelo kernel, permitindo que um número maior de kernels seja implementado em um chip de mesmo tamanho, acelerando o treinamento. Após a implementação paralela em hardware, a SVM é validada por simulação e são feitas análises associadas ao desempenho temporal da estrutura proposta, assim como análises associadas ao uso de área da FPGA.

Palavras-chave: Máquina de Vetores de Suporte. Otimização Sequencial Mínima. Implementação em FPGA. Simulink. System Generator.

Abstract

The importance of Field-Programmable Gate Arrays as compute accelerators has dramatically increased during the last couple of yers. Many companies such as Amazon, IBM and Microsoft included FPGAs in their data centers aiming to accelerate their search engines. In the center of those applications are many machine learning algorithms, such as Support Vector Machines (SVMs). For FPGAs to thrive in this new role, the effective usage of FPGA resources is required. The project's main goal is the parallel FPGA implementation of both the feed-forward phase of a Support Vector Machine as well as its training phase. The feed-forward phase (inference) is implemented using the polynomial kernel in a highly parallel way in order to obtain maximum throughput at the cost of some extra area. Moreover, the inference implementation is capable of computing both classification and regression using a single hardware. The training phase of the SVM is implemented using Sequential Minimal Optimization (SMO), which enables the resolution of a complex convex optimization problem using simple steps. The SMO implementation is also highly parallel and uses some acceleration techniques, such as the error cache. Moreover, the Hardware Friendly Kernel (HFK) is used in order to reduce the kernel's area, enabling the increase in the number of kernels per area. After the parallel implementation in hardware, the SVM is validated by simulation. Finally, analysis associated with the temporal performance of the proposed structure, as well as analysis associated with FPGA's area usage are performed.

Keywords: Support Vector Machine. Sequential Minimal Optimization. FPGA implementation. Simulink. System Generator.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AI	Inteligência Artificial
ASIC	Circuitos integrados de aplicação específica
CLB	Bloco lógico programável
DFF	Flip-flop tipo D
DSP	Processamento digital de sinais
FA	Somador completo
FPGA	Arranjo de Portas Programável em Campo
GPP	Processador de Propósito Geral
HDL	Linguagem de Descrição de Hardware
HFK	Kernel Amigável ao Hardware
HLS	Síntese de Alto Nível
IoT	Internet das coisas
LUT	Tabela De Busca
ML	Aprendizado de máquina
MSE	Erro médio quadrático
NLP	Processamento de linguagem natural
RTL	Nível de Transferência entre Registradores
SG	System Generator
SMO	Otimização Sequencial Mínima
SV	Vetor de Suporte
SVM	Máquina de Vetores de Suporte
VHDL	Linguagem de descrição de hardware de alta velocidade
XOR	Porta lógica OU exclusivo

Lista de símbolos

C	Constante de regularização
d	Grau do kernel polinomial
d_{MAX}	Grau máximo do kernel polinomial
E	Erro do treinamento utilizando otimização sequencial mínima
Ec	Erro da função CORDIC do kernel amigável ao hardware
Hi	Limite superior do valor de α_j durante o treinamento
K	Função de kernel
L	Dimensão dos elementos do conjunto de entrada
Lo	Limite inferior do valor de α_j durante o treinamento
N	Número de elementos do conjunto de entrada
N_{SV}	Número de vetores de suporte
sh	Parâmetro de deslocamento da entrada
sf	Parâmetro de dimensionamento da entrada
x	Elemento do conjunto de entrada
y	Classe ou valor de saída
z	Valor do elemento do conjunto de entrada após dimensionamento
α	Multiplicadores de Lagrange
β	Constante unificada de classificação e regressão
γ	Constante de dimensionalidade do kernel amigável ao hardware
ε	Parâmetro da zona insensível

Sumário

1 Introdução

1.1 Contextualização

É notório que, nos últimos anos, os requisitos dos sistemas eletrônicos vem se alterando muito, principalmente devido ao avanço crescente do poder de processamento desses dispositivos e devido às novas aplicações que estão surgindo para eles. Um dos campos de estudo que vem se destacando bastante nesse contexto é o campo da inteligência artificial (Artificial Inteligence - AI), mais especificamente o subcampo de aprendizagem de máquina (Machine Learning - ML). Dentre tantos fatores relevantes em relação aos novos requisitos dessa área de estudo, destacam-se: a necessidade de dispositivos com tempos de resposta cada vez menores e mais precisos, que atendam aos requisitos de tempo real, e a imprescindibilidade de sistemas que façam uso inteligente de seus recursos energéticos.

Sabe-se que, cada vez mais, algoritmos de aprendizagem de máquina são utilizados em aplicações que demandam altas taxas de amostragem. Exemplos desses casos são as aplicações que envolvem ML aplicado à visão computacional (ZEPEDA; PEREZ, 2015) e ML aplicado ao processamento de *Big Data* (HUANG; LIU, 2014). Essa necessidade crescente por velocidade faz com que haja uma demanda cada vez maior de implementações desses algoritmos em harware.

Além disso, com o avanço do número de sistemas embarcados que utilizam ML e com o advento da internet das coisas (Internet of Things - IoT), o baixo consumo desses dispositivos dependentes de bateria tornou-se um fator essencial. Para tal, faz-se necessário, dentre muitos outros fatores, a diminuição do *clock* desses dispositivos. Entretanto, para que não haja perda de desempenho, paralelizações do algoritmo são constantemente feitas em hardware, principalmente utilizando hardwares reconfiguráveis seja para o projeto do circuito integrado de aplicação específica (Application-Specific Integrated Circuit - ASIC) ou seja para a utilização com o próprio arranjo de portas programável em campo (Field-Programmable Gate Array - FPGA).

Ademais, observa-se que existe uma crescente demanda pela implementação de algoritmos de aprendizado de máquina em hardware, já que fatores como taxa de amostragem e eficiência energética se tornam elementos cada vez mais relevantes.

Dentre os algoritmos de aprendizado de máquina mais utilizados, destaca-se a máquina de vetores de suporte (Support Vector Machine - SVM). Esta apresenta uma grande gama de aplicações, principalmente envolvendo problemas de classificação e regressão no campo do processamento de linguagem natural (Natural Language Processing - NLP). No atual contexto, o algoritmo em questão, além de amplamente utilizado, também apresenta

diversas possibilidades de paralelização, o que o torna extremamente interessante de ser implementado em hardware.

1.2 Metodologia

Objetivando a implementação de hardware do algoritmo já consolidado da SVM, será feito um estudo para identificar os principais elementos que devem estar presentes em sua implementação. Esse estudo inclui as novas propostas de heurísticas para a Otimização Sequencial Mínima e os diferentes kernels propostos a serem implementados em hardware. Em seguida, será elaborada uma proposta de implementação paralela do algoritmo de modo a possibilitar o uso de um número qualquer de vetores de suporte e parâmetros de kernel configuráveis.

Finalmente, baseada na proposta de implementação e nos parâmetros estudados, será feita a implementação em hardware utilizando o System Generator, uma ferramenta gráfica de design a nível RTL com integração com Matlab e Simulink. Para os testes, o mesmo programa será utilizado no modo simulação e, em seguida, sínteses para o FPGA alvo serão feitas a fim de determinar a possibilidade de implementação do algoritmo proposto nesse hardware, assim como analisar sua taxa de amostragem e nível de paralelismo.

1.3 Trabalhos Relacionados

Diversos trabalhos na literatura fazem implementações de algoritmos de inteligência artificial em hardware reconfigurável. A grande quantidade de trabalhos nessa área é justificada pelo enorme ganho em velocidade e em economia de energia possível nesse tipo de tecnologia.

Existe um crescente interesse na utilização de SVMs em aplicações relacionadas com processamento de imagem e com sistemas de classificação embarcados. Nesse contexto, o alto custo computacional da SVM, principalmente em conjuntos de dados grandes, provoca a necessidade de aceleração desse algoritmo. O maior problema das implementações em software é que, apesar dessas implementações possuírem boa precisão numérica, elas não são capazes de atender aos requisitos de baixo consumo energético e processamento em tempo real. Isso ocorre em vários casos, como no processamento de imagem por exemplo. Esse fato motivou as implementações de SVM em hardware reconfigurável.

Os trabalhos que propõem implementação em hardware de máquinas de vetores de suporte podem ser subdivididos em duas principais categorias: trabalhos que fazem apenas a implementação da fase *feed-forward* e trabalhos que propõem tanto a implementação da fase *feed-fowward* quanto da fase de treinamento.

1.3.1 Trabalhos de implementação da fase *feed-forward* da SVM

Diversos trabalhos fizeram a implementação apenas da fase *feed-forward* da SVM, ou seja, da fase de inferência. (PATIL et al., 2012), (HUSSAIN; SEKER, 2013) e (HUSSAIN; SEKER, 2014) fizeram a implementação da fase *feed-forward* utilizando arrays sistólicos objetivando acelerar os cálculos e reduzir a área de uso do FPGA. Inicialmente, (PATIL et al., 2012) propôs uma implementação para a classificação de imagens faciais objetivando diferenciar seis expressões básicas: sorriso, surpresa, tristeza, raiva, desgosto e medo. A estrutura com array sitólico promove o uso eficiente da memória e baixa complexidade da arquitetura proposta. Esse trabalho, entretanto, não foca em criar uma estrutura com alto throughput, mas sim em utilizar uma mesma estrutura para a diversas classificações binárias diferentes, resultando em uma classificação multiclasse. Para isso, o autor faz uso da ferramente de reconfiguração parcial da Xillinx para dinamicamente modificar blocos de lógica enquanto o restante da lógica continua em execução sem interrupções.

Já (HUSSAIN; SEKER, 2013) propôs uma implementação com arrays sistólicos para a classificação de microarranjos de DNA. Esse trabalho, expandido posteriormente pelos mesmo autores em (HUSSAIN; SEKER, 2014), também faz uso de reconfiguração parcial como em (PATIL et al., 2012). A arquitetura proposta, entretanto, foca na aceleração de da classificação de um número pequeno de elementos de enorme dimensão, como é o caso dos microarranjos de DNA. Além disso, o kernel utilizado nesse paper é o kernel linear. Esse kernel, apesar de ser facilmente implementado em hardware, possui baixa capacidade de classificar problemas complexos.

(RUIZ-LLATA; GUARNIZO; YÉBENES-CALVINO, 2010), (JALLAD; MOHAMMED, 2014), (PAN et al., 2013) fizeram a implementação dessa mesma fase utilizando o kernel amigável ao hardware ou *Hardware Friendly Kernel* (HFK) obtendo reduções significativas no espaço utilizado pelo FPGA. Isso ocorre, pois o esse kernel não necessita dos blocos de multiplicação presentes em kernels convencionais. O kernel HFK, que também será utilizado nesse trabalho, utiliza uma estrutura muito similar ao kernel gaussiano e efetua as suas operações mais complexas em um bloco CORDIC que contém apenas deslocamentos e somas como mostrado na Seção 6.1.

A principal contribuição de (RUIZ-LLATA; GUARNIZO; YÉBENES-CALVINO, 2010) é a introdução de uma arquitetura para a fase *feed-forward* que permite tanto a classificação quanto a regressão reutilizando os mesmos componentes de hardware. O hardware proposto foi utilizado para a classificação entre 4 classes diferentes de imagens de 32x32 pixels em escala de cinza de 8 bits. Já a regressão foi testada com a função *sinc*. O clock máximo atingido pela implementação foi de 30MHz, entretanto são necessários diversos ciclos para a classificação devido ao HFK.

Em (JALLAD; MOHAMMED, 2014), o hardware desenvolvido objetiva ser utilizado

em aplicações de classificação de imagem embarcadas em satélites. A principal contribuição do trabalho é o baixo uso de área devido ao uso do HFK em combinação com um bloco de controle. Entretanto, grande parte da computação é feita de modo serial através do bloco de controle, fazendo com que a classificação leve 90 ciclos de clock por cada pixel da imagem a ser analisada.

O hardware proposto em (PAN et al., 2013) foi possui sua arquitetura similar ao proposto em (RUIZ-LLATA; GUARNIZO; YÉBENES-CALVINO, 2010). Entretanto, o cálculo da norma L1 do kernel HFK é feito de modo paralelo ao invés de serial. Assim, o número de clocks necessários para computar o kernel passa a não depender mais da dimensionalidade da entrada.

Objetivando acelerar a classificação de situações mais desafiadoras (PAPADONIKOLAKIS; BOUGANIS, 2010) e (KYRKOU; THEOCHARIDES; BOUGANIS, 2013) propuseram a implementação da fase *feed-forward* em cascata. Classificadores em cascata tomam vantagem de situações onde a probabilidade da classe ser +1 é muito maior do que a classe ser -1. Vários estágios de classificadores são aplicados sequencialmente aos dados de entrada. Caso qualquer um dos estágios assuma que a classe é -1 o algoritmo para e não precisa passar pelas demais camadas. Esse tipo de abordagem apenas é eficiente em casos específicos onde as características de interesse são apenas uma pequena parte do elemento de entrada, como um rosto de uma pessoa em uma foto tirada à distância.

1.3.2 Trabalhos de implementação do treinamento da SVM

Outros poucos trabalhos fizeram a implementação da fase de treinamento da SVM. (TA-WEN et al., 2012) implementou o treinamento da SVM utilizando Otimização Sequencial Mínima (SMO). A arquitetura proposta consiste em três blocos principais que representam as principais funções do SMO e são controladas por uma máquina de estados finito (FSM). Entretanto, o grande ganho de área que se pode obter com o kernel HFK não foi levado em consideração. Além disso, o kernel utilizado foi o linear, limitando a capacidade de classificação da arquitetura proposta.

(CAO; SHEN; CHEN, 2010), por sua vez, propôs também uma arquitetura baseada na SMO. A principal contribuição desse trabalho foi a flexibilidade dada a estrutura proposta, que é customizável entre totalmente paralela e serial. O objetivo dessa arquitetura foi ser utilizada em aplicações embarcadas, onde implementações totalmente paralelas excedem os recursos da FPGA e implementações seriais não atendem aos requisitos temporais do problema.

Já em (BUSTIO-MARTÍNEZ et al., 2010) foi proposto uma arquitetura híbrida de software e hardware para acelerar o treinamento da SMO. Na implementação proposta o algoritmo da SMO foi particionado de modo a fazer com que o processamento

inevitavelmente serial e os sinais de controle fossem executados em um Processador de Propósito Geral (GPP) enquanto uma FPGA executa tarefas que podem ser executadas em paralelo (nesse caso, o produto vetorial do kernel linear). A arquitetura proposta com o coprocessador obteve um speedup de 178.7x quando comparado a uma implementação apenas de software em um GPP.

Em (FILHO et al., 2010) foi proposta uma arquitetura dinamicamente reconfigurável para o treinamento da SVM utilizando a SMO. A maior contribuição desse trabalho foi a proposta de uma arquitetura modular capaz de ser reprogramada durante sua execução. O kernel utilizado foi o HFK, proporcionando redução no uso de área quando comparado com implementações similares utilizando outros kernels. A arquitetura proposta foi capaz de obter speedup de até 12.5x quando comparada a implementações feitas puramente em software. Além disso, foi feito um estudo da precisão numérica necessária para o treinamento utilizando a arquitetura proposta, mostrando que 24 bits [6.18] é o suficiente para representar os dados de entrada.

(VENKATESHAN; VARGHESE, 2014), por sua vez, propôs um algoritmo de treinamento da SVM diferente da SMO, chamado de Hybrid Working Set (HWS). Na arquitetura proposta o FPGA é apenas utilizado como co-processador das funções de kernel. Nessa implementação, o kernel utilizado foi o gaussiado. A implementação do co-processador em FPGA obteve um speedup de 25x quando comparado a uma implementação puramente em software.

1.4 Objetivos e Contribuição

O objetivo desse trabalho é o desenvolvimento de um algoritmo paralelo em Arranjo de Portas Programável em Campo (FPGA) de uma máquina de vetores de suporte (SVM). O hardware desenvolvido deve ser capaz tanto de fazer o cálculo da fase *feed-forward* quanto de fazer o treinamento utilizando Otimização Sequencial Mínima (SMO). O hardware desenvolvido para a fase *feed-forward*, diferente do que foi encontrado na literatura, deve ser capaz de permitir ao usuário a escolha dos parâmetros do kernel, sem que essa opção afete negativamente a taxa de amostragem do sistema. Essa flexibilidade na escolha dos parâmetros do kernel é uma das principais contribuições do trabalho, visto que a maioria dos trabalhos presentes na literatura utiliza uma estrutura de kernel fixa.

Além disso, a implementação do treinamento utilizando a Otimização Sequencia Mínima também deve ser feita de modo fortemente paralelo. Outra contribuição desse trabalho é a proposta de implementação da cache do erro em hardware, possível devido ao uso do kernel amigável ao hardware (HFK), que não necessita de multiplicadores para sua implementação.

Demais objetivos do trabalho envolvem averiguar o seu bom funcionamento, assim

como fazer o estudo a respeito da quantidade de elementos necessários para a implementação do design proposto em hardware. Tal estudo será de extrema importância para a possível futura implementação desse design em um microchip baseado em SVMs.

1.5 Artigos Publicados

1. NORONHA, D. H.; FERNANDES, M. A. C. Implementação em FPGA de Máquina de Vetores de Suporte (SVM) para Classificação e Regressão. *XIII Encontro Nacional de Inteligência Artificial e Computacional (ENIAC)*, 2016.

1.6 Organização do Trabalho

No Capítulo 2 será feito um relato histórico do algoritmo da máquina de vetores de suporte, assim como será apresentada uma visão geral de seu funcionamento e suas atuais aplicações. Grande parte do foco deste capítulo será dado à parte *feed-forward* da classificação e regressão.

Logo em seguida, no Capítulo 3, será apresentado o algoritmo de Otimização Sequencial Mínima (SMO). Neste capítulo, serão discutidas as heurísticas para a seleção dos elementos a serem otimizados, o cálculo da otimização desses elementos, o cálculo do limiar da SVM b, além de métodos de aceleramento do SMO como a utilização da cache do erro.

Em seguida, no Capítulo 4, será feito um breve resumo sobre computação reconfigurável. Neste capítulo, serão apresentadas diversas plataformas de computação reconfigurável e suas principais características. Entretanto, o foco do capítulo será na arquitetura das FPGAs, já que esse é o hardware que será utilizado para a implementação.

No Capítulo 5 será apresentado o design proposto para a implementação paralela da fase *feed-forward* da SVM em hardware. Este capítulo mostrará em detalhes o funcionamento de cada um dos componentes, assim como apresentará toda a informação de modo a possibilitar a implementação da estrutura proposta em uma FPGA genérica de um fabricante qualquer. Em seguida será mostrado o design descrito implementado na FPGA alvo, as nuâncias da implementação serão comentadas, assim como serão justificadas as escolhas de projeto. Posteriormente, serão apresentados os resultados da arquitetura implementada. Tanto a classificação quanto a regressão serão testadas e, por fim, a síntese dessas implementações serão feitas e seus resultados discutidos.

No Capítulo 6 será apresentado o design do hardware proposto para a fase de treinamento utilizando a Otimização Sequencia Mínima. Assim como no Capítulo 5, serão apresentados os detalhes de funcionamentos dos blocos propostos de modo genérico,

permitindo a implementação do algoritmo proposto independentemente do modelo da FPGA. Também serão apresentados os resultados de testes específicos mostrando o bom funcionamento da arquitetura proposta, assim como os resultados da síntese mostrando os recursos necessários para a arquitetura proposta na FPGA alvo.

Finalmente, no Capítulo 7, serão apresentadas as conclusões gerais a respeito do design e de sua implementação.

2 Máquina de Vetores de Suporte (SVM)

Esse capítulo fará uma breve introdução ao conceito da Máquina de Vetores de Suporte (SVM). Inicialmente, será discutida a história da SVM, seguida sobre uma apresentação da visão geral intuitiva e matemática deste algoritmo.

2.1 História

Inicialmente proposta nos anos 90 pelos engenheiros da AT&T Bell Labs, Corinna Cortes e Vladimir Vapnik (CORTES; VAPNIK, 1995), a Máquina de Vetores de Suporte (SVM) foi concebida como um classificador binário. Esse algoritmo de aprendizado supervisionado foi baseado em um trabalho de Bernhard E. Boser, Isabelle M. Guyon e Vladimir Vapnik de 1992 (BOSER; GUYON; VAPNIK, 1992).

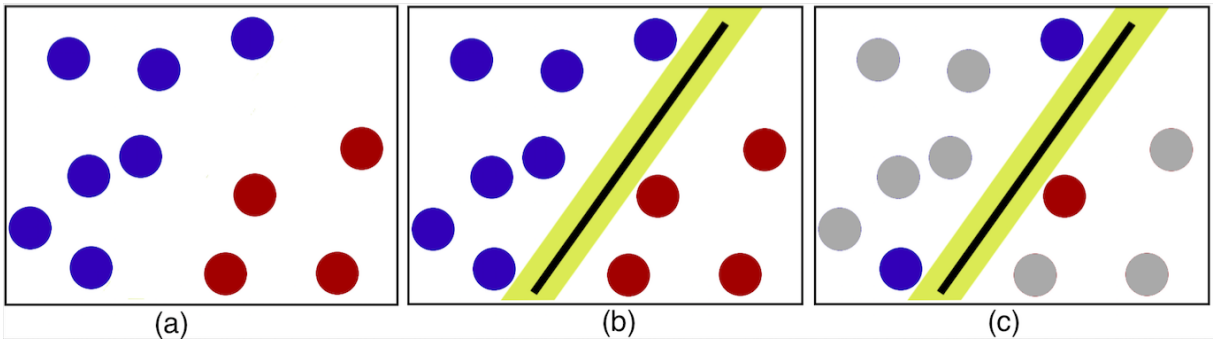
Em seu artigo de 1992, Vapnik sugeriu um método de criar classificadores não lineares, utilizando hiperplanos e funções de semelhança, chamadas de kernels. Tal algoritmo já era bem parecido com a SVM que é conhecida hoje. Entretanto, a máquina de vetores de suporte só ganhou esse nome em 1995, quando foram introduzidas ao algoritmo condições em sua fase de treinamento que o tornaram capaz de classificar dados, mesmo quando essa separação envolve erros. Assim, existe uma ponderação feita entre a largura da margem a ser maximizada e os erros de classificação. A versão mais atual da SVM também é chamada de *soft-margin SVM* e sua antecessora de *hard-margin SVM*.

2.2 Visão Geral Intuitiva

Apesar de possuir embasamento matemático relativamente complexo, a ideia por trás da SVM é simples.

Observe a Figura 1.

Figura 1 – Conjunto de dados linearmente separável



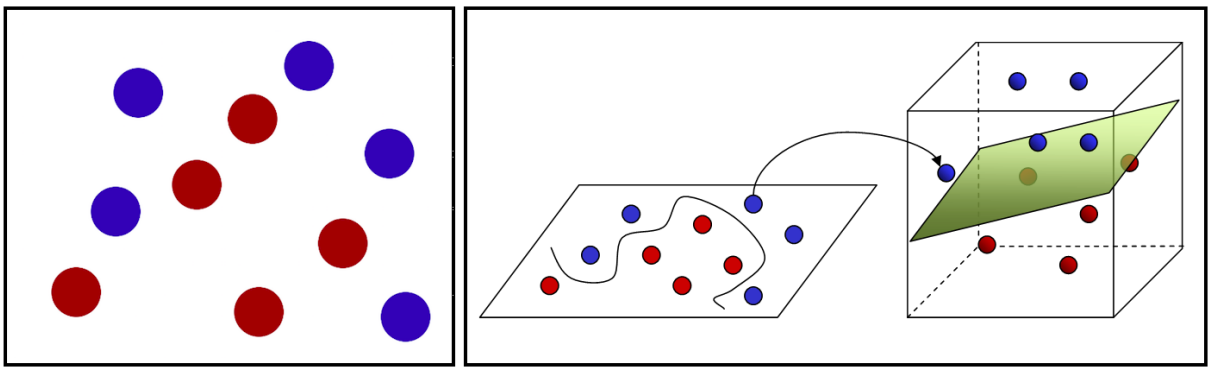
A Figura 1 (a) mostra um conjunto de dados formado por círculos de cores azul e vermelha. Cada uma dessas cores representa uma classe. Tais classes podem ser ditas linearmente separáveis, já que existe uma linha capaz de separá-las, de modo que de um lado da linha esteja apenas uma classe e do outro lado da linha esteja apenas a outra classe. Tal divisão pode ser vista na Figura 1 (b).

Note que não existe apenas uma linha capaz de separar as duas classes. Todavia, existe apenas uma linha capaz de dividir essas classes de modo a fazer com que as margens entre essas duas classes seja a maior possível: esse é o objetivo da SVM. Na Figura 1 (b) pode-se ver a linha que separa ambas as classes de modo ótimo.

Ainda observando a Figura 1 (b), observa-se que, para obter a linha capaz de separar as classes de modo a maximizar o espaço entre elas, não é necessário todo o conjunto de dados, mas apenas dos dados mais próximos da linha de classificação, como mostra a Figura 1 (c). Esses dados relevantes para a classificação são chamados de vetores de suporte (SVs).

Entretanto, existem conjuntos que não são linearmente separáveis. Como exemplo, observe a Figura 2 (a).

Figura 2 – Conjunto de dados não linearmente separável



Ao observar a Figura 2 (a), é possível notar que não existe nenhuma maneira linear capaz de separar as duas classes. Para fazer tal, pode-se mapear os pontos da Figura 2 (a) em pontos em uma dimensão maior, como mostra a Figura 2 (b). Nela, observa-se que, ao mapear o conjunto de dados original em uma outra dimensão, o que é chamado de *kernelling*, é possível separar as duas classes em questão através de um hiperplano.

Observa-se que, assim como no primeiro exemplo, o hiperplano ótimo também pode ser encontrado ao maximizar as bordas entre as duas classes. Além disso, de mesma maneira, os pontos que ajudam a encontrar esse hiperplano são chamados de vetores de suporte.

2.3 Visão Geral Matemática

2.3.1 Classificação

Máquinas de vetores de suporte foram inicialmente desenvolvidas para a classificação binária (HAYKIN, 2008). Nesse tipo de problema, tem-se o conjunto de treinamento $(\vec{x_i}, y_i)_N^{i=1}$, onde x_i são é o I-ésimo com o padrão a ser classificado e y_i sua respectiva classe. As classes devem sempre assumir o valor $y_i = \pm 1$.

O treinamento para a classificação binária consiste em resolver o problema quadrático com restrições lineares como apresentado em

$$\begin{aligned} Maximizar \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\vec{x_i}, \vec{x_j}) \\ Sujeito\ às\ condições \quad & 0 \leq \alpha_i \leq C \quad Para\ i = 1, 2, ..., N \\ & \sum_{i=1}^{N} \alpha_i y_i = 0, \end{aligned} \tag{2.1}$$

onde $K(\vec{x_i}, \vec{x_j})$ representa a função de kernel e C é um parâmetro de regularização que funciona como limitante do valor dos multiplicadores de Lagrange α_i.

Já a fase *feed-forward* da classificação, que consiste na classificação de um novo vetor $\vec{x}$, é dada por

$$y(x) = sgn \left(\sum_{i=1}^{N_{SV}} y_i \alpha_i K(\vec{x_i}, \vec{x_j}) + b \right), \tag{2.2}$$

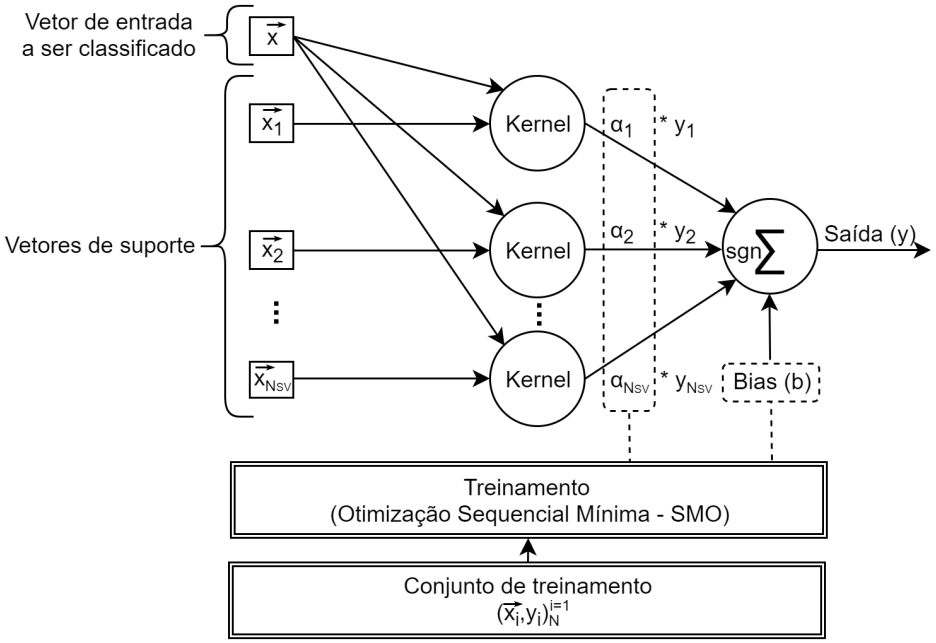
onde os parâmetros α_i e b são dados na fase de treinamento. Todos os multiplicadores de Lagrange α_i que possuírem valores diferentes de zero são chamados de Vetores de Suporte (SV). É importante notar também que o somatório da Equação (2.2) só se estende até o número de vetores de suporte N_{SV}, ao invés de se estender a todos os N pontos do conjunto de treinamento.

A arquitetura da inferência da classificação da SVM pode ser vista na parte superior da Figura 3. Nela, a inferência da SVM é ilustrada em conjunto com a fase de treinamento como visto na Equação 2.1.

2.3.2 Regressão

No caso da regressão, o problema consiste em estimar o valor de uma função para um ponto qualquer a partir de um conjunto de treinamento com um número de pontos finitos. De modo similar a classificação, tem-se como entrada da fase de treinamento um conjunto $(\vec{x_i}, y_i)_N^{i=1}$, onde $\vec{x_i}$ é o i-ésimo ponto de referência e y_i sua respectiva saída. É importante notar que, diferente do que ocorre na classificação, os valores de y_i na regressão podem assumir qualquer valor real.

Figura 3 – Arquitetura geral da inferência e do treinamento da SVM



Logo, incluindo na equação o parâmetro ε proposto por Vapnik (CORTES; VAPNIK, 1995), tem-se que a fase de treinamento da regressão é dada por

$$Maximizar \quad \sum_{i=1}^{N}(\alpha_i^* - \alpha_i)y_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)K(\vec{x_i}, \vec{x_j}) - \varepsilon\sum_{i=1}^{N}(\alpha_i^* + \alpha_i) \tag{2.3}$$

$$Sujeito\ às\ condições \quad 0 \leq \alpha_i^*, \alpha_i \leq C \quad Para\ i = 1, 2, ..., N$$

$$\sum_{i=1}^{N}(\alpha_i^* - \alpha_i) = 0,$$

onde, assim como na classificação, $K(\vec{x_i}, \vec{x_j})$ representa a função de kernel e C é um parâmetro de regularização que funciona como limitante do valor dos multiplicadores de Lagrange α_i. Já ε é um parâmetro positivo que define a *zona insensível* dentro da qual os erros são ignorados. Note que tanto C quanto ε são constantes definidas antes do treinamento de acordo com as características do conjunto de entrada.

Já a fase *feed-forward* da regressão, que consiste em estimar o valor de um novo ponto x, é dada por

$$y(x) = \sum_{i=1}^{N_{SV}}(\alpha_i^* - \alpha_i)K(\vec{x_i}, \vec{x_j}) + b, \tag{2.4}$$

onde os parâmetros α_i^*, α_i e b são resultantes da fase de treinamento. Além disso, assim como na classificação, apenas os vetores de suporte são utilizados para o somatório.

3 Otimização Sequencial Mínima (SMO)

Neste capítulo os detalhes do algoritmo de treinamento da SVM chamado de Otimização Sequencial Mínima (Sequential Minimal Optimization - SMO) serão discutidos. Inicialmente, são apresentadas as condições de convergência para esse algoritmo. Em seguida, os passos para o cálculo dessa otimização são discutidos em detalhe. Por fim, pseudocódigos das principais partes desse algoritmo são apresentados.

3.1 Condições de Karush-Kuhn-Tucker (KKT)

O treinamento de uma Máquina de Vetores de Suporte (SVM) requer a solução de um problema de otimização de programação quadrática (QP) como mostra a Equação (2.2). O SMO quebra esse grande problema de QP em uma série de problemas de QP do menor tamanho possível (PLATT, 1999). Esses problemas pequenos de QP podem ser resolvidos analiticamente, o que evita a computação numérica custosa dos grandes problemas de QP.

Um ponto é um ponto ótimo da Equação (2.2) se, e somente se, as condições de Karush-Kuhn-Tucker (KKT) são obedecidas. As condições de KKT são dadas por

$$\begin{aligned} \alpha = 0 \quad &\Rightarrow \quad y_i f(\vec{x_i}) \geq 1 \\ \alpha = C \quad &\Rightarrow \quad y_i f(\vec{x_i}) \leq 1 \\ 0 < \alpha < C \quad &\Rightarrow \quad y_i f(\vec{x_i}) = 1, \end{aligned} \tag{3.1}$$

onde f é a fase de inferência da SVM. Assim, o SMO itera até que essas condições sejam obedecidas, garantindo a convergência do algoritmo.

Objetivando encontrar esse ponto ótimo, o SMO seleciona através de uma heurística um par de αs (α_i e α_j) para otimizar, os otimiza, recalcula o valor de b de acordo com os novos αs e repete o processo até que as condições de KKT sejam obedecidas (dentro de uma certa tolerância).

3.2 Heurísticas para seleção dos αs

Parte do algoritmo do SMO é dedicado à escolha do par de αs a ser otimizado. Na literatura, existem diversas maneiras diferentes de escolher esses elementos a serem otimizados. Entretanto, não existe um modo "errado"de fazer essa escolha, mas a ordem das escolhas pode alterar a velocidade da convergência da SMO. Neste trabalho será apresentado o método originalmente proposto para a escolha desse elementos.

Existem duas heurísticas feitas na escolha dos multiplicadores de Lagrange, sendo uma para o primeiro multiplicador de Lagrange em um loop externo (α_i) e outra para o segundo em um loop interno (α_j).

Na heurística da primeira escolha (loop externo) é feita uma iteração em todo o conjunto de treinamento. Quando um elemento que não obedece as condições de KKT é encontrado esse elemento é selecionado como o α_i para ser otimizado. Objetivando acelerar o SMO, a cada loop feito em todo o conjunto de dados é feito também um loop apenas nos elementos cujos multiplicadores de Lagrange que não possuem valor 0 ou C (multiplicadores não limitados).

A heurística da segunda escolha (loop interno), por sua vez, consiste em escolher como α_j o elemento que possibilita o maior passo, sendo o erro dado por

$$E_i = f(\vec{x_i}) - y_i \tag{3.2}$$

e o passo dado por

$$|E_i - E_j|. \tag{3.3}$$

Caso o α_j escolhido com essa heurística não resulte em nenhuma melhora na convergência, é feito um loop em todos os αs não limitados até que o primeiro deles resulte em alguma melhora na convergência. Finalmente, caso nenhum deles resultar em melhora, é feito um loop em todo o conjunto de treinamento.

3.3 Otimização de α_i e α_j

Uma vez escolhidos o α_i e α_j a serem otimizados deve-se encontrar as margens Lo e Hi tal que $Lo \leq \alpha_j \leq Hi$ de modo a fazer com que α_j obedeça a restrição $0 \leq \alpha_j \leq C$ após a otimização. Caso y_i e y_j tenham valores distintos

$$Lo = max(0, \alpha_j - \alpha_i), \;\; Hi = min(C, C + \alpha_j - \alpha_i) \tag{3.4}$$

e caso y_i e y_j possuam o mesmo valor

$$Lo = max(0, \alpha_i + \alpha_j - C), \;\; Hi = min(C, \alpha_i - \alpha_j). \tag{3.5}$$

O próximo passo é encontrar o valor de α_j que maximiza a Equação (2.1). O ponto ótimo α_j é dado por

$$\alpha_j^{novo} = \alpha_j - \frac{y_j(E_i - E_j)}{\eta} \tag{3.6}$$

onde

$$\eta = 2K(\vec{x_i}, \vec{x_j}) - K(\vec{x_i}, \vec{x_i}) - K(\vec{x_j}, \vec{x_j}). \tag{3.7}$$

Caso o valor de α_j não esteja entre Lo e Hi, o valor de α_j é simplesmente saturado

$$\alpha_j^{novo} = \begin{cases} Hi & se\ \alpha_j^{novo} > Hi \\ \alpha_j^{novo} & se\ Lo \leq \alpha_j^{novo} \leq Hi \\ Lo & se\ \alpha_j^{novo} < Lo. \end{cases} \tag{3.8}$$

Por fim, uma vez que o valor de α_j foi encontrado, pode-se encontrar α_i utilizando a equação

$$\alpha_i^{novo} = \alpha_i + y_i y_j (\alpha_j - \alpha_j^{novo}). \tag{3.9}$$

É importante notar que α_i só é calculado quando o valor de α_j sofre uma mudança minimamente considerável. Assim, α_i é calculado apenas quando

$$|\alpha_j - \alpha_j^{novo}| \geq \varepsilon(\alpha_j + \alpha_j^{novo} + \varepsilon), \tag{3.10}$$

onde parâmetro positivo ε define a *zona insensível* dentro da qual os erros das condições de KKT são ignorados. Esse parâmetro normalmente possui um valor entre 10^{-2} e 10^{-3}.

3.4 Cálculo do limiar b

Apenas após otimizar α_i e α_j é possível selecionar o limiar b de modo a satisfazer as condições de KKT. O limiar b_1 é dado por

$$b_1 = -E_i - y_i(\alpha_i^{novo} - \alpha_i)K(\vec{x_i}, \vec{x_i}) - y_j(\alpha_j^{novo} - \alpha_j)K(\vec{x_i}, \vec{x_j}) + b \tag{3.11}$$

e é válido quando α_i não está dentro de seu limite (ou seja, $0 \leq \alpha_i \leq C$), pois força a saída correta da SVM quando a entrada é $\vec{x_i}$. Já o limiar b_2 dado por

$$b_2 = -E_j - y_i(\alpha_i^{novo} - \alpha_i)K(\vec{x_i}, \vec{x_j}) - y_j(\alpha_j^{novo} - \alpha_j)K(\vec{x_j}, \vec{x_j}) + b \tag{3.12}$$

é válido quando α_i não está dentro de seu limite, pois força a saída correta da SVM quando a entrada é $\vec{x_j}$. Caso tanto b_1 quanto b_2 sejam válidos, eles terão o mesmo valor. Caso os novos αs estejam ambos nas bordas (ou seja, $\alpha_i = 0$ ou $\alpha_i = C$ e $\alpha_j = 0$ ou $\alpha_j = C$) então todos os valores entre b_1 e b_2 satisfazem as condições de KKT. Nesses casos b é escolhido como o valor intermediário entre b_1 e b_2. Assim, b é dado por

$$b = \begin{cases} b_1 & se\ 0 < \alpha_i < C \\ b_2 & se\ 0 < \alpha_j < C \\ (b_1 + b_2)/2 & caso\ contrário. \end{cases} \tag{3.13}$$

3.5 Cache do erro

A cache do erro se refere a uma técnica de aceleração do SMO que consiste no armazenamento contínuo do erro de todos os elementos não limitados do conjunto de

treinamento. Essa técnica é extremamente vantajosa, já que, devido a linearidade da fase *feed-forward* da classificação presente na Equação (2.2), é possível manter o valor do erro sempre atualizado fazendo o cálculo

$$E_k^{novo} = E_k + y_i(\alpha_i^{novo} - \alpha_i)K(\vec{x_i}, \vec{x_k}) + y_j(\alpha_j^{novo} - \alpha_j)K(\vec{x_j}, \vec{x_k}) - b + b^{novo} \tag{3.14}$$

apenas quando algum α é otimizado.

O cálculo contínuo do erro evita o cálculo extremamente custoso da fase *feed-forward* da classificação, que ocorreria todas as vezes que o erro de qualquer ponto do conjunto de entrada necessitasse ser calculado.

3.6 Pseudocódigos

Os Algoritmos 1 e 2, como explicado na Seção 3.2, mostram o pseudocódigo das heurísticas da primeira e segunda escolha, respectivamente. Já o Algoritmo 3 mostra o pseudocódigo da tentativa de otimização de α_i e α_j como visto na Seção 3.3.

Algorithm 1 – Heurística da primeira escolha

```
1:  procedure mainRoutine
2:      Inicializar array dos alphas igual a zero
3:      Inicializar o limiar b igual a zero
4:      numChanged = 0;
5:      examineAll = true;
6:      while numChanged>0 || examineAll do
7:          numChanged=0
8:          if examineAll then
9:              loop i em todos os elementos
10:                 numChanged += examineExample(i);
11:         else
12:             loop i em todos os elementos não limitados
13:                 numChanged += examineExample(i);
14:         if examineAll then
15:             examineAll = false;
16:         else if numChanged == 0 then
17:             examinelAll = true;
```

Algorithm 2 – Heurística da segunda escolha

```
1: procedure examineExample(i)
2:     if ((y_i E_i < −tol && α_i < C) || (y_i E_i > tol && α_i > 0)) then
3:         if existem elementos não limitados then
4:             j = resultado da heurística da segunda escolha
5:             if takeStep(i, j) then
6:                 return 1
7:         loop em todos os elementos j não limitados começando em ponto randômico
8:             if takeStep(i, j) then
9:                 return 1
10:        loop em todos os elementos j começando em ponto randômico
11:            if takeStep(i, j) then
12:                return 1
13:    return 0
```

Algorithm 3 – Tentativa de otimização de α_i e α_j

procedure takeStep(i, j)
  if $i == j$ **then**
    return 0
  if $y_i == y_j$ **then** ▷ Equação (3.5)
    Lo = max(0,$\alpha_i + \alpha_j - C$)
    Hi = min(C,$\alpha_i - \alpha_j$)
  else ▷ Equação (3.4)
    Lo = max(0,$\alpha_j - \alpha_i$)
    Hi = min(C,$C + \alpha_j - \alpha_i$)
  if $Lo == Hi$ **then**
    return 0
  eta=$2K(\vec{x_i}, \vec{x_j}) - K(\vec{x_i}, \vec{x_i}) - K(\vec{x_j}, \vec{x_j})$; ▷ Equação (3.7)
  if eta<0 **then**
    auxJ=$\alpha_j - y_j(E_i - E_j)$/eta; ▷ Equação (3.6)
    if auxJ<Lo **then** ▷ Equação (3.8)
      auxJ = Lo;
    if auxJ>Hi **then**
      auxJ = Hi;
  else
    return 0 ▷ Ocorre apenas Raramente
  if auxJ<1e-8 **then** ▷ Arredondamento de limitantes
    auxJ=0;
  if auxJ>C-1e-8 **then**
    auxJ=C;
  if |auxJ - α_j| < eps(auxJ + α_j + eps) **then** ▷ Equação (3.10)
    return 0
  auxI = $\alpha_i + y_i * y_j(\alpha_j - auxJ)$ ▷ Equação (3.9)
  auxB=b;
  $b_1 = -E_i - y_i(auxI - \alpha_i)K(\vec{x_i}, \vec{x_i}) - y_j(auxJ - \alpha_j)K(\vec{x_i}, \vec{x_j}) + b$; ▷ Equação (3.11)
  $b_2 = -E_j - y_i(auxI - \alpha_i)K(\vec{x_i}, \vec{x_j}) - y_j(auxJ - \alpha_j)K(\vec{x_j}, \vec{x_j}) + b$; ▷ Equação (3.12)
  if $0 \leq \alpha_i \leq C$ **then** ▷ Equação (3.13)
    $b = b1$;
  else if $0 \leq \alpha_j \leq C$ **then**
    $b = b2$;
  else
    $b = (b1 + b2)/2$
  loop em todos os elementos não limitados iterando em k ▷ Equação (3.14)
    $E_k = E_k + y_i(auxI - \alpha_i)K(\vec{x_i}, \vec{x_k}) + y_j(auxJ - \alpha_j)K(\vec{x_j}, \vec{x_k}) - auxB + b$;
  α_i = auxI;
  α_j = auxJ;
  return 1

4 Computação Reconfigurável

Nesse capítulo será apresentada uma visão geral da computação reconfigurável. Inicialmente, será apresentada a motivação para o surgimento da computação reconfigurável. Em seguida, será apresentado o Arranjo de Portas Programáveis em Campo (Field-Programmable Gate Array - FPGA) e sua estrutura interna será discutida. Posteriormente, serão apresentados as três fases principais do desenvolvimento de um circuito em FPGA.

4.1 Visão Geral

Várias propostas de implementação em Circuitos Integrados de Aplicação Específica (Application Specific Integrated Circuits - ASICs) foram feitas como modo de acelerar o processamento de algoritmos de aprendizado de máquina (BAPTISTA et al., 2012). Apesar de possuírem um processamento extremamente rápido, os ASICs possuem custo de fabricação elevado e as aplicações em ASICs resultam em pouca flexibilidade (SOUZA; FERNANDES, 2014). Essa baixa flexibilidade se deve ao fato do ASIC não poder ser alterado após sua fabricação, o que força a criação de um novo projeto e a nova fabricação do chip toda vez que uma modificação é necessária.

Outro modo comum de implementar algoritmos de aprendizado de máquina é utilizando microprocessadores. Estes possuem extrema flexibilidade e preço de projeto reduzido, entretanto seu desempenho é apenas uma fração do desempenho obtido quando ASICs são utilizados.

Objetivando preencher essa lacuna entre ASICs e microprocessadores, surgiu a computação reconfigurável (COMPTON; HAUCK, 2002). Sua característica principal é o grande aumento de desempenho atrelado a uma flexibilidade de nível semelhante ao que tem-se em um software. Os dispositivos reconfiguráveis mais comuns são as FPGAs e os CPLDs.

4.2 FPGAs

4.2.1 Introdução às FPGAs

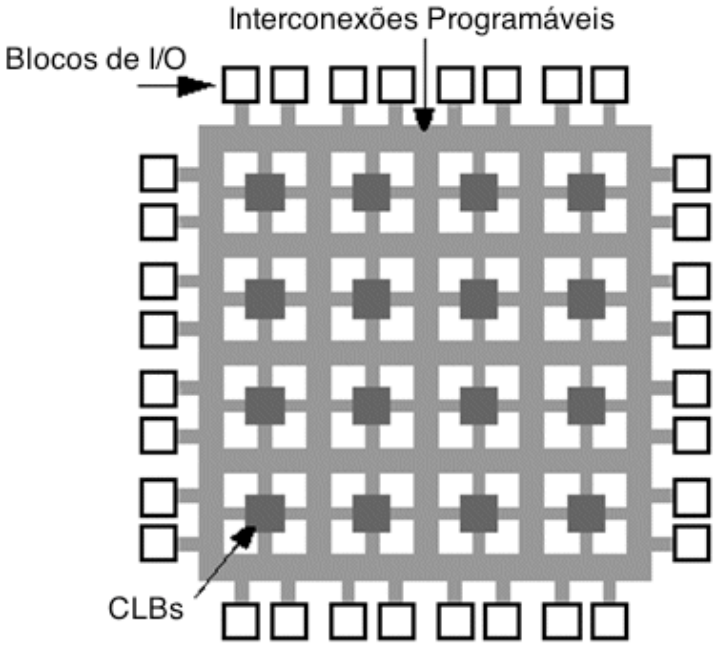
Atualmente, a arquitetura reconfigurável mais amplamente utilizada é a FPGA. Assim como os ASICs, uma FPGA pode ser programada para implementar qualquer circuito digital possível.

A primeira FPGA comercial, chamada de XC2064, foi desenvolvida pelos co-

fundadores da empresa Xilinx, Ross Freeman e Bernard Vonderschmitt, em 1985. Esses dispositivos logo se tornaram extremamente populares e, no começo dos anos 90, houve um grande aumento tanto no volume de produção quanto na sofisticação desses dispositivos. No fim dessa década, as FPGAs, que antes eram utilizadas principalmente para telecomunicações, passaram a ser utilizadas também em aplicações industriais e automobilísticas (MAXFIELD, 2012).

Esses dispositivos de silício pré-fabricados são compostos por uma matriz de blocos lógicos programáveis (CLBs) interligados por interconexões programáveis. Essa matriz possui em suas extremidades blocos de I/O, responsáveis pela entrada e saída dos sinais. A estrutura simplificada de uma FPGA está ilustrada na Figura 4.

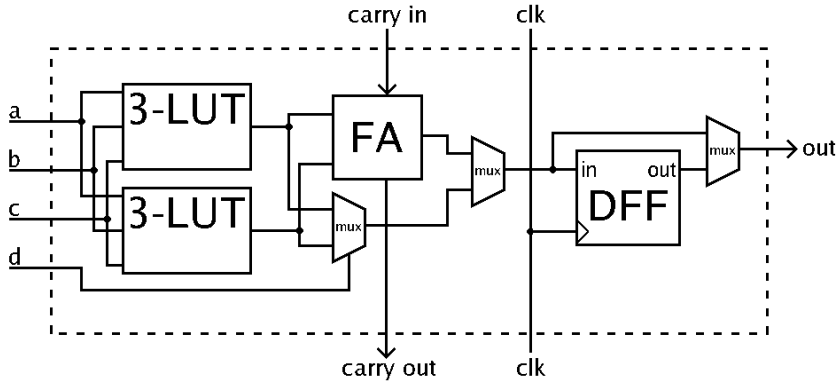
Figura 4 – Estrutura simplificada de uma FPGA



Os CLBs, por sua vez, são compostos por um conjunto de células lógicas. Uma célula típica contém tabelas de busca (LUTs) de 4 a 6 entradas, um somador completo (FA) e um flip-flop tipo D (DFF) como mostra a Figura 5.

Modelos de FPGAs modernas também podem incluir alguns componentes adicionais. Dentre eles, os mais comumente encontrados são blocos específicos para o processamento digital de sinais (DSP). Tais componentes podem ser extremamente úteis, visto que operações de multiplicação em FPGAs, por exemplo, tendem a ocupar uma grande quantidade de LUTs. Um exemplo de DSPs são os DSP48E1s, encontrados nas séries 6 e 7 das famílias Virtex e Spartan das FPGAs produzidas pela Xilinx.

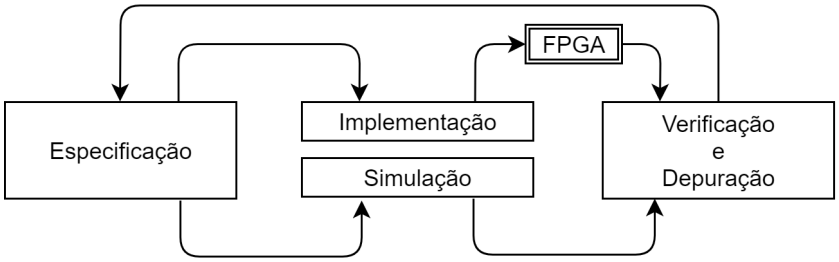
Figura 5 – Estrutura genérica de uma célula lógica



4.2.2 Especificação para hardware

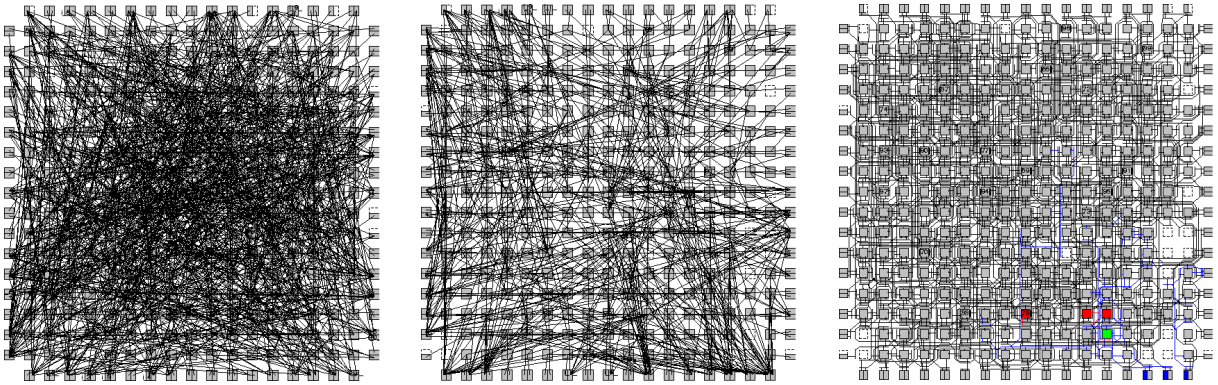
De modo geral, em um projeto que tem como alvo uma FPGA existem três fases principais: especificação, implementação (ou simulação) e verificação e depuração. Como representado na Figura 6, durante o design o projetista passa por cada uma dessas fases mais de uma vez para fazer ajustes e correções de erros.

Figura 6 – Fases do processo de design de um circuito em FPGA



A primeira fase é a de especificação. A maneira mais comum de especificar um circuito em hardware é através de linguagens de descrição de hardware (Hardware Description Language - HDL), como VHDL, Verilog, entre outras. Além disso, existem também ferramentas gráficas de projeto, como é o caso do System Generator comentado na Seção 1.2. No caso das ferramentas gráficas, os blocos básicos são descritos internamente pelo programa utilizando uma HDL. Além disso, tanto a implementação utilizando HDL quanto a utilizando ferramentas gráficas podem ser aceleradas através da utilização de componentes de alto nível, incluindo componentes de propriedade intelectual (Intellectual Property - IP Blocks).

Entretanto, esses métodos de especificação de hardware são considerados difíceis de serem utilizados. Por esse motivo, diversas empresas como Intel e Xilinx estão investindo massivamente em linguagens de síntese de alto nível (High level Synthesis - HLS). Essas linguagens tem como objetivo converter códigos em linguagens de alto nível, como C, em HDL. Isso permite que programadores de software possam criar circuitos em hardware sem conhecimento profundo de hardware. Dentre as linguagens mais atuais de HLS destacam-se



(a) Posicionamento inicial (b) Posicionamento final (c) Circuito após roteamento

Figura 7 – Ilustração do processo de posicionamento e roteamento em uma FPGA

o OpenCL, mantida por diversas empresas de hardware reconfigurável, e o LegUp, uma ferramenta de código aberto mantida pela University of Toronto. Apesar dessas ferramentas acelerarem o processo de design em hardware, elas normalmente ainda geram softwares com maior área, maior delay e maior número de clocks. Em contrapartida essas implementações são mais portáveis e fáceis de serem mantidas.

4.2.3 Implementação ou Simulação

A segunda fase do design de um circuito em FPGA pode ser ou a implementação em hardware ou a simulação do design, dependendo da ferramenta que se está sendo utilizada. A implementação ocorre em três etapas principais. Inicialmente é feito o mapeamento (mapping), onde as operações de computação são combinadas ou decompostas de acordo com os blocos de hardware específicos da FPGA alvo. Após o mapeamento, os blocos de hardware mapeados são posicionados (placement) na FPGA de modo a maximizar a proximidade entre os blocos lógicos que se comunicam. Por fim, é feito o roteamento (routing), onde as interconexões físicas da FPGA conectam os blocos previamente colocados em seus respectivos locais. Ferramentas atuais de implementação fazem esse processo de modo iterativo objetivando diminuir a área ocupada e aumentar o clock máximo ao longo das iterações. Uma ilustração do processo de posicionamento e roteamento pode ser visto na Figura 7.

A estrutura interna de cada FPGA influencia fortemente a complexidade da implementação. De modo geral, a fase de implementação em FPGA é extremamente complexa, o que leva os algoritmos que realizam esses passos a utilizar heurísticas e otimizações genéricas que não garantem encontrar a melhor solução, como é o caso do arrefecimento simulado (simulated annealing). Assim, a implementação de um algoritmo em FPGA pode ser extremamente demorado.

Objetivando diminuir o tempo necessário para o processo de design de um circuito em FPGA são frequentemente utilizadas simulações ao invés da implementação. O software

System Generator, por exemplo, permite a simulação do hardware especificado utilizando entradas e saídas do próprio Matlab, o que facilita imensamente o processo de depuração.

4.2.4 Verificação e Depuração

Uma vez que o circuito especificado já está simulado ou implementado faz-se necessário averiguar se os requisitos do problema foram cumpridos. Para isso, podem ser verificados a área ocupada pelo circuito na FPGA, o número de clocks, o delay máximo do circuito, entre outros. É importante observar que, no caso do circuito simulado, apesar de ser possível obter a informação precisa sobre o número de clocks, é apenas possível aproximar a área ocupada e o delay máximo do circuito.

Além da verificação dos requisitos, é feita também uma análise do bom funcionamento do sistema. Caso seja necessário corrigir erros funcionais do circuito ou o circuito não atenda aos requisitos do problema, a especificação do circuito deve ser modificada e o circuito deve ser novamente simulado ou implementado.

5 Proposta de Implementação da fase *feed-forward*

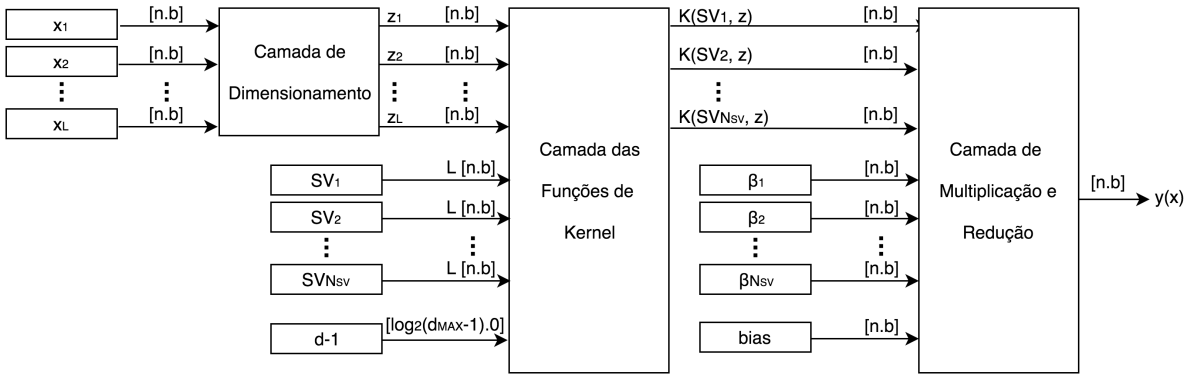
Nesse capítulo será feita a apresentação da proposta de implementação da fase de inferência da SVM. Inicialmente será apresentada a estrutura geral da arquitetura proposta e cada uma das camadas desta arquitetura será discutida em detalhes. Em seguida, serão apresentados resultados da classificação e regressão obtidos através de simulação com o Matlab. Por fim, os resultados da síntese da arquitetura proposta serão apresentados e discutidos.

5.1 Descrição do Projeto

5.1.1 Estrutura geral da implementação

A estrutura geral da arquitetura proposta está presente na Figura 8. Tal estrutura é formada por três camadas principais que serão explicadas em detalhes nas próximas seções. A camada inicial é responsável pelo dimensionamento da entrada e as demais camadas são responsáveis pela implementação da inferência, como visto na parte superior da Figura 3.

Figura 8 – Estrutura geral da arquitetura proposta



É importante notar que, na Figura 8, a notação [n.b] indica que são utilizandos n bits, dos quais b bits são utilizados para a representação da parte fracionária e n-b bits utilizados para a representação da parte inteira. A lógica de funcionamento das três camadas que compõem a estrutura em questão será explicada nas subseções 5.1.3, 5.1.4 e 5.1.5.

5.1.2 Unificação do hardware para classificação e regressão

Ao observar as Equações (2.2) e (2.4) é possível notar que existe uma grande semelhança entre elas. Analisando apenas o somatório dessas duas equações, tem-se, para a classificação

$$y_i \alpha_i K(x_i, x_j) \tag{5.1}$$

e para a regressão

$$(\alpha_i^* - \alpha_i) K(x_i, x_j). \tag{5.2}$$

Como descrito no Capítulo 2, tanto os valores de $y_i\alpha_i$ quanto os valores de $(\alpha_i^* - \alpha_i)$ são constantes atreladas aos vetores de suporte. Devido ao fato desses valores já serem conhecidos nas fases de classificação e regressão, é possível unificá-los (RUIZ-LLATA; GUARNIZO; YÉBENES-CALVINO, 2010). Reescrevendo as Equações (5.1) e (5.2) de modo a unificar o hardware necessário para computar esses valores, tem-se a Equação

$$\beta_i K(x_i, x_j). \tag{5.3}$$

Na Equação (5.3), β_i é a constante unificada de classificação e regressão. Essa constante é dada por $y_i\alpha_i$ na classificação e por $\alpha_i^* - \alpha_i$ na regressão.

5.1.3 Camada de dimensionamento

Diversos estudos comprovaram a eficiência da normalização do conjunto de treinamento na melhoria do desempenho das classificações de redes neurais (HANRAHAN, 2007). Além disso, tal técnica mostra-se especialmente importante para as implementações das arquiteturas em hardware, visto que a normalização causa, logicamente, uma maior padronização das entradas. Essa padronização torna possível para o projetista dimensionar o número de bits dos componentes RTL de modo mais eficiente, fazendo com que um menor número de bits possa ser utilizado sem que ocorram problemas como *overflow* ou *underflow*.

A normalização aqui proposta envolve centrar todo o conjunto de treinamento em sua média e, em seguida, dividir todo o conjunto por seu desvio padrão. Logo, o conjunto de dados redimensionado terá sempre média zero e desvio padrão unitário. O cálculo para a normalização é então dado por

$$z_{ij} = \frac{x_{ij} - \bar{x_j}}{\sigma_{x_j}}, \tag{5.4}$$

onde z_i é o valor autodimensionado de x_i, $\bar{x}$ é a média de todo o conjunto de treinamento e σ_x o desvio padrão desse mesmo conjunto.

É notório que, já que a normalização é feita na fase de classificação, é essencial que o dimensionamento da entrada a ser testada também seja feito na fase *feed-forward*. Para

tal, os valores de sh_j e sf_j são salvos na fase de classificação através dos cálculos

$$sh_j = -\bar{x_j} \tag{5.5}$$

e

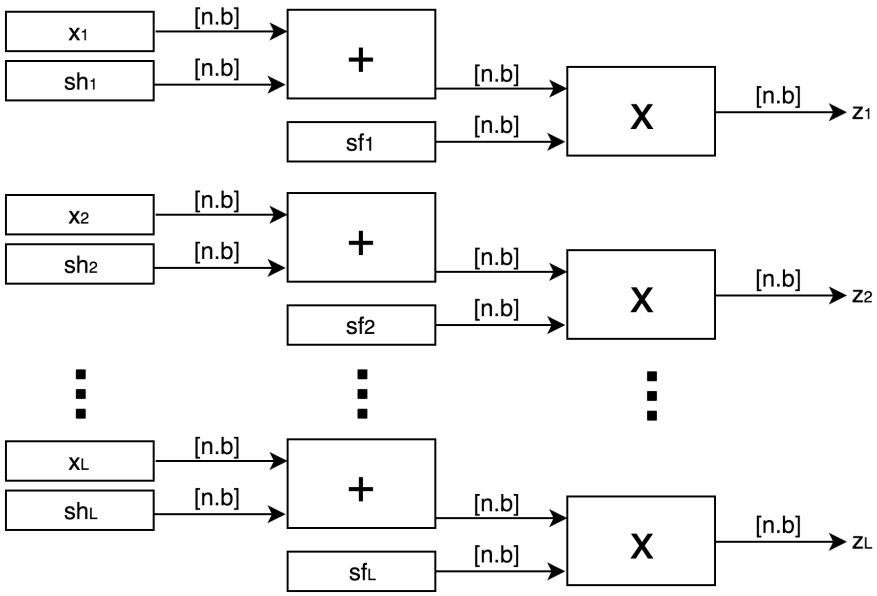
$$sf_j = \frac{1}{\sigma_{x_j}}. \tag{5.6}$$

Nas Equações (5.5) e (5.6), sh_j é o fator de deslocamento e sf_j é o fator de escala. Logo, o dimensionamento de um ponto qualquer na fase *feed-forward* pode ser dado por

$$z_{ij} = (x_{ij} + sh_j) \times sf_j. \tag{5.7}$$

Assim, a arquitetura da camada de dimensionamento pode ser implementada como mostra a Figura 9.

Figura 9 – Arquitetura da camada de dimensionamento



5.1.4 Camada das funções de kernel

Um kernel nada mais é do que uma função de similaridade responsável por receber um par de conjuntos como entrada e ter como saída um número escalar que indica a similaridade entre eles. Os mais utilizados são o polinomial e o Gaussiano. Entretanto, existem outros kernels propostos pela literatura que são menos utilizados, como é o caso do kernel linear e do kernel MLP. Existe também o kernel amigável ao hardware (Hardware-Friendly kernel), que possui propriedades interessantes e será discutido detalhadamente na subseção 6.1.2. A Tabela 1 lista os kernels mencionados.

O kernel polinomial tem bom desempenho de classificação em comparação aos demais, sendo capaz de classificar uma vasta gama de problemas não linearmente separáveis.

Tabela 1 – Funções de kernel

Kernel	Descrição*
Polinomial de grau d	$\left[(x^T x_i) + 1\right]^d$
Gaussiano (ou RBF)	$e^{\left(-\|x_i - x\|^2 / 2\theta^2\right)}$
Hardware-friendly	$2^{-\gamma \|x_i - x\|_1}$
Linear	$x^T x_i$
MLP	$tanh\left(k\, x^T x_i + \theta\right)$

* d, θ e k são parâmetros especificados pelo usuário.

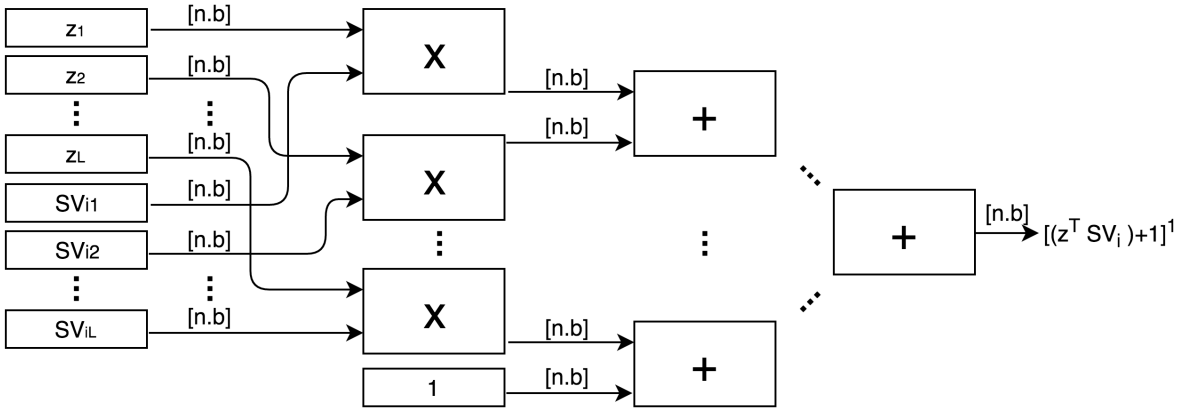
Além disso, ele é muito citado na literatura, principalmente devido a seu bom desempenho em processar linguagem natural (GOLDBERG; ELHADAD, 2008). Este também possui estrutura simples, o que permite a implementação em hardware de maneira intuitiva e altamente paralela. Devido a esses motivos, o kernel polinomial foi escolhido para a implementação em hardware na fase *feed-forward* deste trabalho.

Como mostrado na Tabela 1, o cálculo do kernel polinomial é composto pela multiplicação de $(x^T x_i) + 1$ por ele mesmo um número d de vezes, onde d especifica o grau do kernel polinomial. Note que, como o resultado da normalização de x é chamado de z e os vetores de suporte são chamados de SV, é possível reescrever a equação do kernel polinomial da Tabela 1 da seguinte maneira:

$$K(z, SV_i) = \left[(z^T SV_i) + 1\right]^d. \tag{5.8}$$

Logo, para implementar o kernel polinomial de grau d, deve-se inicialmente implementar o kernel polinomial de grau 1. A estrutura utilizada para tal está presente na Figura 10, onde L é a dimensão da entrada.

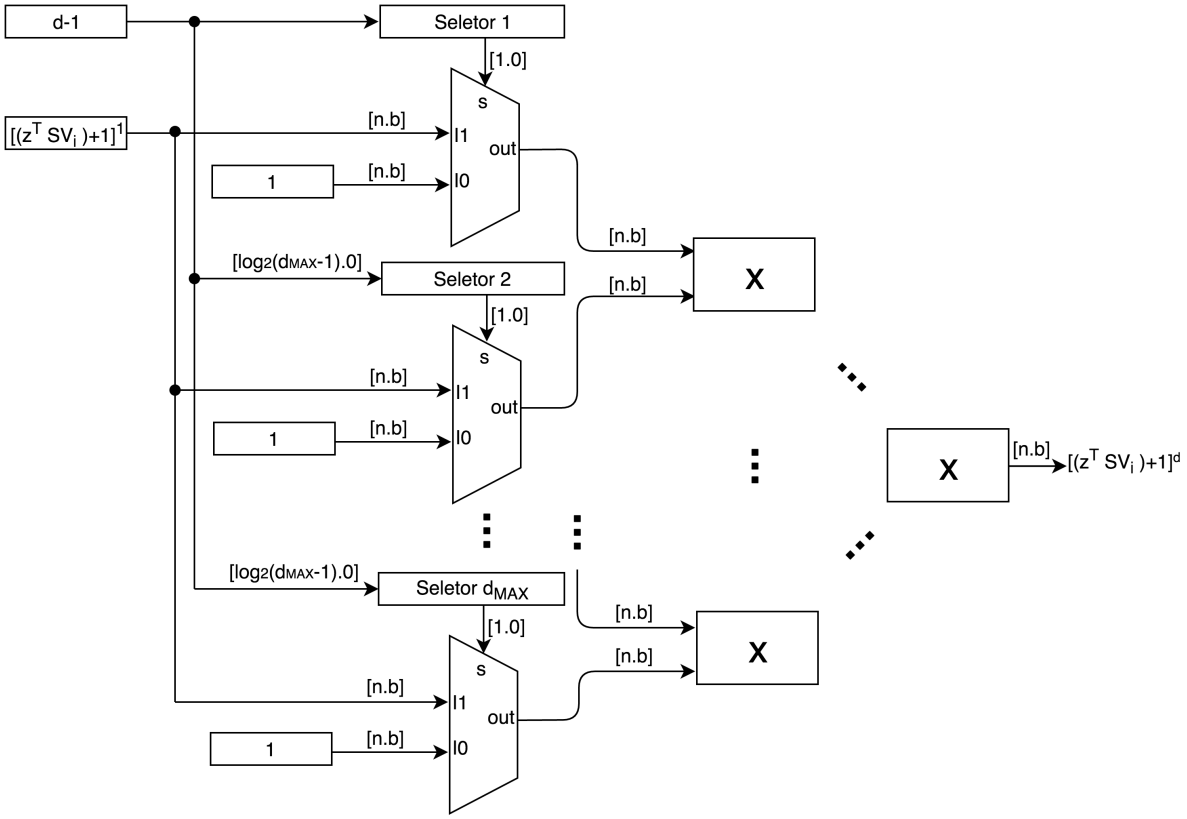
Figura 10 – Cálculo do kernel polinomial de grau 1



O grau do kernel polinomial é um parâmetro que deve ser definido pelo usuário antes da fase de treinamento e, durante a fase *feed-forward*, o mesmo parâmetro d deve ser utilizado. Esse valor é de extrema importância para a classificação, visto que um grau

baixo tende a não ser capaz de resolver problemas complexos enquanto um grau maior pode causar erros de sobreajuste (CHANG et al., 2010). O grau mais comumente utilizado para o kernel em questão é o grau 2. Entretanto, como o grau mais adequado do kernel varia de acordo com o problema, o mais correto é o desenvolvimento de uma arquitetura que permita a utilização de um grau d qualquer. A Figura 11 mostra a arquitetura do kernel polinomial de grau d qualquer.

Figura 11 – Cálculo do kernel polinomial de grau d



Para o cálculo do kernel polinomial de grau d, foram utilizados d kernels polinomiais de grau 1, como mostrado na Figura 10. Objetivando permitir ao usuário escolher um grau qualquer para o kernel, multiplexadores foram utilizados para selecionar entre o kernel de grau 1 ou a constante 1, que funciona como elemento neutro da multiplicação. Em seguida, todas as saídas dos multiplexadores são multiplicadas em árvore, nos dando como resultado $\left[(z^T SV_i) + 1\right]^d$.

É importante observar que o primeiro multiplexador da Figura 11 pode ser removido, tendo em vista que no mínimo um deles deve selecionar a função de kernel de grau 1 em vez da constante 1, já que não faz sentido que exista um kernel polinomial de grau 0. Por esse mesmo motivo, a entrada dos seletores, que são os responsáveis por fazer a seleção dos multiplexadores, é $d - 1$ ao invés de d, o que possibilita que o número de bits da lógica combinacional seja $log_2(d_{max} - 1)$, onde d_{max} é o grau máximo do kernel da implementação.

A lógica dos seletores é dada por uma simples regra: a saída do seletor para s é igual a zero quando a entrada $d-1$ é menor do que o índice do seletor e é igual a 1 caso contrário. O modo de elaborar a lógica combinacional dos seletores, tomando como exemplo um kernel onde d_{max} é 4, está representado na Tabela 2.

Tabela 2 – Tabela verdade dos seletores para $d_{max} = 4$

Seletor	Bits de $d-1$		Saída
	bit1	bit0	s
#1	1	1	1
	1	0	1
	0	1	1
	0	0	1
#2	1	1	1
	1	0	1
	0	1	1
	0	0	0
#3	1	1	1
	1	0	1
	0	1	0
	0	0	0
#4	1	1	1
	1	0	0
	0	1	0
	0	0	0

Observando a Tabela 2 nota-se que, como mencionado anteriormente, o primeiro multiplexador pode ser desprezado. Já a lógica dos demais multiplexadores para o exemplo em questão é dada como mostra a Tabela 3.

Tabela 3 – Lógica combinacional dos seletores para $d_{max} = 4$

Seletor	Lógica Combinacional
#1	1
#2	bit1 | bit0
#3	bit1
#4	bit1 & bit0

5.1.5 Camada de multiplicação e redução

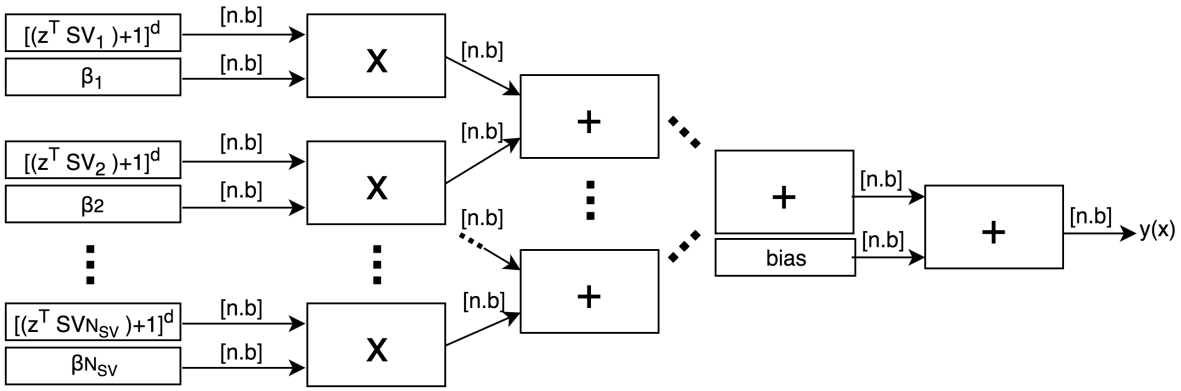
A última camada da arquitetura da SVM proposta é a camada de multiplicação e redução, que tem como principais funções:

- Multiplicar as N_{SV} saídas da camada de kernel por suas respectivas N_{SV} constantes unificadas de classificação e regressão β;

- Reduzir os resultados das N_{SV} multiplicações em uma soma em árvore;
- Somar a bias ao resultado da redução, resultando na saída final $y(x)$.

A arquitetura da camada de multiplicação e redução pode ser vista na Figura 12. É importante notar que a soma da *bias* ao resultado da redução também pode ser feita em árvore. Tal abordagem pode ser útil quando o número de vetores de suporte não é potência de 2.

Figura 12 – Arquitetura da camada de multiplicação e redução



5.2 Resultados

5.2.1 Metodologia

Essa seção tem como objetivo apresentar os testes e resultados da arquitetura descrita na Seção 5.1. Para testar o hardware implementado, são feitas simulações da classificação da porta OU exclusivo (XOR) e da regressão da função $sin(x) + cos(y)$.

A porta XOR foi escolhida como teste para a classificação por ser um exemplo simples, frequente na literatura (GU; HAN, 2013) e não ser linearmente separável. Já a função $sin(x)+cos(y)$ foi escolhida para testar a regressão devido a sua maior complexidade, que serve como teste para averiguar que a estrutura implementada é capaz de atuar satisfatoriamente em casos mais desafiadores.

Um gerador de números pseudo-aleatórios é responsável por gerar as entradas para a SVM implementada, que tem seu resultado comparado com o bloco de referência. A SVM implementada, por sua vez, possui 16 vetores de suporte e entrada x com duas dimensões ($L = 2$), o que nos permite, por exemplo, classificar pontos em um espaço bidimensional e fazer a regressão de superfícies no espaço tridimensional. Além disso, optou-se por implementar o kernel polinomial com $d_{max} = 4$, pois este é suficiente para a classificação e regressão de problemas relativamente complexos sem que muito espaço seja consumido ou problemas de sobreajuste ocorram. Os valores dos testes são armazenados e

processados, permitindo a geração de gráficos e dados numéricos que comprovam a eficácia da arquitetura proposta na classificação e regressão dos casos propostos.

Finalmente, é feita uma análise de uso da FPGA alvo Virtex 6 xc6vlx240t-1ff1156 através do relatório gerado durante a síntese da arquitetura proposta. Esse relatório contém dados relevantes, como o número de LUTs e multiplicadores utilizados, assim como a taxa máxima de amostragem do design implementado utilizando a FPGA alvo.

5.2.2 Resultados de Simulação

5.2.2.1 Classificação

Objetivando treinar a SVM para classificar pontos de acordo com a porta XOR utilizando a arquitetura implementada, foram utilizados kernels polinomiais de graus 2, 3 e 4. Todos esses kernels obtiveram excelentes resultados para classificar a porta em questão. Objetivando mostrar o bom desempenho da SVM mesmo utilizando polinômios de grau baixo, apresenta-se abaixo os resultados da classificação utilizando o grau d=2.

O conjunto de treinamento da porta XOR é dado como mostra a Tabela 4.

Tabela 4 – Conjunto de treinamento para a porta XOR

x_1	x_2	$y(x)$
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

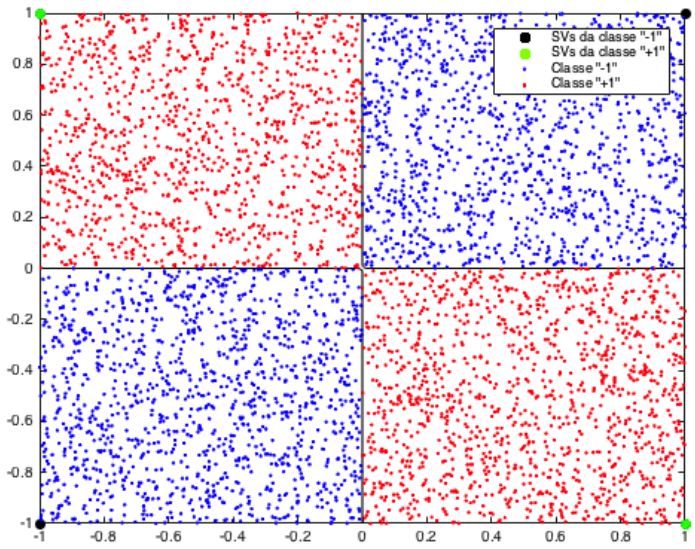
Após o treinamento *offline* no Matlab utilizando o conjunto de treinamento presente na Tabela 4, foram obtidos a *bias* e os vetores de suporte SV, assim como seus respectivos multiplicadores de Lagrange α. Multiplicando os vetores de suporte por seus multiplicadores de Lagrange, as constantes unificadas de classificação e regressão β são obtidas, como visto na Equação (5.1).

Utilizando esses dados obtidos durante a fase de treinamento na SVM implementada e gerando cinco mil pontos pseudo-aleatórios entre -1 e 1 para ambas as entradas, foram obtidos os pontos classificados como mostra a Figura 13.

Ao observar a Figura 13, vê-se que os quatro conjuntos de pontos do treinamento se tornaram vetores de suporte. Assim, como nossa implementação possui capacidade para computar a fase *feed-forward* com até 16 vetores de suporte, basta que os 12 vetores de suporte não utilizados sejam preenchidos com entradas nulas.

A curva de classificação utilizada como referência está centrada no ponto $(0, 0)$ e percorre os eixos x e y de modo a maximizar a margem dos dados do treinamento.

Figura 13 – Classificação da porta XOR



Ao observar os pontos classificados pela SVM implementada, que foram coloridos de azul e vermelho, observa-se que a estrutura proposta foi capaz de classificar os pontos pseudo-aleatórios com grande acurácia. Neste simples exemplo, constatou-se que com apenas 8 bits, sendo 4 deles utilizados para a parte fracionária, foi possível executar o teste proposto com taxa de erros nula. Entretanto, é importante notar que a não ocorrencia de erros no teste em questão não significa que nunca ocorrerão erros em outros conjuntos de valores aleatórios.

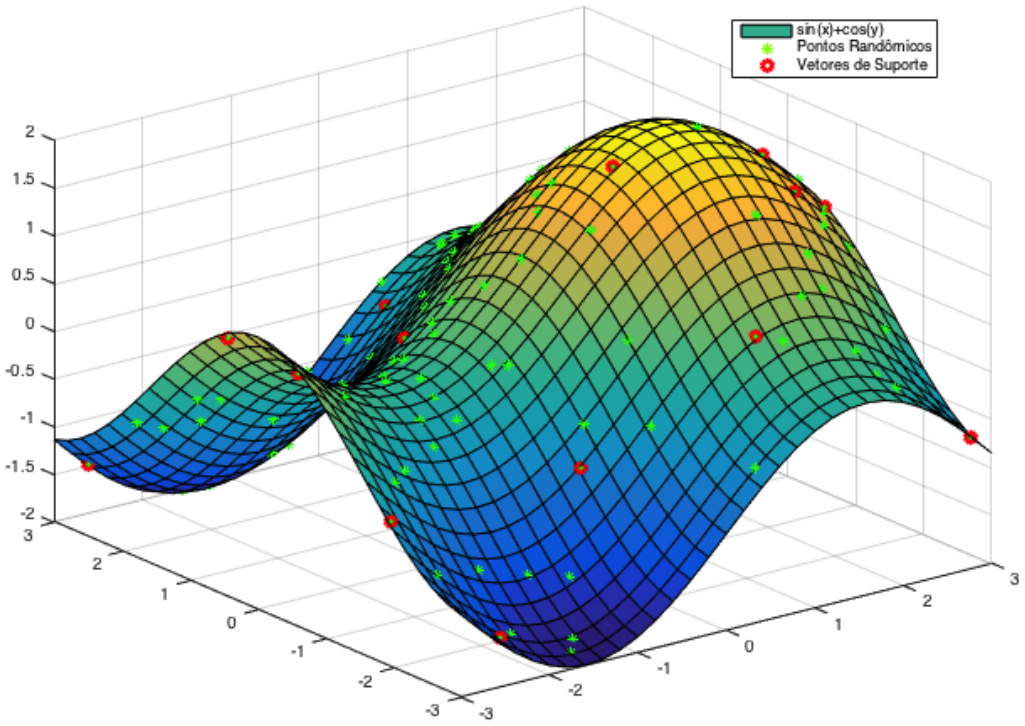
5.2.2.2 Regressão

Para o treinamento da regressão da função $sin(x) + cos(y)$, foram gerados 100 pares de pontos pseudo-aleatórios (x, y) ente $-\pi$ e π. Os valores desses pontos foram computados para a função em questão e esses dados foram utilizados como o conjunto de treinamento da regressão. Tal abordagem garante uma situação de teste válida para o problema de estimativa de uma função. (SCHOELKOPF; SMOLA, 2002). A Figura 14 mostra a função de referência, assim como os pontos aleatórios gerados.

Durante o treinamento *offline* feito no Matlab, diversos graus diferentes foram testados para o kernel polinomial. Os kernels polinomiais de grau d inferiores a 4 não foram capazes de regredir a função $sin(x) + cos(y)$ corretamente. Entretanto, o kernel polinomial de grau 4 apresentou bons resultados. Como nossa implementação da SVM suporta kernels de ordem menor ou igual a 4, a regressão da função em questão pode ser feita utilizando o hardware proposto.

Ao treinar a função $sin(x) + cos(y)$ utilizando o kernel polinomial de grau 4 e os pontos pseudo-aleatórios gerados, 16 vetores de suporte foram obtidos, como mostrado em

Figura 14 – Curva utilizada para regressão e seus vetores de suporte



vermelho na Figura 14. Dado que nossa implementação suporta até 16 vetores de suporte, observa-se que o hardware proposto será capaz de atender ao problema em questão.

Utilizando os dados obtidos durante a fase de treinamento na SVM implementada e gerando uma malha de pontos igualmente espaçados entre $-\pi$ e π, obtêm-se uma curva dos pontos regredidos durante a fase *feed-forward*, como mostra a Figura 15.

Ao comparar a curva regredida presente na Figura 15 com a curva de referência presente na Figura 14, é possível notar que existe uma enorme semelhança entre elas. Para uma análise numérica mais detalhada, pontos aleatórios foram gerados entre $-\pi$ e π e seus valores foram computados tanto pela função de referência quanto pela SVM treinada.

Integrando o erro quadrático obtido durante este experimento e o dividindo pelo número de pontos aleatórios gerados, obtêm-se um erro quadrático médio de $5,3 \times 10^{-3}$. É importante notar que, objetivando maior precisão durante a regressão, o número de bits n utilizados foi 32, sendo 16 bits para sua parte fracionária b.

5.2.3 Resultados de Síntese

A Tabela 5 mostra o resultado da síntese em uma FPGA Virtex 6 xc6vlx240t-1ff1156 da arquitetura proposta na Seção 5.1.

Figura 15 – Curva resultante da regressão após treinamento

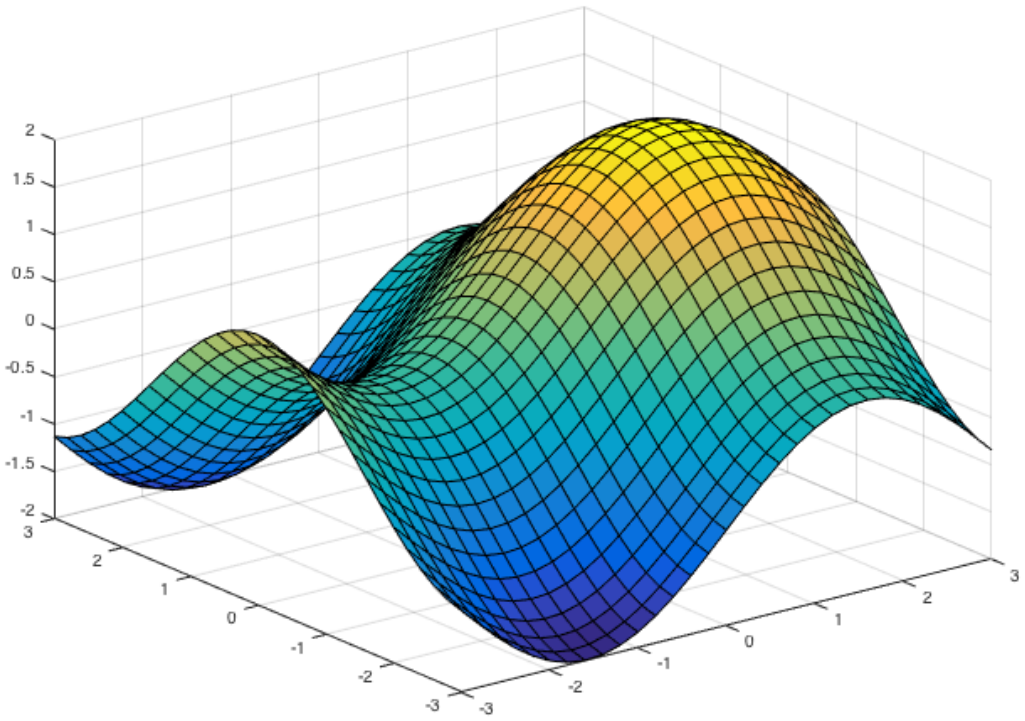


Tabela 5 – Resultados da síntese da SVM proposta

Vetores de suporte	2	4	8	16
Multiplicadores (DSP48E1s)	56 (7,29%)	104 (13,5%)	200 (26,0%)	392 (51,0%)
LUTs	256 (0,17%)	448 (0,30%)	832 (0,55%)	1,600 (1,06%)
Throughput (MHz)	18,282	17,208	15,309	13,948

Para a síntese, todos os elementos do hardware proposto foram otimizados para velocidade. Além disso, para otimizar o uso da área da FPGA, os multiplicadores internos (DSP48E1s) foram utilizados sempre que possível. Objetivando obter uma implementação com maior precisão, foram utilizados componentes de 32 bits, sendo 16 bits para a parte fracionária ([32.16] de acordo com a notação da Figura 8).

É importante notar na Tabela 5 que o clock máximo atingido é igual ao throughput da fase *feed-forward*, visto que apenas um ciclo é necessário para computar essa inferência.

Como é possível observar na Tabela 5, o número de multiplicadores e de LUTs cresceu de modo aproximadamente linear com o aumento do número de vetores de suporte. Esse comportamento é esperado, visto que grande parte da estrutura física é composta pelo cálculo da função de kernel para cada um dos vetores de suporte.

Por fim, observa-se que o hardware proposto é extremamente paralelizável, já que

não houve perda de desempenho significativa com o aumento do número de vetores de suporte. Esse comportamento é esperado, já que a maior parte do processamento está no cálculo das funções de kernel e apenas uma pequena parte no dimensionamento da entrada e na soma em árvore.

Ao extrapolar os dados da Tabela 5 é possível comparar a arquitetura proposta com outras propostas de implementação presentes na literatura. O hardware apresentado em (RUIZ-LLATA; GUARNIZO; YÉBENES-CALVINO, 2010) pode ser executado a um clock máximo de 30 MHz quando são utilizados 10 vetores de suporte. Entretanto, são necessários pelo menos 8 clocks por classificação de cada elemento ao se utilizar apenas 8 bits na parte fracionária. Isso ocorre devido ao uso do kernel HFK. Como o hardware proposto no presente trabalho necessita apenas de 1 clock para a computação da inferência, temos um *speedup* de $\sim$ 4x.

Quando comparado com a arquitetura proposta em (PAN et al., 2013) foi obtido um *speedup* de $\sim 1.2 - 1.8$x (dependendo da precisão numérica). Nesse trabalho, a síntese da arquitetura proposta não foi feita, entretanto a arquitetura foi simulada a 100 MHz para classificações de elementos de dimensão 10. Mesmo com esse alto valor teórico de clock, o *throughput* é reduzido significativamente devido ao número de clocks (8-12) necessários para a computação da inferência.

De forma similar, em (JALLAD; MOHAMMED, 2014), a arquitetura proposta não foi sintetizada, mas foi simulada a 100 MHz. Quando comparado a esse trabalho, a implementação aqui proposta obteve *speedup* de $\sim$ 13.5x, já que são necessários 90 ciclos de clock para a classificação de cada pixel de imagem baseado em três parâmetros de cada pixel. Assim, as imagens de satélite apresentadas por esse trabalho, que possuem 4000X4000 pixels e precisavam de 14.4s para ter todos os seus pixels classificados como nuvem ou terreno, poderiam ser classificadas pelo hardware aqui proposto em apenas 1.06s.

Assim, quando comparada com arquiteturas similares na literatura, vemos que a arquitetura proposta nesse trabalho é capaz de calcular a fase de inferência mais rapidamente do que as demais ao custo de um maior uso de área.

6 Implementação da SMO

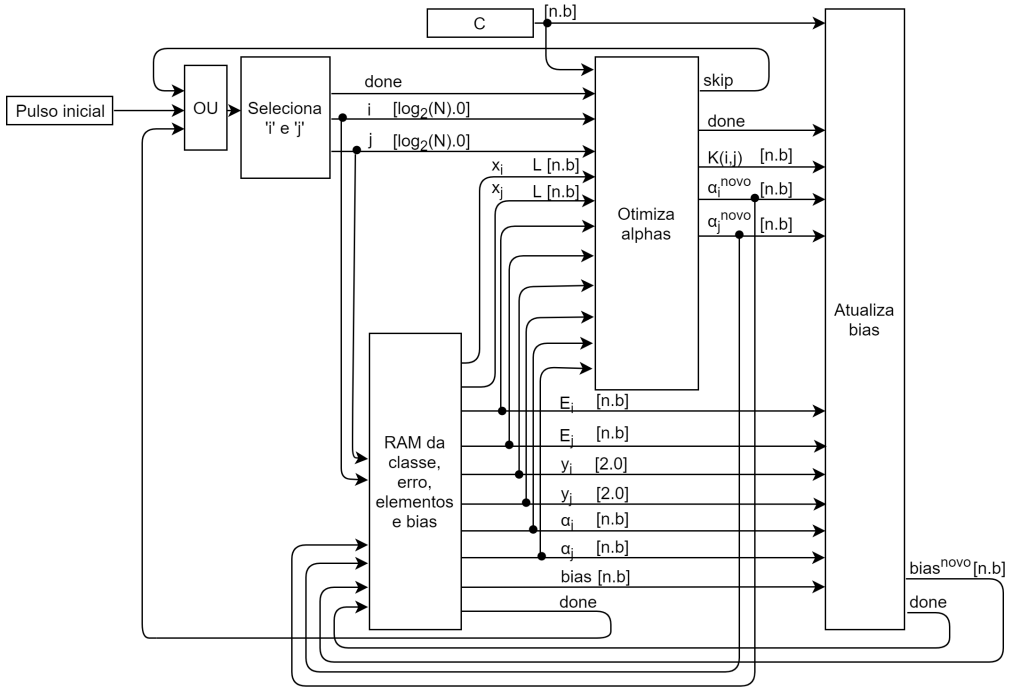
Neste capítulo será apresentada a proposta de implementação em hardware do treinamento da SVM utilizando a otimização sequencial mínima. A estrutura geral será apresentada e em seguida o kernel utilizado para a implementação será introduzido. As camadas da implementação, feitas de acordo com o que foi apresentado no Capítulo 3, serão discutidas em detalhes. Em seguida, serão apresentados resutados de treinamento obtidos utilizando a arquitetura proposta. Por fim, serão apresentados os resultados da síntese, onde serão descritos os recursos necessários para a implementação dessa arquitetura.

6.1 Descrição do Projeto

6.1.1 Estrutura geral da implementação

A estrutura geral da arquitetura proposta para a otimização sequencial mínima (SMO) está presente na Figura 16. Tal estrutura foi desenvolvida de acordo com o algoritmo da SMO proposto no Capítulo 3 e detalhado no pseudocódigo na Seção 3.6 e é formada por quatro camadas principais que serão explicadas em detalhes nas próximas seções. A estrutura da Figura 16 corresponde a parte inferior da Figura 3.

Figura 16 – Estrutura geal da SMO



Novamente, na Figura 16, a notação [n.b] indica que são utilizandos n bits, dos

quais b bits são utilizados para a representação da parte fracionária e n-b bits utilizados para a representação da parte inteira. Os sinais que não apresentam indicação são sinais binários, ou seja, [1.0]. Além disso, como existem diversos modos possíveis de fazer a seleção do par de αs a ser otimizado (ver Seção 3.2), a seleção desses elementos será feita simplesmente de modo aleatório. A lógica de funcionamento das demais camadas que compõem a estrutura em questão será explicada nas subseções 6.1.2, 6.1.3, 6.1.4 e 6.1.5.

6.1.2 Kernel Amigável ao Hardware (HFK)

Como discutido na subseção 5.1.4 um kernel nada mais é do que uma função de similaridade entre dois ou mais elementos. Os kernels polinomial e gaussiano, apesar de serem os mais comuns, não foram desenvolvidos para tomar vantagem das características presentes em arquiteturas reconfiguráveis.

Objetivando resolver esse problema o kernel amigável ao hardware (ou HFK) foi proposto em (ANGUITA et al., 2006). Esse kernel gera uma função de semelhança próxima a do kernel gaussiano. Entretanto, o HFK possui a grande vantagem de poder ser implementado em hardware apenas com shifts e adições ao invés de multiplicações. Tal característica faz com que a área ocupada pelo HFK seja bem menor do que a área ocupada pelos demais kernels apresentados na Tabela 1.

É importante notar que, como na otimização sequencial mínima os elementos a serem otimizados são chamados de $\vec{x_i}$ e $\vec{x_j}$, a equação do hardware amigável ao kernel pode ser reescrita como

$$K(\vec{x_i}, \vec{x_j}) = 2^{-\gamma||\vec{x_i}-\vec{x_j}||_1}, \tag{6.1}$$

onde $||\vec{x_i} - \vec{x_j}||_1$ é a norma L1 da diferença entre os vetores $\vec{x_i}$ e $\vec{x_j}$. Já γ é uma constante que depende da dimensão dim desses vetores e é dada por

$$\gamma = 2^{-log_2(dim)}. \tag{6.2}$$

Por exemplo, se $\vec{x_i} \in \Re^2$ então $\gamma = 2^{-1}$.

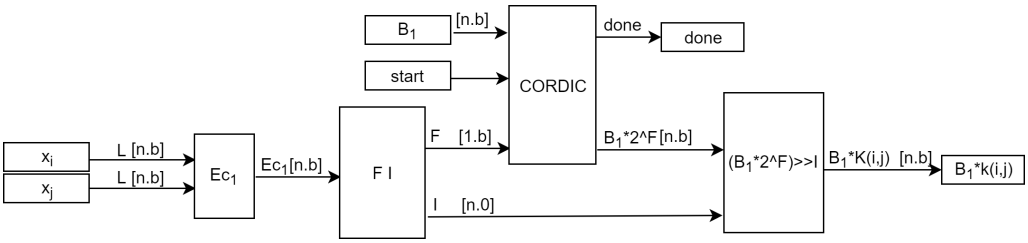
A Figura 17 mostra uma visão geral da implementação do kernel amigável ao hardware. Inicialmente é calculado o erro inicial do bloco CORDIC Ec_1, que é dado por

$$Ec_1 = -\gamma||\vec{x_i} - \vec{x_j}||_1. \tag{6.3}$$

Em seguida, E_1 é dividido entre sua parte inteira I e sua parte fracionária F no bloco FI. O bloco CORDIC então recebe a parte fracionária F e efetua a operação $B_1 2^F$, onde B_1 é um valor a ser multiplicado pelo kernel e pode ser muito útil como mostrado em 6.1.5. Finalmente, o valor de $B_1 2^F$ é deslocado para a direita I vezes, resultando no valor final do kernel, visto que $B_1 K(\vec{x_i}, \vec{x_j}) = B_1 2^{Ec_1} = B_1 2^{F+I} = B_1 2^F 2^I$.

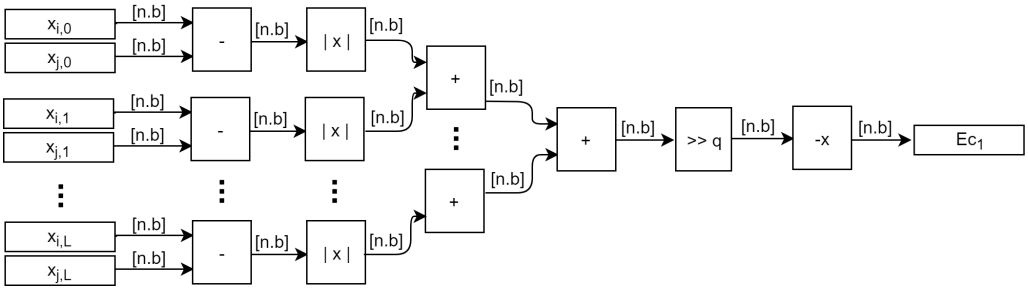
O cálculo de E_1, por sua vez, é feito como mostra a figura 18. Primeiro, é feita a subtração de cada uma das dimensões de $\vec{x_i}$ pela respectiva dimensão em $\vec{x_j}$. Em seguida,

Figura 17 – Visão geral do HFK



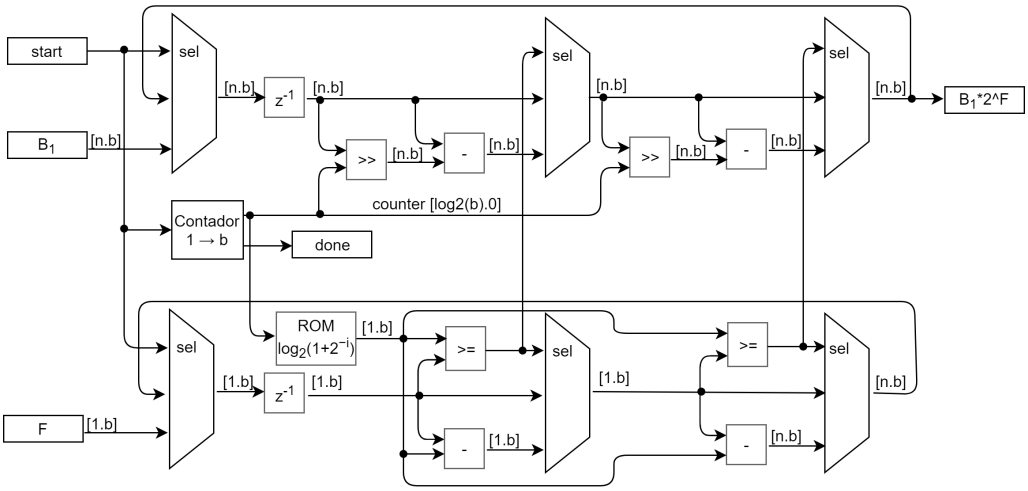
é calculado o módulo de cada um desses elementos e é feita uma soma em árvore. Por fim, o valor é deslocado para a direita q vezes, onde $q = log_2(\gamma)$ e então o seu sinal é invertido. Desse modo, o cálculo de Ec_1 é efetuado em apenas um ciclo de modo paralelo e sem o uso de multiplicações. Entretanto, é importante notar que caso q não seja um valor inteiro, é necessário substituir esse bloco por uma multiplicação por γ.

Figura 18 – Bloco Ec_1 do HFK



O bloco CORDIC (COordinate Rotation DIgital Computer) responsável pelo cálculo de $B_1 \times 2^F$ está ilustrado na Figura 19. Algoritmos CORDIC fazem parte de uma classe de algoritmos que substituem operações complexas, como multiplicações, por deslocamentos e adições.

Figura 19 – Bloco CORDIC do HFK



O bloco CORDIC é responsável pelos cálculo iterativo das equações

$$B_{i+1} = B_i(1 + d_i 2^{-i}) \tag{6.4}$$

e

$$Ec_{i+1} = Ec_i - log_2(1 + d_i 2^{-i}), \tag{6.5}$$

onde d_i pode assumir os valores de 0, +1 e −1 e é escolhido tal que $|Ec_{i+1}| \leq |Ec_i|$, fazendo com que o valor de E_i diminua ao longo das iterações. Quando $Ec_i \to 0$ então $B_i \to B_1 2^{Ec_1}$, o que acontece após b iterações como provado em (ANGUITA et al., 2006).

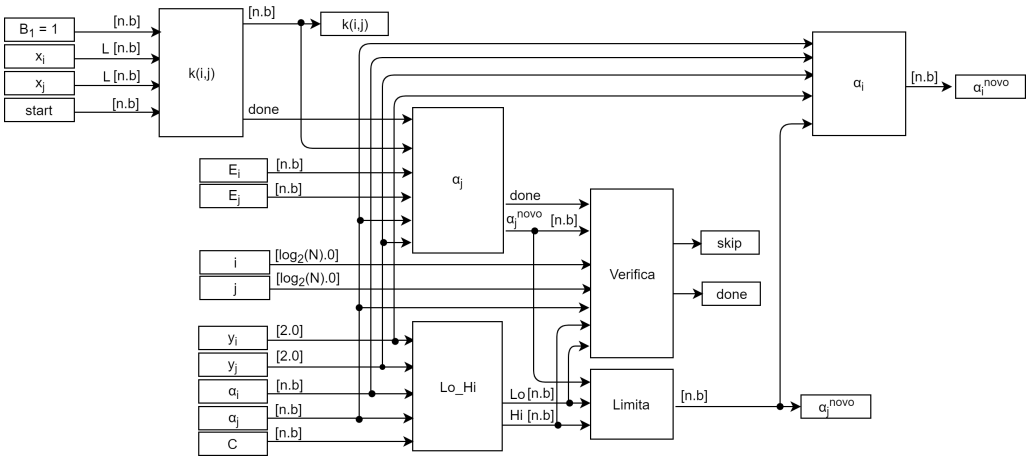
A Equação 6.4 é calculada utilizando a soma de B_i com o resultado de um deslocador Barrel (Barrel-Shifter), responsável por deslocar o valor de B_i para a direita i vezes. Já o valor de Ec_i é calculado através da subtração de Ec_i pelos valores de $log_2(1 + 2^{-i})$ salvos em uma ROM dentro de cada bloco CORDIC.

É importante notar que o valor de Ec_1 é sempre negativo, como mostra a equação 6.3. Assim, objetivando simplificar o hardware proposto, a parte inteira I será sempre positiva e a parte fracionária F sempre negativa. Isso possibilita que d_i seja sempre 0 ou +1, evitando o armazenamento de valores de $log_2(1 - 2^{-i})$ na ROM.

6.1.3 Otimização de α_i e α_j

O cálculo de α_i e α_j é feito como discutido na Seção 3.3 e detalhando no Algoritmo 3. Uma visão geral da implementação da otimização dos valores de α está ilustrada na Figura 20.

Figura 20 – Visão geral da otimização de α_i e α_j



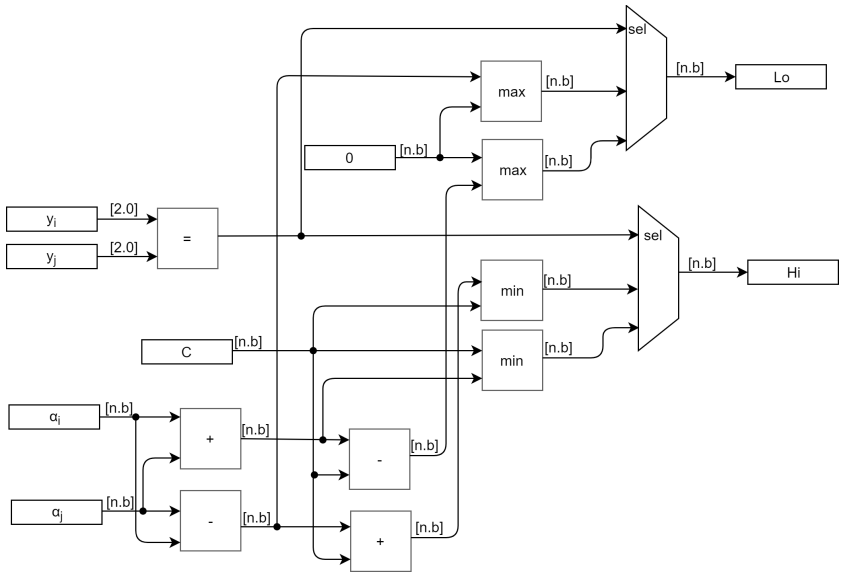
É importante notar na Figura 20 que apenas o cálculo de um kernel é necessário. Isso ocorre pois quando $\vec{x_i}$ é igual a $\vec{x_j}$ o resultado do kernel HFK é 1. Assim, a Equação 3.7 pode ser reescrita como

$$\eta = 2K(\vec{x_i}, \vec{x_j}) - K(\vec{x_i}, \vec{x_i}) - K(\vec{x_j}, \vec{x_j}) = 2K(\vec{x_i}, \vec{x_j}) - 2. \tag{6.6}$$

O bloco a_j é composto de duas simples subtrações e uma divisão. Esse bloco recebe como entrada o valor antigo de α_j y_j e os erros E_i e E_j. Em seguida, o novo valor de a_j é calculado utilizando a Equação 3.6 com o η calculado de acordo com a Equação 6.6.

O cálculo do limitante inferior Lo e do limitante superior Hi são feitos no bloco Lo_Hi como mostra a Figura 21. O cálculo desses limitantes é feito como descrito nas Equações 3.4 e 3.5.

Figura 21 – Cálculo de Lo e Hi



Após o cálculo de Lo e Hi o valor de α_j é limitado utilizando esses valores no bloco Limita como mostra a Equação 3.8. Simultaneamente, no bloco Verifica, é verificado se é vantajoso prosseguir com a otimização. Para isso são verificadas três condições: se i é igual a j, se Lo é igual a Hi e se α_j não mudou significativamente. Caso alguma dessas condições seja verdadeira será emitido um pulso na saída "skip"ao invés da saída "done", fazendo com que a iteração seja descartada e novos valores de i e j sejam escolhidos para a otimização.

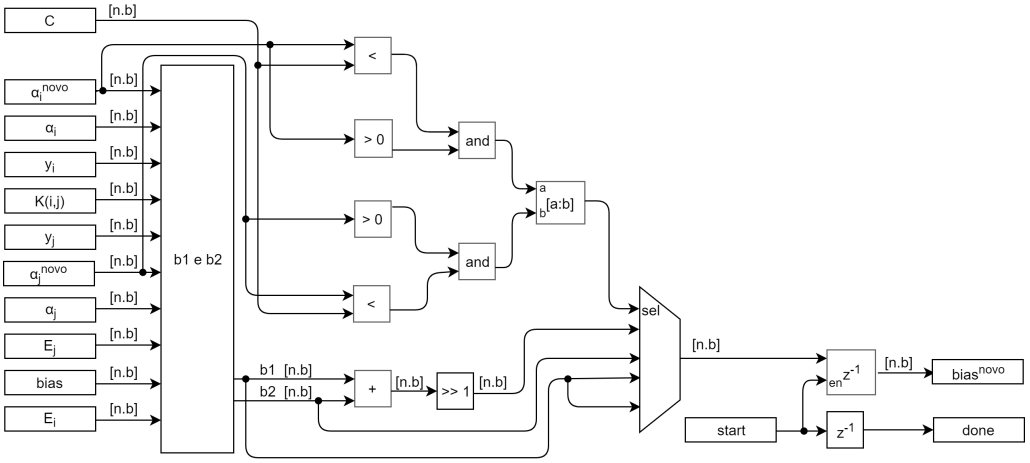
Além disso, como visto na Figura 20, o valor de α_i também é calculado após o cálculo de α_j. Esse valor é calculado como mostra a Equação 3.9.

6.1.4 Otimização da bias

Após a otimização dos valores de α_i e α_j é necessário fazer a atualização do valor da bias. Uma visão geral desse cálculo está ilustrado na Figura 22. Essa figura mostra a seleção entre os valores de $b1$ e $b2$ como descrito na Equação 3.13.

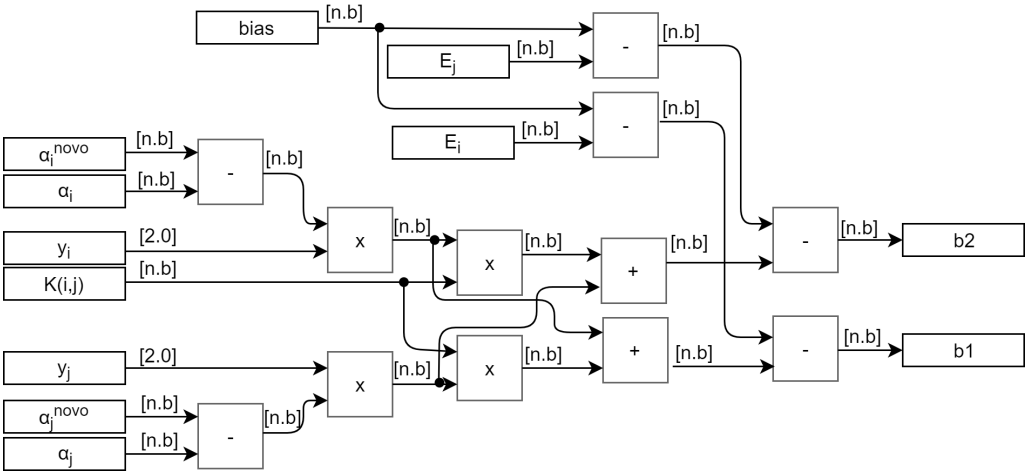
O cálculo dos valores de $b1$ e $b2$ está ilustrado na Figura 23. Nesse bloco, o cálculo de $b1$ é feito como descrito na Equação 3.11 e o cálculo de $b2$ é feito como descrito na Equação 3.12. É importante notar que grande parte das operações necessárias para o

Figura 22 – Cálculo da bias



cálculo de $b1$ e $b2$ são idênticos em ambas equações, o que possibilita o reaproveitamento desses sinais em hardware.

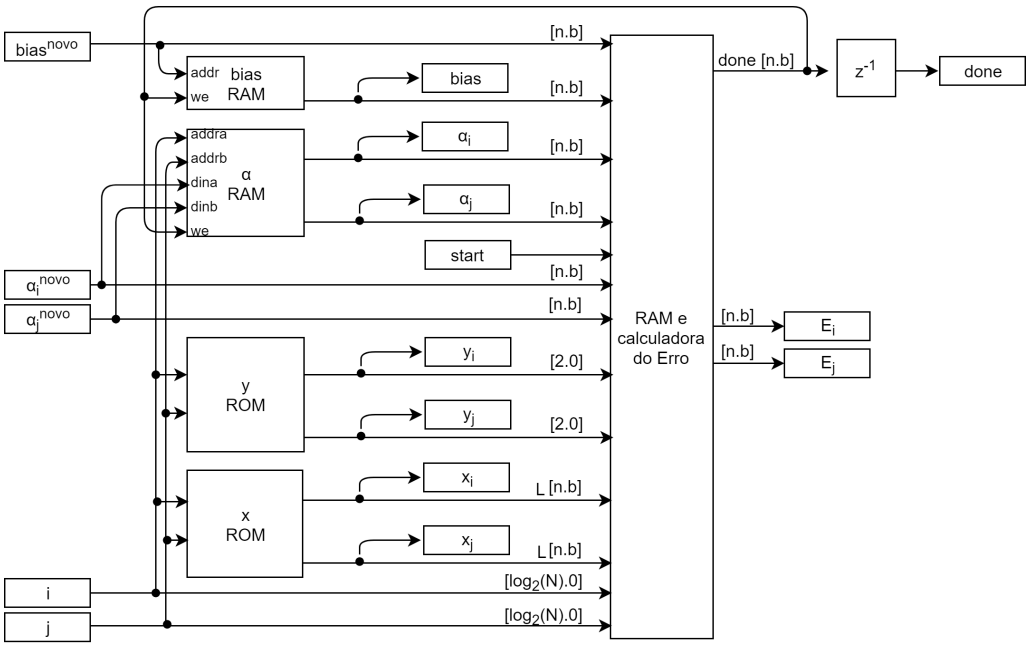
Figura 23 – Cálculo de b1 e b2



6.1.5 Armazenamento dos dados e cálculo do erro

Após otimizar os valores de α_i e α_j e ajustar o valor da *bias* é necessário armazenar esses dados para a próxima iteração. Entretanto, antes disso, um novo valor de erro é calculado para cada um dos elementos do conjunto de treinamento. Na implementação proposta, o cálculo de todos os erros é feito de modo paralelo, já que o kernel HFK utilizado não necessita de blocos de multiplicação e ocupa uma área muito pequena quando comparado ao kernel polinomial e ao gaussiano. Como comentado na Seção 3.5, a cache do erro foi criada com o objetivo de evitar que o custoso cálculo da fase feed-forward seja necessário durante a otimização de α_i e α_j. A arquitetura geral do bloco de armazenamento está ilustrada na Figura 24.

Figura 24 – Visão geral do cálculo do armazenamento dos dados e cálculo do erro



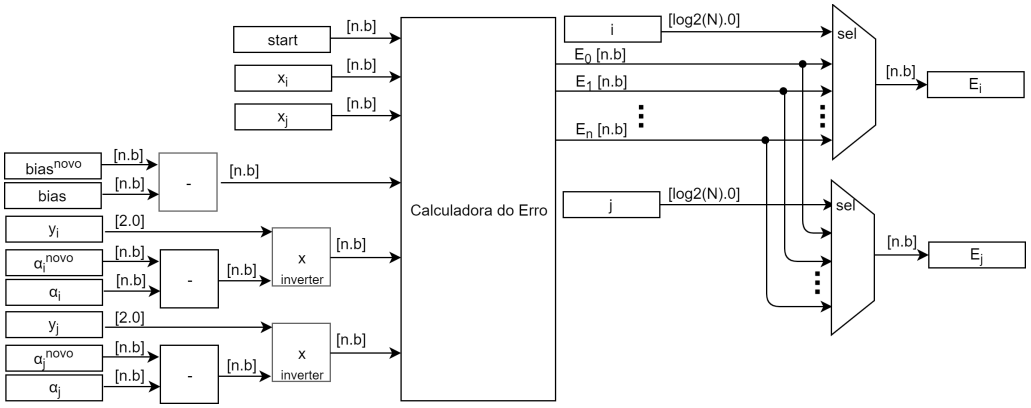
Como mostrado na Figura 24 tanto os valores do elementos do conjunto de entrada quanto a classe as quais esses elementos pertencem são armazenadas em ROMs, já que esses valores não são sobrescritos durante o treinamento. Em um cenário onde o mesmo hardware será utilizado para o treinamento de diversos dados, faz-se necessário a substituição dessas ROMs por RAMs. Já os valores dos multiplicadores de Lagrange e a *bias* são armazenados em RAMs. É importante notar também que o valor desses blocos apenas é atualizado após a conclusão do cálculo dos erros.

Inicialmente, no bloco que faz o cálculo dos erros, são computados os valores que são comuns ao cálculo de todos os erros da Equação 3.14 como mostra a Figura 25. Esses valores são a subtração do novo valor da *bias* pelo antigo valor da *bias*, a multiplicação da classe de i pela subtração do novo valor de α_i pelo valor antigo de α_i e a multiplicação da classe de j pela subtração do novo valor de α_j pelo valor antigo de α_j. É importante notar que como as classes apenas podem assumir o valor de $+1$ ou -1 as multiplicações em questão podem ser substituídas por simples inversões condicionais.

O cálculo do erro é feito por N blocos ilustrados na Figura 26, onde N é o número de elementos do conjunto de treinamento. Em seguida, as saídas do i-ésimo e j-ésimo erros (E_i e E_j) são selecionadas utilizando um simples multiplexador de N entradas.

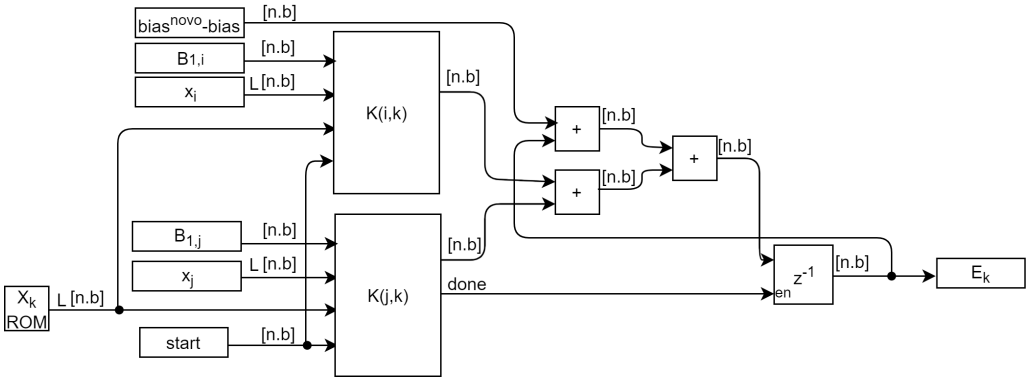
O cálculo individual do erro ilustrado na Figura 26 é constituído principalmente da computação de duas funções de kernel: $k(i,k)$ e $k(j,k)$, onde k é o índice do erro a ser calculado que varia entre 1 e N. Como cada bloco sempre é responsável pelo cálculo do mesmo k-ésimo erro (E_k), o valor do elemento de k é salvo em um registrador dentro do próprio bloco de cálculo de erro. Por fim, após o cálculo do erro segundo a Equação 3.14,

Figura 25 – Cálculo do erro



o valor de E_k é armazenado em um registrador.

Figura 26 – Cálculo individual do erro



6.2 Resultados

6.2.1 Metodologia

Essa seção tem como objetivo apresentar os testes e resultados da arquitetura descrita na Seção 6.1. Para testar o hardware implementado, é feito o treinamento da porta OU exclusivo (XOR) e do conjunto de dados 'fisheriris'. Nesse último conjunto, apresentado em (FISHER, 1950), é feita a distinção entre os tipos 'setosa' e 'versicolor' de plantas do gênero Íris de acordo com a altura e largura tanto da pétala quanto da sépala.

Assim como feito no Capítulo 6.1, a porta XOR foi escolhida como teste para a classificação por ser um exemplo simples, frequente na literatura (GU; HAN, 2013) e não ser linearmente separável. Já o conjunto de dados 'fisheriris' foi escolhido para testar a classificação de um conjunto de dados tridimensional, servindo assim como teste para averiguar que a estrutura implementada é capaz de atuar satisfatoriamente em casos mais desafiadores.

Os valores finais da *bias* assim como o valor dos multiplicadores de Lagrange para cada um dos elementos do conjunto de entrada são armazenados e processados, permitindo a geração de gráficos e dados numéricos que comprovam a eficácia da arquitetura proposta no treinamento da classificação.

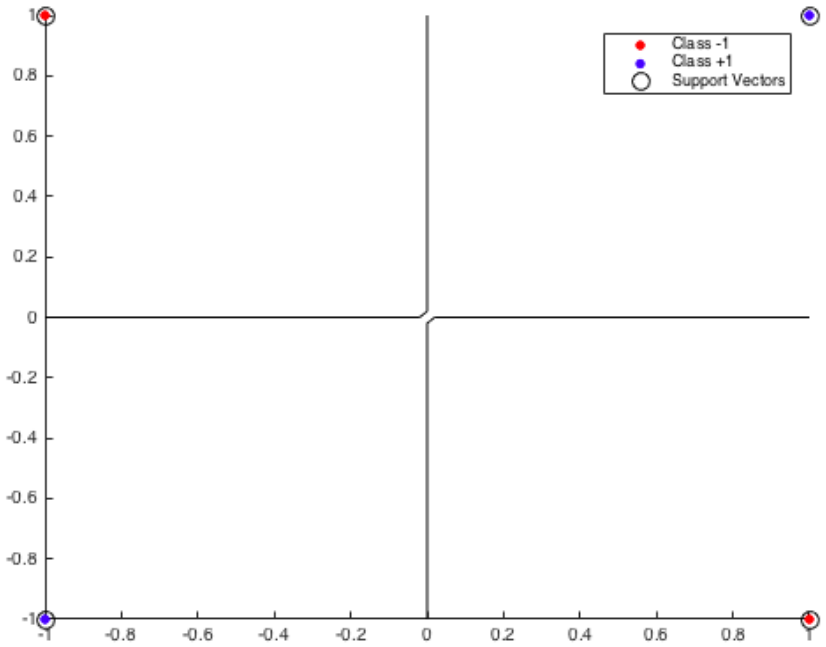
Assim como no Capítulo 6.1, também é feita uma análise de uso da FPGA alvo Virtex 6 xc6vlx240t-1ff1156 através do relatório gerado durante a síntese da arquitetura proposta.

6.2.2 Resultados do Treinamento

Após o treinamento da função XOR utilizando o conjunto de treinamento presente na Tabela 4 e utilizando a estrutura em hardware proposta foram obtidos a *bias* e os vetores de suporte SV, assim como seus respectivos multiplicadores de Lagrange α. Utilizando esses valores e o cálculo da fase *feed-forward* da classificação presente na Equação 2.2 foi possível encontrar a curva que delimita as duas classes.

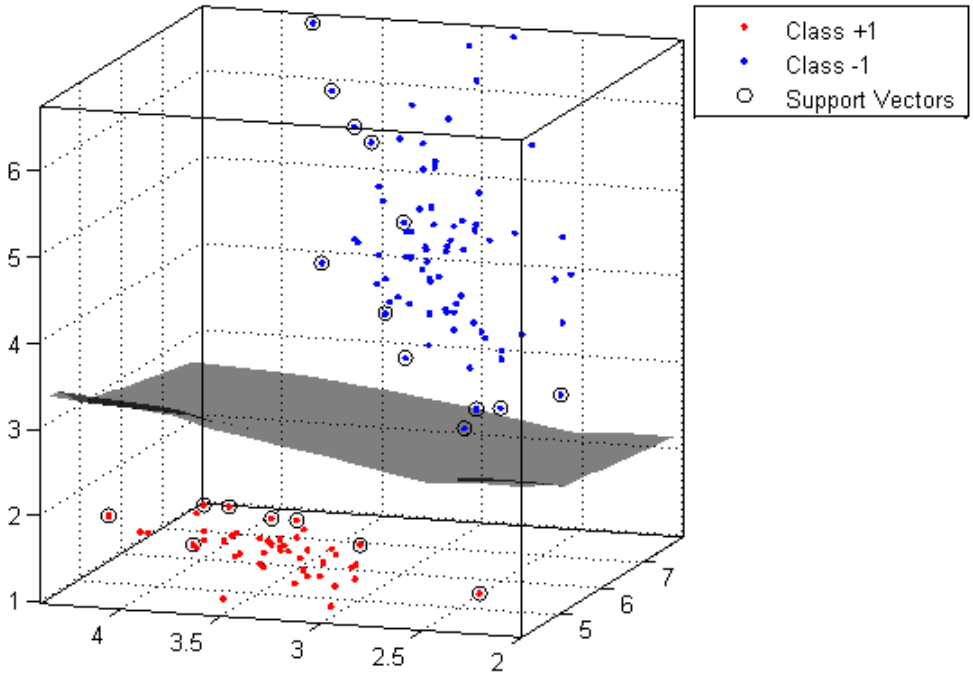
Ao observar a Figura 27, vê-se que os quatro conjuntos de pontos do treinamento tornaram-se vetores de suporte. Além disso, nota-se também que a curva de classificação gerada pelo algoritmo implementado separa as classes de modo ideal, do mesmo modo como visto na Figura 13.

Figura 27 – Resultado do teste com porta XOR



A Figura 28 por sua vez, mostra o resultado do segundo teste proposto, em que a classe 'setosa' é representada pela classe +1 e as classes 'versicolor' e 'virginica' são representadas pela classe -1. Para o teste proposto, foram utilizados os primeiros 128

Figura 28 – Resultado do com conjunto de dados 'fisheriris'



elementos desse conjunto, assim como as três primeiras dimensões de cada um desses elementos.

Os resultados do segundo teste proposto, como visto na Figura 28, mostram que os pontos extremos e os mais próximos à classe oposta tornaram-se vetores de suporte. Visualmente percebe-se que os vetores de suporte geraram uma curva de classificação capaz de dividir as duas classes de modo a maximizar as bordas entre eles, cumprindo assim o objetivo proposto pela SVM.

Em ambos os testes propostos as condições de KKT foram satisfeitas, possibilitando a convergência do algoritmo. Assim, foi possível perceber que o hardware apresentado neste trabalho é funcional e portanto uma alternativa válida para o treinamento da SVM utilizando SMO.

6.2.3 Resultados de Síntese

A Tabela 6 mostra o resultado da síntese em uma FPGA Virtex 6 xc6vlx240t-1ff1156 da arquitetura proposta na Seção 6.1.

Tabela 6 – Resultados da síntese da SMO

Bits	Elementos	Multiplicadores	LUTs	Clock (MHz)
16.10	8	16 (2.0%)	7397 (4.9%)	34.620
32.24	8	31 (4.0%)	15320 (10.1%)	24.684
16.10	16	16 (2.0%)	13534 (8.9%)	33.980
32.16	16	31 (4.0%)	28158 (18.6%)	24.601
16.10	32	16 (2.0%)	25784 (17.1%)	34.426
32.24	32	31 (4.0%)	53887 (35.7%)	24.385
16.10	64	16 (2.0%)	50859 (33.7%)	33.383
32.16	64	31 (4.0%)	106008 (70.3%)	24.657

Como é possível observar na Tabela 6, a síntese do hardware proposto foi feita utilizando tanto 32 bits (com 24 para a parte fracionária) quanto 16 bits (com 10 para a parte fracionária). Além disso, foram feitos testes com o hardware proposto suportando até 8, 16, 32 ou 64 elementos no conjunto de treinamento.

É importante notar que, para cada ciclo de treinamento, são necessários no máximo

$$2b + w \tag{6.7}$$

clocks, onde w é o número de clocks necessário para o bloco de divisão da otimização de α_i e α_j somado ao overhead dos sinais de controle entre os blocos. O valor de w varia com a precisão numérica da implementação e normalmente tem valor entre 6 e 10. Já $2b$ é o número de clocks necessário para o cálculo do kernel durante a otimização de α_i e α_j e durante o cálculo do erro. Entretanto, muitas das vezes são necessários menos clocks do que $2b + w$, visto que frequentemente o hardware proposto é capaz de saltar iterações que não contribuem significativamente para a convergência do treinamento, como discutido na Seção 6.1.

Diferente do que ocorreu com a implementação da fase feed-forward utilizando o kernel polinomial, o número de multiplicadores não cresceu com o aumento do número de elementos. Esse comportamento é esperado, visto que o kernel HFK utilizado foi implementado sem utilizar nenhuma multiplicação. Essa característica mostra uma das grandes vantagens de se utilizar o HFK ao invés do kernel polinomial.

Por fim, observa-se que o hardware proposto é extremamente paralelizável, já que não ocorre perda de desempenho com o aumento do número máximo de elementos do conjunto de treinamento. Esse comportamento é esperado, já que o gargalo da implementação está no cálculo do erro, que é feito totalmente em paralelo.

Objetivando ilustrar a aplicabilidade da arquitetura em hardware do treinamento da SVM apresentada neste trabalho foram estimados o tempo de convergência do algoritmo para três exemplos da literatura. O primeiro exemplo chamado de 'Iris' é a versão completa do conjunto de dados 'fisheriris' do Matlab (utilizado como exemplo na Subseção 6.2.2).

Tabela 7 – Tempo de Convergência e Frequência de Amostragem de Aplicações da literatura Utilizando a Arquitetura em Hardware do Treinamento da SVM com 16.10 bits.

Aplicação	Elementos	Dimensão	Clock	Iterações	Convergência
Iris	100	4	$\sim$ 33 MHz	$\sim$500	$\sim$0.45 ms
Breast Cancer	569	10	$\sim$ 27 MHz	$\sim$5200	$\sim$5.2 ms
Banknote Auth.	1372	4	$\sim$ 30 MHz	$\sim$17000	$\sim$17.0 ms

Os exemplos aqui expostos também estão disponíveis no UCI Machine Learning Repository.

Como visto na Seção 6.2, esse conjunto de dados foi apresentado em (FISHER, 1950) e nele é feita a distinção entre os tipos 'setosa' e 'versicolor' de plantas do gênero Íris de acordo com a altura e largura tanto da pétala quanto da sépala. O segundo conjunto de dados, apresentado em (STREET et al., 1993) e intitulado 'Breast Cancer Wisconsin', tem como objetivo a classificação binária de tumores como maligno ou benigno utilizando 10 parâmetros obtidos de 569 tumores distintos. Esses parâmetros incluem raio, textura, perímetro, área, entre outros. Por fim, o terceiro conjunto de dados, chamado de 'Banknote Authentication', é utilizado para classificar 1372 cédulas bancárias entre real ou falsa como visto em (LOHWEG et al., 2013). Para isso, são utilizados 4 atributos após obtidos após a transformada wavelet da imagem original: a variância da imagem transformada, a obliquidade da imagem transformada, a curtose da imagem transformada e a entropia da imagem original.

O tempo estimado de convergência para os conjuntos de dados citados podem ser vistos na Tabela 7. A primeira coluna contém a referência do conjunto de dados utilizado. Na segunda e terceira coluna são apresentadas as dimensões do problema (número de elementos e dimensionalidade de cada entrada). Na coluna seguinte é apresentado o clock obtido através da extrapolação da Tabela 6, que o problema poderia alcançar se o treinamento da SVM fosse implementado na arquitetura proposta em um hardware suficientemente grande. Na quinta coluna são apresentados a quantidade aproximada de iterações necessárias para que os problemas, de acordo com suas dimensões, convirjam. Na última coluna, a partir do número de iterações e da máxima frequência de amostragem, é feita uma estimativa de tempo de convergência dos problemas citados. Os dados apresentados na Tabela 7 foram gerados de acordo com uma arquitetura com 16 bits, sendo 10 para a parte fracionária. Desse modo, são necessários 30 clocks por iteração da fase de treinamento.

Em um problema como o do conjunto de dados 'Iris' são necessárias aproximadamente 500 iterações para que ocorra a convergência do algoritmo. Assim, dada a frequência máxima de 33 MHz, teríamos o resultado final após apenas 0.45 ms. Já no caso do conjunto de dados 'Breast Cancer', com um clock de aproximadamente 27 MHz, são necessárias aproximadamente 5200 iterações, o que faz com que a convergência ocorra em aproximadamente 5.2 ms. Por fim, no treinamento do conjunto de dados 'Banknote Authentication',

a frequência máxima é de aproximadamente 30 MHz, entretanto são necessárias aproximadamente 17000 iterações devido ao grande número de elementos para o treinamento. Desse modo, o resultado do treinamento estaria disponível após aproximadamente 17 ms.

Ao extrapolar os dados da Tabela 6 é também possível comparar a arquitetura proposta com outras propostas de implementação presentes na literatura. No hardware apresentado em (FILHO et al., 2010), por exemplo, são necessários aproximadamente 100.000 ciclos de clock para cada iteração da classificação do conjunto de dados 'Breast Cancer' utilizando componentes de 20bits. Isso ocorre, pois a implementação em questão serializa partes paralelizáveis do código para poupar área. Assim, mesmo que essa implementação seja executada a 100 MHz, a implementação proposta no atual trabalho possui *speedup* de $\sim$ 600x considerando precisão numérica de 32.24 bits e um clock máximo de 24 MHz como visto na Tabela 6.

Quando comparada a arquitetura apresentada em (CAO; SHEN; CHEN, 2010), a arquitetura proposta apresentaria um *speedup* de $\sim$ 96x no conjunto de dados 'Breast Cancer', assumindo um clock máximo de 24MHz em uma implementação com precisão numérica de 32.24 bits. Segundo as equações apresentadas no trabalho, o número aproximado de clocks por ciclo para esse conjuto de dados seria de aproximadamente 12.000 iterações e a frequência máxima da FPGA seria de 75 MHz. A aceleração obtida em comparação a esse trabalho se deve ao fato do número de clocks independer da dimensão da entrada na implementação aqui proposta, já que o cálculo do kernel que envolve as dimensões (Bloco E_{c1} do HFK) é feito de modo totalmente paralelo.

Por fim, quando comparado com (BUSTIO-MARTÍNEZ et al., 2010), que utilizou a FPGA apenas como acelerador das funções de kernel, o *speedup* do presente trabalho foi de aproximadamente $\sim$ 67x. Nesse trabalho, o clock máximo atingido foi de 35 MHz ao classificar um conjunto de dados com dimensionalidade 14 e computando o kernel linear em apenas 3 clocks. Entretanto, após a implementação em hardware do kernel, o gargalo da implementação passou a ser o restante do algoritmo da SMO executado em software. Nesse algoritmo, cada iteração (software + hardware) levou em média 0,115ms, enquanto na implementação do presente trabalho levaria aproximadamente 0,0017 ms considerando-se um clock de 24 MHz em uma implementação com 32.24 bits.

Assim, vê-se que a implementação proposta é capaz de computar o treinamento da SVM de forma muito mais rápida que outras implementações na literatura em troca de maior uso de área.

7 Conclusões

Os resultados da classificação da porta XOR na Seção 5.2 mostraram que a arquitetura da SVM proposta para a fase *feed-foward* funciona de modo extremamente satisfatório na classificação de problemas não linearmente separáveis, mesmo quando um pequeno número de bits é utilizado para os cálculos.

Já os resultados da classificação da função $sin(x) + cos(y)$, também discutidos na Seção 5.2, mostram que a SVM implementada é capaz de lidar com problemas mais complexos. Durante a regressão da função $sin(x)+cons(y)$, foi possível notar a importância da arquitetura proposta possuir kernel polinomial de grau selecionável e observou-se também que o grau máximo 4 atendeu bem ao problema proposto. Ademais, estimar a função em questão nos fez verificar que o número de vetores de suporte implementados em nossa SVM é suficiente para atender a problemas de maior complexidade.

Além disso, ao analisar os resultados da síntese, constatou-se que a arquitetura proposta obteve resultados compatíveis com o esperado em relação ao uso da FPGA. Apesar de necessitar de uma grande quantidade de multiplicadores, muitas das FPGAs modernas dispõem de um grande número de multiplicadores internos, o que possibilita a implementação de arquiteturas como a nossa. Observou-se que a FPGA alvo Virtex 6 xc6vlx240t-1ff1156 é capaz de atender satisfatoriamente o hardware desenvolvido, mesmo quando o número de vetores de suporte é elevado.

Finalmente, ao verificar durante a síntese a taxa máxima de amostragem do sistema proposto, foi possível constatar que a arquitetura desenvolvida atinge taxas de amostragem de até 18,282 MHz na FPGA alvo. Notou-se também que o design proposto é extremamente paralelo, já que não houve diminuição perceptível da frequência máxima de amostragem do sistema com o aumento do número de vetores de suporte. Quando comparado com outras implementações na literatura, a arquitetura proposta obteve um *speedup* de até 13.5x.

No Capítulo 6, por sua vez, a implementação da fase de treinamento através da Otimização Sequencial Mínima (SMO), introduzida no Capítulo 3 e detalhada no pseudocódigo na Seção 3.6, foi implementada em hardware. Essa implementação foi testada e validada para diversos casos de classificação.

Os resultados do treinamento da porta XOR na Seção 6.2 mostram que o hardware proposto foi capaz de treinar corretamente os pontos em questão, satisfazendo todas as condições necessárias para a convergência. Além disso, os resultados do treinamento do conjunto de dados 'fisheriris' demonstraram o bom funcionamento do hardware proposto mesmo com um maior número de elementos e com a dimensão maior que 2.

Através da síntese do treinamento utilizando a SMO foi possível notar que o uso do kernel HFK evita o uso demasiado de multiplicadores. Assim, extrapolando os resultados obtidos na Tabela 6, vemos que é possível fazer a implementação do treino de até 128 elementos bidimensionais com precisão de 16 bits na FPGA alvo ou até 10x mais utilizando FPGAs com mais recursos, como uma Virtex7 7V2000T. É importante notar que o cálculo do erro de todos os elementos do conjunto de treinamento é feito em paralelo, o que torna essa implementação extremamente eficiente quando comparada a uma implementações em arquiteturas seriais.

Ao analisar a taxa máxima de amostragem da fase de treinamento também foi possível constatar o alto grau de paralelismo dessa implementação. Como esperado, a taxa máxima de amostragem diminui significativamente com o aumento da precisão numérica utilizada. Entretanto, essa mesma taxa não é alterada com o aumento do número máximo de elementos do conjunto de treinamento. Ao ser comparada com outras arquiteturas propostas na literatura, a arquitetura proposta obteve um *speedup* de até $\sim$ 600x ao custo de um maior uso de área.

Referências

ANGUITA, D. et al. Feed- forward support vector machine without multipliers. *IEEE Transactions on Neural Networks*, v. 17, p. 1328–1331, 2006. Citado 2 vezes nas páginas 50 e 52.

BAPTISTA, D. et al. Artificial neural networks - a survey about hardware and software use. *10th Portuguese Conference on Automatic Control*, 2012. Citado na página 32.

BOSER, B. E.; GUYON, I. M.; VAPNIK, V. A training algorithm for optimal margin classifers. *Proceedings of the fifth annual workshop on Computational learning theory*, 1992. Citado na página 22.

BUSTIO-MARTÍNEZ, L. et al. Advances in pattern recognition. In: _____. [S.l.]: Springer, 2010. cap. On the Design of a Hardware- Software Architecture for Acceleration of SVM's Training Phase, p. 281–290. Citado 2 vezes nas páginas 18 e 61.

CAO, K.; SHEN, H.; CHEN, H. A parallel and scalable digital architecture for training support vector machines. *Journal of Zhejiang University SCIENCE C*, v. 11, p. 620–628, 2010. Citado 2 vezes nas páginas 18 e 61.

CHANG, Y.-W. et al. Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research 11*, 2010. Citado na página 41.

COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 2002. Citado na página 32.

CORTES, C.; VAPNIK, V. Support-vector networks. *Kluwer Academic Publishers*, 1995. Citado 2 vezes nas páginas 22 e 25.

FILHO, J. G. et al. A general-purpose dynamically reconfigurable svm. *2010 VI Southern Programmable Logic Conference (SPL)*, p. 107–112, 2010. Citado 2 vezes nas páginas 19 e 61.

FISHER, R. The use of multiple measurements in taxonomic problems. *Contributions to Mathematical Statistics*, 1950. Citado 2 vezes nas páginas 56 e 60.

GOLDBERG, Y.; ELHADAD, M. splitsvm: Fast, space-efficient, non-heuristic, polynomial kernel computation for nlp applications. *Proceedings of ACL-08*, 2008. Citado na página 40.

GU, Q.; HAN, J. Clustered support vector machines. *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2013. Citado 2 vezes nas páginas 43 e 56.

HANRAHAN, G. *Artificial Neural Networks in Biological and Environmental Analysis*. [S.l.]: CRC Press, 2007. Citado na página 38.

HAYKIN, S. *Neural Networks and Learning Machines*. [S.l.]: Pearson Prentice Hall, 2008. Citado na página 24.

HUANG, H.; LIU, H. Big data machine learning and graph analytics: Current state and future challenges. *IEEE International Conference on Big Data*, 2014. Citado na página 15.

HUSSAIN, K. B. H.; SEKER, H. Novel dynamic partial reconfiguration implementations of the support vector machine classifier on fpga. *Turkish Journal of Electrical Engineering and Computer Sciences*, 2014. Citado na página 17.

HUSSAIN, K. B. H. M.; SEKER, H. Reconfiguration- based implementation of svm classifier on fpga for classifying microarray data. *35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, p. 3058–3061, 2013. Citado na página 17.

JALLAD, A. H. M.; MOHAMMED, L. B. Hardware support vector machine (svm) for satellite on-board applications. *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, p. 256–261, 2014. Citado 2 vezes nas páginas 17 e 48.

KYRKOU, C.; THEOCHARIDES, T.; BOUGANIS, C. An embedded hardware-efficient architecture for real-time cascade support vector machine classification. *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, p. 129–136, 2013. Citado na página 18.

LOHWEG, V. et al. Banknote authentication with mobile devices. *Media Watermarking, Security, and Forensics*, 2013. Citado na página 60.

MAXFIELD, C. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. [S.l.]: Elsevier, 2012. Citado na página 33.

PAN, X. et al. Fpga implementation of svm decision function based on hardware-friendly kernel. *International Conference on Computational and Information Sciences - ICCIS 2013*, p. 133–136, 2013. Citado 3 vezes nas páginas 17, 18 e 48.

PAPADONIKOLAKIS, M.; BOUGANIS, C.-S. A novel fpgabased svm classifier. *International Conference on Field- Programmable Technology (FPT)*, p. 283–286, 2010. Citado na página 18.

PATIL, R. et al. Power aware hardware prototyping of multiclass svm classifier through reconfiguration. *25th International Conference on VLSI Design (VLSID)*, p. 62–67, 2012. Citado na página 17.

PLATT, J. Fast training of support vector machines using sequential minimal optimization. *Advances in Kernel Methods—Support Vector Learning*, v. 3, 1999. Citado na página 26.

RUIZ-LLATA, M.; GUARNIZO, G.; YÉBENES-CALVINO, M. Fpga implementation of a support vector machine for classification and regression. *IEEE World Congress on Computational Intelligence*, 2010. Citado 4 vezes nas páginas 17, 18, 38 e 48.

SCHOELKOPF, B.; SMOLA, A. J. Learning with kernels. *MIT Press*, 2002. Citado na página 45.

SOUZA, A. C. D.; FERNANDES, M. A. C. Parallel fixed point implementation of a radial basis function network in an fpga. *Sensors*, 2014. Citado na página 32.

STREET, W. et al. Nuclear feature extraction for breast tumor diagnosis. *1993 International Symposium on Electronic Imaging: Science and Technology*, 1993. Citado na página 60.

TA-WEN, K. et al. Vlsi design of an svm learning core on sequential minimal optimization algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 20, p. 673–683, 2012. Citado na página 18.

VENKATESHAN, A. P. S.; VARGHESE, K. Hybrid working set algorithm for svm learning with a kernel coprocessor on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2014. Citado na página 19.

ZEPEDA, J.; PEREZ, P. Exemplar svms as visual feature encoders. *CVPR*, 2015. Citado na página 15.