DISSERTAÇÃO DE MESTRADO Nº 1043

ESTUDO E DESENVOLVIMENTO ARQUITETURAL PARA IMPLEMENTAÇÃO DE UM CLASSIFICADOR GEOMÉTRICO DE MARGEM LARGA EM SISTEMAS EMBARCADOS

Liliane dos Reis Gade

DATA DA DEFESA: 27/02/2018

Universidade Federal de Minas Gerais

Escola de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

ESTUDO E DESENVOLVIMENTO ARQUITETURAL PARA IMPLEMENTAÇÃO DE UM CLASSIFICADOR GEOMÉTRICO DE MARGEM LARGA EM SISTEMAS EMBARCADOS

Liliane dos Reis Gade

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do Título de Mestre em Engenharia Elétrica.

Orientador: Prof. Cristiano Leite de Castro

Belo Horizonte - MG

Fevereiro de 2018

Gade, Liliane dos Reis.
G124e Estudo e desenvolvimento arquitetural para implementação de um classificador geométrico de margem larga em sistemas embarcados [manuscrito] / Liliane dos Reis Gade. – 2018.
x, 79 f., enc.: il.

Orientador: Cristiano Leite de Castro.

Dissertação (mestrado) Universidade Federal de Minas Gerais, Escola de Engenharia.

Bibliografia: f. 76-79.

1. Engenharia elétrica - Teses. 2. Matrizes de portas programáveis no campo - Teses. 3. Microcontroladores ARM - Teses. I. Castro, Cristiano Leite de. II. Universidade Federal de Minas Gerais. Escola de Engenharia. III. Título.

CDU: 621.3(043)

"Estudo e Desenvolvimento Arquitetural para Implementação de um Classificador Geométrico de Margem Larga em Sistemas Embarcados"

Liliane dos Reis Gade

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Mestre em Engenharia Elétrica.

Aprovada em 27 de fevereiro de 2018.

Por:

Prof. Dr. Cristiano Leite de Castro
DEE (UFMG) - Orientador

Prof. Dr. Janier Arias García
DELT (UFMG) - Coorientador

Prof. Dr. Antônio de Pádua Braga
DELT (UFMG)

Prof. Dr. Henrique Resende Martins
DEE (UFMG)

Resumo

O presente trabalho se destina ao estudo de novas formas de implementação da regra de decisão de um classificador de margem larga com o objetivo de reduzir a complexidade das operações durante a etapa de classificação de novos padrões, facilitando, a sua utilização em sistemas embarcados. Para a implementação do referido classificador em hardware, desenvolveu-se uma arquitetura em fluxos de dados do classificador geométrico, aproveitando-se ao máximo o paralelismo intrínseco dos *FPGAs* (*Field Programmable Gate Arrays*). Essa arquitetura foi testada e simulada com bases de dados reais, que revelou um alto desempenho e consumo de recursos obtidos. Verificou-se que o consumo de recursos cresce de maneira exponencial com o aumento do número de amostras de treinamento, o que torna o uso da arquitetura proposta não muito adequado em sistemas com grandes amostras de treinamento. Diante disso, realizou-se a implementação do classificador no microcontrolador *ARM* 32bits, repetindo-se os testes e os comparando aos obtidos pela arquitetura anterior. Os resultados mostraram que o tempo de execução do algoritmo no microcontrolador é maior que no *FPGAs*, visto que o microcontrolador não possui características de paralelismo. No entanto, nele, o consumo de recursos é menor, o que possibilita seu uso em sistemas que possuem um número de amostras de treinamento mais elevado.

Abstract

The present work is aimed at the study of new ways of implementing the decision rule of a large margin classifier aiming at of reducing the complexity of the operations during the classification step of the new patterns, making it easier the use of this classifier in embedded systems. An architecture in data flows of the geometric classifier was also developed, taking full advantage of the intrinsic parallelism of Field Programmable Gate Arrays (*FPGAs*). This architecture was tested and simulated on real data sets commonly used in the literature. The results showed a high performance and consumption of resources obtained by the architecture. The consumption of resources increase exponentially with the number of training samples, thus making it not very adequate in systems with large training samples. Because of that, a 32-bits *ARM* microcontroller implementation was performed and the tests were repeated and compared with the previous architecture. The results showed that the running time of the algorithm in microcontroller is larger than in *FPGAs*, since it does not have the characteristics of parallelism. However the resource consumption is smaller, in systems that have a higher number of training samples.

Agradecimentos

Agradeço primeiramente a **Deus** por ter me sustentado durante toda minha jornada acadêmica e ter me dado o discernimento para tomar as decisões corretas.

Aos meus pais pelos ensinamentos, palavras de afeto e ânimo nos momentos em que mais precisava.

A minhas irmãs e cunhados pelo incentivo e por sempre acreditarem no meu esforço.

Ao meu marido Jânio pelo amor, carinho e ajuda nas revisões dos textos escritos ao longo do mestrado.

A meu orientador Cristiano e Co-orientador Janier pelos ensinamentos, paciência e por acreditar no meu trabalho.

Aos professores e colegas do LITC pelas orientações, conselhos e troca de experiências.

Aos meus amigos e familiares.

Sumário

1 Introdução **1**

1.1 Motivação . . . 2

1.2 Objetivos . . . 4

1.2.1 Objetivo Geral . . . 4

1.2.2 Objetivos Específicos . . . 4

1.3 Principais Contribuições deste Trabalho . . . 4

1.4 Organização do Texto . . . 5

2 Fundamentação Teórica **7**

2.1 Classificação de Padrões . . . 7

2.2 Classificador de margem larga baseado em distância . . . 8

2.2.1 Mistura de hiperplanos . . . 11

2.2.2 CHIP-clas Reduzido . . . 13

2.3 kNN . . . 13

2.4 Sistemas Embarcados . . . 14

2.4.1 Microcontroladores . . . 15

2.4.2 FPGA . . . 18

2.4.3 ASIC . . . 21

2.4.4 SoC FPGAs . . . 22

2.5 Representação numérica . . . 22

2.5.1 Representação em ponto fixo . . . 22

2.5.2 Representação em ponto flutuante . . . 26

3 Nova abordagem para o classificador de margem larga e metodologia para sua implementação em hardware **31**

3.1 NN-clas . . . 31

3.2 Metodologia . . . 33

4 Arquitetura em hardware proposta **40**

4.1 Fase de treinamento . . . 41

4.1.1 Cálculo de distância 41
4.1.2 Cálculo do grafo de proximidade 46
4.1.3 Cálculo do grafo de borda 49
4.1.4 Armazenamento de dados situados na borda 51
4.2 Fase de classificação 53
4.2.1 Cálculo de distância 53
4.2.2 Classificação de dados 56

5 Experimentos e Resultados Computacionais 58
5.1 Testes com o NN-clas e o CHIP-clas reduzido 58
5.2 Testes da arquitetura em hardware proposta 61
5.3 Testes da implementação do NN-clas em microcontrolador ARM 69

6 Conclusões e Trabalhos Futuros 74

Referências Bibliográficas 76

Lista de Figuras

1.1 Histórico das revoluções industriais 2

2.1 Diagrama explicando a criação do grafo de Gabriel 9
2.2 Grafo de Gabriel de duas gaussianas com arestas de suporte indicadas pelos quadrados 9
2.3 Remoção de amostras ruidosas de uma base de dados com sobreposição entre as classes 11
2.4 Arquitetura da Mistura Hierárquica de Especialistas 11
2.5 Arquitetura de um microcontrolador baseado no modelo *Von Neumann* . . 17
2.6 Arquitetura de um microcontrolador baseado no modelo *Harvard* 17
2.7 Fluxo do ciclo de instruções de um microcontrolador utilizando a técnica pipeline 18
2.8 Estrutura básica de uma FPGA 19
2.9 Etapas do desenvolvimento de um projeto em FPGA 20
2.10 Soma de números inteiros em complemento de dois (modificada de (Stallings, 2010)) 24
2.11 Subtração de números inteiros em complemento de dois (modificada de (Stallings, 2010)) 24
2.12 Diagrama em blocos para as operações de soma e subtração (modificada de (Stallings, 2010)) 25
2.13 Fluxograma da multiplicação de dois números inteiros em complemento de dois (modificada de (Stallings, 2010)) 26
2.14 Fluxograma da adição e subtração em ponto flutuante (modificada de (Stallings, 2010)) 28
2.15 Fluxograma da multiplicação em ponto flutuante (modificada de (Stallings, 2010)) 29
2.16 Fluxograma da divisão em ponto flutuante (modificada de (Stallings, 2010)) 30

4.1 Fluxo de dados das duas etapas da arquitetura proposta 41
4.2 Arquitetura do cálculo de distância pela métrica de *Manhattan* 42

4.3 Arquitetura do cálculo de distância pela técnica Euclidiana quadrática . . 43
4.4 Arquitetura do grafo de proximidade 47
4.5 Arquitetura do grafo de borda 50
4.6 Arquitetura do Armazenamento de dados situados na borda 52
4.7 Arquitetura do cálculo de distância da fase de classificação pela métrica de *Manhattan* 54
4.8 Arquitetura do cálculo de distância da fase de classificação pela técnica Euclidiana quadrática 55
4.9 Arquitetura da classificação de dados 57

5.1 Superfícies de decisão geradas pelo método NN-clas e pelo CHIP-clas . . . 59
5.2 Relação entre o número de LUTs e o número de amostras 65
5.3 Relação entre o número de LUTs e o número de dimensões 66
5.4 Relação entre o número de LUTs e o número de dimensões da fase de classificação 68
5.5 Diagrama em blocos do microcontrolador *STM32F429ZI* (imagem extraída do *datasheet* desse microcontrolador) 70
5.6 Fluxograma do firmware desenvolvido para o NN-clas 71

Lista de Tabelas

5.1 Resultados dos algoritmos 60
5.2 Resultados do teste de Wilcoxon 61
5.3 Erro do cálculo de distância 62
5.4 Resultado de síntese dos blocos de soma/subtração e multiplicação 64
5.5 Resultados de síntese da arquitetura da fase de treinamento do NN-clas utilizando a técnica euclidiana quadrática para alguns números de amostras 64
5.6 Resultados de síntese da arquitetura da fase de treinamento do NN-clas utilizando a técnica euclidiana quadrática para alguns números de dimensão 64
5.7 Resultados do 1-NN implementado em hadrware e em software 66
5.8 Resultados de síntese da arquitetura da fase de classificação do NN-clas utilizando a técnica euclidiana quadrática para diferentes números de dimensões 67
5.9 Resultados do NN-clas sem filtro implementado em hadrware e em software 68
5.10 Resultados do NN-clas implementado em software e em microcontrolador . 73
5.11 Comparação de desempenho do NN-clas sem filtro em FPGA e no ARM . 73

Lista de Abreviaturas

ALU	*Arithmetic Logic Unit*
ARM	*Advanced RISC Machine*
ASIC	*Application Specific Integrated Circuit*
AUC	*Area Under The Curve*
CISC	*Complex Instruction Set Computing*
CPS	*Cyber – Physical Systems*
CPU	*Central Processor Unit*
DSP	*Digital Signal Processor*
EEPROM	*Electrically – Erasable Programmable Read – Only Memory*
EP	Elementos de processadores
EPROM	*Erasable Programmable Read – Only Memory*
FPGA	*Field Programmable Gate Array*
FPU	*Float Point Unit*
HDL	*Hardware Description Language*
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
kNN	*k – nearest neighbors*
LUT	*Look Up Table*
MHE	Mistura Hierárquica de Especialistas
MSE	*Mean Squared Error*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computing*
RNA	Redes Neurais Artificiais
SoC	*System – On – Chip*
SRAM	*Static Random Access Memory*
SVM-Poly	*Support Vector Machine – Polynomial Function*
SVM-RBF	*Support Vector Machine – Radial Basis Function*
UC	Unidade de Controle
VHDL	*Very High Speed Integrated Circuits Hardware Description Language*

Lista de Algoritmos

1 Classificação - CHIP-clas 12
2 Classificação - CHIP-clas Reduzido 13

3 Cálculo de distância - Treinamento NN-clas 32
4 Cálculo do grafo de proximidade - Treinamento NN-clas 33
5 Filtro - Treinamento NN-clas 34
6 Cálculo de amostras de suporte - Treinamento NN-clas 35
7 Classificação - NN-clas 35

Capítulo 1

Introdução

As três revoluções industriais foram caracterizadas por mudanças radicais nos âmbitos econômico, social e político, motivadas pela incorporação de tecnologias para o auxílio nas atividades manuais. A primeira revolução ocorreu entre 1760 e 1840 na Inglaterra, quando os métodos artesanais foram aos poucos sendo substituídos pelo uso de máquinas a vapor e vários tipos de ferramentas (da Costa, 2017). A segunda revolução industrial se deu entre o final do século XIX e início do século XX, trazendo como principais inovações o emprego do aço, o surgimento da linha de montagem e produção em massa, a utilização da energia elétrica e dos combustíveis derivados do petróleo, a invenção do motor a explosão e o desenvolvimento de produtos químicos. Por fim, a terceira revolução, teve início na década de 1960 e foi responsável pelo rompimento de paradigmas em vista do desenvolvimento de semicondutores e tecnologias como mainframes, computadores pessoais e, posteriormente, nos anos 1990, a Internet.

No início do século XXI, com a difusão da Internet, o desenvolvimento de sensores cada vez menores, mais potentes e sofisticados, o surgimento de máquinas autônomas capazes de aprender e colaborar na criação de grandes redes, promoveu-se uma quarta revolução na indústria, nomeada de "indústria 4.0" durante a Feira Industrial de Hannover, na Alemanha, pelos professores Erilk Braynjolfsson e Andrew McAfee (da Costa, 2017). A figura 1.1 mostra as revoluções industriais que ocorreram ao longo dos anos.

A quarta revolução industrial tem integrado novas tecnologias como Sistemas Ciber-físicos(CPS) e Internet das Coisas(IoT), unindo máquinas e humanos em cadeias de valor, e também compondo uma rede de entidades que podem estar localizadas em qualquer lugar, fornecendo serviços e produtos de forma autônoma (Silva et al., 2015).

Os *CPSs* são sistemas compostos por elementos computacionais colaborativos com o intuito de controlar entidades físicas (Khaitan e McCalley, 2015). A geração anterior à dos Sistemas Ciber-físicos é geralmente conhecida como sistemas embarcados, e encontraram aplicações em áreas diversas, tais como aeroespacial, automotiva, processos

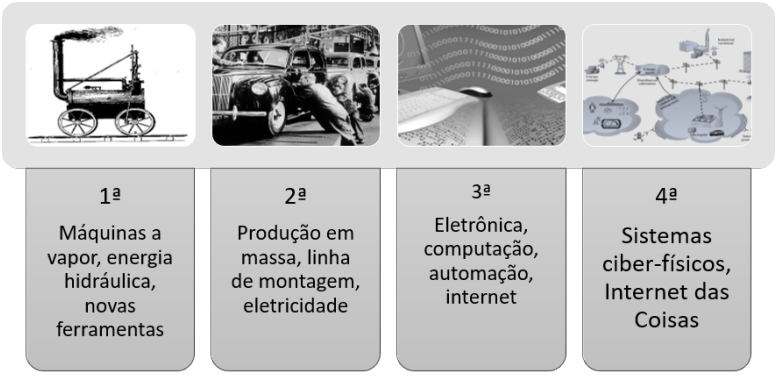


Figura 1.1: Histórico das revoluções industriais

químicos, infraestrutura civil, energia, saúde, manufatura, transporte, entretenimento, e aplicações voltadas ao consumidor (Khaitan e McCalley, 2015). Sistemas embarcados, no entanto, tendem a focar mais nos elementos computacionais, enquanto que Sistemas Ciber-físicos enfatizam o papel das ligações entre os elementos computacionais e elementos físicos (Lee e Seshia, 2016).

A Internet das Coisas, por sua vez, é definida como uma rede de objetos físicos que possuem tecnologia embarcada, sensores e conexão com rede capaz de coletar e transmitir dados. Também pode ser considerada como uma extensão da Internet atual, que permite os objetos de uso diário se conectarem à Internet. Essa conexão com a rede mundial de computadores, viabiliza o controle remoto de objeto e também permite que tais objetos sejam acessados como provedores de serviços (Al-Fuqaha et al., 2015).

A junção da Internet das Coisas aos Sistemas Ciber-físicos permite a comunicação e cooperação entre processos físicos distintos em tempo real, possibilitando a execução de "Fábricas Inteligentes" totalmente autônomas e cada vez menos dependentes da intervenção humana.

1.1 Motivação

No cenário da indústria 4.0, muitos trabalhos e estudos têm sido realizados buscando desenvolver Sistemas Ciber-físicos autônomos inteligentes capazes de aprender e tomar decisões próprias. No entanto, a aplicação de métodos de aprendizado de máquina nestes sistemas não é uma tarefa trivial, uma vez que os algoritmos utilizados são computacionalmente custosos em termos de processamento e memória, que são recursos limitados nos sistemas embarcados. Por isso é necessário otimizar estes algoritmos para reduzir sua complexidade assintótica sem prejuízo à sua eficácia.

O uso dos algoritmos de aprendizagem de máquina em sistemas embarcados surgiu nos anos de 1990 e início dos anos 2000 com as placas de circuitos integrados que

implementavam Redes Neurais Artificias (*RNA*) (He et al., 2008). O chip neural da *Intel ETANN* (Holler et al., 1989), por exemplo, lançado em 1989, já possuía sinapses adaptativas, apesar de ainda ter a fase de treinamento realizada externamente devido aos complexos algoritmos de otimização, e da necessidade de uma interface com o usuário para configuração de seus parâmetros.

Atualmente, outros estudos (Mohammed e Ali (2013), Muthuramalingam et al. (2008) e Holanda (2013)) têm sido desenvolvidos no campo da implementação de redes neurais em *FPGA*. Nestes, a fase de treinamento das *RNAs* também é realizada em um computador externo devido ao custo dos métodos de otimização utilizados para o aprendizado dos pesos das redes. Todavia, essa prática diminui a flexibilidade do sistema por não poderem ser treinados durante seu uso, impossibilitando sua aplicação em sistemas adaptativos *on-line*.

Nos trabalhos realizados por Manolakos e Stamoulias (2010a), Hussain et al. (2012) e Stamoulias e Manolakos (2013a), o método dos k vizinhos mais próximos (*kNN*) é implementado em *FPGA*. Esse método não depende da resolução de problemas complexos de otimização, pois são baseados apenas no cálculo de distância. Contudo, a eficácia desse método depende da configuração do parâmetro k. O que torna o sistema dependente do usuário para o melhor ajuste do k.

Para aplicações que exigem maior flexibilidade e autonomia ou para o uso de métodos de aprendizagem de máquina em sistemas *on-line*, mostra-se necessário o uso de algoritmos que não exigem a resolução de problemas de otimização complexos nem dependam de parâmetros configurados pelo usuário.

Encontra-se na literatura um método independente de parâmetros que foi desenvolvido por Torres et al. (2015), o qual se mostrou estatisticamente equivalente aos métodos *SVM-RBF* e *SVM-Poly*, sendo adequado para o uso nas aplicações descritas acima.

Sem depender de um método formal de otimização, o classificador proposto por Torres et al. (2015) leva em consideração apenas a estrutura dos dados representados pelo grafo de Gabriel (Gabriel e Sokal, 1969) para minimizar o erro do conjunto de treinamento e maximizar a margem de separação entre as classes. Nesse método, a superfície de decisão é resultado de um modelo de mistura de hiperplanos cujos parâmetros são extraídos dos padrões próximos à margem. A decisão sobre a classificação de um padrão arbitrário x depende da avaliação de todos os elementos da mistura. Esse procedimento, embora factível de ser implementado em hardware, tem custos envolvidos no procedimento de estimação da superfície de decisão.

1.2 Objetivos

1.2.1 Objetivo Geral

Diante desse cenário, o objetivo geral do presente trabalho é realizar um estudo e avaliar as melhores maneiras de se implementar o classificador de margem larga não paramétrico completo, ou seja, desenvolver tanto a fase de classificação como a de treinamento em sistemas embarcados.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral desse trabalho foram estabelecidos objetivos específicos, tais como:

1. Estudar e avaliar novas técnicas de se implementar a regra de decisão do classificador de margem larga, objetivando-se com isso a redução dos custos computacionais e sua implementação de maneira mais facilitada em hardware, sem a perda, contudo, do desempenho alcançado por Torres et al. (2015) na qualidade da classificação;

2. Separar as fases de treinamento e classificação do algoritmo em subetapas para modularizar o classificador e permitir que sejam reaproveitados no desenvolvimento de outros métodos de aprendizagem de máquina;

3. Reduzir o número de operações, quando possível, de cada um dos módulos do classificador;

4. Desenvolver um desenho arquitetural em hardware para cada subetapa com o melhor desempenho possível que possa ser parametrizável para qualquer número de amostras e características, dependendo dos requisitos de desenho impostos;

5. Desenvolver uma metodologia que permita a criação de uma ferramenta automática da arquitetura em hardware projetada em *VHDL*;

6. Realizar a simulação de cada um dos desenhos desenvolvidos para cada subetapa e posteriormente da arquitetura completa;

7. Definir uma metodologia que permita comparar os resultados obtidos.

1.3 Principais Contribuições deste Trabalho

As principais contribuições do trabalho são:

- Desenvolvimento de uma nova metodologia que se baseia na abordagem de vizinhos mais próximos (kNN) para a classificação de um novo padrão x;
- Estudo comparativo entre a formulação reduzida (baseada no hiperplano mais próximo) da regra de decisão proposta por Torres et al. (2015), a nova metodologia desenvolvida e o classificador original do Torres et al. (2015);
- Desenvolvimento de desenhos arquiteturais para cada uma das subetapas das duas fases do método de margem larga e posteriormente um desenho arquitetural do classificador completo para a implementação em *FPGA*;
- Implementação de uma ferramenta automática para a criação do código em *VHDL* da arquitetura projetada, parametrizável em número de amostras e características;
- Desenvolvimento do classificador de margem larga em linguagem C de maneira modularizada e eficiente, e posteriormente implementação desse método em um microcontrolador *ARM* de 32 bits *STM32F429ZI*;
- Publicação do artigo: GADE, L. R.; CASTRO, C. L.; TORRES, L. C. B.; COELHO, F. G. F.; BRAGA, A. P.; GARCIA, J. A.; TORRES, F. S. "NN-clas: classificador geométrico de margem larga baseado na regra do vizinho mais próximo." **Congresso Brasileiro de Inteligência Computacional, 2017, Niterói. XIII Brazilian Congress on Computational Intelligence. , 2017. v.XIII.**

1.4 Organização do Texto

O presente trabalho está organizado da seguinte forma:

No capítulo 2 é feita, inicialmente, uma fundamentação teórica dos métodos de aprendizado de máquina nos quais o classificador proposto foi baseado. Em seguida é feita uma breve introdução aos sistemas embarcados exemplificando os tipos existentes e suas arquiteturas, bem como as vantagens e desvantagens de cada uma delas. Por último, é apresentada a representação numérica utilizada na implementação do classificador em sistemas embarcados.

No capítulo 3 descreve-se a nova abordagem proposta para o classificador de margem larga e a metodologia escolhida para desenvolver a arquitetura em hardware desse classificador. Já o capítulo 4 descreve de maneira pormenorizada a arquitetura em hardware digital desenvolvida. O capítulo 5 apresenta todos os experimentos realizados

em cada etapa do trabalho e os resultados alcançados. Finalmente, no capítulo 6 são descritas as conclusões, sugestões de melhoria e continuidade do trabalho.

Capítulo 2

Fundamentação Teórica

O primeiro tópico desse capítulo descreverá os conceitos teóricos da técnica de classificação de padrões. Posteriormente será descrito o classificador de margem larga proposto por Torres et al. (2015), apresentando as duas técnicas que podem ser utilizadas na regra de decisão do método. A primeira técnica se baseia em uma mistura hierárquica de especialista e a segunda utiliza apenas o hiperplano mais próximo.

Já o segundo tópico apresentará o método de k vizinhos mais próximos (kNN), no qual a nova metodologia proposta se baseia. Os demais tópicos irão detalhar os fundamentos dos sistemas embarcados, descrevendo as plataformas existentes. Adicionalmente, alguns conceitos e definições de representação numérica serão apresentados para uma melhor compreensão das propostas sugeridas nesse trabalho.

2.1 Classificação de Padrões

A classificação de padrões está inserida no contexto de aprendizagem de máquina, área estudada na inteligência computacional com foco em pesquisas de técnicas capazes de extrair conceitos a partir da amostra de dados. Em outras palavras, pode-se afirmar que essas técnicas são programas de computador capazes de fazer com que máquinas tomem decisões baseadas em informações e experiências acumuladas e armazenadas em bases de dados a partir de soluções de problemas anteriores. O objetivo, portanto, dos referidos algoritmos é o de levar as máquinas a discriminarem e classificarem autonomamente exemplos de grupos diversos por meio da identificação de aspectos e características comuns dentre os exemplos presentes em uma determinada base de dados.

Existem basicamente dois tipos de classificação, no primeiro tem-se informações sobre objetos e o intuito é estabelecer a existência de classes. Esse tipo de classificação é conhecido como agrupamento ou aprendizagem não-supervisionada. Já o segundo

tipo de classificação supõe que existe um número conhecido de classes e o objetivo é estabelecer uma regra pela qual se possam alocar novos dados a uma das classes. Tal técnica é conhecida como aprendizagem supervisionada.

O tipo de classificação que mais vem sendo utilizado é o de aprendizagem supervisionada. Essa classificação supervisionada atribui uma classe a um dado desconhecido através de uma medida de similaridade desse novo dado com outros conhecidos previamente rotulados. Os dados conhecidos são chamados de conjunto de treinamento. A partir da medição de determinadas características dos dados do conjunto de treinamento, estimam-se os parâmetros que caracterizam cada classe no espaço de características (Gomes, 2008).

No domínio do espaço de característica opera uma função discriminante, também chamada de classificador, que, baseada na informação denotada pelo conjunto de treinamento, determina a semelhança de um novo padrão a uma das classes. Geometricamente, o trabalho do classificador pode ser entendido como a tarefa de particionar o espaço de características em sub-espaços, cada qual domínio de uma classe (Gomes, 2008).

2.2 Classificador de margem larga baseado em distância

O classificador de margem larga é um método supervisionado utilizado em sistemas que possuem apenas duas classes. Esse classificador binário, baseado em distância, (CHIP-clas) (Torres et al., 2015) obtém a informação estrutural dos dados através do grafo de Gabriel, que, por sua vez, depende apenas da distância entre as amostras de treinamento. O grafo de Gabriel de um conjunto de pontos consiste em um grafo cujo conjunto de vértices V e arestas A deve obedecer à definição expressa pela Equação 2.1, onde os pontos (p_i, p_j) constituirão uma aresta se e somente se nenhum outro ponto estiver contido na hiperesfera de diâmetro $D(p_i, p_j)$, sendo que tal diâmetro consiste na distância entre os pontos p_i e p_j. A figura 2.1 exemplifica essa teoria, mostrando a formação de uma aresta entre os vértices p_i e p_j, uma vez que, não existe nenhum outro ponto contido na hiperesfera.

$$(p_i, p_j) \in A \leftrightarrow D^2(p_i, p_j) \leq [D^2(p_i, p_z) + D^2(p_j, p_z)] \, \forall z \in V, p_i, p_j \neq p_z \tag{2.1}$$

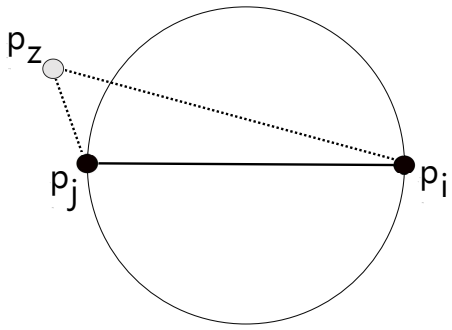


Figura 2.1: Diagrama explicando a criação do grafo de Gabriel

O grafo de Gabriel $G = \{V, A\}$ de um conjunto de dados de treinamento $T = \{(x_i, y_i)|i = 1, ..., N\}$, onde $y_i \in \{+1, -1\}$ e $x_i \in \mathbb{R}^n$ também possui um conjunto de Arestas de Suporte AS, o qual representa todas as arestas de A que possuem um par de vértices (x_i, x_j) de classes distintas. Caso não haja sobreposição entre as amostras, é possível afirmar que os vértices de AS localizam-se na margem de separação entre as classes, como pode ser visto na Figura 2.2. Dessa forma, o hiperplano H_l que passa pelo ponto médio de um par de vértices x_i e x_j pertencentes a AS, refere-se ao classificador de margem máxima em relação aos ditos vértices (Torres, 2016).

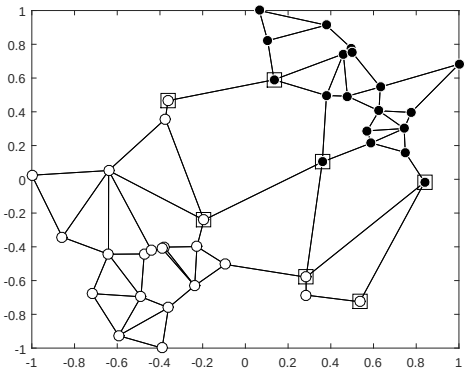


Figura 2.2: Grafo de Gabriel de duas gaussianas com arestas de suporte indicadas pelos quadrados

Para que a metodologia descrita acima possa ser aplicada em problemas com sobreposição ou ruídos entre as classes de dados, deve-se utilizar no CHIP-clas uma técnica de filtragem, como a de Garcia et al. (2015), para eliminar as amostras ruidosas antes da detecção do conjunto de Arestas de Suporte AS. Essa técnica de filtragem utiliza operações próprias da teoria dos Grafos, como por exemplo, o grau do vértice, o qual representa o número de arestas conectadas ao referido vértice. Segundo Garcia

et al. (2015), os vértices de grau baixo podem ser rotulados como um possível ruído. Adotou-se assim, uma medida de qualidade para cada vértice do grafo, aqui formalmente definida pela Equação 2.2, onde $Gr(x_i)$ representa o grau do vértice x_i e $\hat{Gr}(x_i)$ o grau de x_i menos os vértices de classe distinta de x_i (Torres, 2016).

$$\begin{aligned} q(x_i) &= \frac{\hat{Gr}(x_i)}{Gr(x_i)}, \\ \hat{Gr}(x_i) &= \{\forall x_j \in Gr(x_i) | y_j = y_i\} \end{aligned} \tag{2.2}$$

A filtragem é baseada na medida de qualidade $q(.)$ e pode ser dividida em três fases:

1. Calcula-se, inicialmente $q(x_i)$ para todo $x_i \in G$. Em seguida, os $q(x_i)$ são agrupados por classe, de forma que Q^+ e Q^- representam a medida de qualidade entre as classes $+1$ e -1, respectivamente;

2. Posteriormente, deve se calcular de acordo com a Equação 2.3 o valor do limiar t^+ e t^- de cada classe como a média da medida de qualidade pertencente à Q^+ e Q^-;

3. Por último, deve-se remover todos os vértices de G que possuem $q(x_i)$ menores que t^+ e t^- (Torres, 2016). No caso específico do CHIP-clas, a filtragem somente é realizada se $q(x) > 0, \ \forall x \in T$.

A Figura 2.3 exemplifica a importância do processo de filtragem em bases com sobreposição e ruídos. O gráfico mais a esquerda mostra um conjunto de dados que possui amostras ruidosas. Como se pode observar os ruídos, indicados pelos quadrados, dificultam o processo de obtenção das arestas de suporte, podendo então definir algumas arestas, como de suporte, que não estejam localizadas na margem de separação entre as classes. Eliminando as amostras ruidosas, como exemplificado pelo gráfico a direita, é possível identificar de maneira mais precisa os vértices de AS localizados na borda de separação entre as classes.

$$t^+ = \frac{\sum_{q(x_i) \in Q^+} q(x_i)}{|Q^+|}, \ t^- = \frac{\sum_{q(x_i) \in Q^-} q(x_i)}{|Q^-|} \tag{2.3}$$

Após a eliminação das amostras ruidosas e obtenção dos vértices de AS, será gerado um classificador representado pelo hiperplano H_i que passa pelo ponto médio de tais vértices. É importante ressaltar, que um único hiperplano H_i não separa todas as

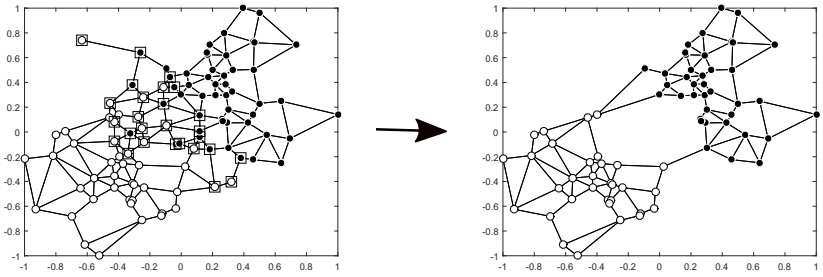


Figura 2.3: Remoção de amostras ruidosas de uma base de dados com sobreposição entre as classes

amostras do conjunto de treino, mas a combinação de todos os hiperplanos produz, segundo Torres et al. (2015), um classificador com a informação espacial de todas as amostras em T.

2.2.1 Mistura de hiperplanos

A classificação final resulta de uma Mistura Hierárquica de Especialistas (MHE), na qual cada hiperplano terá um peso diferente diante a um novo padrão de entrada x a ser rotulado. De acordo com Torres et al. (2015), a arquitetura da MHE é representada através de uma rede, conforme a Figura 2.4, onde a primeira camada corresponde aos especialistas locais $\{H_1, ..., H_m\}$. Já a saída de m especialistas para um novo padrão de entrada x a ser rotulado é representada pelas funções $h_1(x), ..., h_m(x)$. Cada especialista é ponderado por um módulo *Gating Network*, onde o peso $c_i(x)$ para o i-ésimo especialista é obtido de acordo com a Equação 2.4, o $pl = (x_i + x_j)/2$ representa o ponto médio da aresta de suporte formada pelos vértices x_i e x_j e $D(x, p_l)$ é a distância entre a amostra x a ser rotulada e o ponto médio p_l.

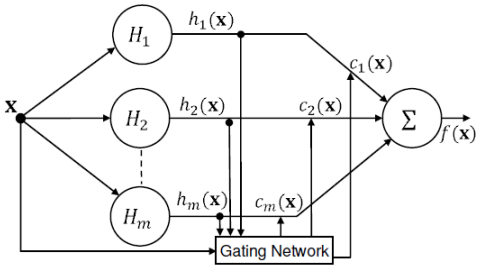


Figura 2.4: Arquitetura da Mistura Hierárquica de Especialistas

$$c_l(x) = exp - \left(\frac{(max(D(x, p_k)))^2}{D(x, p_l)} \right), \forall k = 1, ..., m \tag{2.4}$$

Após o cálculo dos pesos pela Equação 2.4 uma normalização é imposta, tal que $\sum_{l=1}^{m} c_l(x) = 1$. Em seguida, o resultado final da classificação é obtido por $f(x) = sign\left(\sum_{l=1}^{m} h_l(x)c_l(x)\right)$, sendo que $h_l = sign(x^T w_l - b_l)$. O $sign(.)$ representa a função sinal, $w_l = (x_l - x_j)$ e o termo independente $b_l = [(1/2)(x_l + x_j)]w_l^T$. O algoritmo 1 descreve de forma detalhada o processo de classificação realizado por esse método de margem larga.

Algoritmo 1: Classificação - CHIP-clas

Entradas : conjunto de pontos médios das arestas de suporte P, conjunto de parâmetros B e W dos hiperplanos e conjunto de teste Xt.
Saída : classes do conjunto de teste CTe

```
for j in Xt do
    for i in P do
        Calcula a distância entre o novo padrão e o ponto médio da aresta de suporte
        d(i) = D(Xt(j), P(i))
    end for
    Calcula a distância máxima
    mD = max(d)
    for i in W do
        c(i) = exp − ((mD)^2 / d(i))
        h(i) = sign(Xt(j)^T W(i) − B(i))
    end for
    c = 1 / (c + (1e^−16))
    c = c / sum(c)
    CTe(j) = sign(sum(h * c))
end for
retorna CTe
```

Percebe-se que a mistura de hiperplanos ainda é custosa para implementação em hardware, principalmente por causa das operações que utilizam exponencial, como representado pela Equação 2.4, e pelos cálculos para se obter a função representada pela Equação 2.5.

$$f(x) = sign\left(\sum_{l=1}^{m} h_l(x)c_l(x)\right) \tag{2.5}$$

.

2.2.2 CHIP-clas Reduzido

O CHIP-clas reduzido realiza a classificação de acordo com o hiperplano mais próximo. A técnica desse método é obtida da seguinte maneira: dado uma amostra x a ser rotulada, encontra-se o ponto médio l mais próximo de x através de:

$$\arg\min_{l} D(x, P_l), \quad l = 1, ..., m \tag{2.6}$$

onde $D(\cdot)$ é a função que retorna o valor da distância entre dois vetores. O hiperplano de separação que passa pelo ponto médio l mais próximo de x é definido como $H_l(x) = (x^T w_l - b_l)$, onde $b_l = [(1/2)(x_i + x_j)]w_l^T$ e $w_l = (x_i + x_j)$, sendo (x_i, x_j) o par de vértices que formam a Aresta de Suporte associada ao ponto médio l. O rótulo de x é dado pela função de classificação $f(x)$, descrita em 2.7. Esse processo pode ser visto de maneira mais detalhada pelo Algoritmo 2.

$$f(x) = \begin{cases} +1 & \text{se} \quad H_l(x) > 0 \\ -1 & \text{se} \quad H_l(x) \leq 0. \end{cases} \tag{2.7}$$

Algoritmo 2: Classificação - CHIP-clas Reduzido

Entradas :conjunto de parâmetros B e W dos hiperplanos, conjunto de pontos médios das arestas de suporte P e conjunto de teste Xt.
Saída : classes do conjunto de teste CTe

```
for j in Xt do
    for i in P do
        Calcula a distância entre o novo padrão e os pontos médios
        d(i) = D(Xt(j), P(i))
    end for
    Hiperplano h_l que passa pelo ponto médio P_l mais próximo de Xt(j)
    h_l = sign(x^T w_l − b_l)
    if h_l > 0 then
        CTe(j) = +1
    else
        CTe(j) = −1
    end if
end for
retorna CTe
```

2.3 kNN

O k-vizinhos mais próximos, kNN (*k Nearest Neighbors*), é um classificador clássico baseado em memória, que é frequentemente usado em aplicações do mundo real devido a sua simplicidade. Apesar de simples, ele tem conseguido considerável exatidão na

classificação em diversas aplicações e é consequentemente muito usado como uma base para comparação com novos classificadores (da Silva Júnior, 2007).

O kNN possui duas fases, a primeira denominada de fase de treino simplesmente armazena todos as amostras de treinamento rotuladas. Na segunda fase, ou também chamada de fase de classificação, primeiramente calcula-se as distâncias de uma nova amostra a ser classificada com todos os padrões do conjunto de treinamento. Em seguida, o algoritmo considera os k padrões do conjunto de treinamento com as menores distâncias ao padrão a ser classificado. Por último, o novo padrão é classificado como pertencente a classe da maioria dos k padrões mais próximos do conjunto de treinamento.

Apesar da técnica ser muito simples, o custo computacional do kNN é elevado. Do ponto de vista do custo da memória, o kNN armazena todos os padrões de treinamento e quanto ao desempenho, para cada padrão a ser classificado o kNN deve calcular sua distância para todos os padrões de treinamento. A complexidade desse algoritmo é $O(mn)$ (Manolakos e Stamoulias, 2010b) por vetor de teste, onde m é a dimensão das amostras de treino e n o número de tais amostras.

2.4 Sistemas Embarcados

Os denominados "sistemas embarcados" ou "embutidos" são sistemas microprocessados de funcionalidades específicas (Queiroz, 2015), o que os distinguem dos computadores convencionais, os quais são utilizados para propósito geral.

Embora não sejam reconhecidos pela maioria das pessoas, esses sistemas estão presentes em vários dispositivos eletrônicos, tais como centrais telefônicas, drives de armazenamento, impressoras, equipamentos médicos e de redes de computadores (roteadores, *hubs*, *switches*), além de videogames, eletrodomésticos (micro-ondas, máquinas de lavar, *smart* tv), carros (controladores de tração, de motor, freios *ABS*), aeronaves (sistemas de controle inercial, controle de voo), dentre outros.

Os sistemas embarcados são implementados através de diferentes tipos de tecnologia como por exemplo: microcontroladores, circuitos de hardware reconfigurável (*FPGAs - Field Programmable Gate Array*), circuitos integrados de aplicação específica (*ASICs - Application Specific Integrated Circuit*) e sistemas em um único chip (*SoC - System-On-A-Chip*). Cada uma dessas tecnologias possui vantagens e desvantagens, ficando a critério do projetista escolher a que mais se adeque a sua aplicação.

As subseções, a seguir, descrevem as tecnologias citadas, apresentando suas características e arquiteturas de hardware.

2.4.1 Microcontroladores

Os microcontroladores são dispositivos eletrônicos que possuem em um único chip os principais componentes presentes em um computador: *CPU*(*Central Processor Unit* ou Unidade de Processamento Central), memória de programa, memória de dados, circuito de *clock* e dispositivos de entrada e saída (de Almeida et al., 2017). Contudo, diferentemente dos computadores de propósito geral, os microcontroladores são mais limitados em termos de memória e processamento, e também possuem um consumo de potência bem reduzido. Eles são ideais para o uso em aplicações específicas que exigem baixo consumo de energia, custo e tamanho reduzido.

A unidade de processamento central (*CPU*) é o principal componente de um microcontrolador e tem como finalidade executar as instruções do programa e processar dados. Ela possui internamente os seguintes blocos, dentre outros: (Stallings, 2010)

- Unidade lógica aritmética (*ALU - Arithmetic-Logic Unit*): responsável por realizar todas as operações lógicas e aritméticas;
- Registradores: armazenam dados temporários e podem ser divididos em registradores de uso geral e específico;
- Unidade de Controle (*UC - Control Unit*): realiza o controle de todas as unidades da *CPU* de maneira lógica e sincronizada;
- Barramentos Internos: permite a comunicação entre a Unidade de Controle, a Unidade Lógica e Aritmética e os Registradores.

A memória de programa contém o software embarcado desenvolvido pelo projetista, também chamado de *firmware*, traduzido em linguagem de máquina. Cada posição dessa memória contém uma instrução que é executada pela *CPU* de maneira sequencial. Essas memórias são do tipo não-volátil, ou seja, conservam suas informações internas mesmo quando a alimentação é retirada.

A memória de dados, por sua vez, armazena as informações e dados utilizados ao longo da execução do programa. Tais memórias são do tipo voláteis, ou seja, perdem todos seus dados quando desligada.

Já os circuitos de *clock* auxiliam na geração do sinal de *clock* utilizado para sincronizar os componentes do microcontrolador e controlar a velocidade de execução de uma instrução.

Os dispositivos de entrada e saída são responsáveis por realizar uma interface do microcontrolador com o ambiente externo, permitindo o controle e leitura de dispositivos externos, como por exemplo, sensores e indicadores visuais.

Cada microcontrolador possui uma arquitetura interna específica variando de acordo com o conjunto de instruções que lhes são atribuídas, com o modelo pelo qual ocorre o acesso aos dados e programas, largura do barramento de dados e dos registradores representada em quantidade de bits, número de dispositivos de entrada e saída e diferentes tipos de periféricos.

O conjunto de instruções do microcontrolador pode ser do tipo *CISC*(*Complex Instruction Set Computing*) ou RISC(*Reduced Instruction Set Computing*). A arquitetura *RISC* possui um conjunto simples e reduzido de instruções, resultando em uma Unidade de controle simples, barata e rápida. Já as do tipo *CISC* possuem um conjunto complexo de instruções, produzindo programas de máquina mais curtos e reduzindo então, o número de acesso à memória para buscar instruções (Bhandarkar e Clark, 1991).

Como as arquiteturas *RISC* visam Unidades de Controle mais simples, rápidas e baratas, elas geralmente optam por instruções mais simples possível, com pouca variedade e com poucos endereços. A pouca variedade dos tipos de instrução e dos modos de endereçamento, além de demandar uma Unidade de Controle mais simples, também traz outro importante benefício, que é a previsibilidade. Como as instruções variam pouco de uma para outra, é mais fácil para a Unidade de Controle prever quantos ciclos serão necessários para executá-las. Ao saber quantos ciclos serão necessários para executar um estágio de uma instrução, a Unidade de Controle saberá exatamente quando será possível iniciar o estágio de uma próxima instrução (Bhandarkar e Clark, 1991).

Já as arquiteturas *CISC* investem em Unidades de Controle poderosas e capazes de executar tarefas complexas como a Execução Fora de Ordem e a Execução Superescalar. Na Execução Fora de Ordem, a Unidade de Controle analisa uma sequência de instruções ao mesmo tempo. Muitas vezes há dependências entre uma instrução e a seguinte, assim, a Unidade de Controle busca outras instruções para serem executadas que não são as próximas da sequência e que não sejam dependentes das instruções atualmente executadas. A Execução Superescalar é a organização do microcontrolador em diversas unidades de execução, como Unidades de Pontos Flutuante e Unidades de Inteiros. Essas unidades trabalham simultaneamente. Enquanto uma instrução é executada por uma das unidades de inteiros, outra pode ser executada por uma das unidades de Pontos Flutuantes. Com a execução Fora de Ordem junto com a Superescalar, instruções que não estão na sequência definida podem ser executadas para evitar que as unidades de execução fiquem ociosas (Bhandarkar e Clark, 1991).

Com relação ao modelo de acesso aos dados e instruções existem dois tipos de arquitetura, a proposta por *John von Neumann* denominada de "*von Neumann*" e a criada pela universidade de *Harvard* conhecida como arquitetura de *Harvard*.

A arquitetura de *von Neumann* se caracteriza por compartilhar no mesmo espaço

de memória os dados e o programa, e possuir apenas um barramento para a transferência de dados e instruções entre a memória e a *CPU* (Stallings, 2010). Dessa forma, a transferência de bits desses dados e instruções obedece a um único padrão de endereçamento e tamanho de bits, o que causa um "gargalo" devido a limitação de banda de transferência entre a *CPU* e a memória. Geralmente a *CPU* tem capacidade de processar uma maior quantidade de dados em comparação com a memória, contudo precisa aguardar a leitura de dados para então executar a próxima instrução. A figura 2.5 mostra um digrama dessa arquitetura.

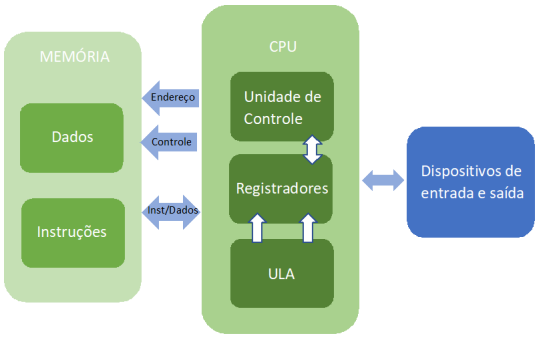


Figura 2.5: Arquitetura de um microcontrolador baseado no modelo *Von Neumann*

Já a arquitetura *Harvard* possui barramentos independentes para a comunicação com memória de instrução e de dados, permitindo a *CPU* buscar uma nova instrução enquanto executa outra. Dessa maneira, obtém-se um desempenho melhor do que a de *Von Neumann*. A figura 2.6 exemplifica essa arquitetura de *Harvard*.

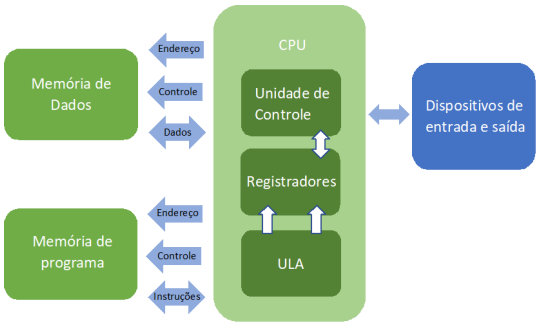


Figura 2.6: Arquitetura de um microcontrolador baseado no modelo *Harvard*

Com relação a largura do barramento de dados e dos registradores as arquiteturas mais comuns são as de 8, 16 e 32bits. Uma arquitetura com maior número de bits permite a movimentação de uma maior quantidade de dados por unidade de tempo, aumentando o desempenho do microcontrolador.

Uma outra variação comum nas arquiteturas dos microcontroladores é a capacidade de dividir o processamento em vários estágios distintos, técnica conhecida como *pipeline*. Quando uma nova instrução é carregada, inicialmente passa pelo primeiro estágio permanecendo nessa etapa durante apenas um ciclo de *clock*, em seguida é processada pelo segundo e sucessivamente pelos demais estágios. A vantagem desta técnica, é que o primeiro estágio não precisa aguardar a instrução passar por todas as etapas para carregar a próxima. Dessa maneira, o microcontrolador é capaz de processar simultaneamente, em um único ciclo de *clock*, várias instruções que normalmente demorariam vários ciclos para serem processadas. A figura 2.7 explica o funcionamento de um microcontrolador utilizando essa técnica.

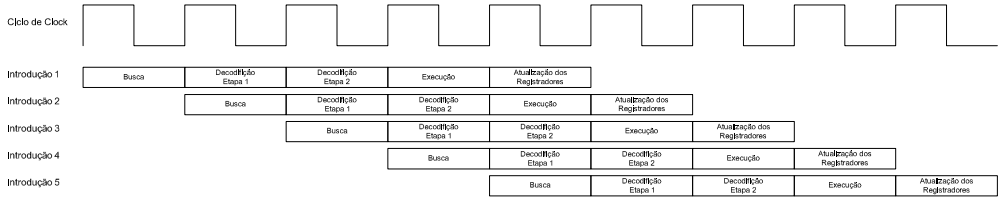


Figura 2.7: Fluxo do ciclo de instruções de um microcontrolador utilizando a técnica pipeline

Os microcontroladores baseados na arquitetura *ARM*(*Advanced RISC Machine*), denominados de *ARM Cortex-M* têm sido muito utilizados em projetos de sistemas embarcados devido ao seu alto desempenho alcançado pelo uso da arquitetura RISC de 32-bits e da técnica pipeline (Yiu, 2013).

A família *ARM Cortex-M* é formada por distintas subfamílias que possuem características diversas. As subfamílias *Cortex-M0*, *Cortex-M0+* e *Cortex-M1* são baseadas no modelo *Von Neumann*, não possuem instruções *DSP* e nem Unidade de Ponto Flutuante(*FPU*) (tagkey2015iii, 2015). Já as subfamílias *Cortex-M3*, *Cortex-M4* e *Cortex-M7* são baseadas no modelo de *Harvard*, possuem divisão em hardware e conseguem alcançar alta velocidade de processamento. As linhas *Cortex-M4* e *M7* possuem instruções *DSP* e em alguns casos são formados por Unidade de Ponto Flutuante(*FPU*), o que acelera a execução de algoritmos que utilizam operações em ponto flutuante (Yiu, 2013).

2.4.2 FPGA

Os *FPGAs*(*Field Programable Gate Arrays*) são dispositivos semicondutores programáveis que consistem em uma matriz de blocos lógicos capazes de serem configurados para reproduzir o comportamento de diversos circuitos digitais (Gonçalves, 2005). A figura 2.8 ilustra a estrutura geral de um *FPGA*, onde os três componentes básicos

são: os blocos de entrada e saída representados pelos retângulos, os blocos lógicos demonstrados pelos quadrados brancos e as interconexões chaveadas simbolizadas pelos quadrados azuis.

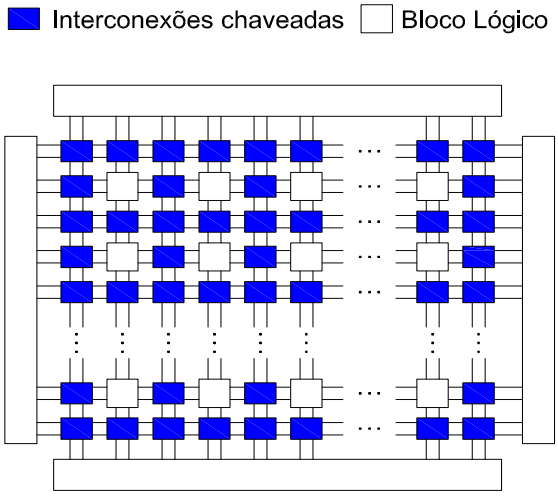


Figura 2.8: Estrutura básica de uma FPGA

Alguns modelos de *FPGA* também possuem recursos lógicos adicionais como por exemplo: unidades lógicas e aritméticas(*ULAs*), blocos de memória, decodificadores, dispositivos de processamento digital *DSP* e blocos otimizados de multiplicadores. Com relação aos tipos de elementos de programação, existem quatro tipos diferentes: a *RAM* estática (*SRAM*), o anti-fusível(*antifuse*), *EPROM* e *EEPROM*, sendo que as tecnologias mais utilizadas são as do tipo *SRAM* e *antifuse* (Gonçalves, 2005).

A tecnologia baseada nas *RAMs* estáticas possuem as conexões programáveis compostas por transistores de passagem, portas de transmissão ou multiplexadores. Essa tecnologia possibilita uma configuração rápida, contudo, toda vez que o dispositivo é energizado, o arquivo de configuração precisa ser carregado e configurado. Dessa maneira, é necessário o uso de memórias externas permanentes, onde os bits de configuração do dispositivos sejam armazenados (Gonçalves, 2005).

Nesse tipo de tecnologia as células lógicas são compostas por uma tabela de consulta *LUT - Look Up Table*, utilizada no lugar de portas convencionais, que determina a saída baseada nos valores de entrada. Essas *LUTs* representam na prática o conceito de tabela verdade.

Já a tecnologia baseada no *antifuse* não requer dispositivos de memória externa para armazenar os bits de configuração, uma vez que mantêm as conexões de configuração de forma permanente. As células lógicas utilizam a alocação de portas lógicas tradicionais ao invés de *LUTs*. A vantagem dessa tecnologia em relação a do tipo *SRAM* é a necessidade de uma área menor para implementação. A desvantagem é que o dispositivo

só pode ser programado uma única vez, dessa forma, todo vez que houver a necessidade de uma mudança na lógica do programa, essa nova configuração deve ser feita em um novo dispositivo.

Outra característica dos *FPGAs* é o tipo de granularidade que os compõem. O grão é considerado a menor unidade configurável de um *FPGA* e é classificado de acordo com as seguintes categorias:

1. Grão Grande: o grão é composto por unidades lógicas e aritméticas, pequenos microprocessadores e memórias;
2. Grão Médio: o grão consiste em dois ou mais LUTs e *flip-flops*;
3. Grão Pequeno: o grão é formado por blocos lógicos que contêm uma função lógica de duas entradas ou um multiplexador 4x1 e um *flip-flop*.

A metodologia de projeto para implementar uma arquitetura em um *FPGA* envolve várias etapas que geralmente são automatizadas com a utilização de ferramentas de *software*. A figura 2.9 representa o fluxo dessas etapas.

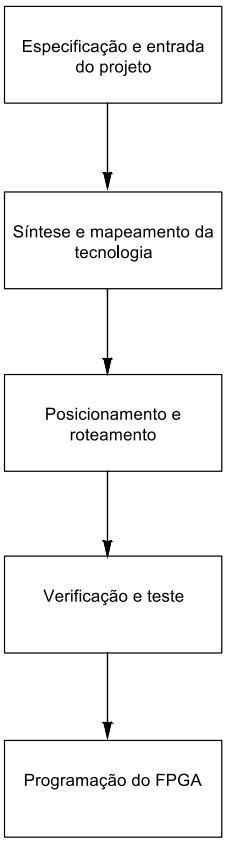


Figura 2.9: Etapas do desenvolvimento de um projeto em FPGA

A primeira etapa chamada de especificação do projeto consiste no desenvolvimento dos requisitos do sistema, estudo da viabilidade de implementação e desenho da arquitetura. A entrada do projeto pode ser realizada através de um editor gráfico, utilizando portas lógicas e macroinstruções, ou por meio de um editor de texto que utilize uma linguagem de descrição de hardware como por exemplo o *VHDL - Very High Speed Integrated Circuit Hardware Description Language* ou o Verilog (Oliveira, 2010).

A fase síntese lógica é utiliza para otimizar o projeto através de algoritmos de síntese que simplificam as equações booleanas geradas, reduzindo assim a área a ser ocupada no circuito integrado e o atraso de propagação(*delay*) dos sinais envolvidos. Já o mapeamento da tecnologia seleciona um conjunto de portas lógicas de uma biblioteca para implementar as representações abstratas, enquanto melhora a área, o atraso ou a combinação de ambos, considerando as restrições arquiteturais do *FPGA* (Oliveira, 2010).

Na etapa de posicionamento é realizado a atribuição de componentes particulares do circuito integrado aos elementos lógicos de projeto. No roteamento, realiza-se a ligação de trilhas e elementos programáveis, utilizando os recursos disponíveis de interconexão para a comunicação entre os componentes (Oliveira, 2010).

Durante a fase de verificação e teste é realizada uma simulação do projeto para verificar o funcionamento do sistema e investigar as restrições e temporização. Os simuladores mais utilizados são o *ModelSim* e o *QuestaSim* da empresa *Mentor Graphics*.

Depois de realizadas todas as etapas descritas acima, gera-se um arquivo de configuração e programa-se o *FPGA*, finalizando assim a implementação do projeto.

2.4.3 ASIC

O *ASIC*(*Application Specific Integrated Circuits* - Circuito Integrado de aplicação específica), como o próprio nome diz, é um circuito integrado customizado para ser utilizado em uma aplicação específica. Esse tipo de circuito é composto por uma estrutura de interconexão personaliza que exige uma construção única para um determinado sistema. Geralmente são utilizados em sistemas embarcados com o intuito de reduzir custos, tamanho, consumo e aumentar o desempenho. No entanto o projeto e fabricação desses circuitos integrados personalizados podem ser demorados e dispendiosos (Smith, 1997).

A personalização envolve um longo ciclo de projeto durante a fase de definição do produto e altos custos de Engenharia Não Recorrente (NRE) durante a fase de fabricação. Além disso, se ocorrer erros de projeto nos circuitos personalizados, o ciclo de projeto e fabricação deve ser repetido, agravando o *time to market*. Dessa maneira, os ASICs são apropriados para aplicações específicos de alto volume e baixo custo (Smith, 1997).

2.4.4 SoC FPGAs

O *SoC* (*System-on-Chip*) FPGA integra arquiteturas de microcontroladores e FPGA em um único dispositivo. Consequentemente, proporciona uma maior integração, menor tamanho e maior comunicação de banda entre o microcontrolador e FPGA (Qin e Berekovic, 2015). O SoC FPGA também é formado por um conjunto de periféricos, memórias e transceptores de alta velocidade.

Esse tipo de tecnologia aproveita a flexibilidade da arquitetura FPGA, para acompanhar a mudança de padrões e as demandas dos usuários finais, combinadas com a complexidade reduzida para o desenvolvimento de algoritmos sequenciais utilizando microcontroladores.

2.5 Representação numérica

Os aspectos mais importantes para a realização de cálculos em computadores de propósito geral ou em sistemas embarcados são a maneira como os números são representados e os algoritmos utilizados para realizar as operações básicas como adição, subtração, multiplicação e divisão (Stallings, 2010). Os números podem ser representados em ponto fixo ou em ponto flutuante, já os algoritmos usados para operações básicas variam de acordo com a representação dos números.

As subseções, a seguir, descrevem os dois tipos de representação numérica e seus algoritmos utilizados para realizar as operações básicas.

2.5.1 Representação em ponto fixo

Um número inteiro A sem sinal pode ser representado por uma sequência de n bits de dígitos binários $a_{n-1}\ a_{n-2}\ ...\ a_1\ a_0$ como mostra a Equação 2.8. Já para representação de um número inteiro com sinal pode ser utilizado dois tipos de técnicas distintas denominadas de representação em sinal-magnitude e representação em complemento de dois. Nesses dois tipos de representação de inteiros com sinal o bit mais significativo é considerado como bit de sinal, caso esse bit seja 0 o número é positivo, do contrário, negativo.

$$A = \sum_{i=0}^{n-1} 2^i a_i \tag{2.8}$$

A representação em sinal-magnitude pode ser expressa de acordo com a Equação 2.9. Essa técnica, no entanto, não é muito utilizada pelo fato das operações de adição e subtração exigirem uma consideração dos sinais dos números e de suas relativas mag-

nitudes e pelo o número 0 possuir duas representações, dificultando assim a operação de verificação de um valor ser igual a zero.

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i, & \text{if } a_{n-1\ =\ 0}. \\ -\sum_{i=0}^{n-2} 2^i a_i, & \text{otherwise.} \end{cases} \tag{2.9}$$

A representação em complemento de dois pode ser expressa pela Equação 2.10. Quando $a_{n-1} = 0$, o termo $-2^{n-1}a_{a-1}$ se iguala a zero e a equação define um número inteiro não negativo. Se $a_{n-1} = 1$, o termo 2^{n-1} é subtraído do somatório, resultando assim em um número negativo (Stallings, 2010). O número 0 é identificado como um número positivo e apresenta o bit de sinal e os demais como 0 (Stallings, 2010).

$$A = -2^{n-1}a_{a-1} + \sum_{i=0}^{n-2} 2^i a_i \tag{2.10}$$

A operação de negação em complemento de dois é realizada da seguinte forma:

1. Inicialmente é feito o complemento booleano bit a bit do número inteiro. Ou seja, é realizado uma inversão de cada um dos bits, dessa maneira, caso um bit seja 1 se tornará 0, e se for 0 mudará para 1;

2. Depois é realizada a soma do resultado da inversão com 1.

A adição em complemento de dois é mostrada na figura 2.10. Como se pode observar os quatros primeiros exemplos demonstram operações bem sucedidas, ou seja, o resultado não é maior que o tamanho da palavra. Já os dois últimos exemplos mostram operações em que se ocorreu um estouro (*overflow*), invalidando o resultado. Também é possível notar na figura que em alguns casos, existe um bit de *carry* no início da palavra, destacado pelo sombreado, que é ignorado (Stallings, 2010).

Já na operação de subtração inicialmente é realizado a negação do subtraendo e em seguida o resultado é somado ao minuendo. A figura 2.11 exemplifica essa operação. As operações de soma e subtração podem ser representadas em conjunto através de um diagrama em blocos como mostrado na figura 2.12, onde os registradores A e B armazenam os dois operandos, caso seja uma operação de subtração o subtraendo é armazenado no registrador B, o flag OF indica a ocorrência de *overflow* e o seletor SW verifica se a operação é soma ou subtração, se for subtração o subtraendo armazenado no registrador B passará por um processo de negação antes de ser somado ao minuendo armazenado no registrador A. O resultado da operação é armazenado em um dos dois registradores ou em um terceiro (Stallings, 2010).

A multiplicação é uma operação um pouco mais complexa que as anteriormente citadas. A figura 2.13 descreve os passos dessa operação. Como se pode observar,

1001 = -7 + 0101 = 5 1110 = -2 (a) (-7) + (+5)	1100 = -4 + 0100 = 4 1000 = 0 (b) (-4) + (+4)
0011 = 3 + 0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = -4 +1111 = -1 11011 = -5 (d) (-4) + (+-1)
0101 = 5 + 0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = -7 +1010 = -6 10011 = Overflow (f) (-7) + (-6)

Figura 2.10: Soma de números inteiros em complemento de dois (modificada de (Stallings, 2010))

0010 = 2 +1001 = -7 1011 = -5 (a) M = 2 = 0010 S = 7 = 0111 -S = 1001	0101 = 5 +1110 = -2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 -S = 1110
1011 = -5 +1110 = -2 11001 = -7 (c) M = -5 = 1011 S = 2 = 0010 -S = = 1110	0101 = 5 + 0010 = 2 0111 = 7 (d) M = 5 = 0101 S = -2 = 1110 -S = = 0010
0111 = 7 + 0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = -7 = 1001 -S = = 0111	1010 = -6 + 1100 = -4 10110 = Overflow (f) M = -6 = 1010 S = 4 = 0100 -S = = 1100

Figura 2.11: Subtração de números inteiros em complemento de dois (modificada de (Stallings, 2010))

inicialmente o multiplicador e o multiplicando são armazenados nos registradores Q e M respectivamente, o registrador A e o registrador Q_{-1}, de um bit adicionado logicamente à direita do bit menos significativo do registrador Q, são inicializados com 0 e um contador é inicializado com o número de bits que representa os números. Em seguida, verifica-se os bits do multiplicador um de cada vez, e a medida que cada bit é examinado, o bit à sua direita também é verificado. Caso os dois bits sejam iguais, então todos os bits dos registradores A, Q e Q_{-1} são deslocados à direita por 1 bit.

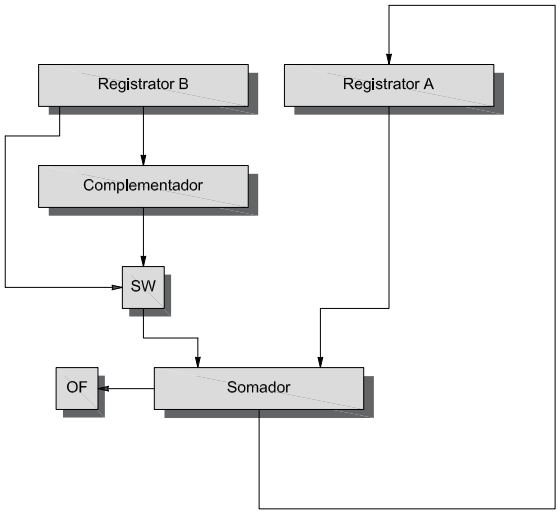


Figura 2.12: Diagrama em blocos para as operações de soma e subtração (modificada de (Stallings, 2010))

Se forem diferentes, é realizado a soma ou subtração do multiplicando com o registrador A e o resultado é então armazenado no registrador A. Posteriormente, ocorre um deslocamento à direita de 1 bit dos registradores A, Q e Q_{-1}. Depois, o contador é decrementado e tudo é repetido até que o contador se iguale a zero. O resultado final é composto pelo valor armazenado em A e Q (Stallings, 2010).

A divisão é um pouco mais complexa que a multiplicação, porém se baseia nos mesmo princípios. O algoritmo pode ser resumido da seguinte maneira: (Stallings, 2010)

1. Inicialmente é realizado a negação dp divisor e o resultado é armazenado no registrador M. O dividendo é carregado nos registradores A e Q, pois o dividendo necessita ser expresso como um número positivo de $2n$ bits;
2. Os registradores A e Q são deslocados à esquerda de 1 bit;
3. O registrador A é subtraído do divisor e o resultado é armazenado em A.
4. Verifica-se o valor de A, caso não seja negativo, é armazenado 1 em Q_{-1}, se for negativo o valor anterior de A é restaurado e o valor 0 é atribuído a Q_{-1};
5. As etapas 2 a 4 são repetidas n vezes, onde n é o número de bits de Q;
6. O resto está armazenado em A e o quociente em Q.

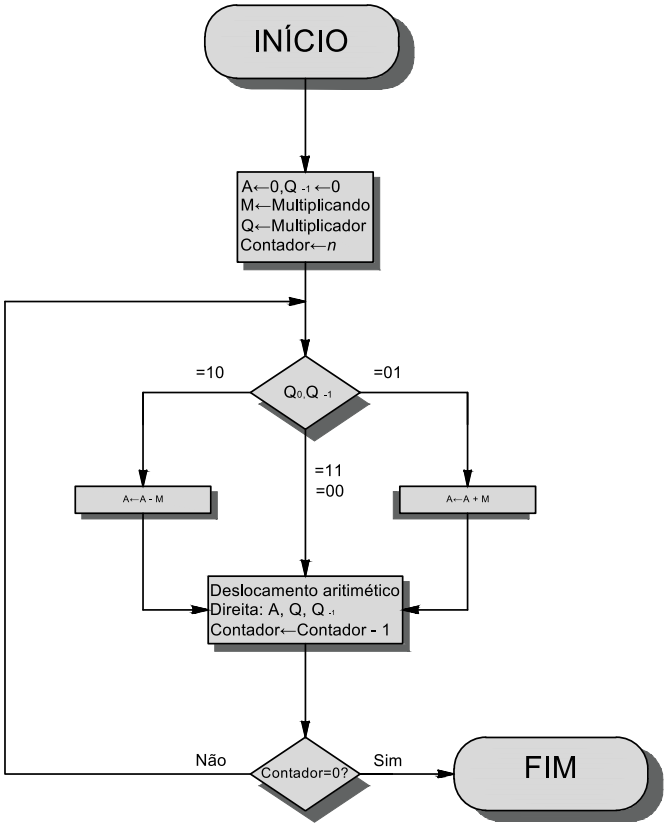


Figura 2.13: Fluxograma da multiplicação de dois números inteiros em complemento de dois (modificada de (Stallings, 2010))

2.5.2 Representação em ponto flutuante

A representação de ponto flutuante mais utilizada é a definida pelo *IEEE Standart 754* que foi adotada em 1985 com o intuito de facilitar a portabilidade dos programas de um processador para outro e incentivar o desenvolvimento de algoritmos sofisticados, orientados numericamente (Stallings, 2010). Esse padrão representa um número por uma cadeia de bits caracterizado por três componentes principais: um bit de sinal S, um expoente E com 8 bits quando o formato for simples e 11 bits se for duplo, e uma mantissa M com 23 bits no formato simples e 52 bits no duplo.

O sinal S pode conter dois valores, 0 representando números positivos e 1 números negativos. O expoente E armazena seu valor pela notação com peso, também chamada de notação por excesso de valor. Essa notação pode ser representada por $2^{k-1}-1$, onde k é o número de bits que o expoente possui. Dessa maneira, em precisão simples, os limites dos valores são dados por excesso de 127 e em precisão dupla por excesso de 1023. A mantissa M representa um número fracionário normalizado, de maneira que o bit mais a esquerda da mantissa é sempre 1. Como esse valor é fixo não há necessidade de armazená-lo, contudo implicitamente ele faz parte da representação. A mantissa

pode ser calculada de acordo com a Equação 2.11. Assim número real por ser expresso pela Equação 2.12.

$$M = m_{22}2^{-1} + m_{21}2^{-2} + ... + m_1 2^{-22} + m_0 2^{-23} \tag{2.11}$$

$$N = (-1)^s(1 + M)(2)^{E-Peso} \tag{2.12}$$

As operações de soma e subtração em ponto flutuante são mais complexas que as de multiplicação e divisão, pelo fato da necessidade de realizar uma etapa de alinhamento. O algoritmo de adição e subtração pode ser dividido em quatro fases:

1. Verificar zeros;
2. Alinhar os significandos;
3. Somar ou subtrair os significando;
4. Normalizar o resultado.

Como a adição e a subtração são idênticas, exceto pela mudança de sinal, o algoritmo começa alterando o sinal do subtraendo, caso a operação seja de subtração. Depois, é verificado se algum dos operandos é 0, caso seja, o outro número é informado como o resultado (Stallings, 2010).

A fase de alinhamento é necessária para que os números tenham os expoentes iguais, facilitando assim o algoritmo. O alinhamento é realizado deslocando-se o número menor para a direita e esse processo é repetido até que os dois expoentes se igualem. Caso esse processo resulte em um valor 0 para o significando, o outro número é informado como resultado. Dessa maneira, se dois números tiverem expoentes que diferem de forma significativa, o número menor é perdido (Stallings, 2010).

Depois da fase de alinhamento, os dois significandos são somando, considerando seus sinais. Assim o resultado pode ser 0 caso os sinais sejam diferentes. Existe também, a possibilidade de ocorre um *overflow* do significando por 1 dígito. Caso isso ocorra, o significando do resultado é deslocado para a direita e o expoente é incrementado. Se ainda houver um *overflow* de expoente, a operação é encerrada e isso é informado (Stallings, 2010).

Por último, o resultado é normalizado. A normalização consiste no deslocamento dos dígitos do significando para esquerda até que o dígito mais significativo seja diferente de zero. O deslocamento causa um decremento do expoente, dessa maneira, pode ocorrer um estouro para baixo (*underflow*) do expoente, encerrando assim a operação. Se não houver um *underflow* o resultado é arredondado, finalizando assim a operação.

A figura 2.14 mostra em formato de fluxograma todas as etapas das operações de adição e subtração (Stallings, 2010).

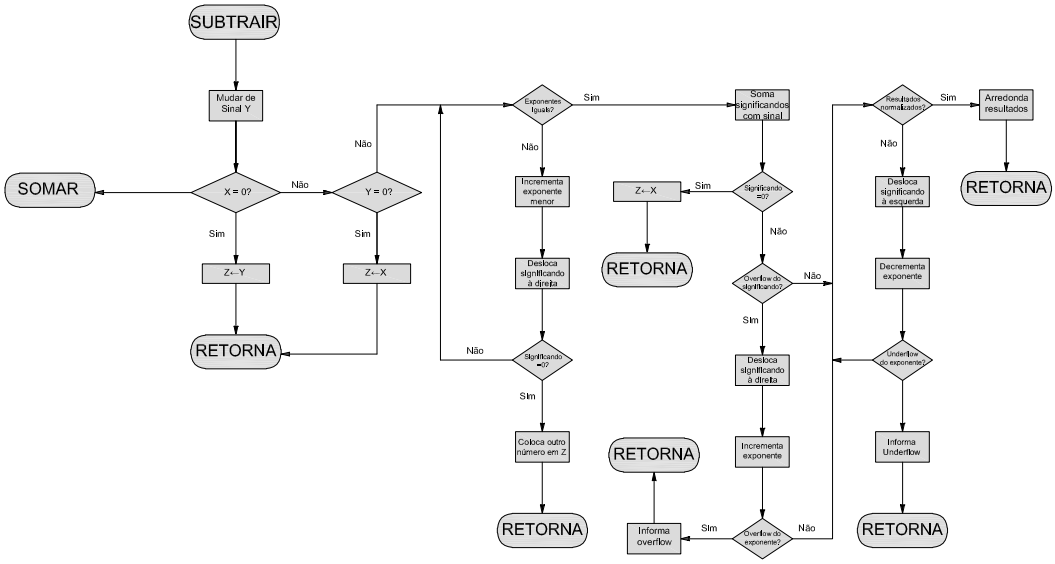


Figura 2.14: Fluxograma da adição e subtração em ponto flutuante (modificada de (Stallings, 2010))

Como dito anteriormente, a multiplicação e a divisão em ponto flutuante são processos mais simples que a adição e a subtração. As etapas da multiplicação são demonstradas pela figura 2.15. Como se pode observar inicialmente é verificado se algum operando é igual a zero, caso seja o valor 0 é informado como resultado. Se nenhum dos operandos for 0 o próximo passo é realizar a soma dos expoentes. Como os expoentes geralmente são armazenados de maneira polarizada, a soma do expoente dobra a polarização, dessa forma, o valor da polarização precisa ser subtraído da soma. Se ocorrer um *overflow* ou um *underflow* o algoritmo é finalizado. Caso não ocorra nenhum desse eventos, o passo seguinte é multiplicar os significandos, considerando seus sinais. O produto resultante será o dobro do tamanho do multiplicador e do multiplicando, contudo, os bits extras são extraídos durante a fase de arredondamento. Depois do cálculo do produto, o resultado é normalizado e arredondado da mesma maneira que é feito nas operações de adição e subtração.

A operação de divisão em ponto flutuante é mostrada através da figura 2.16. Observa-se nessa figura que o primeiro passo, assim como na multiplicação é verifi-

car se algum operando é igual a zero. Caso o divisor seja 0, um erro é emitido ou o resultado é definido como infinito, dependendo da aplicação. Se o dividendo for 0 o resultado final também será 0. Depois, o expoente do divisor é subtraído do expoente do dividendo, contudo essa etapa remove a polarização, então precisa ser somada novamente. Em seguida, realiza-se os testes de *overflow* e *underflow*, e por último, é feito a normalização e posteriormente o arredondamento.

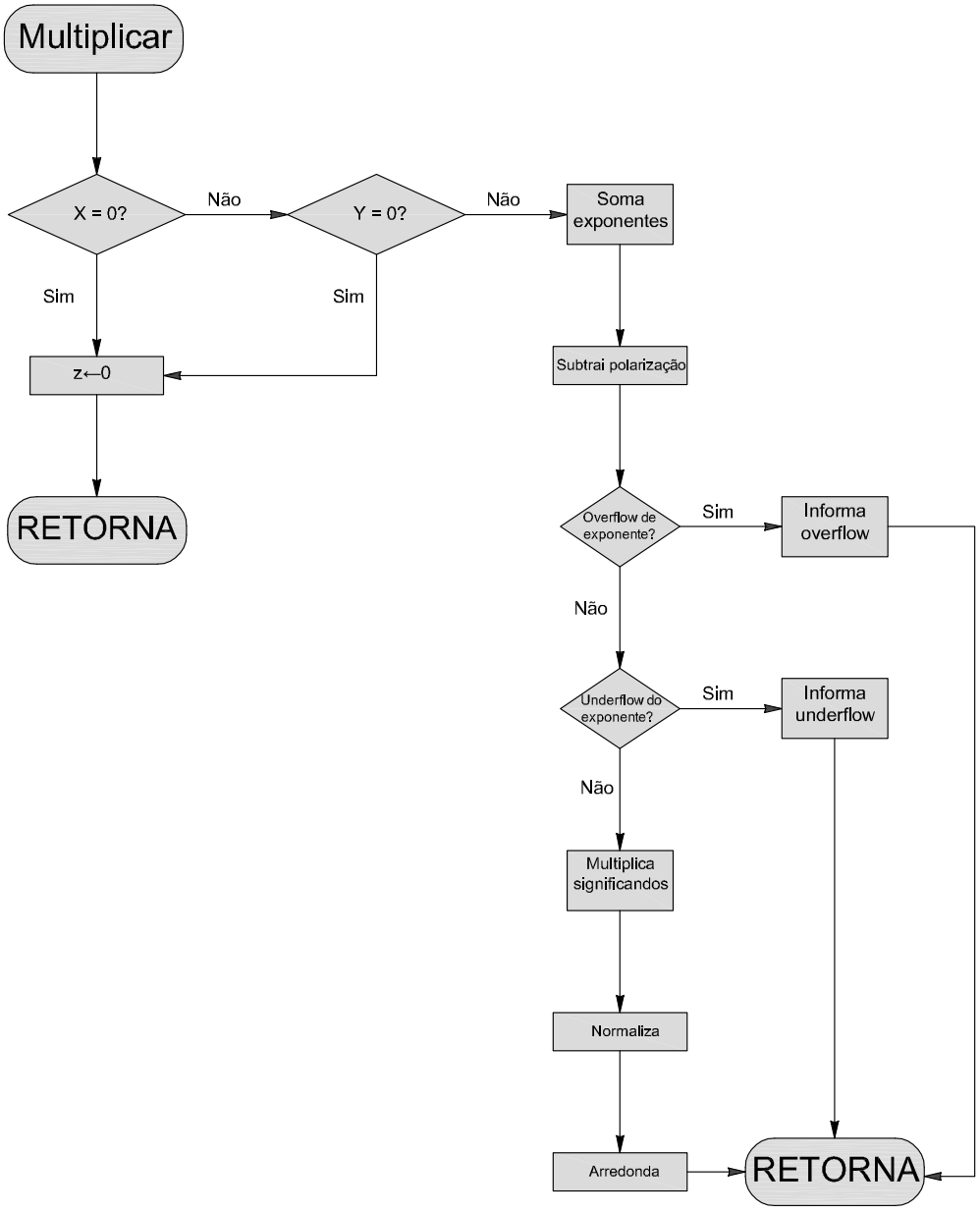


Figura 2.15: Fluxograma da multiplicação em ponto flutuante (modificada de (Stallings, 2010))

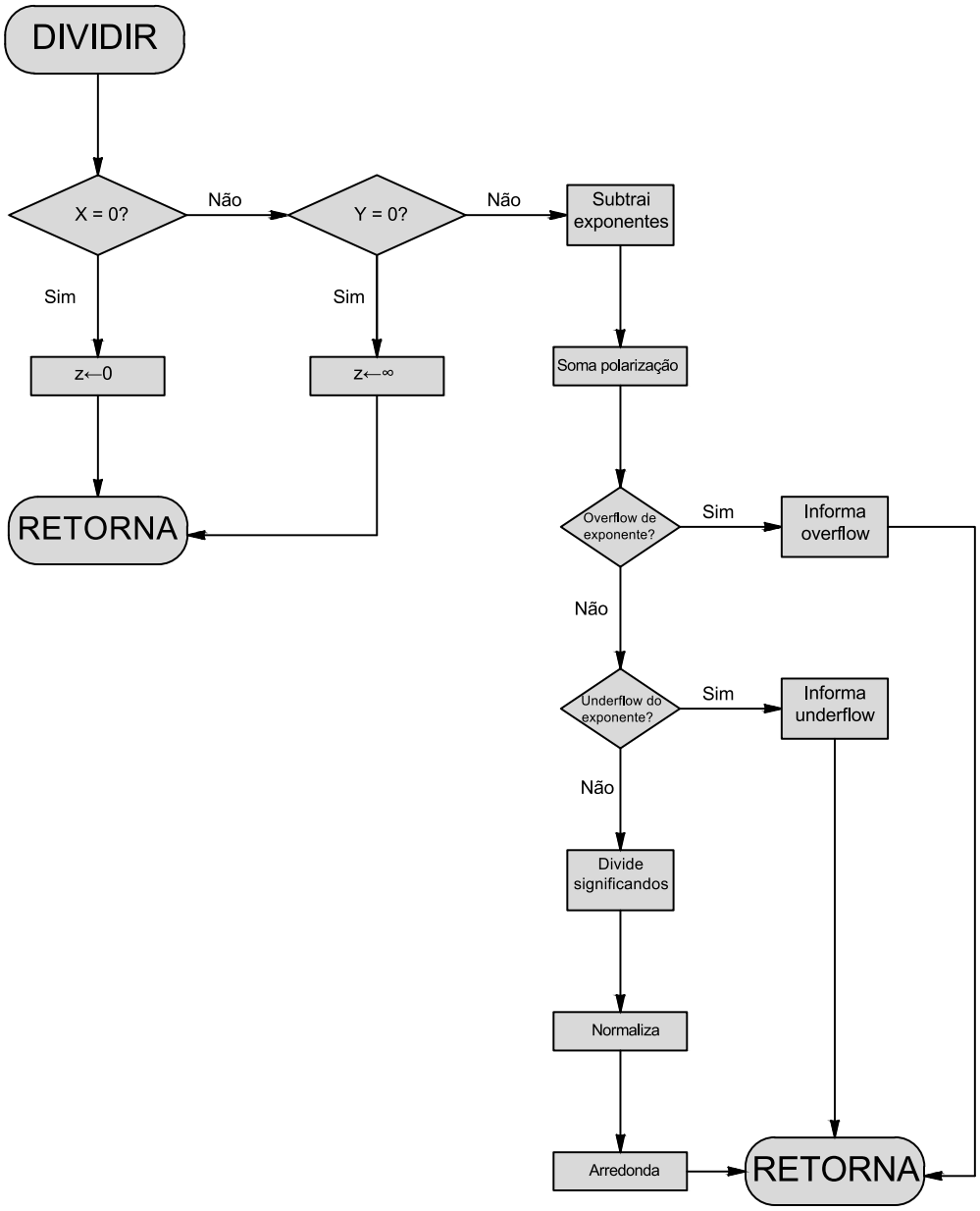


Figura 2.16: Fluxograma da divisão em ponto flutuante (modificada de (Stallings, 2010))

Capítulo 3

Nova abordagem para o classificador de margem larga e metodologia para sua implementação em hardware

Esse capítulo irá abordar a nova proposta para o classificador de margem larga CHIP-clas e a metodologia utilizada para o desenvolvimento da arquitetura em hardware para esse algoritmo. Inicialmente é detalhada a nova abordagem proposta, denominada NN-clas, desenvolvida de forma a simplificar à implementação do classificador em sistemas embarcados. Em seguida, será mostrada a metodologia utilizada para desenvolver uma arquitetura para o algoritmo de aprendizado de máquina utilizando o *FPGA*. Essa plataforma foi escolhida por causa de seu paralelismo intrínseco, permitindo assim desenvolver arquiteturas de alto desempenho com *time-to-market* menor do que em projetos utilizando *ASIC*.

3.1 NN-clas

O NN-clas consiste em uma extensão do classificador de margem larga baseado em distância, porém, com um menor custo computacional. A diferença da nova abordagem em relação ao CHIP-clas está na regra de decisão. O CHIP-clas utiliza uma mistura de hiperplanos para estimar a superfície de decisão, já o NN-clas baseia-se na regra de vizinhos mais próximos.

Assim como realizado no CHIP-clas, inicialmente, obtêm-se, no NN-clas, o Grafo de Gabriel (G) a partir de um conjunto de dados de treinamento $T = \{(x_i, y_i)|i = 1, ..., N\}$, onde $y_i \in \{+1, -1\}$ e $x_i \in \mathbb{R}^n$, sendo o conjunto de vértices formado por todas as amostras de treinamento, ou seja, $V = \{x_i|i = 1, ..., n\}$ e o conjunto de arestas A satisfazendo à condição: $D^2(p_i, p_j) \leq [D^2(p_i, p_z) + D^2(p_j, p_z)]$.

Em seguida, realiza-se a filtragem dos dados ruidosos, utilizando a técnica que foi devidamente descrita pela Equação 2.3. Após a eliminação dos dados ruidosos repete-se a construção do Grafo de Gabriel, porém apenas com as amostras não ruidosas. Em seguida, encontra-se o conjunto de vértices das arestas de suporte presentes na borda entre as classes, finalizando assim a fase de treinamento.

Algoritmo 3: Cálculo de distância - Treinamento NN-clas

Entradas : conjunto de dados de treinamento T
Saída : distância entre todas as amostras d
  for j in T **do**
    for i in T **do**
      $d(j,i) = D(T(j), T(i))$
    end for
  end for
  retorna d

De acordo com essas etapas descritas, a fase de treinamento do NN-clas pode ser dividida nos seguintes passos:

- Cálculo de distância entre o conjunte de amostras de treino T;
- Cálculo do grafo de proximidade para definir o conjunto de arestas A que satisfação $D^2(p_i, p_j) \leq [D^2(p_i, p_z) + D^2(p_j, p_z)]$;
- Cálculo da medida de qualidade de todos os vértices do conjunto T, cálculo dos limiares t^+ e t^- e eliminação dos vértices presentes no conjunto T que possuam uma medida de qualidade menor que t^+ para as amostras de classe $+1$ ou menor que t^- para as de classe -1;
- Novo cálculo de distância entre as amostras não ruidosas;
- Novo cálculo do grafo de proximidade, também apenas com as amostras não ruidosas;
- Cálculo do conjunto de vértices das arestas de suporte presentes na borda entre as classes.

Os algoritmos 3, 4, 5 e 6 exemplificam de maneira detalhada cada um dos passos da fase de treinamento. Como se pode observar os cálculos de distância e grafo de proximidade são repetidos, por isso foram apresentados apenas 4 algoritmos para os 6 passos da etapa de treinamento do NN-clas.

Algoritmo 4: Cálculo do grafo de proximidade - Treinamento NN-clas

Entradas : distância entre todas as amostras d
Saída : conjunto de arestas A

```
for j in d do
  for i in d do
    aux = 1
    for w in d do
      if d(j,i)^2 > (d(j,w)^2 + d(i,w)^2 then
        aux = 0
      end if
    end for
    if aux = 1 then
      Vértice i forma uma aresta com vértice j
      A(i,j) = 1
    else
      Vértice i não forma uma aresta com vértice j
      A(i,j) = 0
    end if
  end for
end for
retorna A
```

Já na fase de classificação, diferentemente da forma como é realizado no CHIP-clas, o NN-clas calcula a distância entre cada amostra de teste x com todos os vértices das arestas de suporte, em seguida a nova amostra é classificada com o rótulo do vértice mais próximo. Essa fase de classificação é detalhada de maneira pormenorizada pelo algoritmo 7. Observa-se que essa técnica utilizada na regra de decisão é equivalente à aplicação do algoritmo kNN com $k = 1$, porém, considerando apenas o subconjunto de vértices de borda.

Percebe-se portanto, que esta nova metodologia elimina as operações mais custosas do CHIP-clas geradas pela Mistura Hierárquica de Especialistas, tais como os cálculos dos pesos de todos hiperplanos, definido pela Equação 2.4 e a soma ponderada dos hiperplanos $f(x)$. Facilitando assim, a implementação do NN-clas em hardware.

3.2 Metodologia

A metodologia escolhida para o desenvolvimento arquitetural em hardware do NN-clas baseou-se na figura 2.9 apresentada na fundamentação teórica sobre *FPGA*. Tal metodologia consiste primeiramente em realizar a especificação e entrada do projeto, em seguida fazer o mapeamento e síntese da tecnologia, depois efetuar o posicionamento e roteamento e posteriormente a verificação e teste. Como o objetivo inicial foi propor

Algoritmo 5: Filtro - Treinamento NN-clas

Entradas : conjunto de arestas A, conjunto de dados de treinamento T e rótulos das amostras de treinamento Y
Saída : novo conjunto de amostra de treinamento sem dados ruidosos T e novo conjunto de rótulos das amostras de treinamento Y

```
Q+ = 0
somaP = 0
Q- = 0
somaN = 0
for j in A do
    grauI = 0
    grauD = 0
    for i in A do
        if A(i,j) = 1 then
            if Y(i) = Y(j) then
                grauI = grauI + 1
            else
                grauD = grauD + 1
            end if
        end if
    end for
    Medida de qualidade do vértice j
    q(j) = grauI/(grauI + grauD)
    if Y(j) = 1 then
        Q+ = Q+ + 1
        somaP = somaP + q(j)
    else
        Q- = Q- + 1
        somaN = somaN + q(j)
    end if
end for
t+ = somaP/Q+
t- = somaN/Q-
for j in T do
    if Y(j) = 1 then
        if q(j) < t+ then
            Remove o vértice j do conjunto de amostras de treinamento T e seu rótulo do conjunto Y
        end if
    else
        if q(j) < t- then
            Remove o vértice j do conjunto de amostras de treinamento T e seu rótulo do conjunto Y
        end if
    end if
end for
retorna T
```

uma arquitetura parametrizável para qualquer número de amostras de treinamento e dimensão, não realizou-se a programação em um *FPGA* específico, apenas simulou-se seu comportamento através de um *software*.

A especificação e entrada do projeto foi realizada em diversas etapas. Inicialmente dividiu-se as duas fases do algoritmo NN-clas (treinamento e classificação) em subetapas de forma que a arquitetura ficasse modularizada e muitas das sub-fases pudessem ser utilizadas no desenvolvimento de arquiteturas de outros métodos de aprendizagem de máquina.

A fase de treinamento foi desenvolvida primeiramente sem a subetapa de filtro para que o modelo pudesse ser implementado em fluxo de dados. Dependendo do resultado alcançado com essa arquitetura seria feito um estudo de viabilidade da incorporação

Algoritmo 6: Cálculo de amostras de suporte - Treinamento NN-clas

Entradas : conjunto de dados de treinamento T, conjunto de arestas A, rótulos das amostras de treinamento Y

Saída : conjunto de vértices das arestas de suporte V e rótulos das arestas de suporte C

```
index = 1
for j in T do
    aux = 0
    for i in T do
        if −Y(j) = (A(i, j) ∗ Y(j)) then
            aux = aux + 1
        end if
    end for
    if aux > 0 then
        V(index) = T(j)
        C(index) = Y(j)
        index = index + 1
    end if
end for
retorna C e V
```

Algoritmo 7: Classificação - NN-clas

Entradas : conjunto de vértices das arestas de suporte V, rótulos das arestas de suporte C e conjunto de teste Xt.

Saída : classes do conjunto de teste CTe

```
for j in Xt do
    for i in V do
        Calcula a distância entre o novo padrão e os vértices da aresta de suporte
        d(i) = D(Xt(j), V(i))
    end for
    CTe(j) = rótulo C(l) do vértice mais próximo do padrão Xt(j)
end for
retorna CTe
```

da filtragem ao modelo desenvolvido. Esse algoritmo de treinamento sem o filtro pode ser dividido nas seguintes subetapas:

1. Cálculo de distância entre as amostras de treinamento;
2. Cálculo do grafo de proximidade, para detectar quais vértices formam arestas entre si;
3. Cálculo do grafo de borda, para detectar quais amostras estão situadas na margem de separação;

4. Armazenar as amostras de dados que se encontram na borda.

Já a fase de classificação foi projetada de acordo com o algoritmo 7. Dessa maneira suas sub-tarefas são:

1. Cálculo de distância entre as amostras armazenadas na etapa anterior e o novo dado a ser rotulado;

2. Encontrar a amostra mais próxima a esse novo dado e classificá-lo com o rótulo dessa amostra.

Após a separação do algoritmo NN-clas em subetapas, realizou-se o desenho arquitetural de cada uma dessas sub-tarefas, inicialmente para um conjunto de 4 amostras de treinamento e 4 características. Durante o desenvolvimento de cada módulo buscou-se alternativas de otimizar ainda mais os algoritmos de cálculo de distância, grafo de proximidade e grafo de borda.

Para realizar o cálculo de distância várias métricas podem ser utilizadas, contudo, a mais usual é a distância euclidiana que pode ser definida como:

$$D(X,Y) = \sqrt{\sum_{i=1}^{M}(x_i - x_i)^2} \tag{3.1}$$

Ou seja, $D(X,Y)$ representa a distância entre os vetores $X = \{x_1, x_2, x_3, x_4, \ldots, x_M\}$ e $Y = \{y_1, y_2, y_3, y_4, \ldots, y_M\}$ de dimensão M, utilizando a métrica Euclidiana. Contudo essa métrica apresenta um alto consumo de recursos computacionais, quando implementada em hardware, devido às operações de multiplicação utilizadas para a obtenção da raiz quadrada (Hussain et al., 2011).

Uma alternativa para o cálculo de distância é o uso da métrica de *Manhattan* definida pela Equação 3.2. Como se pode observar essa métrica possui um menor consumo computacional, uma vez que não requer o cálculo da raiz quadrada. Essa métrica tem sido adotada em diversos trabalhos como os Manolakos e Stamoulias (2010a), Hussain et al. (2012), Stamoulias e Manolakos (2013a), Stamoulias e Manolakos (2013b) e Hussain et al. (2011), os quais implementam algoritmos de aprendizado de máquina, baseado em distância, em hardware digital. Contudo, apesar dessa métrica ser bastante utilizada, é importante ressaltar que seu uso pode prejudicar a acurácia do método para algumas bases de dados, quando comparado com o uso da métrica Euclidiana, como foi descrito nos trabalhos Estlick et al. (2001), Estlick (2002) e James Theiler et al. (2000).

$$D(X,Y) = \sum_{i=1}^{M}|x_i - x_i| \tag{3.2}$$

Diante disso, buscou-se outras técnicas que pudessem ser utilizadas como alternativas para as duas métricas citadas anteriormente, contudo que tivesse um menor consumo computacional que a Euclidiana e uma acurácia superior a de *Manhattan*.

A distância Euclidiana quadrática, exemplificada pela Equação 3.3, não é considerada uma métrica, pois não satisfaz a desigualdade triangular, contudo, é uma técnica frequentemente aplicada em problemas em que a distância é utilizada apenas para ser comparada, como ocorre no caso do algoritmo *kNN* e NN-clas. O uso dessa técnica nesses algoritmos não reduz a acurácia quando comparada com a métrica Euclidiana original e ainda elimina a operação de raiz quadrada. Entretanto, esse técnica é um pouco mais custosa em relação à métrica de *Manhattan*, pois necessita de uma operação a mais de multiplicação.

$$D(X,Y) = \sum_{i=1}^{M}(x_i - x_i)^2 \tag{3.3}$$

Dessa maneira, propõe-se o desenvolvimento de dois blocos distintos para realizar o cálculo de distância. Um bloco utiliza a métrica de *Manhattan* e o outro a Euclidiana quadrática. Com isso, dependendo da aplicação pode-se utilizar um bloco ou outro.

Independentemente da técnica utilizada para realizar o cálculo de distância, o espaço métrico entre um conjunto de amostras pode ser definido por uma matriz, exemplificada pela Equação 3.4. Cada elemento dessa matriz representa a distância entre a amostra i e j, por exemplo, o elemento D_{12} representa a distância entre a amostra a_1 e a_2.

$$\begin{bmatrix} D_{11} & D_{12} & D_{13} & \dots & D_{1n} \\ D_{21} & D_{22} & D_{23} & \dots & D_{2n} \\ D_{31} & D_{32} & D_{33} & \dots & D_{3n} \\ D_{41} & D_{42} & D_{43} & \dots & D_{4n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ D_{n1} & D_{n2} & D_{n3} & \dots & D_{nn} \end{bmatrix}, \tag{3.4}$$

Percebe-se que a diagonal dessa matriz de distâncias é formada por zeros ($D_{11} = D_{22} = D_{33} = \ldots = D_{nn} = 0$), uma vez que, a distância entre uma amostra i e ela mesma é zero. Também é possível observar que os elementos à esquerda da diagonal são iguais aos elementos à direita da diagonal ($D_{12} = D_{21}$, $D_{13} = D_{31}$, $D_{14} = D_{24}$, $D_{23} = D_{32}$, $D_{24} = D_{42}$, $\ldots$, $D_{(n-1)n} = D_{n(n-1)}$). Dessa forma, como só é preciso realizar o cálculo dos elementos à direita da diagonal, reduziu-se então esse cálculo de

distância eliminando as operações desnecessárias.

$$\begin{bmatrix} D_{11}^2 & D_{12}^2 & D_{13}^2 & \dots & D_{1n}^2 \\ D_{21}^2 & D_{22}^2 & D_{23}^2 & \dots & D_{2n}^2 \\ D_{31}^2 & D_{32}^2 & D_{33}^2 & \dots & D_{3n}^2 \\ D_{41}^2 & D_{42}^2 & D_{43}^2 & \dots & D_{4n}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ D_{n1}^2 & D_{n2}^2 & D_{n3}^2 & \dots & D_{nn}^2 \end{bmatrix}, \tag{3.5}$$

O cálculo do grafo de proximidade, como mostrado pelo algoritmo 4, utiliza as distâncias ao quadrado entre as amostras para verificar quais desses dados formam arestas entre si. Ou seja, inicialmente eleva ao quadrado a matriz de distância exemplificada pela Equação 3.4, resultando em uma matriz mostrada pela Equação 3.5. Em seguida, para cada par de amostras i e j realiza-se a comparação definida por $D^2(p_i, p_j) \leq [D^2(p_i, p_z) + D^2(p_j, p_z)]$. Se a condição for satisfeita define-se que a amostra i forma uma aresta com a amostra j. Por exemplo, para que a amostra 1 forme uma aresta com a 2 a condição definida pela Equação 3.6 deve ser satisfeita.

$$\begin{gathered} D_{12}^2 \leq [D_{11}^2 + D_{21}^2] \;\&\&\; D_{12}^2 \leq [D_{11}^2 + D_{21}^2] \;\&\& \\ D_{12}^2 \leq [D_{12}^2 + D_{22}^2] \;\&\&\; ... \;\&\&\; D_{12}^2 \leq [D_{1n}^2 + D_{2n}^2] \end{gathered} \tag{3.6}$$

Contudo, observa-se que as operações $D_{12}^2 \leq [D_{11}^2 + D_{21}^2]$ e $D_{12}^2 \leq [D_{12}^2 + D_{22}^2]$ são desnecessárias, pois as distâncias D_{11}^2 e D_{22}^2 são iguais a zero, dessa forma as duas condições sempre serão verdadeiras. Assim, tais operações foram eliminadas, reduzindo então, a implementação desse cálculo em hardware.

Já o cálculo do grafo de borda verifica quais amostras estão localizadas na superfície de separação entre as classes. Para tanto, verifica-se em cada aresta se seus dois vértices pertencem a classes distintas, caso as duas amostras que formam uma aresta sejam de classes diferentes, esses dois dados são definidos como "arestas de suporte". O algoritmo 6 descreve os passos para a detecção das amostras localizadas na borda. Observa-se nesse algoritmo que o conjunto A é o resultado do grafo de proximidade, ou seja, representa quais dados formam uma aresta entre si. Por exemplo, o elemento $A(i, j)$ é igual a 1 se as amostras i e j formam uma aresta, e 0 caso contrário. Já o conjunto Y é formado pelos rótulos de todos os dados de treinamento, tais rótulos informam se uma amostra pertence à classe 1 ou -1.

Diante desses dados de entrada, o cálculo necessário para detectar se a amostra i está localizada na superfície de separação por ser representado pela Equação 3.7, ou seja, se o vértice i forma uma aresta com alguma outra amostra j que pertence à classe

distinta de i, tais amostras são identificadas como "arestas de suporte".

$$\begin{aligned}[-Y(i) == A(i,1) * Y(1)] \;||\; [-Y(i) == A(i,2) * Y(2)] \;||\\ [-Y(i) == A(i,3) * Y(3)] \;||\; ... \;||\; [-Y(i) == A(i,n) * Y(n)]\end{aligned} \tag{3.7}$$

Apesar desse cálculo de borda ser relativamente simples, estudou-se alternativas para reduzir a Equação 3.7 em uma expressão booleana apenas. Para tanto, faz-se necessário mudar a classe -1 para classe 0, essa mudança permite detectar se a amostra i está localizada na borda através da Equação 3.8. Ou seja, como os elementos $A(i,j)$, $Y(i)$ e $Y(j)$ são números binários, o cálculo de borda pode ser realizado apenas com portas lógicas, facilitando assim a implementação em hardware.

$$\begin{aligned}[A(i,1) \;\&\&\; (Y(i) \wedge Y(1))] \;||\; [A(i,2) \;\&\&\; (Y(i) \wedge Y(2))] \;||\\ [A(i,3) \;\&\&\; (Y(i) \wedge Y(3))] \;||\; ... \;||\; [A(i,n) \;\&\&\; (Y(i) \wedge Y(n))]\end{aligned} \tag{3.8}$$

Após a redução de cálculos e melhorias nas sub-tarefas descritas acima, realizou-se o desenho de cada uma delas para um conjunto inicial de 4 amostras de treinamento e 4 características. Cada desenho foi implementado na linguagem de descrição de *hardware* (*HDL*) utilizando os blocos da ferramenta *XSG Xilinx System Generator* anexa ao *Simulink* do *Matlab*, assim como blocos do próprio *Simulink*. Essa ferramenta *XSG* permite a criação e verificação de designs de hardware para *FPGAs* da *Xilinx* de maneira rápida e com pouco esforço, fazendo o uso de elementos do *Simulink* e *Matlab*.

Depois da validação de cada desenho em HDL das sub-fases do NN-clas através da ferramenta *XSG*, realizou-se a implementação de todos os desenhos em linguagem VHDL também para um conjunto de 4 amostras e 4 características, e em seguida simulou-se cada subetapa utilizando o software *QuestaSim*.

Posteriormente à avaliação de cada subetapa do NN-clas implementada em código *VHDL* para um conjunto de 4 amostras e 4 dimensões, desenvolveu-se um *script* em *Matlab* para a geração automática de código *VHDL*, facilitando assim a implementação em hardware do algoritmo com diferentes números de amostras de treinamento e características. Inicialmente, criou-se um *script* separado para cada subetapa das duas fases do classificador, depois realizou-se a junção de todas as subetapas de cada fase em um único código, gerando dessa forma, um *script* para cada uma das duas fases do NN-clas.

O *script* desenvolvido além de gerar o código em *VHDL* de acordo com parâmetros configurados pelo usuário, realiza a chamada do software *QuestaSim* para simular o algoritmo em *VHDL* criado, posteriormente compara os resultados obtidos com o do classificador implementado em *Matlab*. Dessa maneira, o processo de validação da arquitetura fica bem eficiente e automático.

Capítulo 4

Arquitetura em hardware proposta

A arquitetura para o NN-clas foi projetada objetivando o máximo desempenho, aproveitando do paralelismo intrínseco do *FPGA*. Dessa forma, uma arquitetura síncrona de arranjo de processadores foi desenvolvida onde cada etapa é constituída de múltiplos elementos de processadores (*EPs*), relacionado a cada sub-tarefa implementada em hardware digital. Em cada subetapa foram realizadas operações lógicas e aritméticas em ponto flutuante com a precisão simples de 32bits, seguindo o padrão *IEEE-754*.

A representação em ponto flutuante foi escolhida, pois apresenta uma ampla faixa dinâmica para representar qualquer valor real restrito apenas à precisão escolhida. Os cálculos utilizando ponto fixo são mais simples contudo, como na maioria das aplicações a ordem de grandeza de cada característica é diferente das demais, é necessário um processo de normalização, que pode ser custoso para ser implementado em sistemas embarcados. Além disso, o processo de normalização deve ser executado para todas as amostras de treinamento e para cada novo dado recebido para ser rotulado. Então, por mais esse motivo optou-se pelo uso de ponto flutuante.

Cada *EP* foi projetado baseado em uma topologia linear mediante a implementação da arquitetura pipeline usando uma descrição em hardware parametrizável, no intuito de atender mudanças referentes aos valores das amostras a serem processadas. O funcionamento de cada *EP* está associado à sua própria unidade de controle (*UC*). Assim, permitindo o envio dos diferentes sinais de controle internos projetados no *EP* e, direcionando os dados para os diferentes elementos de memória local distribuídos.

A figura 4.1 mostra o fluxo dos dados nas duas etapas da arquitetura proposta, onde cada elemento representa um EP. Os elementos de leitura e armazenamento de dados acessam os elementos de memória para processar tais dados e redirecioná-los aos *EPs* subsequentes.

As subseções a seguir descreverão de maneira detalhada o desenho arquitetural projetado de cada um dos blocos que compõem as duas fases do NN-clas. Também

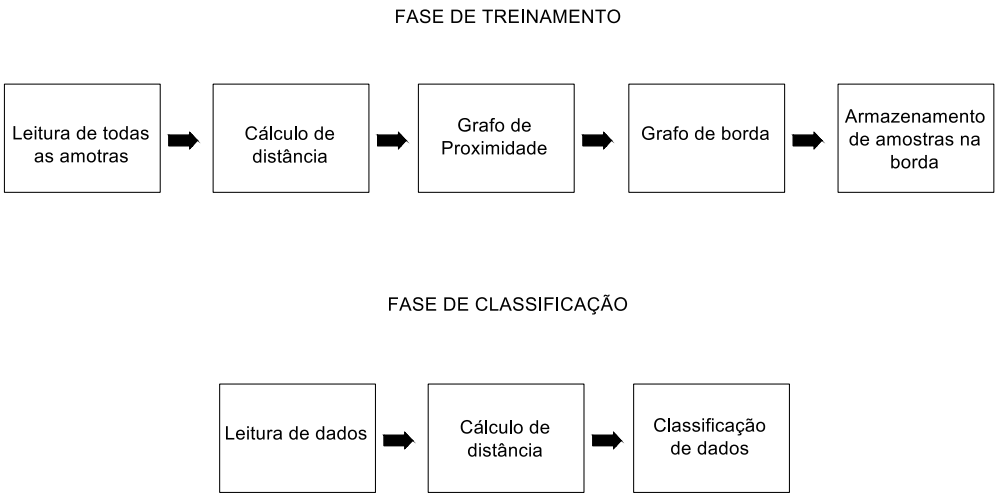


Figura 4.1: Fluxo de dados das duas etapas da arquitetura proposta

será descrito o uso da ferramenta criada para a geração automática de código *VHDL* no *Matlab*, o que facilita a implementação em hardware do algoritmo com diferentes parâmetros.

4.1 Fase de treinamento

4.1.1 Cálculo de distância

Como foi abordado na metodologia, foram desenvolvidos dois desenhos para o cálculo de distância, um utilizando a métrica de *Manhattan* e outro a técnica Euclidiana quadrática. Para tanto, considerou-se apenas o cálculo dos elementos à direita da diagonal da matriz expressa pela Equação 3.4. As figuras 4.2 e 4.3 exemplificam a arquitetura utilizando *Manhattan* e Euclidiana quadrática, respectivamente, para um conjunto de amostras de treinamento que possua 4 elementos e 4 dimensões. Como se pode observar, os modelos são praticamente idênticos, a única diferença está na adição do bloco de multiplicação em cada estágio.

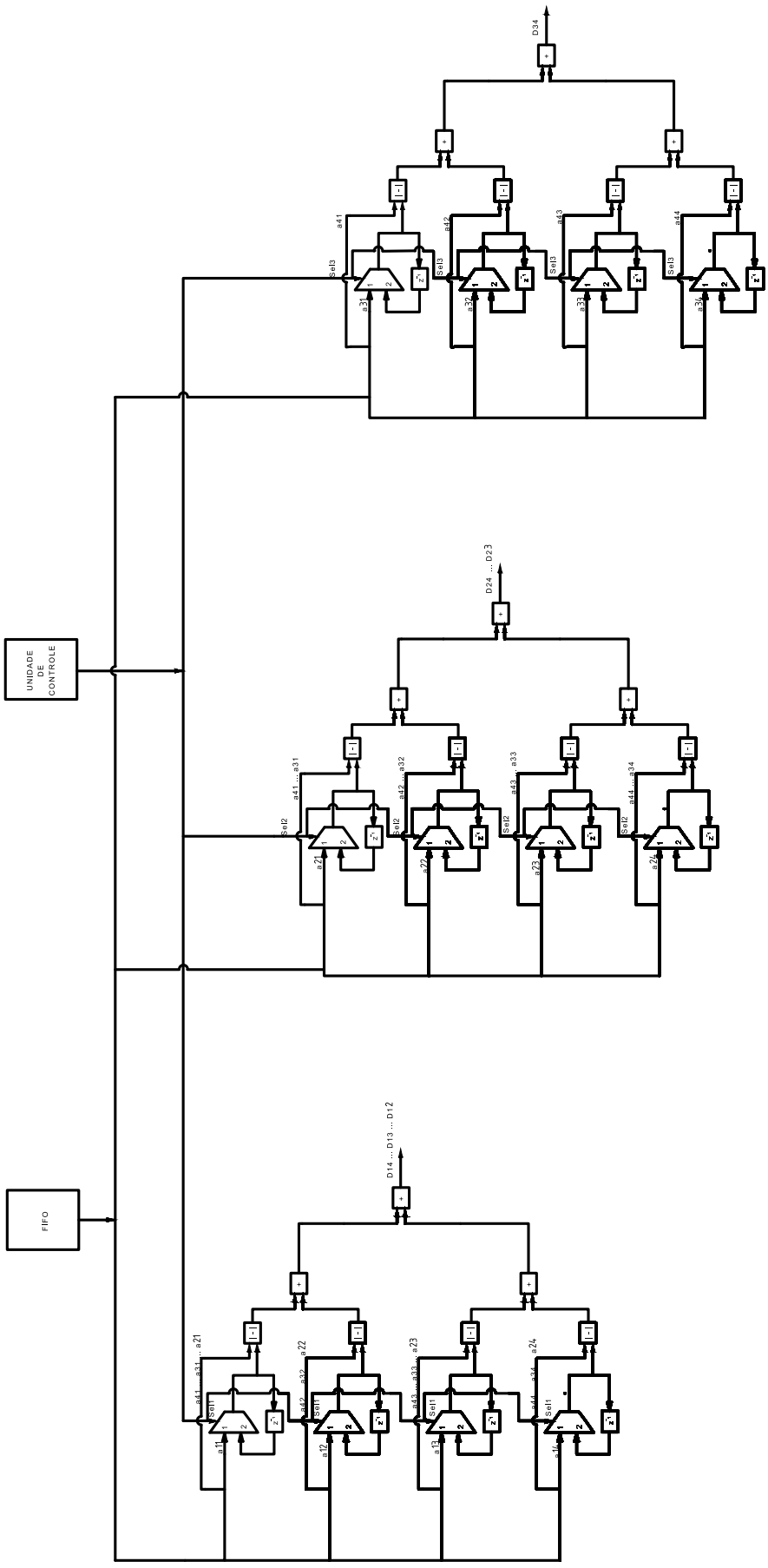


Figura 4.2: Arquitetura do cálculo de distância pela métrica de *Manhattan*

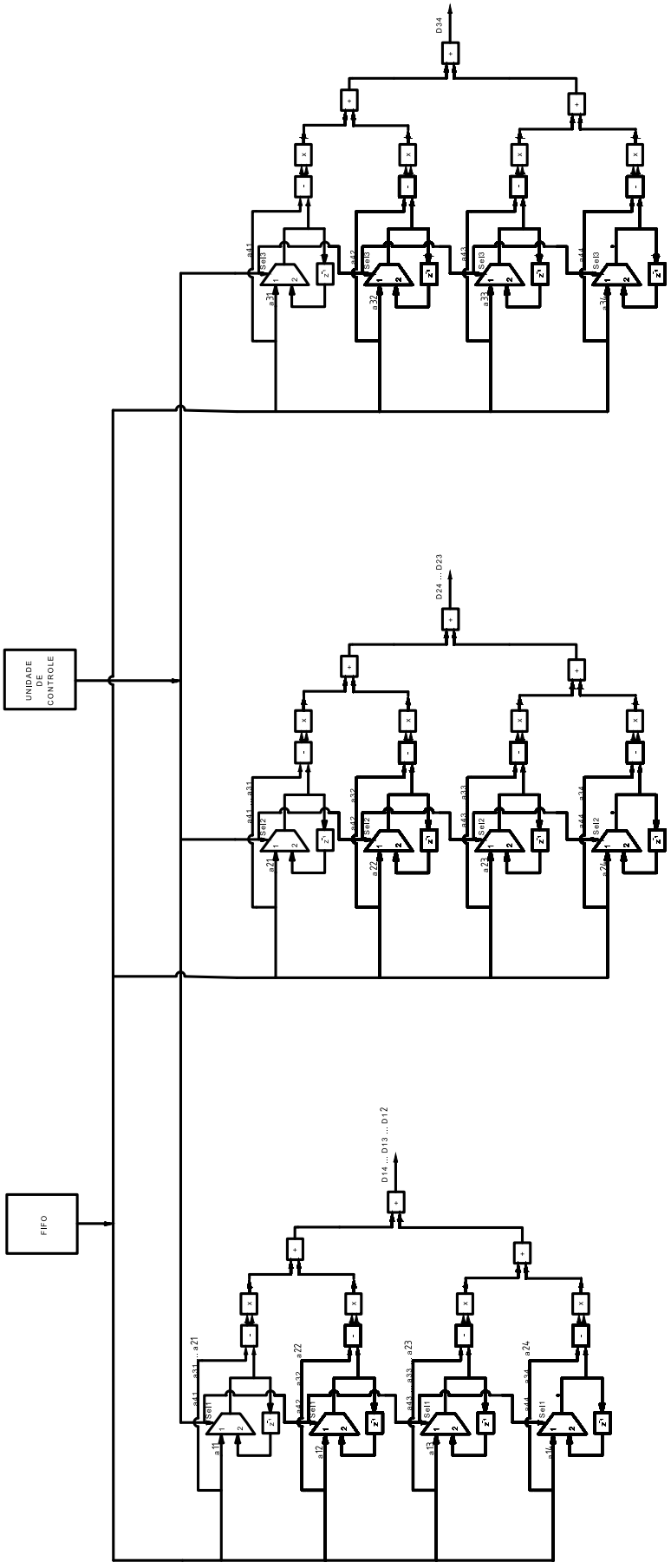


Figura 4.3: Arquitetura do cálculo de distância pela técnica Euclidiana quadrática

O bloco nomeado de *fifo* representa um elemento de memória local que armazena todas as amostras de treinamento, e a cada ciclo de *clock* fornece uma amostra com suas respectivas características. Como optou-se por utilizar uma representação em ponto flutuante simples de 32bits o tamanho desse elemento de memória é $32xMxN$, onde N representa o número de amostras de treinamento e M a dimensão.

Como abordado anteriormente, as figuras 4.2 e 4.3 exemplificam o cálculo de distância de um conjunto de amostras de 4 elementos, dessa maneira, o desenho arquitetural apresenta três estágios $(4-1)$, sendo que cada estágio irá realizar o cálculo de distância entre uma amostra e as demais. Ou seja, o primeiro estágio irá calcular a distância entre a amostra a_1 e a_2, a_1 e a_3, e a_1 e a_4, o segundo estágio por sua vez realizará o cálculo de distância entre a amostra a_2 e a_3, e a_2 e a_4, já o terceiro estágio calculará a distância entre a amostra a_3 e a_4.

Para tanto, faz-se necessário o uso de multiplexadores e atrasadores para manter o valor de uma determinada amostra, e assim realizar o cálculo de distância dessa amostra com as demais. Por exemplo, no primeiro ciclo de *clock* é fornecida a amostra a_1 pelo bloco *fifo*, os blocos multiplexadores e atrasadores irão manter o valor dessa amostra para que nos demais ciclos de *clock* ao receber as amostras seguintes, possa-se realizar o cálculo de distância entre a amostra a_1 e as demais. Dessa maneira, cada bloco multiplexador/atrasador possui uma entrada de controle chamada de seletor para indicar o momento que o valor contido na entrada 1 será mantido através do atrasador. Enquanto a entrada seletora permanecer em 1 o valor mantido através do atrasador será passado para a saída do multiplexador, e quando a entrada seletora estiver em 0 o valor contido na entrada 1 do multiplexador será passado para a saída desse multiplexador.

O número de multiplexadores e atrasadores presentes na arquitetura do cálculo de distância irá depender do número de dimensões m e do número de amostras de treinamento n. Como se pode observar nas figuras 4.2 e 4.3 cada estágio possui um número m de multiplexadores e atrasadores para manter os valores das características da amostra a_i, onde i representa o número do estágio. Dessa maneira a quantidade desses blocos pode ser expressa de forma genérica de acordo com a Equação 4.1.

$$num_{multiplexadores\_atrasadores} = m * (n-1) \tag{4.1}$$

No modelo que emprega a métrica de *Manhattan*, são utilizados blocos somadores e subtratores de valor absoluto. Cada subtrator de valor absoluto realiza o cálculo de subtração em módulo de uma característica de duas amostras. Por exemplo, o primeiro bloco subtrator de valor absoluto do primeiro estágio da figura 4.2 irá realizar a subtração em módulo da dimensão 1 das amostras a_1 e a_n, ou seja $|a_{11} - a_{n1}|$. A quantidade desse bloco na arquitetura de cálculo de distância é expressa pela Equação

4.2. O bloco somador por sua vez, irá realizar a operação de soma par a par do resultado da subtração em módulo de cada uma das dimensões de duas amostras, finalizando assim o cálculo de distância entre duas amostras. O número de somadores nessa arquitetura é expresso pela Equação 4.3.

$$num_{subtratores} = m * (n - 1) \tag{4.2}$$

$$num_{somadores} = (n - 1) * (\sum_{i=1}^{\lceil log_2 m \rceil} \left\lceil (\frac{m}{2^i}) \right\rceil) \tag{4.3}$$

Já o modelo que utiliza a técnica Euclidiana possui blocos subtratores que realizam a mesma função descrita no modelo de *Manhattan*, blocos multiplicadores para elevar ao quadrado o resultado de cada subtração e blocos somadores idênticos aos do modelo de *Manhattan*. A quantidade de blocos subtratores e somadores também pode ser expressa pela Equação 4.2 e 4.3, respectivamente, e o número de blocos multiplicadores é definido pela Equação 4.4.

$$num_{multiplicadores} = m * (n - 1) \tag{4.4}$$

Os blocos somadores, multiplicadores e subtratores foram implementados em *pipeline*, utilizando a representação em ponto flutuante simples. Os blocos somadores e subtratores implementam o algoritmo similar ao descrito pela figura 2.14 e o multiplicador ao algoritmo exemplificado pela figura 2.15. A diferença está no fato de que o algoritmo implementado não possui uma fase de arredondamento, o valor final é truncado apenas. Tais blocos foram implementados segundo a arquitetura em fluxo de dados proposta por García (2014). Essa arquitetura permite a saída de um novo resultado a cada ciclo de *clock* após uma latência de três ciclos de *clock* para o bloco somador e subtrator e um ciclo de *clock* para o bloco multiplicador.

A latência do cálculo de distância medida em ciclos de *clocks* utilizando a métrica de *Manhattan* é exemplificada pela Equação 4.5. Já a Equação 4.6 expressa a latência do cálculo de distância empregando a técnica Euclidiana quadrática.

$$Temp(clks)\_Manhattan = 3 + \lceil log_2 m \rceil * 3 + 1 \tag{4.5}$$

$$Temp(clks)\_EuclidianaQ = 3 + \lceil log_2 m \rceil * 3 + 2 \tag{4.6}$$

Para controlar as entradas das amostras de dados, a saída das distâncias entre as amostras e os seletores, desenvolveu-se uma unidade de controle. Essa unidade de controle e os demais blocos foram implementados em *VHDL*. Para facilitar a parame-

trização da arquitetura em termos do número de amostras de treinamento e dimensão criou-se uma ferramenta no *Matlab* para geração automática de código *VHDL*, onde o usuário pode definir esses parâmetros e escolher qual técnica deseja utilizar para realizar o cálculo de distância.

4.1.2 Cálculo do grafo de proximidade

O cálculo do grafo de proximidade foi projetado separadamente, contudo considerou-se que a entrada dos dados nesse *EP* referentes à distância entre as amostras seguisse o mesmo fluxo e estrutura dos dados de saída do *EP* que realiza o cálculo de distância. Uma vez que, o objetivo final é realizar a junção de todos os *EPs* da fase de treinamento.

Como abordado anteriormente, o resultado do cálculo de distância entre as amostras é fornecido a cada ciclo de *clock* após $3+\lceil log_2 m \rceil * 3+1$ ciclos de *clock* quando utilizado a métrica de *Manhattan* e $3+\lceil log_2 m \rceil * 3+2$ aplicando a técnica Euclidiana quadrática. Dessa maneira, como observado nas figuras 4.2 e 4.3, no primeiro ciclo de *clock* após a latência é fornecida a distância D_{12} entre as amostras a_1 e a_2, no segundo são fornecidas as distâncias D_{13} e D_{23} e no terceiro as distâncias D_{14}, D_{24} e D_{34}.

Desenvolveu-se então o desenho arquitetural desse *EP* de forma que pudesse ser expandido para qualquer número de amostras e dimensão, assim como foi feito no projeto do *EP* de cálculo de distância. A figura 4.5 exemplifica a arquitetura do grafo de proximidade para um conjunto de amostras que possua 4 elementos e 4 dimensões. Os cálculos realizados para esse conjunto de amostras são feitos de forma paralela, aproveitando ao máximo as características do *FPGA*, pode-se definir esses cálculos através dos seguintes passos:

1. Verificar se a amostra a_1 forma uma aresta com a_2 através de $D_{12}^2 \leq [D_{13}^2 + D_{23}^2]$ && $D_{12}^2 \leq [D_{14}^2 + D_{24}^2]$;

2. Averiguar se a_1 forma aresta com a_3 se $D_{13}^2 \leq [D_{12}^2 + D_{23}^2]$ && $D_{13}^2 \leq [D_{14}^2 + D_{34}^2]$ for verdadeira;

3. Conferir se a_1 forma uma aresta com a_4 através de $D_{14}^2 \leq [D_{12}^2 + D_{24}^2]$ && $D_{14}^2 \leq [D_{13}^2 + D_{34}^2]$;

4. Verificar se a amostra a_2 forma uma aresta com a_3 através de $D_{23}^2 \leq [D_{12}^2 + D_{13}^2]$ && $D_{23}^2 \leq [D_{24}^2 + D_{34}^2]$;

5. Averiguar se a_2 forma aresta com a_4 se $D_{24}^2 \leq [D_{12}^2 + D_{14}^2]$ && $D_{24}^2 \leq [D_{23}^2 + D_{34}^2]$ for verdadeira;

6. Conferir se a_3 forma uma aresta com a_4 através de $D_{34}^2 \leq [D_{13}^2 + D_{14}^2]$ && $D_{34}^2 \leq [D_{23}^2 + D_{24}^2]$;

Observa-se na figura 4.4 que os primeiros blocos são multiplicadores, uma vez que o grafo de proximidade realiza a comparação entre as distâncias ao quadrado. Cada um desses blocos possui como entrada a saída de um dos estágios do EP de cálculo de distância. Dessa maneira a quantidade de blocos multiplicadores pode ser generalizada através da Equação 4.7.

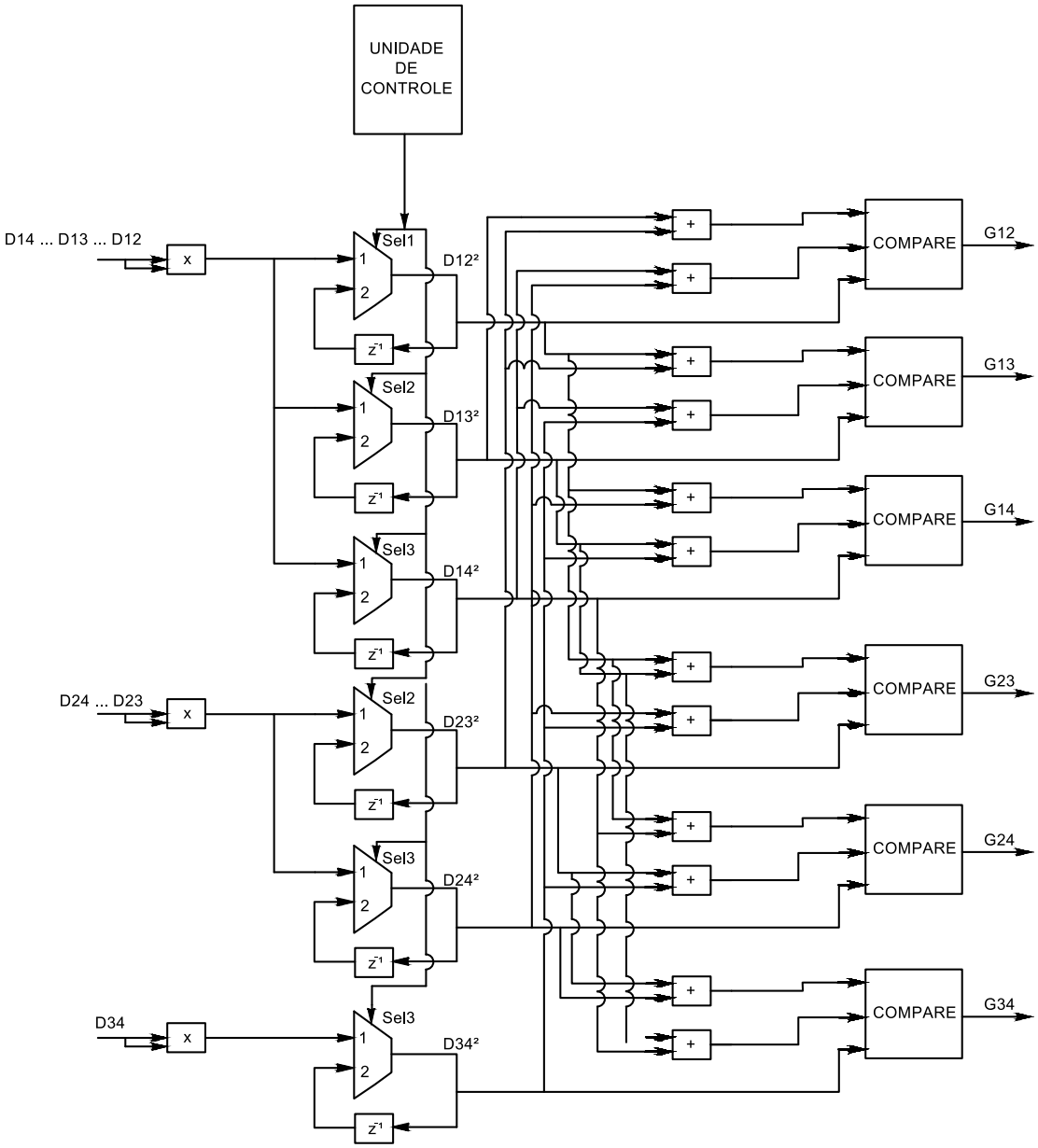


Figura 4.4: Arquitetura do grafo de proximidade

$$num_{multiplicadores\_grafo} = n - 1 \tag{4.7}$$

Como abordado anteriormente, a figura 4.4 exemplifica uma arquitetura para um conjunto de amostras de treinamento com 4 elementos, ou seja $n = 4$, assim o número de blocos multiplicadores é 3. O primeiro bloco multiplicador fornece no primeiro ciclo de *clock* após a latência a distância D^2_{12}, no segundo ciclo D^2_{13} e no terceiro D^2_{14}. O segundo bloco por sua vez, só começa a gerar um resultado válido no segundo ciclo de *clock*, que é a distância D^2_{23}, e no terceiro ciclo D^2_{24}. O terceiro bloco produz um resultado válido apenas no terceiro ciclo de *clock*, sendo ele a distância D^2_{34}.

Observa-se que algumas distâncias ao quadrado das amostras são fornecidas em diferentes ciclos de *clock*, contudo caso o valor não seja mantido, ele é perdido após o próximo ciclo de *clock*. Dessa maneira, faz-se necessário o uso de blocos multiplexadores e atrasadores para manter o valor dessas distâncias quadráticas para que se possa realizar posteriormente as operações de soma e comparação, que iram definir quais amostras formam arestas entre si. Cada bloco multiplexador/atrasador é responsável por armazenar uma distância quadrática, sendo assim a quantidade desses blocos é igual ao número de elementos à direita da diagonal da matriz definida pela Equação 3.4, ou seja, $num_{multiplexadores\_Graf} = \frac{n*(n-1)}{2}$. Os sinais de controle, denominados de seletores, de cada um desses blocos irão definir o momento em que o dado será armazenado e o tempo que ficará retido.

Os blocos posteriores são os somadores, os quais realizarão a soma par a par entre as distâncias ao quadrado, para que o resultado possa ser comparado como exemplifica a Equação 2.1. A quantidade dos blocos somadores é definida pela Equação 4.8. Por último, tem-se os blocos comparadores que realizam a comparação entre os valores de entrada e, em seguida realiza uma operação lógica *AND* entre eles. A saída de cada um desses blocos comparadores é um valor binário, 0 caso as amostras não formem uma aresta e 1 caso contrário. Os valores de saída desses blocos são fornecidos em um mesmo instante de tempo.

$$num_{somadores\_grafo} = \frac{n * (n-1) * (n-2)}{2} \tag{4.8}$$

Todos os blocos que compõem esse *EP*, exceto o comparador, são idênticos aos blocos utilizados no *EP* do cálculo de distância. Sendo assim, a latência desse *EP* é detalha pela Equação 4.9.

$$Temp(clks)\_GrafP = 1 + 3 + (n-1) \tag{4.9}$$

Assim como foi feito no *EP* do cálculo de distância, criou-se uma ferramenta no

Matlab para geração automática de código *VHDL*, permitindo ao usuário configurar o número de amostras de treinamento e dimensão.

4.1.3 Cálculo do grafo de borda

O grafo de borda especifica as amostras localizadas na margem de separação entre as classes. Essa definição é feita caso uma amostra forme aresta com outra de classe distinta. Percebe-se dessa forma, que para realizar o cálculo do grafo de borda, são necessários os rótulos de cada amostra e os resultados do cálculo do grafo de proximidade.

Esse *EP* também foi desenvolvido separadamente, considerando como entrada de dados o mesmo formato e estrutura das saídas do *EP* do grafo de proximidade. Conforme foi dito anteriormente, as saídas do grafo de proximidade são fornecidas em um mesmo instante de tempo. Sendo assim, o projeto foi feito conforme mostra a figura 4.5, que exemplifica a arquitetura para um conjunto de 4 amostras de treinamento de 4 dimensões.

Observa-se na figura 4.5 um bloco *fifo label* que representa um elemento de memória local que armazena os rótulos das amostras de treinamento. Para facilitar o desenvolvimento dessa arquitetura os rótulos foram armazenados como números binários, 0 para a classe -1 e 1 para a base 1. Sendo assim, a profundidade desse elemento é N bits. Todos os rótulos são fornecidos em um mesmo instante de tempo, ou seja, assim que os dados resultantes do grafo de proximidade estiverem presentes na entrada desse *EP* os rótulos são fornecidos pelo elemento de memória *fifo label* para que o grafo de borda possa ser calculado e seu resultado produzido em um mesmo instante de tempo para todas as amostras de treinamento.

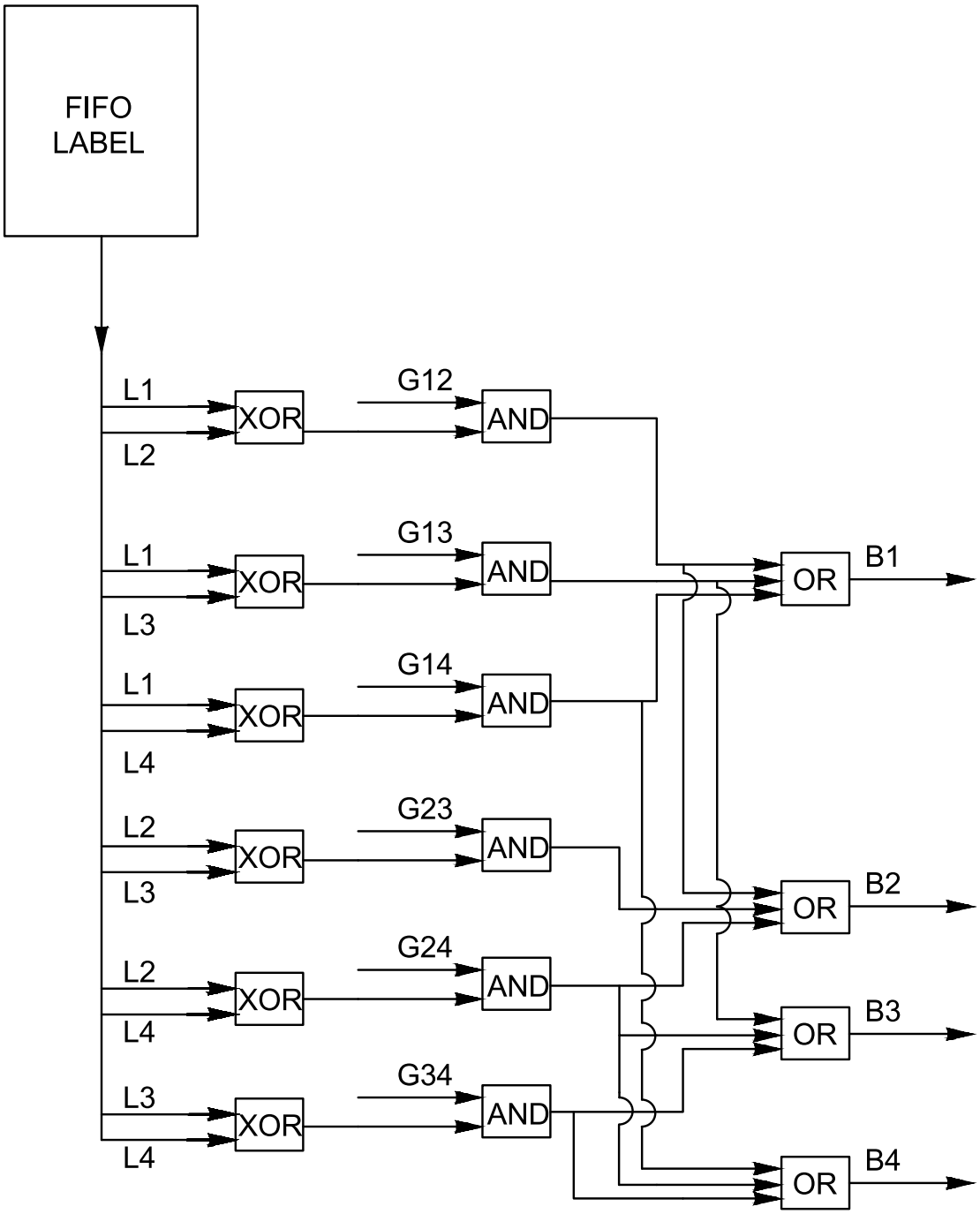


Figura 4.5: Arquitetura do grafo de borda

Considerando o conjunto de 4 amostras de treinamento, o grafo de borda também pode ser calculado de forma paralela, de acordo com as funcionalidades abaixo:

1. Verificar se a amostra $a1$ está na borda, caso $[A(1,2)$ && $(Y(2) \wedge Y(1))]$ || $[A(1,3)$ && $(Y(3) \wedge Y(1)]$ || $[A(1,4)$ && $(Y(4) \wedge Y(1)]$ seja verdadeira, onde

$Y(i)$ representa o rótulo da amostra a_i e $A(i,j)$ expressa se as amostras a_i e a_j formam uma aresta entre si;

2. Averiguar se a_2 está na borda através de $[A(1,2)\ \&\&\ (Y(2)\ \wedge\ Y(1))]||\ [A(2,3)\ \&\&\ (Y(3)\ \wedge\ Y(2)]\ ||\ [A(2,4)\ \&\&\ (Y(4)\ \wedge\ Y(2)]$;

3. Conferir se a_3 está na borda, caso $[A(2,3)\ \&\&\ (Y(2)\ \wedge Y(3)]\ ||\ [A(1,3)\ \&\&\ (Y(3)\ \wedge Y(1)]\ ||\ [A(3,4)\ \&\&\ (Y(4)\ \wedge\ Y(3)]$ seja verdadeira;

4. Verificar se a amostra a_4 através de $[A(1,4)\ \&\&\ (Y(4) \wedge Y(1)]\ ||\ [A(2,4)\ \&\&\ (Y(2)\ \wedge Y(4)]\ ||\ [A(3,4)\ \&\&\ (Y(4)\ \wedge\ Y(3)]$.

O *EP* do grafo de borda é formado apenas por blocos lógicos: *and*, *or* e *xor*. A quantidade de blocos *and* e *xor* pode ser expressa pela Equação 4.10, já o número de blocos *or* é igual a n e a quantidade de entradas desse bloco é igual a $n-1$. Como esse *EP* possui apenas operações lógicas a sua latência é praticamente zero.

$$num_{and\_xor} = \frac{n * (n-1)}{2} \tag{4.10}$$

O código *VHDL* desse *EP* também foi gerado de forma automática através da ferramenta criada no *Matlab*. Assim o usuário pode configurar o número de amostras de treinamento que possui sua aplicação e gerar o *EP* do grafo de borda.

4.1.4 Armazenamento de dados situados na borda

A última etapa da fase de treinamento consiste em armazenar as amostras que estão situadas na margem de separação em um elemento de memória. Para tanto, são necessários os resultados do *EP* do grafo de borda, os rótulos das amostras e os valores de todas as dimensões das amostras de treinamento.

Assim como realizado nas etapas anteriores, esse bloco também foi projetado de forma separada considerando como entrada de dados o mesmo formato e estrutura das saídas do *EP* do grafo de borda. A figura 4.6 exemplifica o desenho arquitetural desse *EP* para um conjunto de amostras de 4 elementos de 4 dimensões.

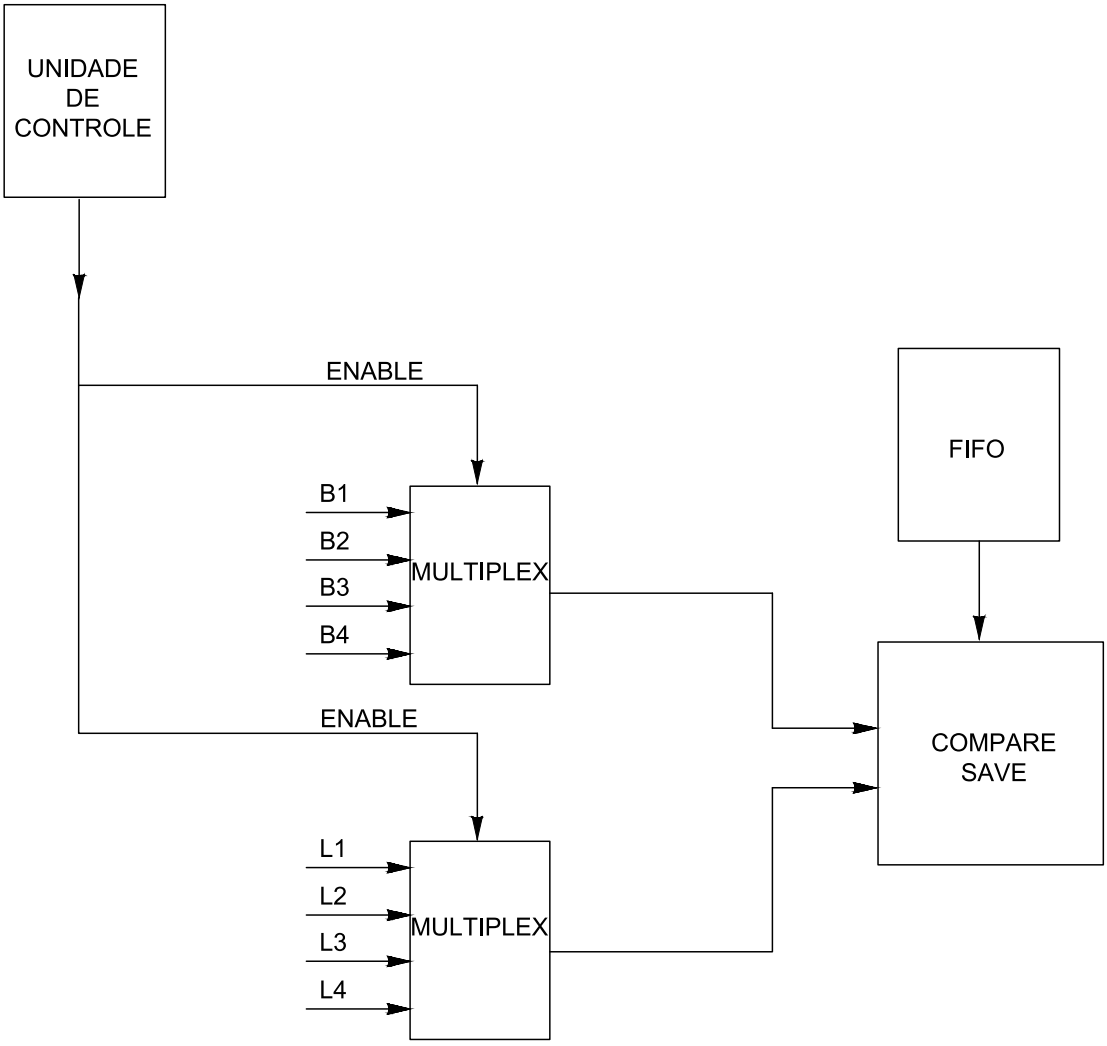


Figura 4.6: Arquitetura do Armazenamento de dados situados na borda

Percebe-se que esse *EP* utiliza o mesmo elemento de memória local *fifo* que o *EP* do cálculo de distância. Como os resultados do grafo de borda e os rótulos das amostras são fornecidos em um mesmo instante de tempo, e os valores de uma amostra são lidos a cada ciclo de *clock*, é necessário projetar blocos que mantenham os resultados do grafo de borda e os rótulos das amostras. Dessa maneira, desenvolveu-se dois blocos distintos que armazenam os valores de suas entradas assim que o sinal de controle *enable* estiver ativado. E a cada ciclo de *clock* é fornecido um dos valores armazenados em sua saída de forma crescente, ou seja, inicialmente, é fornecido por um desses o rótulo da amostra 1, e no outro bloco a informação se amostra 1 está situada na borda, no próximo ciclo de *clock* as informações da amostra 2 e assim por diante. O número de entrada de cada bloco é igual ao número de elementos do conjunto de treinamento n. O bloco mais acima da figura 4.6 armazena os dados que informam se a amostra

está situada na borda, e o bloco mais abaixo armazena os rótulos de cada amostra, sendo que ambos dados armazenados são valores binários.

O último bloco desse *EP* recebe uma amostra por vez, o rótulo dessa amostra e o dado que informa se a amostra está na margem de separação, caso essa amostra esteja situada na borda, os valores da amostra e de seu rótulo são armazenados em um elemento de memória.

Todos os *EPs* da fase de treinamento foram testados e validados de forma separada para distintos números de amostras e dimensão. Em seguida, todos esses elementos foram ajuntados e gerou-se uma arquitetura única para o treinamento do NN-clas. Da mesma maneira que foi feito na geração do código *VHDL* para cada *EP*, utilizou-se a ferramenta do *Matlab* para gerar o código em *VHDL* da arquitetura completa.

4.2 Fase de classificação

4.2.1 Cálculo de distância

O *EP* do cálculo de distância da fase de classificação é bem semelhante ao projetado para a fase de treinamento. Diferentemente do treinamento que é necessário o cálculo de distância de todas as amostras entre si, na classificação calcula-se apenas a distância entre um novo dado a ser rotulado e as amostras de treinamento localizadas na margem de separação. O desenho arquitetural desse *EP* para uma base de dados de 4 dimensões utilizando a métrica de *Manhattan* é demonstrado pela figura 4.7 e pela figura 4.8 aplicando a técnica da Euclidiana Quadrática.

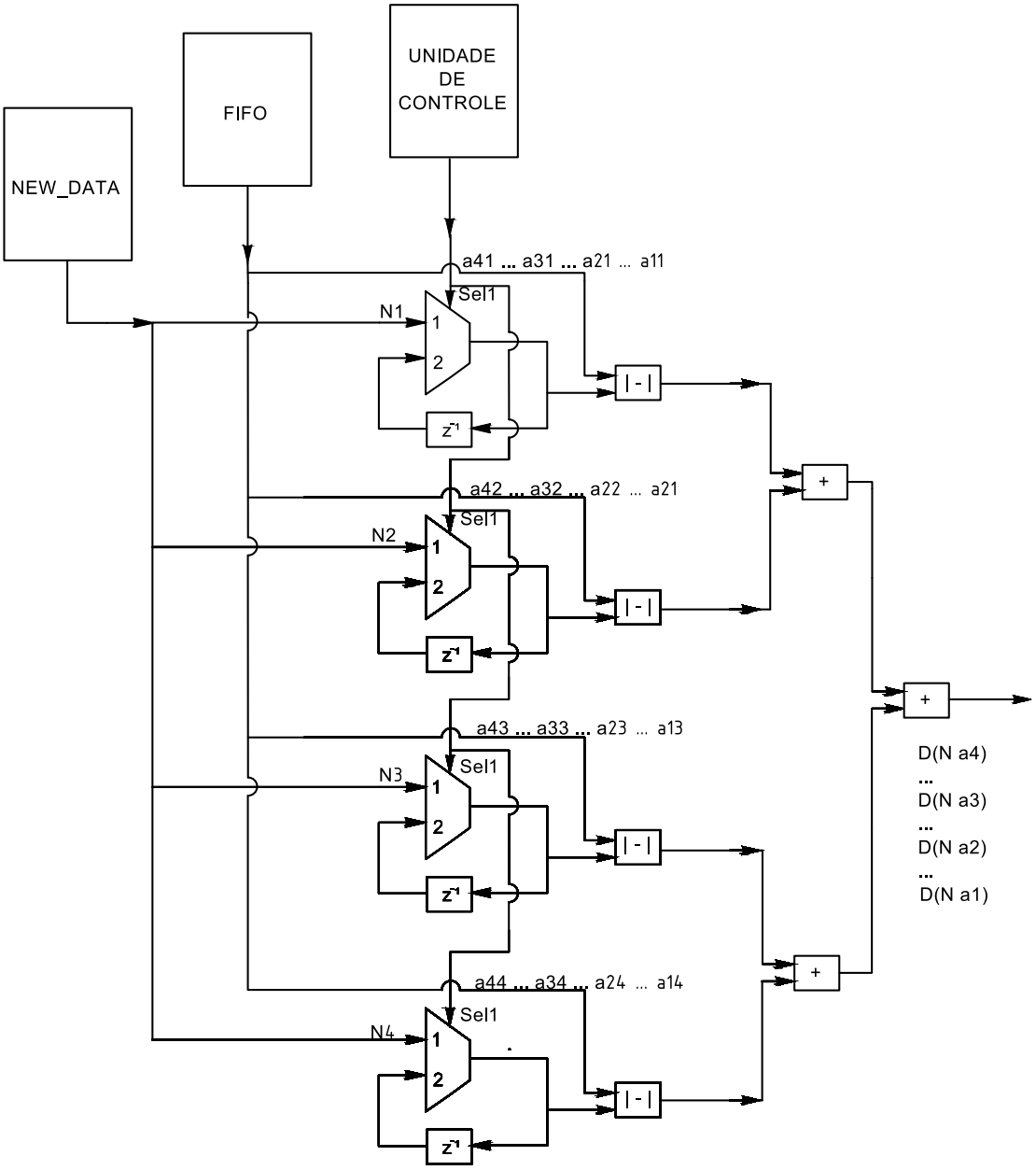


Figura 4.7: Arquitetura do cálculo de distância da fase de classificação pela métrica de *Manhattan*

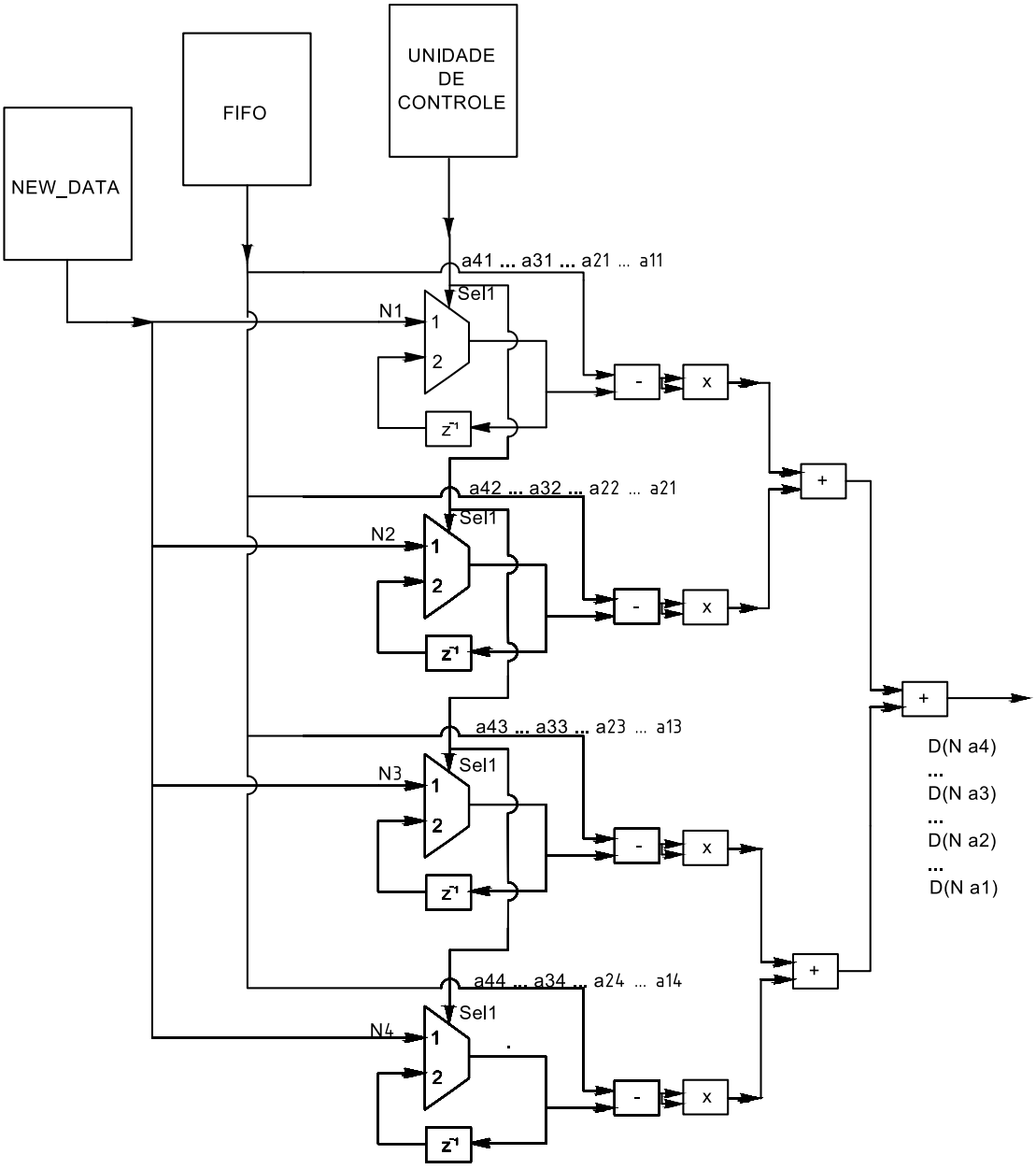


Figura 4.8: Arquitetura do cálculo de distância da fase de classificação pela técnica Euclidiana quadrática

Durante o desenvolvimento dessa arquitetura considerou-se que seria fornecido para o classificador um novo dado por vez, o que ocorre em grande parte das aplicações. As amostras situadas na borda seriam lidas do mesmo local de memória, onde foram armazenadas na fase de treinamento, e fornecidas uma a uma a cada ciclo de *clock* através de um elemento de memória *fifo* similar ao utilizado no *EP* do cálculo de distância da etapa de treinamento.

Dessa maneira é necessário armazenar os valores do dado a ser classificado, para

que a distância entre ele e as demais amostras possa ser calculada, caso contrário os valores seriam perdidos no próximo ciclo de *clock* após o valor desse novo dado ser fornecido. Para tanto, utilizou-se os blocos multiplexadores e atrasadores, já citados anteriormente. A quantidade desses blocos em ambas arquiteturas desse cálculo de distância depende apenas do número de dimensão, sendo definida através da Equação 4.11.

$$num_{multiplexadores\_atrasadores\_classificacao} = m \tag{4.11}$$

Como pode ser visto nas figuras 4.7 e 4.8, esse *EP* também é formado pelos blocos somadores, subtratores e multiplicadores no caso da arquitetura que utiliza a técnica Euclidiana quadrática. A quantidade dos blocos somadores é defina pela Equação 4.12, dos subtratores pela Equação 4.13 e dos multiplicadores pela Equação 4.14. Todos esses blocos são idênticos aos utilizados no *EP* do cálculo de distância da fase de treinamento, dessa maneira os resultados das distâncias calculadas são produzidos um a um a cada ciclo de *clock*. O primeiro resultado é fornecido após $3 + \lceil log_2 m \rceil * 3$ ciclos de *clock* quando utilizado a métrica de *Manhattan* e $3 + \lceil log_2 m \rceil * 3 + 1$ ciclos de *clock* pela técnica Euclidiana quadrática.

$$num_{somadores\_classificacao} = (\sum_{i=1}^{\lceil log_2 m \rceil} \left\lceil (\frac{m}{2^i}) \right\rceil) \tag{4.12}$$

$$num_{subtratores\_classificacao} = m \tag{4.13}$$

$$num_{multiplicadores\_classificacao} = m \tag{4.14}$$

Utilizou-se também a mesma ferramenta criada no *Matlab* para geração automática do código em *VHDL* da arquitetura desse *EP* de cálculo de distância. Assim o usuário define essa arquitetura através do número de dimensões de sua aplicação. Os sinais de controle dos blocos e os acessos aos elementos de memória são todos manipulados por uma unidade de controle.

4.2.2 Classificação de dados

A última subetapa da fase de classificação atribui ao novo dado o mesmo rótulo da amostra situada na borda e que está mais próxima desse dado. Como o *EP* anterior calcula as distâncias entre esse novo dado e as amostra que estão na margem de separação, o *EP* de classificação do dado fica responsável por verificar qual distância recebida é menor e entregar como resultado o rótulo da amostra mais próxima.

Esse *EP* necessita então, receber também como entrada os rótulos das amostras situadas na borda. Para tanto é necessário acessar o local de memória onde estão armazenados os rótulos dessas amostras e em seguida utilizar um elemento de memória para fornecer um rótulo a cada ciclo de *clock* a partir do instante que começasse a receber o primeiro valor de distância. A figura 4.9 exemplifica esse *EP*.

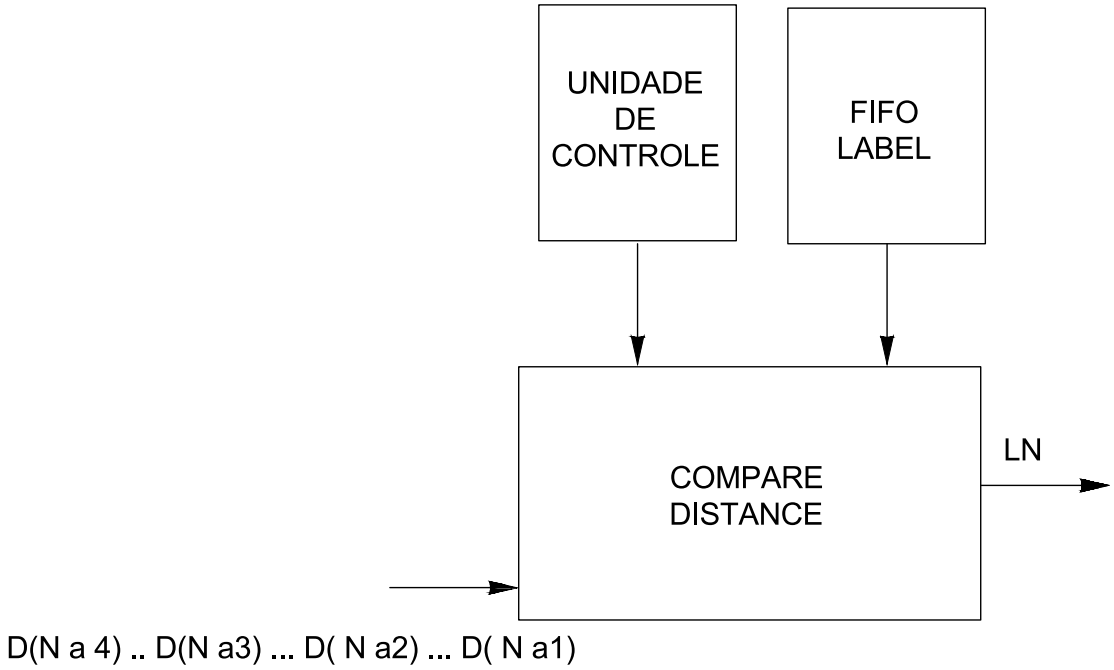


Figura 4.9: Arquitetura da classificação de dados

No momento em que chega o primeiro valor de distância na entrada desse *EP*, ele o mantém armazenado juntamente com o rótulo recebido, e nos demais ciclos de *clock* compara o valor de distância armazenado com o atual recebido, caso a nova distância recebida seja menor que a armazenada, seu valor é armazenado, substituindo o anterior. Após n ciclos de *clock*, onde n é a quantidade de amostras localizadas na borda, o resultado final é fornecido, sendo ele o mesmo valor do último rótulo armazenado.

Assim como foi feito na fase de treinamento, os dois *EP* da etapa de classificação foram testados e validados de forma separada para distintos valores de dimensão e amostras situadas na borda. Em seguida, os *EPs* foram ajuntados, gerando assim uma arquitetura única para a fase de classificação do NN-clas.

Capítulo 5

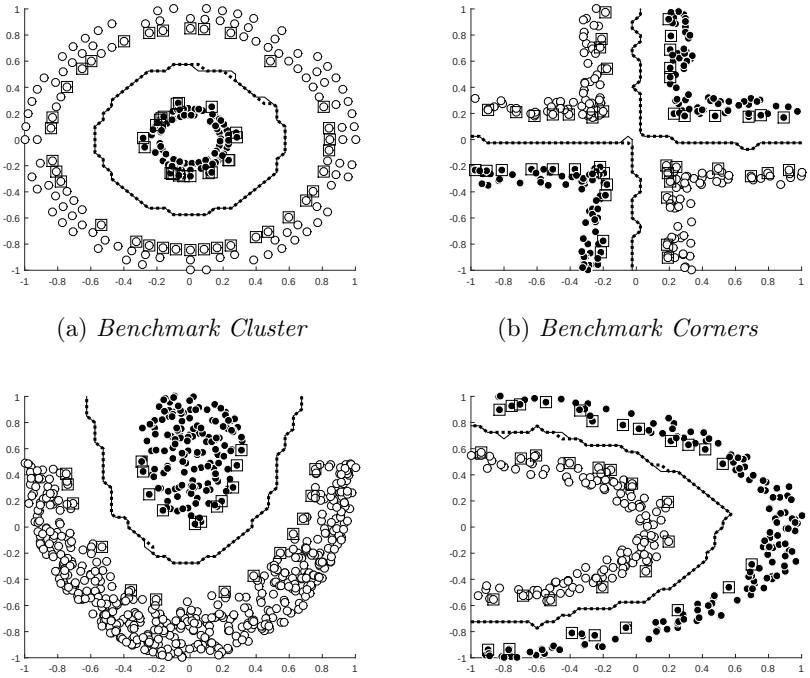
Experimentos e Resultados Computacionais

5.1 Testes com o NN-clas e o CHIP-clas reduzido

Os primeiros experimentos foram feitos no software *Matlab* com o intuito de se validar o NN-clas aqui proposto e avaliar o desempenho da técnica por hiperplanos mais próximos. Inicialmente os testes foram realizados em dados sintéticos de *benchmarks* e no momento posterior com dados reais. Os testes realizados em dados sintéticos serviram para que se fosse feito uma comparação entre a superfície de decisão obtida pela metodologia proposta e pelo CHIP-clas. A Figura 5.1 mostra as superfícies de decisão sobre as bases sintéticas. Os contornos exibidos pelas linhas tracejadas representam as superfícies de separação geradas pela nova abordagem. Já os contornos representados pelas linhas contínuas, as superfícies geradas pelo CHIP-clas.

Como se pode observar na Figura 5.1, alguns dados ruidosos foram eliminados para se garantir que os vértices das arestas de suporte estejam localizados na fronteira de separação entre as classes. Na figura, os pontos destacados com quadrados representam os tais vértices das arestas de suporte. Percebe-se portanto, com o experimento que as superfícies de separação dos *benchmarks* geradas pelos dois métodos são similares, em alguns casos é quase imperceptível a diferença, apesar de ter havido uma redução do custo computacional no novo método. É importante ressaltar também que, embora o método proposto não se baseie em um princípio de margem larga, ele tende a gerar superfícies equidistantes das classes, inclinando-se assim para a maximização da margem.

Os demais experimentos foram realizados em 13 bases de dados reais retiradas do repositório (UCI - *Machine Learning Repository*) (Lichman, 2013), e também em 2 problemas de expressão gênica: "Golub" (Golub et al., 1999) e "BcrHess" (Hess et al.,



(a) *Benchmark Cluster* (b) *Benchmark Corners*

(c) *Benchmark Full Moons* (d) *Benchmark Half Kernel*

Figura 5.1: Superfícies de decisão geradas pelo método NN-clas e pelo CHIP-clas

2006). Primeiramente, realizou-se um pré-processamento, removendo-se as amostras que continham atributos faltantes e normalizando-se os dados entre $\{-1, 1\}$. Utilizou-se uma validação cruzada *10-folds*, objetivando-se a garantia da relevância estatística. Escolheu-se, como medida de desempenho a AUC (área abaixo da curva ROC) por serem desbalanceadas algumas das bases que se pretende testar.

Os métodos comparados foram: o kNN com $k = 1$ (NN), o método aqui proposto (NN-clas), o CHIP-clas e o CHIP-Clas reduzido (descrito no capítulo de fundamentação teórica na Seção 2.2.2), nomeado nos experimentos de RCHIP-clas. Optou-se por realizar a comparação com kNN pela semelhança do NN-clas com tal método. Entretanto é importante ressaltar que o kNN calcula a distância entre cada amostra de teste x com todo o conjunto de treinamento, sendo necessário assim o armazenamento de todas as amostras de treinamento ao invés de guardar apenas os vértices das arestas de suporte como ocorre no NN-clas. Além disso, como o kNN não possui nenhuma fase de filtragem de ruído, a classificação de bases com sobreposição pode ser afetada pelo *overfitting*. Uma forma de minimizar o *overfitting* seria através de uma busca exaustiva pelo melhor k, contudo essa prática é inviável para sistemas embarcados.

A Tabela 5.1 mostra a AUC média e o desvio padrão obtidos pelos métodos em cada base de dados. Nela, também são apresentados o número total de amostras N,

a dimensionalidade N_d, o número de amostras de treinamento N_t e o número total dos vértices das arestas de suporte N_b, com o intuito de se exemplificar a redução de armazenamento de dados exigido pelo método proposto em relação ao *kNN*, bem como mostrar a complexidade de cada base de dados. Os valores destacados em negrito demonstram o método que obteve o melhor desempenho em cada base de dados.

Tabela 5.1: Resultados dos algoritmos

Base de dados	CHIP-clas	NN-clas	RCHIP-clas	NN	N	N_d	N_t	N_b
Australian Cr.	0.846 ± 0.041	**0.848 ± 0.041**	0.846 ± 0.04	0.790 ± 0.047	690	14	395	62
Banknote Auth.	0.980 ± 0.026	**0.999 ± 0.003**	0.979 ± 0.028	0.998 ± 0.003	1372	4	1235	61
BcrHess	0.811 ± 0.117	**0.814 ± 0.125**	0.807 ±0.123	0.714 ± 0.228	133	30	71	6
B. Cancer W.P	0.956 ± 0.029	**0.964 ± 0.028**	0.963 ± 0.026	0.949 ± 0.031	683	9	520	20
Climate M.S.C.	**0.840 ± 0.068**	0.771 ± 0.129	0.768 ± 0.131	0.594 ± 0.088	540	18	274	199
Fertility	**0.588 ± 0.259**	0.553 ± 0.162	0.559 ± 0.163	0.578 ± 0.224	100	9	48	9
German Cr.	0.672 ± 0.042	**0.685 ± 0.033**	0.668 ± 0.043	0.605 ± 0.064	1000	24	463	263
Golub	0.774 ± 0.172	0.788 ± 0.160	0.784 ± 0.160	**0.789 ± 0.086**	72	50	39	9
Haberman's S.	0.569 ± 0.089	0.559 ± 0.108	**0.579 ± 0.084**	0.539 ± 0.082	306	3	143	49
ILPD	0.560 ± 0.086	0.568 ± 0.087	0.559 ± 0.090	**0.595 ± 0.072**	579	10	275	117
Liver disorders	**0.614 ± 0.102**	0.598 ± 0.114	0.566 ± 0.124	0.597 ± 0.079	345	6	164	110
P. ind. diabetes	0.721 ± 0.036	**0.728 ± 0.046**	0.716 ± 0.049	0.672 ± 0.058	768	8	392	124
Parkinsons	0.898 ± 0.149	0.894 ± 0.146	0.895 ± 0.147	**0.945 ± 0.125**	195	22	175	62
Sonar. M against R.	**0.876 ± 0.079**	0.872 ± 0.061	0.846 ± 0.067	0.872 ± 0.061	208	60	187	163
Stalog heart	**0.798 ± 0.082**	0.788 ± 0.071	0.784 ± 0.160	0.789 ± 0.086	270	13	134	36
Média do rank $R(\mathcal{L})$	**2.0333**	**2.0333**	**3.1000**	**2.8333**				

Para realizar uma avaliação estatística dos resultados obtidos pelos classificadores, utilizou-se o teste estatístico de *Friedman*, o qual é considerado por (Demšar, 2006) o teste mais indicado para se comparar mais de dois métodos. O teste de *Friedman* cria um ranqueamento entre os métodos para cada base de dados. A última linha da Tabela 5.1 mostra o *rank* médio obtido por cada um dos classificadores. É importante ressaltar que quanto menor o valor do *rank* melhor o desempenho do algoritmo. Assumiu-se como hipótese nula H_0 a equivalência entre os quatro métodos com um nível de significância de $\alpha = 0.05$. O valor p obtido com esse teste foi de 0.04, ou seja, não é possível afirmar que os métodos sejam equivalentes estatisticamente.

Como o teste de *Friedman* demonstrou que os métodos não são equivalentes estaticamente, um teste *post-hoc* pareado (*Wilcoxon*) foi realizado para verificar significância estatística entre cada par de algoritmos. Para todos os testes utilizou-se como hipótese nula H_0 a equivalência entre os métodos comparados com um nível de significância de $\alpha = 0.05$. A Tabela 5.2 mostra o $p - valor$ obtido em cada um dos testes pareados. Percebe-se que o NN-clas e o CHIP-clas são equivalentes estatisticamente, já o RCHIP-clas possui desempenho inferior ao CHIP-clas.

Tabela 5.2: Resultados do teste de Wilcoxon

Métodos	NN-clas	RCHIP-clas
CHIP-clas	0.476	0.036

5.2 Testes da arquitetura em hardware proposta

Após realizar o teste do NN-clas e comprovar a equivalência desse método em relação ao CHIP-clas, implementou-se essa nova abordagem em *VHDL*, utilizando a metodologia descrita no Capítulo anterior. Como dito anteriormente, o NN-clas foi implementado primeiramente sem filtro, e dependendo dos resultados alcançados seria feito um estudo de viabilidade da incorporação do processo de filtragem à arquitetura desenvolvida.

Inicialmente, foram realizados alguns testes no *EP* do cálculo de distância da fase de treinamento. Conforme já mencionado, desenvolveu-se uma ferramenta no *Matlab* que gera o código em *VHDL* de cada um dos *EPs* de acordo com a configuração de parâmetros do usuário. Nesses experimentos não foram utilizados os elementos de memória local, os dados foram lidos diretamente de um arquivo de texto, contudo utilizou-se a mesma estrutura de transferência de dados definida na arquitetura, facilitando assim a simulação do projeto.

Para validar os dois tipos de *EPs* para cálculo de distância, gerou-se dados aleatórios de n amostras por m dimensões através da função *rand* do *Matlab* em um intervalo de [0 100], sendo que tais parâmetros foram alterados ao longo do teste. Os dados gerados foram convertidos em binário de acordo com o padrão *IEEE-754* de precisão simples e em seguida armazenados em um arquivo de texto. Cada amostra de dado com suas respectivas dimensões foi salva em uma linha do arquivo. Dessa maneira, durante a simulação, a unidade de controle fica responsável por realizar a leitura de uma linha do arquivo a cada ciclo de *clock*, e em seguida fornecer a amostra lida com todas suas características às entradas dos blocos multiplexadores e atrasadores.

Os resultados fornecidos pelo *EP* do cálculo de distância foram armazenados em um outro arquivo de texto, através da unidade de controle. Assim, após a simulação os resultados armazenados no arquivo em formato binário foram lidos e convertidos em decimal pela ferramenta desenvolvida, e logo após comparados com os valores obtidos através do mesmo algoritmo implementado no *Matlab*.

Para facilitar todo o fluxo do experimento, a ferramenta desenvolvida inicialmente solicita ao usuário o número de amostras e dimensões e o tipo de métrica que deseja simular. Em seguida, gera os dados de maneira aleatória segundo os parâmetros pré-estabelecidos e os armazena em um arquivo de texto. Posteriormente, gera os códigos em *VHDL* e chama o simulador *QuestaSim* para executar todo o *EP* do cálculo de

distância. Por último, os resultados produzidos durante a simulação que foram armazenados em um outro arquivo de texto são lidos, convertidos para decimal e comparados com os valores obtidos pelo algoritmo executado no *Matlab*.

Os resultados foram comparados utilizando a métrica de erro médio quadrático(*MSE*), listados na tabela 5.3. A primeira coluna dessa tabela ilustra o número de amostras e o tamanho de sua dimensão, a segunda coluna por sua vez mostra o erro médio quadrático obtido utilizando a métrica de *Manhattan*, e por último a terceira coluna exibe o erro médio quadrático alcançado através da técnica Euclidiana quadrática.

Tabela 5.3: Erro do cálculo de distância

Tamanho dos dados	MSE *Manhattan*	MSE Euclidiana quadrática
$4x4$	$1.8853e^{-11}$	$6.1767e^{-07}$
$16x4$	$4.5838e^{-11}$	$5.6288e^{-07}$
$30x8$	$2.29373e^{-10}$	$2.4916e^{-06}$
$60x10$	$7.6504e^{-10}$	$5.7339e^{-06}$
$120x20$	$5.5677e^{-09}$	$3.6662e^{-05}$

Observa-se que o erro aumenta conforme o número de dimensões cresce, uma vez que com um número maior de dimensões ocorre mais operações de soma. Ressalta-se também que o erro obtido para a técnica Euclidiana quadrática é maior do que o da métrica de *Manhattan*, pois a arquitetura desenvolvida para técnica Euclidiana quadrática também utiliza o bloco multiplicador. Contudo, os erros foram coerentes com os obtidos pelo trabalhado de García (2014) que também utiliza blocos somadores/subtratores e multiplicadores em ponto flutuante de precisão simples para realizar o cálculo de inversão de matriz. O erro médio quadrático obtido pelo citado trabalho para o bloco somador/subtração foi de $2.76e^{-11}$ e para o multiplicador de $1.53e^{-07}$. Esse erro é devido ao fato de tais operações não possuírem a fase de arredondamento, o dado é apenas truncado no final de cada operação.

Como o cálculo de distância é usado apenas para comparação, tanto na fase de treinamento como na fase de classificação, estima-se que esse erro não tenha um impacto significativo no resultado final do método NN-clas. Isso será comprovado nos experimentos posteriores.

O segundo *EP* testado e simulado foi o do grafo de proximidade. Os testes também foram realizados através de uma ferramenta desenvolvida no *Matlab* que possui as mesmas características da que foi criada para realizar o teste do *EP* do cálculo de distância. Como a entrada de dados nesse *EP* se refere às distância calculadas pelo bloco anterior, o *software* desenvolvido, gera os dados aleatórios de acordo com os parâmetros configurados de n e m, calcula as distâncias entre esses dados gerados e

em seguida, converte tais distâncias em binário e salva em um arquivo de texto. As demais fases dessa ferramenta são idênticas as da criada para a simulação do cálculo de distância.

Como os resultados desse *EP* são números binários, os valores foram comparados um a um com os obtidos através do *software* implementado em *Matlab*. Em todos os experimentos realizados, os resultados do *EP* foram idênticos aos obtidos por *software*.

Os mesmos procedimentos foram repetidos para os *EPs* restantes da fase de treinamento. Para o grafo de borda os rótulos das amostras de dados também foram gerados de maneira aleatória, variando entre os binários 0 e 1, e depois tais valores foram salvos em uma única linha de um arquivo de texto para simular o mesmo formato de saída de dados do o bloco *fifo label*. No EP de armazenamento de dados, as amostras foram armazenadas cada uma em uma linha de um arquivo de texto no formato binário e seus respectivos rótulos em um outro arquivo de texto. Assim como ocorreu no EP do grafo de proximidade os resultados desse *EPs* foram idênticos aos obtidos via software.

Após o teste separado de cada um dos *EPs*, realizou-se a simulação da junção de todos os EPs em uma arquitetura única. Os resultados para diferentes parâmetros também foram idênticos aos obtidos por meio de software. Comprovando assim que o havia sido previsto, ou seja, o erro médio quadrático encontrado no *EP* do cálculo de distância não teve impacto no resultado final de treinamento do NN-clas.

Foi realizado também um outro teste no software *Quartus Prime 16.1* da *Intel Corporation (Altera)* de análise e síntese, utilizando o *FGPA 5SGXMA7N1F45C2* da família *Stratix V*, com o objetivo de medir o consumo de recurso gasto pela arquitetura da fase de treinamento. Inicialmente foi realizada uma análise e síntese apenas dos blocos somadores/subtratores e multiplicadores, os quais são os blocos mais complexos de toda arquitetura, em seguida foram feitas análises de tal arquitetura com diferentes números de amostras, com o intuito de mostrar a variação do consumo de recursos com esse parâmetro. Esse teste foi realizado apenas com a arquitetura que utiliza a técnica Euclidiana quadrática, pois a diferença de consumo de recursos entre os dois tipos de arquiteturas não é muito grande.

A tabela 5.4 mostra o consumo de recursos em termos de números de *Look Up Tables (LUTs)* e blocos dedicados de processamento digital (*DSP*(27x27)), também foi listada a máxima frequência de operação alcançada por cada bloco no *FPGA 5SGXMA7N1F45C2*. Já a tabela 5.5 demonstra o consumo de recursos para diferentes números de amostras de duas dimensões. Observa-se nessa tabela que a medida que o número de amostras aumenta, cai a frequência máxima de operação do *FPGA*, devido ao aumento do caminho crítico do circuito criado, e aumenta o número de *LUTs* e *DSPs*, sendo que essa elevação do número de *LUTs* é praticamente exponencial com o aumento do número de amostras. A figura 5.2 mostra um gráfico que exemplifica

essa relação entre o número de *LUTs* e amostras.

A tabela 5.6 demonstra o consumo de recursos para diferentes números de dimensões de 4 amostras. Observa-se que a medida que o número de dimensões aumenta, cai a frequência máxima de operação do *FPGA* e aumenta o número de *LUTs* e *DSPs*, contudo a elevação do número de *LUTs* não é exponencial, como ocorre na relação entre o número de *LUTs* e o número de amostras. A relação entre o número de *LUTs* e o número de dimensões é praticamente linear, como mostra a figura 5.3.

Tabela 5.4: Resultado de síntese dos blocos de soma/subtração e multiplicação

Tipo do bloco	N. LUTs	N. DSP(27x27)	Freq. (MHz)
soma/subtração	509	-	303.4
multiplicação	83	1	717.36

Tabela 5.5: Resultados de síntese da arquitetura da fase de treinamento do NN-clas utilizando a técnica euclidiana quadrática para alguns números de amostras

N. amostra	N. LUTs	N. DSP(27x27)	Freq. (MHz)
4	11053	9	278.78
5	22560	12	267.47
6	40381	15	259.88
7	61480	18	247.46
8	102961	21	225.07
9	147774	24	214.41
10	205910	27	210.26

Tabela 5.6: Resultados de síntese da arquitetura da fase de treinamento do NN-clas utilizando a técnica euclidiana quadrática para alguns números de dimensão

N. dimensão	N. LUTs	N. DSP(27x27)	Freq. (MHz)
2	11053	9	278.78
3	15272	12	261.64
4	17209	15	258.70
5	22476	18	251.76
6	25015	21	232.72

O teste seguinte foi a simulação da arquitetura da fase de classificação. Não foi necessário realizar o teste de cada *EP* dessa etapa de forma separada, pois o *EP* do cálculo de distância dessa fase é bem semelhante ao da etapa de treinamento. Como a fase de classificação do NN-clas é igual ao método *kNN* com $k = 1$, porém utilizando

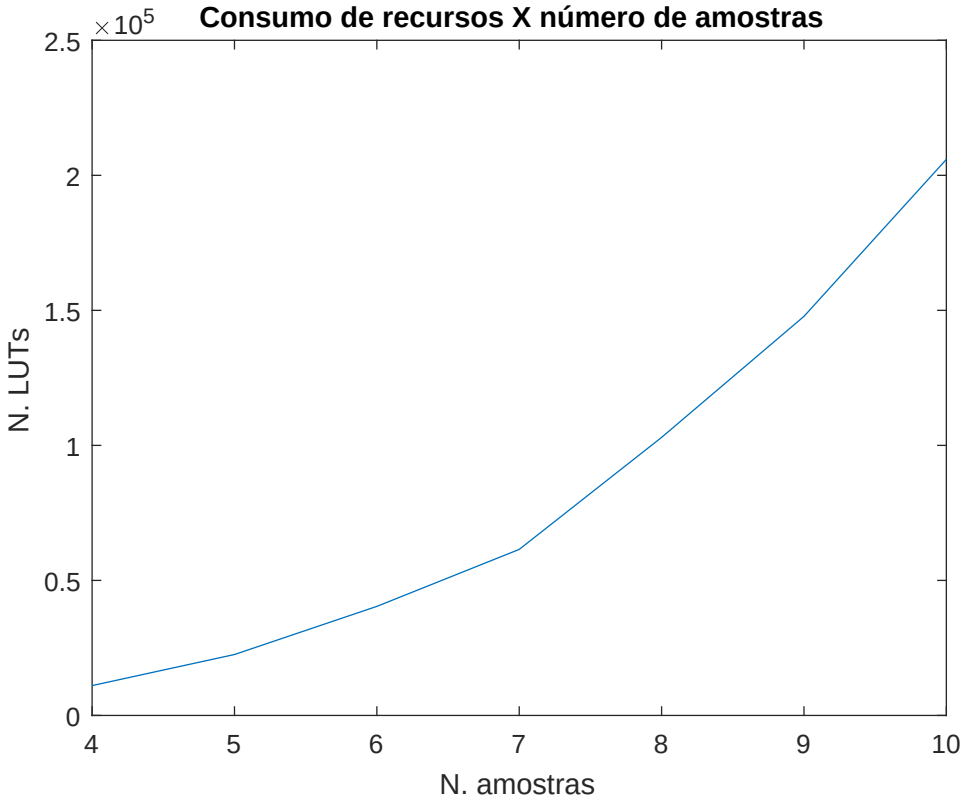


Figura 5.2: Relação entre o número de LUTs e o número de amostras

apenas as amostras que estão localizadas na borda, realizou-se o teste inicial como se essa fase de classificação fosse o próprio método *kNN*. Ou seja, utilizou-se todo o conjunto de amostras de treinamento para a classificação de um novo dado.

Para tanto foi criado um *script* no *Matlab* para ler as 15 bases de dados utilizadas para validar o NN-clas, em seguida, realizar uma validação cruzada de *10-folds*, e para cada *fold* gerar um código *VHDL* de acordo com o número de amostras e dimensão da base de dado, e chamar o *QuestaSim* para simular a classificação dos dados de teste. Os resultados gerados durante a simulação foram comparados como os reais utilizando a medida de desempenho *AUC*. Os mesmos dados e medida de desempenho foram utilizados para a implementação do *kNN* no *Matlab*, com o intuito de se realizar a comparação entre a arquitetura implementada em *hardware* e o algoritmo executado em software *Matlab*.

Os testes foram feitos utilizando a métrica de *Manhattan* e a técnica Euclidiana quadrática. A tabela 5.7 exemplifica os resultados alcançados com tais testes. Como se pode observar, os resultados obtidos via software são idênticos aos simulados no *FPGA*, pois o erro quadrático médio no cálculo de distância não impacta o resultado final do algoritmo para essas bases de dados. Verifica-se também, que o uso da técnica

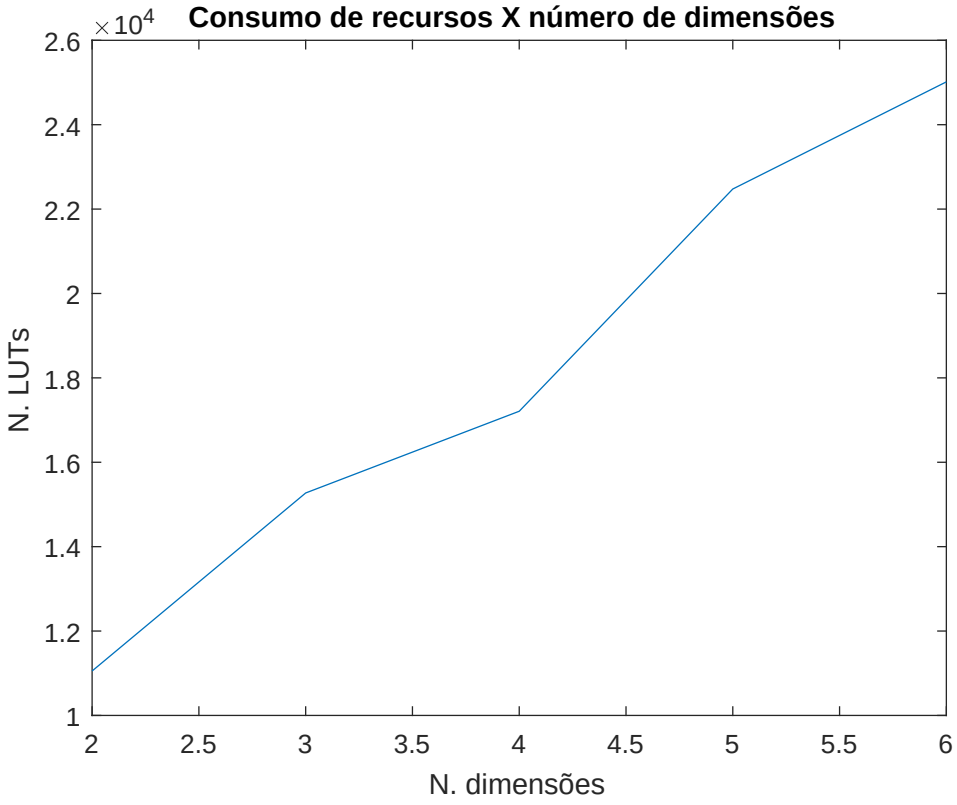


Figura 5.3: Relação entre o número de LUTs e o número de dimensões

Euclidiana quadrática promove um desempenho melhor para a maioria das bases de dados.

Tabela 5.7: Resultados do 1-NN implementado em hadrware e em software

Base de dados	1-NN *Manhattan* Software	1-NN Hardware *Manhattan*	1-NN Euclidiana quadrática Software	1-NN Euclidiana quadrática Software
Australian Cr.	**0.7933 ± 0.0436**	**0.7933 ± 0.0436**	0.7904 ± 0.0473	0.7904 ± 0.0473
Banknote Auth.	0.9987 ± 0.0028	0.9987 ± 0.0028	0.9987 ± 0.0028	0.9987 ± 0.0028
BcrHess	0.6711 ± 0.1853	0.6711 ± 0.1853	**0.7144 ± 0.2279**	**0.7144 ± 0.2279**
B. Cancer W.P	**0.9613 ± 0.0302**	**0.9613 ± 0.0302**	0.9489 ± 0.0307	0.9489 ± 0.0307
Climate M.S.C.	**0.6559 ± 0.1386**	**0.6559 ± 0.1386**	0.5953 ± 0.0878	0.5953 ± 0.0878
Fertility	**0.5965 ± 0.2247**	**0.5965 ± 0.2247**	0.5778 ± 0.2241	0.5778 ± 0.2241
German Cr.	**0.6128 ± 0.0584**	**0.6128 ± 0.0584**	0.6048 ± 0.0642	0.6048 ± 0.0642
Golub	0.7150 ± 0.2529	0.7150 ± 0.2529	**0.7450 ± 0.2999**	**0.7450 ± 0.2999**
Haberman's S.	0.5251 ± 0.0882	0.5251 ± 0.0882	**0.5391 ± 0.0823**	**0.5391 ± 0.0823**
ILPD	0.5835 ± 0.0827	0.5835 ± 0.0827	**0.5953 ± 0.0718**	**0.5953 ± 0.0718**
Liver disorders	**0.6178 ± 0.0570**	**0.6178 ± 0.0570**	0.5967 ± 0.0786	0.5967 ± 0.0786
P. ind. diabetes	0.6387 ± 0.0495	0.6387 ± 0.0495	**0.6722 ± 0.0579**	**0.6722 ± 0.0579**
Parkinsons	0.9354 ± 0.1258	0.9354 ± 0.1258	**0.9454 ± 0.1255**	**0.9454 ± 0.1255**
Sonar. M against R.	0.8501 ± 0.0714	0.8501 ± 0.0714	**0.8720 ± 0.0615**	**0.8720 ± 0.0615**
Stalog heart	0.7771 ± 0.0812	0.7771 ± 0.0812	**0.7898 ± 0.0862**	**0.7898 ± 0.0862**

Para essa etapa de classificação também foi feito o mesmo teste realizado na fase de treinamento para avaliar o consumo de recursos. Na classificação, como não é possível ter o conhecimento prévio do número de amostras situadas na borda, os *EPs* foram projetados em função apenas do número de dimensões. O número de amostras locali-

zadas na borda terá um impacto somente na quantidade de ciclos de *clock* necessária para concluir a etapa de classificação. Dessa maneira, o teste de consumo de recursos dessa fase foi realizado apenas em função do número de dimensões. A tabela 5.8 mostra os resultados desses testes. Como se pode observar o número de *LUTs* e *DSPs* crescem com o aumento do número de dimensões, já a frequência máxima de trabalho do *FPGA* diminui. O gráfico exemplificado pela figura 5.4 representa a relação entre o número de dimensões e o número de *LUTs*. Percebe-se que essa relação é praticamente linear e o consumo de recursos não é tão alto quanto o da fase de treinamento.

Tabela 5.8: Resultados de síntese da arquitetura da fase de classificação do NN-clas utilizando a técnica euclidiana quadrática para diferentes números de dimensões

N. dimensão	N. LUTs	N. DSP(27x27)	Freq. (MHz)
2	1645	2	291.5
3	3040	3	287.3
4	3700	4	280.76
5	5408	5	276.41
6	6052	6	272.69
7	7062	7	270.1
8	7740	8	269.3
9	9911	9	267.5
10	10607	10	265.18
11	11651	11	263.65
12	12335	12	261.76

Após a validação das duas etapas do NN-clas sem filtro, o método foi simulado por completo, utilizando a mesma ferramenta desenvolvida no *Matlab*, e em seguida os resultados foram comparados com os obtidos através da implementação em software. Por causa do crescimento exponencial de recursos conforme o aumento do número de amostras na arquitetura da etapa de treinamento, só foi possível realizar os testes para apenas 3 bases de dados das 15 utilizadas nos experimentos anteriores. A tabela 5.9 exemplifica os resultados alcançados com esses experimentos. Observa-se que os valores obtidos em ambas implementações são idênticas, pois o MSE do EP do cálculo de distância não impacta o resultado final do algoritmo.

Pode-se observar, comparando os resultados das Tabelas 5.1 e 5.9, que o desempenho alcançado pelo NN-clas sem filtro é bem menor do que com filtro pelo fato de algumas bases de dados serem bem ruidosas. Verifica-se também, que a arquitetura é não muito adequada para o uso em aplicações que possuam um número de amostras de treinamento um pouco elevado, devido ao seu alto consumo de recursos. Para reduzir

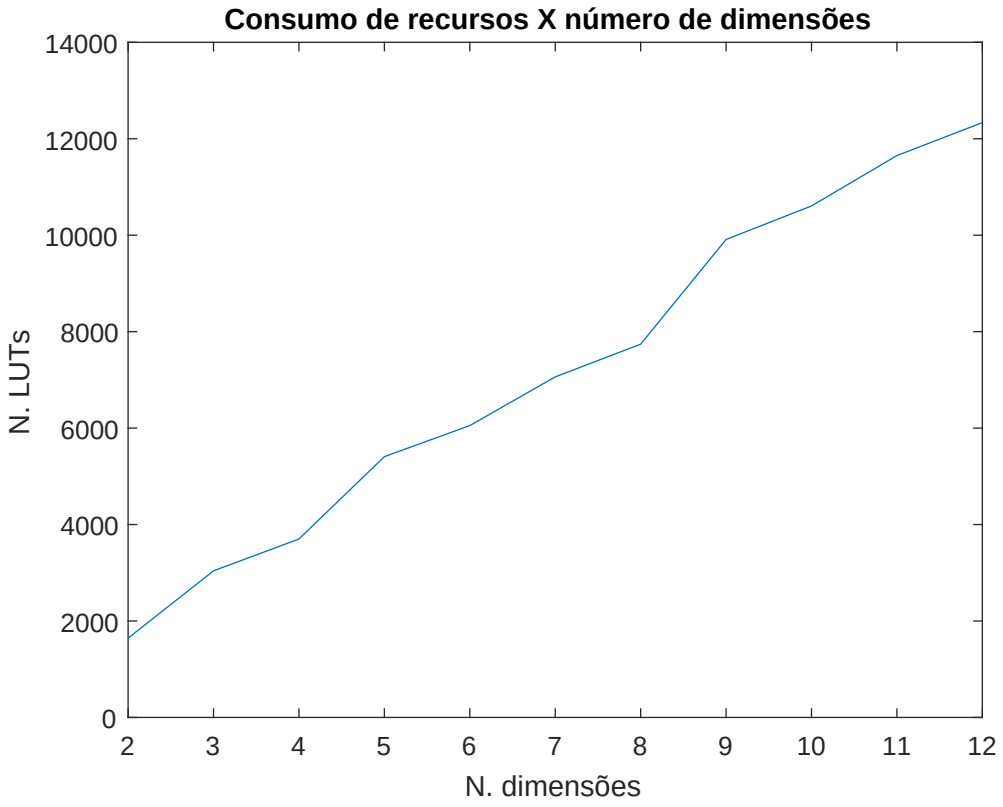


Figura 5.4: Relação entre o número de LUTs e o número de dimensões da fase de classificação

Tabela 5.9: Resultados do NN-clas sem filtro implementado em hadrware e em software

Base de dados	NN-clas sem filtro em Software	NN-clas sem filtro em Hardware
BcrHess	0.7061 ± 0.22705	0.7061 ± 0.22705
Fertility	0.5667 ± 0.2339	0.5667 ± 0.2339
Golub	0.7450 ± 0.2999	0.7450 ± 0.2999

esse consumo, a arquitetura teria que ser reprojetada de maneira sequencial, eliminado, assim, o paralelismo nos *EPs*.

A inserção do filtro nessa arquitetura proposta também não é um processo trivial, pois o algoritmo do filtro teria que interromper o fluxo de dados entre o grafo de proximidade antes da filtragem e o grafo de borda. Além disso, o consumo de recursos aumentaria mais ainda. Sendo assim, o ideal para a implementação do NN-clas completo, ou seja, com filtro, seria o desenvolvimento de uma nova arquitetura sequencial com alguns módulos sem fluxo de dados para a fase de treinamento.

A arquitetura sequencial em *FPGA* apesar de ter vantagens aos microcontroladores, possui um *time-to-market* bem superior. Dessa forma, essa solução de redesenho da

arquitetura demandaria muito tempo de desenvolvimento e testes. Então, uma solução alternativa e mais rápida encontrada foi a de realizar uma implementação do NN-clas em um microcontrolador *ARM Cortex-M4* com o intuito de ilustrar a viabilidade da implementação em hardware do NN-clas para aplicações que não exijam tão alto desempenho.

5.3 Testes da implementação do NN-clas em microcontrolador ARM

A plataforma escolhida para realizar a implementação do NN-clas foi a placa de desenvolvimento *Discovery STM32F429ZI*, pois é composta por um microcontrolador *ARM Cortex-M4 STM32F429ZI* com *FPU* de baixo consumo de energia, uma memória *SDRAM* externa de 64-Mbit, dentre outros periféricos. Dessa forma, é possível utilizar essa placa para testar todas as 15 bases de dados e outros tipos de aplicações com diferentes números de amostras e dimensões.

O microcontrolador *STM32F429ZI* é um *ARM Cortex-M4* de 32bits de arquitetura RISC baseada no modelo de *Harvard* produzido pela fabricante *STMicroelectronics*. Esse microcontrolador possui uma unidade de ponto flutuante e instruções *DSP*, características essas que melhoram o desempenho de operações que utilizam ponto flutuante de precisão simples. Também possui 2MB de memória de programa do tipo *flash*, 256+4kB de memória de dados *RAM*, controle externo flexível de memória *RAM* externa através de um barramento de 32bits, 225 *DMIPS*, frequência de trabalho até 180MHz e diversos tipos de periféricos, tais como: conversores analógico-digital, *timers*, *usarts/uarts*, *spi*, *i2c*, *usb*, *ethernet*, dentre outros. A figura 5.5 mostra um diagrama em blocos da arquitetura desse microcontrolador.

O algoritmo NN-clas foi implementado nesse microcontrolador em linguagem C utilizando a IDE(*Integrated Development Environment*) *IAR Embedded Workbench*. O código foi desenvolvido de maneira modularizada em funções separadas, de forma que alguns tipos de funções genéricas, como por exemplo o cálculo de distância, pudessem ser utilizadas na implementação de outros algoritmos de aprendizado de máquina.

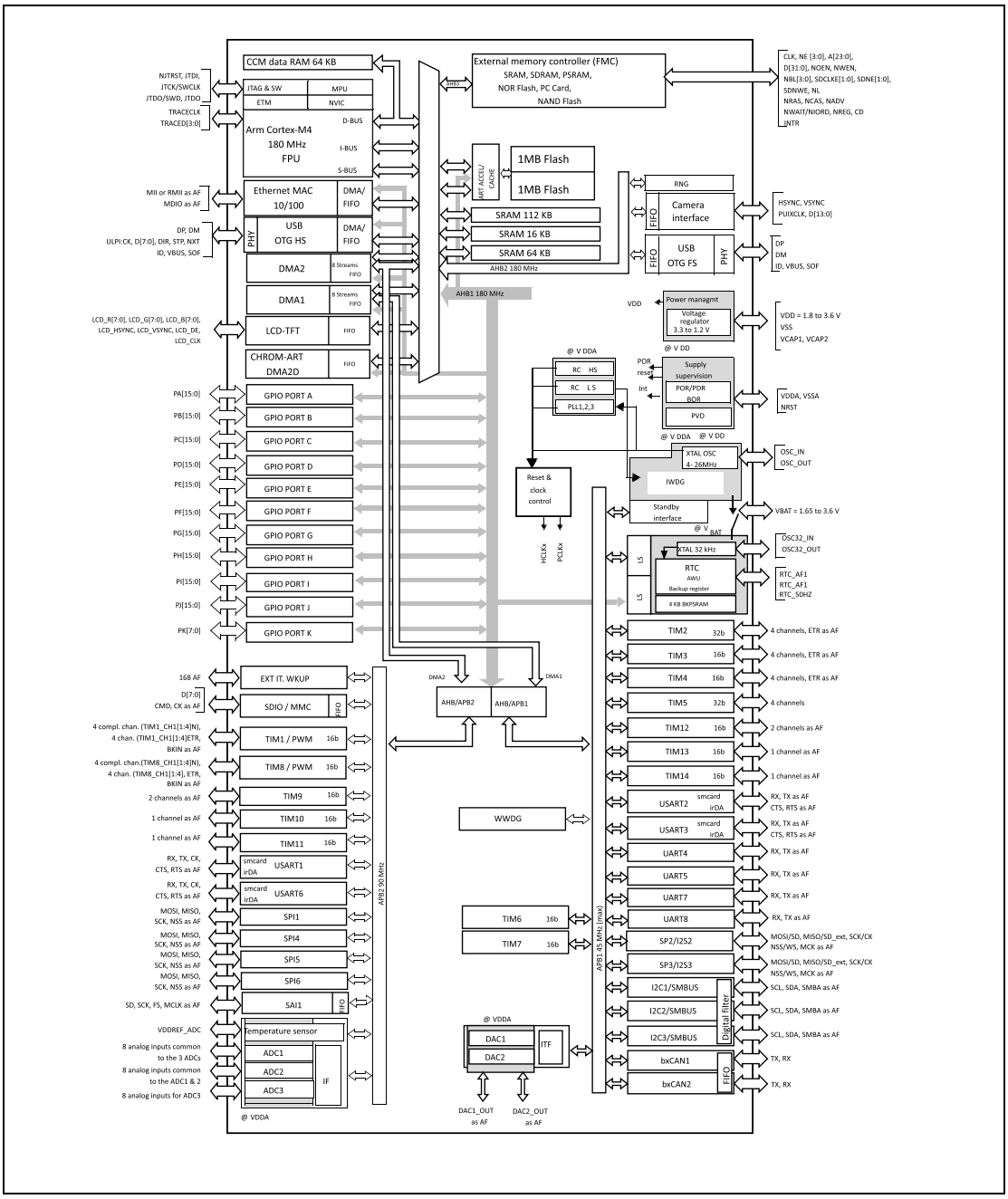


Figura 5.5: Diagrama em blocos do microcontrolador *STM32F429ZI* (imagem extraída do *datasheet* desse microcontrolador)

A figura 5.6 mostra o fluxograma do *firmware* desenvolvido para o NN-clas. Como se pode observar, inicialmente todos os periféricos utilizados são inicializados, em seguida as amostras de treinamento e seus respectivos rótulos são lidos e armazenadas na memória flash interna do microcontrolador. Depois ocorre a fase de treinamento, onde

primeiramente realiza-se o cálculo de distância entre todas as amostras de treinamento, através da técnica euclidiana quadrática. Em seguida, é feito o grafo de proximidade que define quais amostras formam vértices entre si, segundo a Equação 2.1. Logo após é realizado o processo de filtragem descrido no capítulo 2. Durante esse processo é criado um vetor de valores booleanos com o tamanho do número de amostras de treinamento, e em cada posição desse vetor é armazenado 1 caso a amostra seja detectada como ruído, ou 0 caso o contrário. Depois é necessário realizar um novo cálculo do grafo de proximidade apenas com as amostras não ruidosas. Para finalizar essa fase de treinamento, calcula-se o grafo de borda para detectar quais amostras estão localizadas na superfície de separação. Durante esse cálculo, é criado um novo vetor de valores booleanos para informar quais amostras estão na borda.

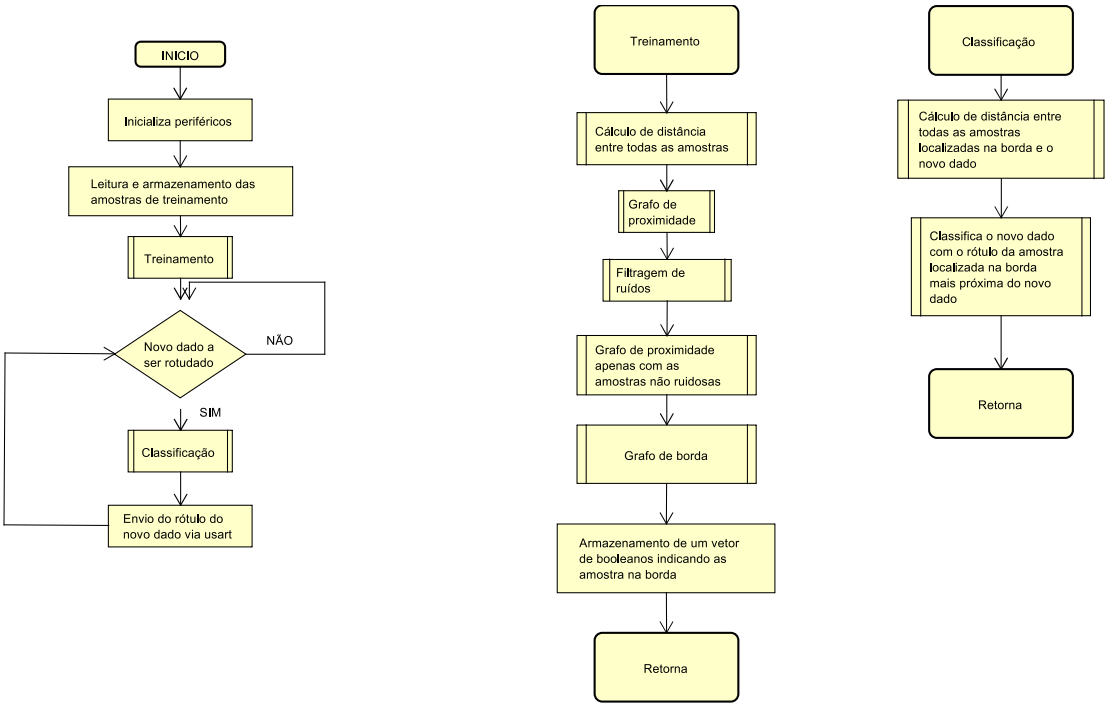


Figura 5.6: Fluxograma do firmware desenvolvido para o NN-clas

Após a fase de treinamento é verificado se existe um novo dado a ser classificado, caso tenha, ocorre a fase de classificação. Nessa fase, é realizado o cálculo de distância entre o novo dado e as amostras localizadas na superfície de separação. Em seguida, verifica-se qual amostra está mais próxima do novo dado e o resultado da classificação é o rótulo dessa amostra. O rótulo desse novo dado é enviado via *usart* para ser recebido em um computador através de um conversor *usart/usb*, ou para se comunicar com algum módulo de comunicação sem fio. O *firmware* fica nesse *loop* infinito aguardando a chegada de um novo dado para a classificação.

O *firmware* foi desenvolvido de forma a consumir o mínimo de memória possível, dessa maneira, toda a transferência de dados entre as funções ocorre através de ponteiros. A organização dos dados nas memórias se deu dá seguinte forma: as amostras de treinamento com seus respectivos rótulos são salvos na memória *flash* e o seu espaço ocupado é de $(n\ x\ m\ x\ 32\ +\ n)$ bits, onde n representa o número das amostras de treinamento e m sua dimensão.

A matriz de distância calculada é armazenada na *SDRAM* externa, sendo que essa matriz ocupa um espaço de $(n\ x\ n\ x\ 32)$ bits. Na memória *RAM* interna, são salvos os dois vetores binários, um que indica quais amostras são ruidosas e outro quais amostras estão localizadas na borda, cada vetor ocupa um espaço de n bits. Também é salvo na memória *RAM* um vetor de $(n\ x$ 32bits) no qual será armazenada a medida de qualidade de cada amostra de treinamento, pois como foi dito no capítulo 2 essa informação é necessária para realizar o processo de filtragem.

Após o desenvolvimento desse *firmware*, realizou-se experimentos com as 15 bases de dados utilizadas em testes anteriores. Assim como foi feito na simulação no *FPGA*, as bases de dados foram dividas em *10-folds* e para cada *fold* as amostras de treinamento e seus rótulos foram salvos na memória *flash*. O *firmware* foi executado e os dados de testes foram enviados através de comunicação serial como o computador pelo software *RealTerm*. O rótulo de cada dado de teste foi enviado pelo microcontrolador também pela comunicação serial. Após a chegada dos rótulos de todos os dados, exportou-se os mesmos para um arquivo de texto.

Depois da execução do *firmware* com os *10-folds*, os arquivos de texto foram lidos pelo software *Matlab* e realizou-se em seguida uma medida de desempenho utilizando a métrica *AUC*. A tabela 5.10 mostra os resultados para as 15 bases de dados através da implementação no microcontrolador e em *software*. Como se pode observar os valores foram idênticos para todas as bases de dados.

Por último, foi realizada uma comparação de desempenho entre a arquitetura proposta em *FPGA* e o algoritmo implementado no microcontrolador *ARM*, retirando a etapa de filtragem. Para tanto, foi utilizada uma base de dados de 4 amostras e 4 dimensões gerada de maneira aleatória com a função *rand* do *Matlab* em um intervalo de [0 10]. Essa base de dados foi gerada de maneira que todas as 4 amostras estejam localizadas na margem de separação entre as classes.

Para a comparação utilizou-se uma simulação no *software QuestaSim* para a arquitetura do *FPGA* e o *ARM* foi testado na *IDE IAR* no modo *debug* de simulação. Os ciclos de *clock* de cada fase do NN-clas sem filtro foi medida no *ARM* através do registrador interno *CYCLECOUNTER*. A tabela 5.11 mostra o número de ciclos de *clock* necessário para cada uma das duas fases do classificador e o tempo em segundos, considerando a máxima frequência de trabalho alcançada por cada plataforma.

Tabela 5.10: Resultados do NN-clas implementado em software e em microcontrolador

Base de dados	NN-clas software	NN-clas microcontrolador
Australian Cr.	0.848 ± 0.041	0.848 ± 0.041
Banknote Auth.	0.999 ± 0.003	0.999 ± 0.003
BcrHess	0.814 ± 0.125	0.814 ± 0.125
B. Cancer W.P	0.964 ± 0.028	0.964 ± 0.028
Climate M.S.C.	0.771 ± 0.129	0.771 ± 0.129
Fertility	0.553 ± 0.162	0.553 ± 0.162
German Cr.	0.685 ± 0.033	0.685 ± 0.033
Golub	0.788 ± 0.160	0.788 ± 0.160
Haberman's S.	0.559 ± 0.108	0.559 ± 0.108
ILPD	0.568 ± 0.087	0.568 ± 0.087
Liver disorders	0.598 ± 0.114	0.598 ± 0.114
P. ind. diabetes	0.728 ± 0.046	0.728 ± 0.046
Parkinsons	0.894 ± 0.146	0.894 ± 0.146
Sonar. M against R.	0.872 ± 0.061	0.872 ± 0.061
Stalog heart	0.788 ± 0.071	0.788 ± 0.071

Tabela 5.11: Comparação de desempenho do NN-clas sem filtro em FPGA e no ARM

Plataforma	Fase	N. ciclos de clock	Max. Freq(MHz)	Tempo (s)
ARM cortex M4	treinamento	14504	180	$8.05e^{-5}$
FPGA	treinamento	22	258	$8.52e^{-8}$
ARM cortex M4	classificação	2714	180	$1.50e^{-5}$
FPGA	classificação	13	280	$4.64e^{-8}$

Observa-se na tabela 5.11 que o desempenho da arquitetura proposta para *FPGA* é bem superior ao do microcontrolador *ARM*. O tempo gasto em segundos para essa base de dados de 4 amostras por 4 dimensões é bem pequeno, contudo para amostras maiores esse valor aumenta bastante para o microcontrolador. No *FPGA* esse aumento de tempo não será muito alto para um número maior de amostras de treinamento, uma vez que a maioria das operações são realizadas de forma paralela

Capítulo 6

Conclusões e Trabalhos Futuros

Nesse trabalho inicialmente, foram avaliadas duas técnicas para redução das operações mais complexas utilizadas na regra de decisão do método CHIP-clas (Torres et al., 2015). Os resultados mostraram que a redução obtida através do NN-clas não gera perdas significativas no desempenho do método, tornando-o ainda mais atrativo para implementação em sistemas embarcados. Já a abordagem utilizando a formulação do CHIP-clas reduzido, apesar de conseguir reduzir as referidas operações complexas, possui perdas consideráveis de desempenho.

Também foi realizada uma comparação do NN-clas com o *kNN* com $k = 1$, uma vez que essa nova metodologia baseia-se nesse algoritmo. Com essa comparação foi possível observar que, apesar do kNN também ser um método factível de ser implementado em sistemas embarcados, ele exige um maior consumo de memória, pelo fato de armazenar todo o conjunto de treinamento, além de apresentar um desempenho inferior ao do NN-clas.

Como o NN-clas se mostrou atrativo para implementação em sistemas embarcados devido a sua baixa complexidade nas operações realizadas, desenvolveu-se uma arquitetura em fluxo de dados de alto desempenho para que o método pudesse ser executado em *FPGA*. A arquitetura foi simulada em um software *QuestaSim* com o objetivo de se verificar a sua funcionalidade, em seguida, realizou-se a sua análise e síntese para medir a variação do consumo de recursos de acordo com o aumento do número de amostra de treinamento. Os resultados mostraram que apesar da arquitetura proposta ter apresentado respostas idênticas ao da implementação em software, o consumo de recursos cresce de forma exponencial com o aumento do número de amostras de treinamento. Dessa maneira, essa arquitetura seria indicada para aplicação com um número reduzido de amostras de treinamento, não sendo adequada em aplicações com um maior número de amostras.

Para a utilização do método em sistemas com um número maior de amostras, seria

necessário desenvolver uma arquitetura sequencial. Essa nova arquitetura teria um consumo de recursos reduzido e esse consumo não iria crescer com o aumento do número de amostras. Contudo, o desempenho diminuiria de forma considerável, uma vez que seria eliminado grande parte do paralelismo da arquitetura inicial.

Apesar da arquitetura sequencial em *FPGA* ainda possuir um desempenho superior ao do microcontrolador, o projeto de uma nova arquitetura em *FPGA* é mais custoso e dispendioso do que a implementação do método em microcontrolador, então optou-se por desenvolver um projeto para a execução do NN-clas em um microcontrolador *ARM Cortex-M4*.

O projeto desenvolvido foi testado em uma placa de desenvolvimento *Discovery STM32F429ZI*. Com essa placa foi possível realizar os experimentos com diversas bases de dados reais. Os resultados desses testes foram idênticos aos obtidos através de *software*. Com isso, foi possível comprovar que o método de aprendizagem de máquina NN-clas pode ser implementado em sistemas embarcados e utilizado em diversas aplicações. Contudo, pode-se dizer que o uso das implementações aqui propostas não é recomendado para cenários que exijam o processamento online das fases de treinamento e operação do classificador, quando se tem um grande número de amostras.

Uma alternativa viável para esses problemas encontrados seria o desenvolvimento da arquitetura *SoC FPGA*. A arquitetura apenas em *FPGA*, apesar de ter um bom desempenho, consome muito recurso, não sendo muito adequada para implementação em uma única *FPGA* em aplicação com um número de amostras de treinamento mais elevado. Já o projeto em microcontroladores apresentou um consumo de recursos razoável, porém o tempo de execução do algoritmo é alto. Diante disso, a fase de treinamento poderia ser implementada na parte *ARM* do *SoC*, utilizando o projeto desenvolvido para o microcontrolador. E a fase de classificação ser desenvolvida no *FPGA* do *SoC*, através da arquitetura projetada nesse trabalho. Dessa maneira, a fase de treinamento gastaria um tempo um pouco maior, porém o desempenho da classificação seria mais elevado. E o consumo de recursos seria menor, possibilitando, assim a aplicação de um método de aprendizado de máquina com uma boa acurácia em diversos sistemas que não necessitem de um treinamento em tempo real.

Outro trabalho que poderia ser desenvolvido seria redefinir como requisitos de desenho alguns parâmetros como área e frequência mínima de operação, dessa forma, uma nova arquitetura poderia ser desenvolvida de maneira apropriada aos recursos físicos dos *FPGAs* disponíveis no mercado.

Também seria interessante realizar em trabalhos futuros o desenvolvimento arquitetural utilizando ponto fixo e projetando processos distintos de normalização. Em seguida, fazer um estudo comparativo de ganhos e perdas dessa nova arquitetura em relação a proposta nesse trabalho.

Referências Bibliográficas

Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M. e Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376.

Bhandarkar, D. e Clark, D. W. (1991). Performance from architecture: Comparing a risc and a cisc with similar hardware organization. *SIGPLAN Not.*, 26(4):310–319.

da Costa, C. (2017). Indústria 4.0: O futuro da indústria nacional. *POSGERE*, 1(4):5–14.

da Silva Júnior, E. R. (2007). Investigação de técnicas de extração e seleção de características e classificadores aplicados ao problema de classificação de dígitos manuscritos de imagens de documentos históricos. Master's thesis, Universidade de Pernambuco.

de Almeida, R.; de Moraes, C. e de Faria Piola Seraphim, T. (2017). *Programação de Sistemas Embarcados: Desenvolvendo Software para Microcontroladores em Linguagem C*. Elsevier Brasil.

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30.

Estlick, M. (2002). *An FPGA Implementation of the K-means Algorithm for Image Processing*. PhD thesis, MS thesis, Dept. Elect. Eng., Northeastern Univ., Boston, MA.

Estlick, M.; Leeser, M.; Theiler, J. e Szymanski, J. J. (2001). Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pp. 103–110. ACM.

Gabriel, K. R. e Sokal, R. R. (1969). A new statistical approach to geographic variation analysis. *Systematic Biology*, 18(3):259–278.

García, J. A. (2014). *Implementação em hardware reconfigurável de operadores matriciais para solução numérica de sistemas lineares*. Tese de doutorado, Faculdade de Tecnologia da Universidade de Brasília.

Garcia, L. P.; de Carvalho, A. C. e Lorena, A. C. (2015). Effect of label noise in the complexity of classification problems. *Neurocomputing*, 160:108–119.

Golub, T. R.; Slonim, D. K.; Tamayo, P.; Huard, C.; Gaasenbeek, M.; Mesirov, J. P.; Coller, H.; Loh, M. L.; Downing, J. R.; Caligiuri, M. A. et al. (1999). Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *science*, 286(5439):531–537.

Gomes, O. d. F. M. (2008). Microscopia co-localizada: Novas possibilidade na caracterização de minérios. Master's thesis, Pontífica Universidade Católica do Rio de Janeiro - PUC-RIO.

Gonçalves, F. A. S. (2005). *Uma abordagem de desenvolvimento de linha de produtos orientada a modelos para a construção de famílias de sistemas embarcados críticos*. Tese de doutorado, Universidade Estadual Paulista-UNESP.

He, M.; Klein, J.-O. e Belhaire, E. (2008). Design and electrical simulation of on-chip neural learning based on nanocomponents. *Electronics Letters*, 44(9):575–576.

Hess, K. R.; Anderson, K.; Symmans, W. F.; Valero, V.; Ibrahim, N.; Mejia, J. A.; Booser, D.; Theriault, R. L.; Buzdar, A. U.; Dempsey, P. J. et al. (2006). Pharmacogenomic predictor of sensitivity to preoperative chemotherapy with paclitaxel and fluorouracil, doxorubicin, and cyclophosphamide in breast cancer. *Journal of clinical oncology*, 24(26):4236–4244.

Holanda, P. C. (2013). Desenvolvimento de uma rede neural multilayer perceptron embarcada em fpga. Master's thesis, Universidade Federal do Ceará.

Holler, M.; Tam, S.; Castro, H. e Benson, R. (1989). An electrically trainable artificial neural network (etann) with 10240 floating gate synapses. In *International Joint Conference on Neural Networks*, volume 2, pp. 191–196.

Hussain, H. M.; Benkrid, K. e Seker, H. (2012). An adaptive implementation of a dynamically reconfigurable k-nearest neighbour classifier on fpga. pp. 205–212.

Hussain, H. M.; Benkrid, K.; Seker, H. e Erdogan, A. T. (2011). Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pp. 248–255. IEEE.

James Theiler, M. L.; Estlick, M. e Szymanski, J. J. (2000). Design issues for hardware implementation of an algorithm for segmenting hyperspectral imagery. In *proceedings of SPIE*, volume 4132, p. 100.

Khaitan, S. K. e McCalley, J. D. (2015). Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365.

Lee, E. A. e Seshia, S. A. (2016). *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd ediï£¡ï£¡o.

Lichman, M. (2013). UCI machine learning repository.

Manolakos, E. S. e Stamoulias, I. (2010a). Ip-cores design for the knn classifier. pp. 4133–4136.

Manolakos, E. S. e Stamoulias, I. (2010b). Ip-cores design for the knn classifier. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 4133–4136. IEEE.

Mohammed, E. Z. e Ali, H. K. (2013). Hardware implementation of artificial neural network using field programmable gate array. *International Journal of Computer Theory and Engineering*, 5(5):780.

Muthuramalingam, A.; Himavathi, S. e Srinivasan, E. (2008). Neural network implementation using fpga: issues and application. *International journal of information technology*, 4(2):86–92.

Oliveira, Caio Augusto, A. J. A. e. F. M. G. S. (2010). Dipositivos lógicos programáveis.

Qin, S. e Berekovic, M. (2015). A comparison of high-level design tools for soc-fpga on disparity map calculation example. *CoRR*, abs/1509.00036.

Queiroz, P. G. G. (2015). *Uma abordagem de desenvolvimento de linha de produtos orientada a modelos para a construção de famílias de sistemas embarcados críticos*. Tese de doutorado, ICMC/USP.

Silva, R.; Santos Filho, D. e Miyagi, P. (2015). Modelagem de sistemas de controle da insdústria 4.0 baseada em holon, agente, rede de petri e arquitetura orientada a serviços. *XII SBAI*, 12.

Smith, M. J. S. (1997). *Application-specific Integrated Circuits*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Stallings, W. (2010). *Arquitetura e Organização de Computadores*. Pearson Pratice Hall, São Paulo, 8.ed ediï£¡ï£¡o.

Stamoulias, I. e Manolakos, E. S. (2013a). Parallel architectures for the knn classifier – design of soft ip cores and fpga implementations. *ACM Trans. Embed. Comput. Syst.*, 13(2):22:1–22:21.

Stamoulias, I. e Manolakos, E. S. (2013b). Parallel architectures for the knn classifier–design of soft ip cores and fpga implementations. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):22.

tagkey2015iii (2015). Front matter. In Yiu, J., editor, *The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors (Second Edition)*, pp. iii –. Newnes, Oxford, second edition ediï£¡ï£¡o.

Torres, L.; Castro, C.; Coelho, F.; Torres, F. S. e Braga, A. (2015). Distance-based large margin classifier suitable for integrated circuit implementation. *Electronics Letters*, 51(24):1967–1969.

Torres, L. C. B. (2016). *Classificador por arestas de suporte(CLAS): Métodos de aprendizado baseados em grafos de Gabriel*. Tese de doutorado, UFMG.

Yiu, J. (2013). *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*. Newnes, Newton, MA, USA, 3rd ediï£¡ï£¡o.