UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Proposta de implementação em Hardware de Rede Neural Profunda baseada em Stacked Sparse Autoencoder

Maria Gracielly Fernandes Coutinho

Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes

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

Número de ordem PPgEEC: M543
Natal, RN, janeiro de 2019

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

Coutinho, Maria Gracielly Fernandes.
Proposta de implementação em hardware de rede neural profunda baseada em Stacked Sparse Autoencoder / Maria Gracielly Fernandes Coutinho. - Natal, 2019.
71 f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Programa de Pós-Graduação em Engenharia Elétrica e de Computação. Natal, RN, 2019.
Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes.

1. Rede neural - Dissertação. 2. Aprendizagem profunda - Dissertação. 3. Stacked Sparse Autoencoder - Dissertação. 4. FPGA - Dissertação. 5. Matriz sistólica - Dissertação. I. Fernandes, Marcelo Augusto Costa. II. Título.

RN/UF/BCZM CDU 004.032.26(043.3)

Elaborado por FERNANDA DE MEDEIROS FERREIRA AQUINO - CRB-15/301

Universidade Federal do Rio Grande do Norte

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

ATA Nº 543

ATA Nº 543

Aos dezessete dias do mês de janeiro do ano de dois mil e dezenove, foi realizada a 543ª sessão de defesa de dissertação de mestrado do Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN, na qual a mestranda Maria Gracielly Fernandes Coutinho apresentou o trabalho que tem como título: Proposta de implementação em Hardware de Rede Neural Profunda baseada em Stacked Sparse Autoencoder. A sessão teve início às 13h30min, tendo a banca examinadora sido constituída pelos seguintes participantes: Marcelo Augusto Costa Fernandes (Dr. UFRN, Presidente), Carlos Alberto Valderrama Sakuyama (Dr. UMONS, Examinador Externo à Instituição), Diomadson Rodrigues Belfort (Dr. UFRN, Examinador Interno à Instituição e Adriao Duarte Doria Neto (Dr. UFRN, Examinador Interno à Instituição). Após a apresentação do trabalho e o exame pela banca, o mestrando foi considerado APROVADO, tendo sido lavrada a presente ata, que vai assinada pelos examinadores e pelo mestrando. A versão final da dissertação deverá ser entregue ao programa, no prazo máximo de 60 dias, contendo as modificações sugeridas pela banca examinadora e constante na folha de correção anexa. Conforme o Artigo 49 da Resolução 197/2013 - CONSEPE, o candidato não terá o título se não cumprir as exigências acima.

Dr. CARLOS ALBERTO VALDERRAMA SAKUYAMA, UMONS

Examinador Externo à Instituição

Dr. ADRIAO DUARTE DORIA NETO, UFRN

Examinador Interno

Dr. DIOMADSON RODRIGUES BELFORT, UFRN

Examinador Interno

Dr. MARCELO AUGUSTO COSTA FERNANDES, UFRN

Presidente

MARIA GRACIELLY FERNANDES COUTINHO

A Deus, realizador de todas as
coisas, e à Maria Santíssima,
medianeira de todos os prodígios.

Agradecimentos

A Deus, por toda força, graça e sabedoria que me fizeram chegar até aqui. Sem Ele, eu nada seria.

Ao meu orientador, professor Dr. Marcelo Augusto Costa Fernandes, pela confiança em mim depositada, pela orientação e por todo apoio durante a realização deste trabalho.

Aos colegas do Grupo de Pesquisa em Sistemas Embarcados e Computação Reconfigurável, por tantos momentos, experiências e conhecimentos compartilhados.

Aos meus familiares, sobretudo, aos meus pais, Graça e Antônio, e às minhas irmãs, Gracineide e Gracinalda, pela compreensão e pelo apoio de cada dia.

Aos meus sobrinhos, Mariana e Miguel, por tornarem meus dias mais coloridos.

A todos os amigos que torceram por mim e que de alguma forma fizeram parte desta história, em especial à Thalita e Aline, pelo companheirismo na convivência diária.

À Universidade Federal do Rio Grande do Norte, por me proporcionar uma formação de qualidade.

À CAPES, pelo apoio financeiro.

Resumo

As técnicas de aprendizagem profunda (*Deep Learning*) aplicáveis a problemas de diversas áreas vêm ganhando grande destaque no âmbito da pesquisa mundial nos últimos anos. No entanto, os algoritmos com aprendizagem profunda possuem um custo computacional elevado, dificultando sua utilização em várias aplicações comerciais. Por outro lado, novas alternativas vêm sendo estudadas para acelerar algoritmos complexos, e entre elas, as baseadas em computação reconfigurável vêm apresentando resultados bastante significativos. Sendo assim, este trabalho tem como objetivo a implementação em hardware de uma rede neural para utilização de algoritmos com aprendizagem profunda. O hardware proposto foi desenvolvido em *Field Programmable Gate Array* (FPGA) e suporta Redes Neurais Profundas (*Deep Neural Network* - DNN) treinadas com a técnica *Stacked Sparse Autoencoder* (SSAE). Para permitir DNNs com muitas entradas e camadas no FPGA, foi utilizada a técnica de matriz sistólica (*systolic array*) em todo hardware desenvolvido. Os detalhes da arquitetura desenvolvida no FPGA são evidenciados, bem como, os dados de ocupação em hardware, o tempo de processamento e o consumo de potência para duas implementações distintas. Resultados mostram que as implementações conseguem atingir *throughputs* elevados, permitindo a utilização de técnicas de *Deep Learning* em problemas de dados massivos.

Palavras-chave: Aprendizagem Profunda, Stacked Sparse Autoencoder, FPGA, Matriz Sistólica.

Abstract

The deep learning techniques have been gaining prominence in world research in the past years. However, the deep learning algorithms have high computational cost, making it hard to apply in several commercial applications. On the other hand, new alternatives have been studying to accelerate complex algorithms, among these, those based on reconfigurable hardware has been showing very significant results. Therefore, the objective of this work is the hardware implementation of a neural network for the use of algorithms with deep learning. The hardware was developed on Field Programmable Gate Array (FPGA) and supports Deep Neural Network (DNN) trained with the Stacked Sparse Autoencoder (SSAE) technique. In order to allow DNNs with many inputs and layers on the FPGA, the systolic array technique was used in all developed hardware. The details of the architecture designed on the FPGA were evidenced, as well as the occupation data on hardware, the processing time and the power consumption to two different implementations. The results show that both implementations achieve high throughputs allowing the use of Deep Learning techniques in massive data problems.

Keywords: Deep Learning, Stacked Sparse Autoencoder, FPGA, Systolic Array.

Sumário

Sumário i

Lista de Figuras iii

Lista de Tabelas v

Lista de Siglas e Abreviaturas vii

1 Introdução 1
1.1 Principais Contribuições 2
1.2 Artigos publicados 3
1.3 Organização do Trabalho 3

2 Trabalhos relacionados 5
2.1 Implementações de técnicas de IA em FPGA 5
2.2 Implementações de técnicas de DL em FPGA 6
2.3 Implementações de técnicas de AE em FPGA 7

3 Computação Reconfigurável 11
3.1 Introdução à Computação Reconfigurável 11
3.2 *Field Programmable Gate Array* 12
3.2.1 Especificação de circuitos em FPGA 13
3.2.2 Consumo de Potência 15

4 Aprendizagem Profunda 17
4.1 Contextualização Histórica 17
4.2 Técnicas de Aprendizagem Profunda 18
4.2.1 *Autoencoder* 19
4.2.2 *Stacked Sparse Autoencoder* 21

5 Descrição da proposta 23
5.1 Proposta 1 23
5.1.1 Camadas do SSAE 24
5.1.2 Unidades de Processamento 25
5.1.3 Funções de Ativação 26
5.1.4 Tempo de Processamento 26
5.2 Proposta 2 28

5.2.1 Camadas do SSAE 28
5.2.2 Unidades de Processamento 29
5.2.3 Tempo de Processamento 30

6 Resultados 33
6.1 Análise de Ocupação em Hardware 33
6.2 Validação 36
6.2.1 Validação da Proposta 1 37
6.2.2 Validação da Proposta 2 41
6.3 Comparação com o estado da arte 42

7 Considerações Finais 45
7.1 Conclusões 45
7.2 Perspectivas Futuras 46

Referências bibliográficas 47

Lista de Figuras

3.1 Comparação de dispositivos de hardware quanto a velocidade e flexibilidade. 12
3.2 Estrutura simplificada de um FPGA. 13
3.3 Estrutura dos blocos lógicos configuráveis básicos (CLBs) dentro de interconexões programáveis globais em linha/coluna. 14
3.4 Estrutura do módulo lógico. 15
3.5 Fases do processo de design de um circuito em FPGA. 15
3.6 Fluxo do processo de síntese do circuito em FPGA. 16

4.1 Ilustração de um modelo de *Deep Learning*. 19
4.2 Exemplo de uma estrutura de *Autoencoder*. 20
4.3 Arquitetura do *Stacked Sparse Autoencoder* proposto. 21

5.1 Arquitetura geral da proposta 1. 24
5.2 Arquitetura da k-ésima camada oculta do SSAE da proposta 1. 25
5.3 Arquitetura do i-ésimo PE_i^1 da proposta 1. 26
5.4 Arquitetura do i-ésimo PE_i^k, para $k > 1$, da proposta 1. 27
5.5 Arquitetura da k-ésima função de ativação, FA^k, associada as camadas ocultas. 27
5.6 Arquitetura geral da proposta 2. 29
5.7 Arquitetura da k-ésima camada oculta do SSAE da proposta 2. 30
5.8 Arquitetura do i-ésimo PE_i^1 da proposta 2. 30
5.9 Arquitetura do i-ésimo PE_i^k, para $k > 1$, da proposta 2. 31

6.1 Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 1 ($k = 1$). 34
6.2 Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 1 ($k \geq 2$). 35
6.3 Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 2 ($k = 1$). 37
6.4 Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 2 ($k \geq 2$). 38
6.5 Escalabilidade da proposta 1 com relação ao número de PEs (n_{PE}). 39
6.6 Escalabilidade da proposta 2 com relação ao número de PEs (n_{PE}). 40
6.7 Exemplos de imagens da base de dados MNIST. 41

Lista de Tabelas

6.1 Ocupação em hardware (células lógicas e multiplicadores) por PE para a proposta 1. 33
6.2 Ocupação em hardware (células lógicas e multiplicadores) por PE para a proposta 2. 36
6.3 Estimativa de n_{PE} para alguns FPGAs comerciais. 36
6.4 Síntese da proposta 1 - Taxa de ocupação 39
6.5 Síntese da proposta 1 - Associação ao tempo de processamento 40
6.6 Síntese da proposta 2 - Taxa de ocupação 41
6.7 Síntese da proposta 2 - Associação ao tempo de processamento 42
6.8 Comparação da proposta 2 com um trabalho relacionado 43
6.9 Comparação da proposta 1 com um trabalho relacionado 43

Lista de Siglas e Abreviaturas

AE *Autoencoder*

ASIC Circuito Integrado de Aplicação Específica, do inglês *Application Specific Integrated Circuit*

CL Célula Lógica

CLB Bloco Lógico Configurável, do inglês *Configurable Logic Block*

CNN Rede Neural Convolucional, do inglês *Convolutional Neural Network*

CPLD *Complex Programmable Logic Device*

CPU Unidade Central de Processamento, do inglês *Central Processing Unit*

CR Computação Reconfigurável

DL *Deep Learning*

DNN Rede Neural Profunda, do inglês *Deep Neural Network*

DSP Processamento Digital de Sinais, do inglês *Digital Signal Processing*

FA Função de Ativação

FPGA *Field Programmable Gate Array*

FPS *Frames* por segundo

GPP Processador de uso geral, do inglês *General Purpose Processor*

GPU Unidade de Processamento Gráfico, do inglês *Graphics Processing Unit*

HDL Linguagem de Descrição de Hardware, do inglês *Hardware Description Language*

HLS *High Level Synthesis*

HPC Computador de Alto Desempenho, do inglês *High Performance Computer*

IA Inteligência Artificial

LUT *Lookup Table*

ML	Aprendizado de Máquina, do inglês *Machine Learning*
MLP	*Multilayer Perceptron*
MSE	Erro Quadrático Médio, do inglês *Mean Square Error*
PE	*Processing Element*
PLD	Dispositivo Lógico Programável, do inglês *Programmable Logic Devices*
RNA	Rede Neural Artificial
ROM	*Read-only Memory*
RTL	*Register Transfer Level*
SCG	Gradiente Conjugado Escalonado, do inglês *Scaled Conjugate Gradient*
SoC	*System on Chip*
SSAE	*Stacked Sparse Autoencoder*

Capítulo 1

Introdução

Tendo em vista a crescente utilização de técnicas de Inteligência Artificial (IA) para resolução de problemas de diversas áreas, as técnicas de Aprendizagem Profunda (*Deep Learning* - DL), também conhecidas como Redes Neurais Profundas (*Deep Neural Network* - DNN), vêm ganhando grande destaque nos últimos anos. As DNNs são capazes de prover alto poder computacional, concomitantemente com a utilização de várias camadas ocultas. Isso ocorre porque a profundidade destas redes permite aprender uma sequência de funções que realizam a transformação de vetores, mapeando-os de um espaço a outro, até que se atinja o resultado pretendido (Ponti & da Costa 2018).

As DNNs podem ser aplicadas a problemas de classificação, predição, reconhecimento facial, detecção de objetos em imagens, extração de características, entre muitos outros. Entre as várias técnicas de *Deep Learning* existentes na literatura, encontram-se as baseadas em *Autoencoders* (AEs), aplicadas principalmente a problemas de predição e classificação (Baldi 2012, Deng et al. 2014, Schmidhuber 2015).

A técnica de utilizar vários *autoencoders* encadeados, formando uma única DNN, vem se mostrando muito útil no treinamento de redes de aprendizagem profunda, como pode ser visto em Vincent et al. (2010) e Maria et al. (2016). Os *Stacked Sparse Autoencoders* (SSAE) vêm sendo utilizados em problemas de classificação (Maria et al. 2016). Nesta abordagem, cada camada oculta é composta por um *sparse autoencoder* treinado individualmente, de forma não supervisionada. A saída da camada oculta de cada AE é utilizada como entrada no AE seguinte, de modo que as características dos dados de entrada sejam propagadas pela rede camada a camada, possibilitando que a camada de saída seja capaz de realizar a classificação dos dados, após um treinamento supervisionado.

Muitos trabalhos da literatura vêm explorando a eficiência dos SSAEs em problemas emergentes, como reconhecimento de expressão facial (Liu et al. 2014), aplicações da indústria, como o diagnóstico de falhas de máquinas rotativas (Qi et al. 2017), além de aplicações da área médica, como análise de imagens histopatológicas (Zhang et al. 2016), detecção de núcleos em imagens de histopatologia de câncer de mama (Xu et al. 2016), detecção de glaucoma em imagens de fundo (Pratiher et al. 2018), entre outros.

Entretanto, as redes neurais profundas possuem uma complexidade computacional elevada em decorrência da grande quantidade de camadas ocultas adicionadas, o que dificulta ou até inviabiliza sua utilização em várias aplicações comerciais. Este problema se agrava ainda mais ao aplicarmos estas técnicas em problemas de dados massivos. Em

contrapartida, vários trabalhos voltados para acelerar algoritmos complexos vêm sendo desenvolvidos, e entre eles a utilização de computação reconfigurável (CR) tem se mostrado uma ótima alternativa.

A computação reconfigurável permite o desenvolvimento de arquiteturas de hardware personalizáveis adaptadas aos algoritmos, de maneira oposta da Unidade Central de Processamento (*Central Processing Unit* - CPU) tradicional ou do hardware de uso geral (Deotale & Dole 2014). Essa abordagem permite que os algoritmos sejam paralelizados e otimizados, resultando em ganho de desempenho e até consumo de energia.

Os *Field Programmable Gate Array* (FPGA) são dispositivos revolucionários que combinam os benefícios do hardware e do software (Hauck & DeHon 2010). Estes dispositivos podem ser reprogramados de maneira barata e fácil e permitem alcançar desempenho muito superior à implementações em software. A utilização de FPGAs pode trazer resultados significativos com relação à ocupação de área de hardware, consumo de energia, entre outros benefícios. Diversos trabalhos da literatura apresentam a adoção de implementações em FPGA para técnicas de inteligência artificial, como apresentado em de Souza & Fernandes (2014) e Torquato (2017), nos quais foi possível atingir elevados ganhos de velocidade em comparação com implementações em processadores gráficos (*Graphics Processing Unit* - GPU) e processadores de uso geral utilizados em computadores de alto desempenho (*High Performance Computer* - HPC).

No entanto, apesar dos SSAEs mostrarem-se muito eficientes para resolução de diversos problemas, sua implementação em hardware ainda é pouco difundida, o que indica a importância de explorar as possibilidades de acelerar esta técnica. Sendo assim, este trabalho apresenta uma proposta de implementação em FPGA de uma Rede Neural Profunda, utilizando *Stacked Sparse Autoencoder*. Todo o projeto foi implementado em *Register Transfer Level* (RTL), utilizando ponto fixo e a técnica de matriz sistólica, o que possibilitou utilizar recursos reduzidos de área de hardware e alcançar *throughputs* elevados que indicam a viabilidade da utilização desta técnica de *Deep Learning* em problemas de dados massivos. Não obstante, a estrutura proposta neste trabalho poderia ser evoluída para outras técnicas de aprendizagem profunda, como as Redes Neurais Convolucionais (*Convolutional Neural Networks* - CNNs), sem grandes dificuldades.

1.1 Principais Contribuições

A principal contribuição deste trabalho trata-se de uma proposta de implementação em hardware de uma Rede Neural Profunda, baseada na técnica *Stacked Sparse Autoencoder*, até então pouco explorada utilizando computação reconfigurável. O hardware foi desenvolvido apenas para a fase *feedforward*, tendo em vista que a etapa de treinamento ocorre uma única vez, o que viabiliza sua execução de forma prévia separadamente. Sendo assim, os pesos obtidos no treinamento prévio são inseridos no hardware, que realiza a etapa de inferência. Em todo o hardware foi utilizada a técnica de matriz sistólica, o que possibilitou a obtenção dos resultados de saída da rede em baixo tempo de processamento, além de garantir a baixa ocupação de área de hardware.

Diferentemente de muitos trabalhos relacionados da literatura, a implementação proposta aqui foi desenvolvida em *Register Transfer Level*, o que permitiu um maior con-

trole e paralelização do circuito, tornando a implementação em FPGA bastante vantajosa. Além disso, a implementação utilizou resolução em ponto fixo, que possibilita alcançar menor consumo de recursos e menor consumo de potência que uma implementação equivalente em ponto flutuante (Finnerty & Ratigner 2017).

Os *throughputs* elevados alcançados viabilizam a utilização desta técnica de *Deep Learning* em problemas de dados massivos. Também foi possível obter baixo consumo de potência, outro fator de extrema importância em diversas aplicações. Ademais, uma análise de ocupação em hardware mostrou a possibilidade de adicionar arquiteturas de SSAEs com muitas camadas e neurônios, viabilizando sua utilização em aplicações de problemas reais.

1.2 Artigos publicados

1. Maria Gracielly F. Coutinho e Marcelo A. C. Fernandes. "Proposta de Implementação de Rede Neural Artificial em Hardware para Aprendizagem Profunda". *Em: XXIII Congresso Brasileiro de Automática (CBA 2018 - João Pessoa, PB).*
2. Maria G. F. Coutinho e Marcelo A. C. Fernandes. "Proposta de Implementação em Hardware de Rede Neural Profunda baseada em *Stacked Sparse Autoencoder*". *Em: VIII Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC 2018 - Salvador, BA).*

1.3 Organização do Trabalho

A presente dissertação está organizada em 7 capítulos, conforme apresentado nos parágrafos a seguir.

Neste primeiro Capítulo foi apresentada uma introdução, em que a motivação e o tema do trabalho são contextualizados, além disso foram evidenciadas as principais contribuições desta pesquisa e os artigos publicados.

No Capítulo 2 serão apresentados alguns trabalhos relacionados e do estado da arte, que dizem respeito à implementações em FPGA para acelerar técnicas de Inteligência Artificial, *Deep Learning* e técnicas baseadas em *Autoencoders*.

No Capítulo 3 será exibida uma fundamentação teórica à respeito da Computação Reconfigurável no contexto dos Dispositivos Lógico Programáveis, com foco na utilização de FPGAs.

No Capítulo 4 será exposta uma fundamentação teórica à respeito das técnicas de aprendizagem profunda, iniciando com uma contextualização histórica acerca da inteligencia artificial até o surgimento da *Deep Learning*, colocando em evidência a técnica de DNN, *Stacked Sparse Autoencoder*, implementada neste trabalho.

No Capítulo 5 serão apresentados os detalhes da arquitetura em hardware para a fase *feedforward* de duas propostas de SSAE, utilizando ponto fixo e a técnica de matriz sistólica. Além disso, serão detalhadas as informações com relação ao tempo de processamento de cada implementação.

O Capítulo 6 irá expor uma análise de ocupação em hardware, além dos resultados da validação do hardware proposto, incluindo os resultados de síntese de cada implementação, referentes a taxa de ocupação dos recursos de hardware do FPGA alvo, ao tempo de processamento e ao consumo de potência de cada circuito. Além disso, serão apresentadas comparações das propostas implementadas com um trabalho do estado da arte.

Por fim, no Capítulo 7 serão realizadas as considerações finais, mostrando as conclusões sobre os resultados obtidos e as perspectivas futuras com a finalização deste trabalho.

Capítulo 2

Trabalhos relacionados

Este capítulo destina-se a evidenciar alguns trabalhos relacionados e do estado da arte, divididos em três categorias: trabalhos que envolvem implementações de técnicas de Inteligência Artificial (IA) em FPGA, trabalhos que envolvem implementações de técnicas de *Deep Learning* (DL) em FPGA e trabalhos que envolvem implementações de técnicas baseadas em *Autoencoder* (AE) em FPGA.

2.1 Implementações de técnicas de IA em FPGA

É possível encontrar na literatura uma grande variedade de trabalhos com implementações em hardware para algoritmos de Inteligência Artificial. Como é o caso dos trabalhos desenvolvidos em de Souza & Fernandes (2014), Silva (2015) e Torquato (2017).

Em de Souza & Fernandes (2014) foi implementada uma Rede Neural Artificial (RNA) em hardware, do tipo Funções de Base Radial (RBF), treinada com o algoritmo *Least Mean Square* (LMS). A arquitetura implementada em RTL foi analisada em termos de taxa de ocupação do hardware, resolução em bits e atraso de processamento. Os resultados das sínteses para dois cenários distintos e várias resoluções de ponto fixo apontam para a possibilidade de utilização da proposta em situações práticas mais complexas.

Em Silva (2015), foi proposto a implementação de uma matriz de neurônios dinamicamente e parcialmente reconfigurável para a descrição de topologias de RNAs *Multilayer Perceptron* (MLP) em FPGA. O sistema permitiu, além da realimentação e a reutilização de neurônios, realizar a configuração de várias topologias de redes. Para isso utilizou-se a ferramenta PlanAhead® da Xilinx®, que permite a criação de partições reconfiguráveis no FPGA. A implementação foi construída parte em VHDL e parte utilizando os blocos do DSP Buider da Xilinx, para um FPGA Virtex-6 XC6VLX240T.

Já no trabalho de Torquato (2017) foi proposto uma implementação paralela de algoritmo genético, em FPGA. A arquitetura foi implementada em RTL e compreendeu testes com diferentes parâmetros. As funções utilizadas pelo algoritmo genético foram armazenadas em *Lookup Tables* (LUTs), eliminando assim a necessidade de um circuito dedicado para realizar cada função e, consequentemente, permitiu a redução da área ocupada no FPGA e tornou a implementação mais flexível, além disso, proporcionou o aumento do *throughput*, já que construir um circuito complexo para estas funções aumentaria o caminho crítico da implementação.

2.2 Implementações de técnicas de DL em FPGA

Diversos trabalhos envolvendo implementações de técnicas de *Deep Learning* em FPGA são encontrados na literatura. No trabalho de Zhou & Jiang (2015) foi proposto a implementação de uma Rede Neural Convolucional (*Convolutional Neural Network* - CNN) em FPGA, utilizando ponto fixo. O circuito foi construído com o auxílio da ferramenta Vivado HLS. A implementação em hardware, utilizando um FPGA Virtex 7, conseguiu operar cerca de 16,42 vezes mais rápido que uma CNN implementada em Matlab executando em um processador Intel i7-4790K.

Zhang et al. (2015) apresentou a implementação de um acelerador de CNNs em FPGA de forma otimizada. Para isso foi adotado um esquema de projeto analítico utilizando o modelo *roofline*, no entanto, a implementação do hardware foi desenvolvida utilizando *high level synthesis* (HLS). A CNN implementada utilizou como FPGA alvo um Virtex7, e obteve um speedup de 17,42x em comparação com a implementação em um processador de uso geral Intel Xeon E5-2430 2.2GHz. Já no trabalho de Gupta et al. (2015) foram evidenciados os efeitos da utilização de representação de dados com precisão limitada em treinamento de redes neurais. Os resultados mostraram que ao utilizar arredondamento estocástico, com uma resolução de apenas 16 bits em ponto fixo, foi possível obter desempenho aproximado ao obtido utilizando 32 bits em ponto flutuante. O trabalho utilizou a estratégia de matriz sistólica para construir matrizes de multiplicadores, contribuindo assim para o aumento do *throughput* da implementação.

Outra arquitetura para acelerar técnicas de DL em FPGA foi proposta em Sharma et al. (2016). Neste trabalho foi desenvolvido o framework DNNWeaver, que permite acelerar DNNs implementadas através do Caffe, um framework de *deep learning* de código aberto utilizado para especificação de DNNs para execução em GPUs e CPUs. O DNNWeaver foi testado utilizando três diferentes FPGAs (Xilinx Zynq, Altera Stratix V e Altera Arria 10), e os resultados obtidos foram comparados com os resultados para duas CPUs multicore e três GPUs many-core. Os FPGAs obtiveram desempenho superior com relação à performance, além disso foi possível obter melhores resultados com relação a performance por watt para dois dos três FPGAs utilizados.

Wang et al. (2017) propôs uma unidade aceleradora de aprendizagem profunda em FPGA, chamada de DLAU. A arquitetura desenvolvida foi dividida em 3 unidades de processamento que utilizam pipeline e podem ser aplicadas a diferentes topologias de rede. Outra estratégia adotada foi particionar os massivos dados de entrada em segmentos menores. O hardware foi desenvolvido utilizando a plataforma de desenvolvimento Xilinx Zynq Zedboard, que possui um processador ARM Cortex-A9. A estrutura permite que diferentes tipos de topologias de redes sejam implementadas. Os experimentos mostraram que o DLAU obteve um *speedup* de 36,1x em comparação com a implementação em um processador de uso geral Intel Core2 2.3GHz, além de alcançar baixa potência em comparação com a implementação em uma GPU NVIDIA Tesla K40c.

Uma outra proposta de implementação de CNN em FPGA é apresentada em Bettoni et al. (2017). O trabalho utilizou *high level synthesis*. Os experimentos mostraram que a implementação em FPGA alcançou até 3 vezes mais eficiência energética que uma implementação em CPU e eficiência energética equivalente a implementação executando

em 16 *threads* na mesma CPU. Além disso, com relação a implementação em SoC GPU para aplicações mobile, foi possível obter um ganho de quase 15 vezes em termos de velocidade e 16 vezes em termos de eficiência energética.

Em Huynh (2017) foi proposto uma implementação de DNN em FPGA, utilizando VHDL e ponto flutuante, que viabilizava a utilização de uma única camada de computação física para executar todo o *feedforward* da rede. Foram realizadas sínteses para diversas arquiteturas de rede, onde constatou-se a maior arquitetura de rede suportada por cada hardware utilizado nos experimentos. No FPGA Xilinx Virtex-5 XC5VLX-110T foi possível adicionar a arquitetura $784-40-40-40-10$, e no FPGA Xilinx ZynQ-7000 XC7Z045, a arquitetura $784-126-126-126-10$. No entanto, observou-se que com as arquiteturas $784-40-40-10$ e $784-126-126-10$ foram obtidos os melhores resultados em termos de taxa de reconhecimento e tempo de processamento, alcançando um desempenho de até 15.810 e 15.900 *frames* de dígitos manuscritos (do dataset MNIST) por segundo, respectivamente.

No trabalho de Zhang et al. (2018) foi apresentada uma proposta para acelerar CNNs e DNNs, utilizando FPGA, através de uma biblioteca baseada em hardware/software, chamada de Caffeine, construída utilizando HLS. A implementação possibilita que o hardware seja configurado através do software, permitindo ainda que seja integrado a frameworks de *deep learning*, como o Caffe. Neste trabalho, a técnica de matriz sistólica foi adotada em partes do design, com o objetivo de proporcionar escalabilidade. Os resultados mostraram um *speedup* de 7,3x com relação a performance e 43,5x com relação a energia em comparação com uma CPU 12-core.

Já no trabalho de Ma et al. (2018) foi proposto um compilador RTL em FPGA para acelerar algoritmos de *Deep Learning*, que visava alcançar desempenho semelhante às implementações em RTL, tendo em vista que implementações que utilizam *high level synthesis* não conseguem otimizar o consumo dos recursos do FPGA. Comparações com trabalhos da literatura deixam claro que o compilador RTL proposto consegue resultados bastante superiores às implementações com HLS.

No trabalho de Zhao et al. (2018) foi implementado em FPGA, utilizando Verilog, uma rede Auto-encoder Convolucional (*Convolutional auto-encoder* - CAE), que corresponde a um tipo de CNN. A implementação em FPGA foi validada efetuando compressão de imagens. Foram realizadas comparações da implementação em hardware com implementações em CPU e GPU. A velocidade de compressão no FPGA foi superior a velocidade na CPU e inferior a velocidade na GPU. No entanto, com relação a performance por watt, o FPGA foi bastante superior em ambos os casos.

2.3 Implementações de técnicas de AE em FPGA

Uma proposta de implementação de rede tradicional utilizando *autoencoder* é apresentada no trabalho de Jin & Kim (2014), em que foi implementada uma arquitetura de *sparse autoencoder*, em *VerilogHDL*. Os testes foram realizados com uma rede de arquitetura $196-100-196$, utilizando o conjunto de dados de imagens naturais da cidade de Kyoto no Japão.

Uma das primeiras implementações de DNN em FPGA utilizando *Stacked Sparse Autoencoders*, é apresentada no trabalho de Maria et al. (2016). A DNN foi utilizada para realizar classificação de imagens do conjunto de dados chamado de CIFAR-10. A arquitetura da rede utilizou duas camadas escondidas, contendo 2000 neurônios na primeira camada escondida e 750 na segunda, além de 3072 entradas e 10 neurônios na camada de saída (chamada de arquitetura $3072-2000-750-10$). Para a implementação foi aplicado um *framework* de *high level synthesis* em OpenCL, o que possivelmente contribuiu para que os resultados para o FPGA, em termos de velocidade, não fossem superiores quando comparados a implementação em GPUs.

Outra proposta de implementação de SSAEs em FPGA é apresentada no trabalho de Jiang et al. (2016), onde são mostradas as vantagens da utilização de ponto fixo em implementações de *autoencoder*, tendo em vista que, ao utilizar uma resolução de apenas 4 bits na parte inteira foi possível obter a mesma acurácia da classificação da resolução em ponto flutuante. Para os experimentos foi utilizada a arquitetura de rede $784-400-10$ e a base de dados MNIST. No entanto, não foram apresentadas informações referentes a taxa de ocupação dos recursos de hardware do FPGA, e ao tempo de processamento.

Nos trabalhos de Suzuki et al. (2018) e Moss et al. (2018) são apresentadas propostas de redes tradicionais utilizando *autoencoders*. No trabalho de Suzuki et al. (2018) foi proposta, além de arquiteturas tradicionais de autoencoders, uma estrutura com dois *autoencoders* encadeados, a arquitetura $4-2-1-2-4$, no entanto, objetivando apenas reconstruir a entrada na saída, como nos modelos convencionais. Neste trabalho, diferentemente dos trabalhos que implementaram técnicas de *autoencoders* citados anteriormente, o circuito foi implementado em RTL, o que possibilitou alcançar eficiência superior às demais implementações de *autoencoder* comparadas no trabalho. O FPGA alvo foi uma Xilinx Virtex-6 xc6vlx240t.

Semelhante a estrutura do trabalho de Suzuki et al. (2018), em Moss et al. (2018) utilizou-se *autoencoders* para implementar a arquitetura $32-16-8-16-32$. No entanto, neste trabalho foi adotada a ferramenta Xilinx Vivado HLS para efetuar a síntese da linguagem C para RTL. Um FPGA Xilinx Kintex-7 xc7k410tffg900-2 foi utilizado para os experimentos. A implementação conseguiu alcançar alto *throughput* (cerca de 200 Mega samples por segundo (MSPS)) e baixa latência (cerca de 105 *ns*). No entanto, observou-se que grande parte dos recursos de hardware do FPGA alvo foram ocupados, o que pode tornar inviável a utilização de uma arquitetura de rede maior que a implementada no trabalho.

Diante disto, verificou-se que a maioria dos trabalhos que utilizam computação reconfigurável para acelerar técnicas baseadas em *autoencoders*, exploram a utilização de modelos tradicionais destas redes, diferentemente da proposta aqui apresentada. Assim sendo, este trabalho apresenta uma proposta de implementação em hardware de uma Rede Neural Profunda baseada na técnica *Stacked Sparse Autoencoder*. Com o objetivo de garantir a eficiência da implementação em FPGA, através da otimização da utilização de seus recursos, a implementação proposta neste trabalho foi construída em RTL, diferentemente do trabalho de Maria et al. (2016), que utilizou HLS, assim como os trabalhos de Zhou & Jiang (2015), Zhang et al. (2015), Bettoni et al. (2017) e Zhang et al. (2018), que implementaram outras técnicas de DL em FPGA utilizando HLS. A implementação

em RTL também foi adotada em Suzuki et al. (2018), no entanto, neste trabalho foram implementadas estruturas de *autoencoders* mais semelhantes às tradicionais do que às DNNs.

O hardware proposto neste trabalho foi desenvolvido para a fase *feedforward* adotando a técnica de matriz sistólica, o que permitiu a utilização de muitos neurônios e várias camadas. Os trabalhos de Gupta et al. (2015) e Zhang et al. (2018), que implementaram outras técnicas de DL em FPGA, também utilizaram a técnica de matriz sistólica em partes do hardware. Neste trabalho, dados com relação a taxa de ocupação em hardware, ao tempo de processamento e a potência consumida serão apresentados para um FPGA Virtex 6 xc6vlx240t-1ff1156.

Capítulo 3

Computação Reconfigurável

Neste capítulo serão introduzidos os conceitos à respeito dos Dispositivos Lógicos Programáveis (*Programmable Logic Devices* - PLDs), onde estão inseridas as tecnologias que utilizam computação reconfigurável. As vantagens de utilizar esta estratégia, em comparação com outros dispositivos de hardware serão evidenciadas. Por fim, será apresentada a tecnologia de hardware reconfigurável adotada neste trabalho, os FPGAs.

3.1 Introdução à Computação Reconfigurável

Os avanços tecnológicos dos últimos anos vêm atingindo os mais variados setores da sociedade e revolucionando o modo como as pessoas vivem. Entretanto, a capacidade tecnológica para resolver problemas complexos, anteriormente intratáveis, vêm resultando em uma grande demanda de recursos computacionais, e no elevado consumo de energia. Diante disso, diversas estratégias vêm sendo adotadas para viabilizar a resolução de problemas complexos de forma rápida e com baixo consumo de energia, e entre elas, a utilização de Computação Reconfigurável (CR) vêm apresentando resultados bastante satisfatórios.

Segundo Todman et al. (2005), a utilização de computação reconfigurável pode atingir ganhos de velocidade de até 500 vezes e 70% de economia de energia, em comparação com implementações de microprocessadores para aplicações específicas. A Computação Reconfigurável está inserida no contexto dos Dispositivos Lógicos Programáveis e apresenta-se como uma alternativa para agregar a velocidade do hardware com uma flexibilidade semelhante a de implementações em software. Os PLDs são circuitos integrados que podem ser configurados pelo próprio usuário. Nessa abordagem, as funções lógicas são programadas pelo usuário, sem que haja a necessidade da fabricação do circuito integrado, o que facilita as possíveis mudanças de projeto (Costa 2009).

A Figura 3.1 apresenta uma comparação entre alguns dispositivos de hardware quanto a velocidade e flexibilidade. Em lados opostos estão os Circuitos Integrados de Aplicação Específica (*Application Specific Integrated Circuits* - ASICs) e os Processadores de uso geral (*General Purpose Processor* - GPP). Os ASICs são dispositivos projetados e construídos especificamente para uma determinada tarefa, capazes de alcançar maior velocidade, comprometendo o quesito flexibilidade. Já os processadores de uso geral são providos da maior flexibilidade entre os dispositivos, possibilitando a execução de qual-

quer função, no entanto, são incapazes de garantir velocidade, principalmente ao executar tarefas complexas. Desse modo, a utilização de dispositivos como PLDs, *Complex Programmable Logic Devices* (CPLDs) e FPGAs mostra-se como uma melhor solução mediante este problema (Azarian & Ahmadi 2009).

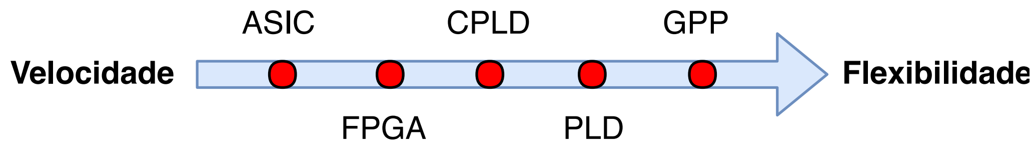

Figura 3.1: Comparação de dispositivos de hardware quanto a velocidade e flexibilidade. Fonte: Azarian & Ahmadi (2009).

Além disso, os dispositivos de lógica programável possuem um ciclo de projeto menor e custos reduzidos, em comparação com outras tecnologias de circuitos integrados digitais (Costa 2009). Segundo Kuon & Rose (2007), em muitas aplicações, os FPGAs têm todos os benefícios dos ASICs sem maiores custos de construção. Todas estas características vêm tornando a utilização dos FPGAs muito atrativa aos desenvolvedores de soluções em hardware e impulsionando diversas pesquisas ao redor do mundo.

3.2 *Field Programmable Gate Array*

Um *Field Programmable Gate Array* (FPGA) consiste em um grande arranjo de células lógicas ou blocos lógicos configuráveis contidos em um único circuito integrado, onde cada célula possui capacidade computacional para implementar funções lógicas e realizar roteamento para comunicação entre elas (Costa 2009). A Figura 3.2 apresenta a estrutura simplificada de um FPGA, que é composto basicamente por blocos lógicos configuráveis (*Configurable Logic Block* - CLBs), blocos de entrada e saída, e chaves de interconexão.

Os blocos lógicos integram uma matriz bidimensional, e as chaves de interconexão são organizadas como canais de roteamento horizontal e vertical entre as linhas e colunas dos blocos lógicos. Dessa forma, os canais de roteamento possuem chaves de interligação programáveis, capazes de conectar os blocos lógicos de acordo com as necessidades de cada projeto (Costa 2009). Além disso, os blocos de entrada e saída, dispostos nas extremidades, permitem realizar a comunicação com os sinais do mundo externo.

O primeiro FPGA comercial foi desenvolvido pela empresa Xilinx em 1983. Estes dispositivos evoluíram muito desde o seu surgimento. Os primeiros FPGAs eram compostos por apenas alguns milhares de blocos lógicos. Atualmente, os FPGAs mais modernos possuem milhões de blocos lógicos, além de blocos específicos para processamento digital de sinais (*Digital Signal Processing* - DSPs), blocos de memórias internas, entre outros blocos dedicados. Algumas plataformas dispõem ainda de processadores embutidos, como é o caso da Cyclone V SoC da Altera e da Zynq da Xilinx, que possuem processadores ARM integrados.

A Figura 3.3 exibe a arquitetura básica dos blocos lógicos configuráveis, dentro das interconexões programáveis globais em linha/coluna que são usadas para interconectar

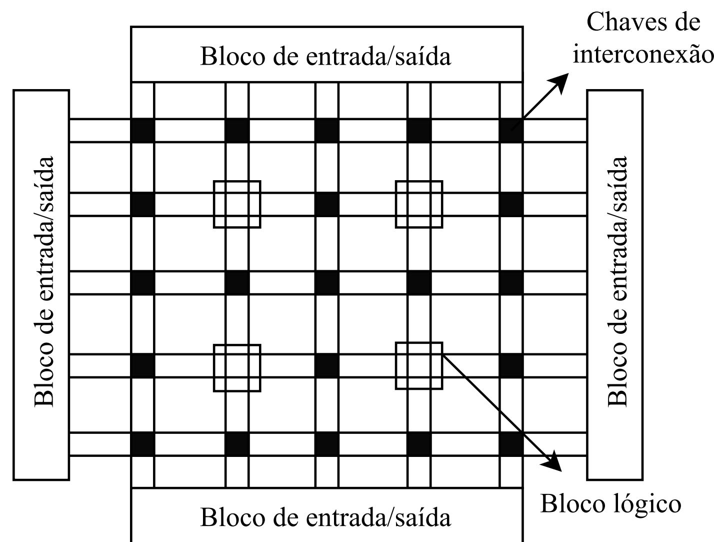

Figura 3.2: Estrutura simplificada de um FPGA.
Fonte: Costa (2009).

blocos lógicos.

Cada CLB é formado de múltiplos módulos lógicos menores e uma interconexão programável local que é usada para interconectar estes módulos dentro do CLB (Floyd 2009). Os módulos lógicos podem ser configurados para implementar lógica combinacional e/ou lógica registrada.

A Figura 3.4 apresenta a arquitetura interna de um módulo lógico baseado em LUT, que funciona como um tipo de memória programável utilizada para gerar funções lógicas combinacionais de soma de produtos.

Vale destacar que a arquitetura específica de cada FPGA pode variar de acordo com o fabricante. Por exemplo, na maioria dos FPGAs Xilinx, a área lógica configurável é dividida em CLBs que contém múltiplas unidades lógicas, chamadas de Células Lógicas (CLs). Duas células lógicas idênticas são denominadas de *slice* (fatia) (Floyd 2009). Cada célula lógica é baseada na lógica LUT de quatro entradas tradicional mais uma lógica adicional e um flip-flop.

3.2.1 Especificação de circuitos em FPGA

A especificação de circuitos em FPGA compreende basicamente três fases principais, que são: especificação, implementação ou simulação, e verificação e depuração (Noronha 2017). A Figura 3.5 apresenta um diagrama que detalha o processo de design de um circuito em FPGA. Nota-se que durante todo o processo, o desenvolvedor pode retornar à todas as fases, quando necessário.

A primeira fase corresponde a etapa de especificação do projeto de hardware. Para isso, utiliza-se geralmente Linguagens de Descrição de Hardware (*Hardware Description Language* - HDL), como VHDL e Verilog, ou design em *Register Transfer Level*, que manipula elementos como registradores, multiplexadores, somadores, entre outros

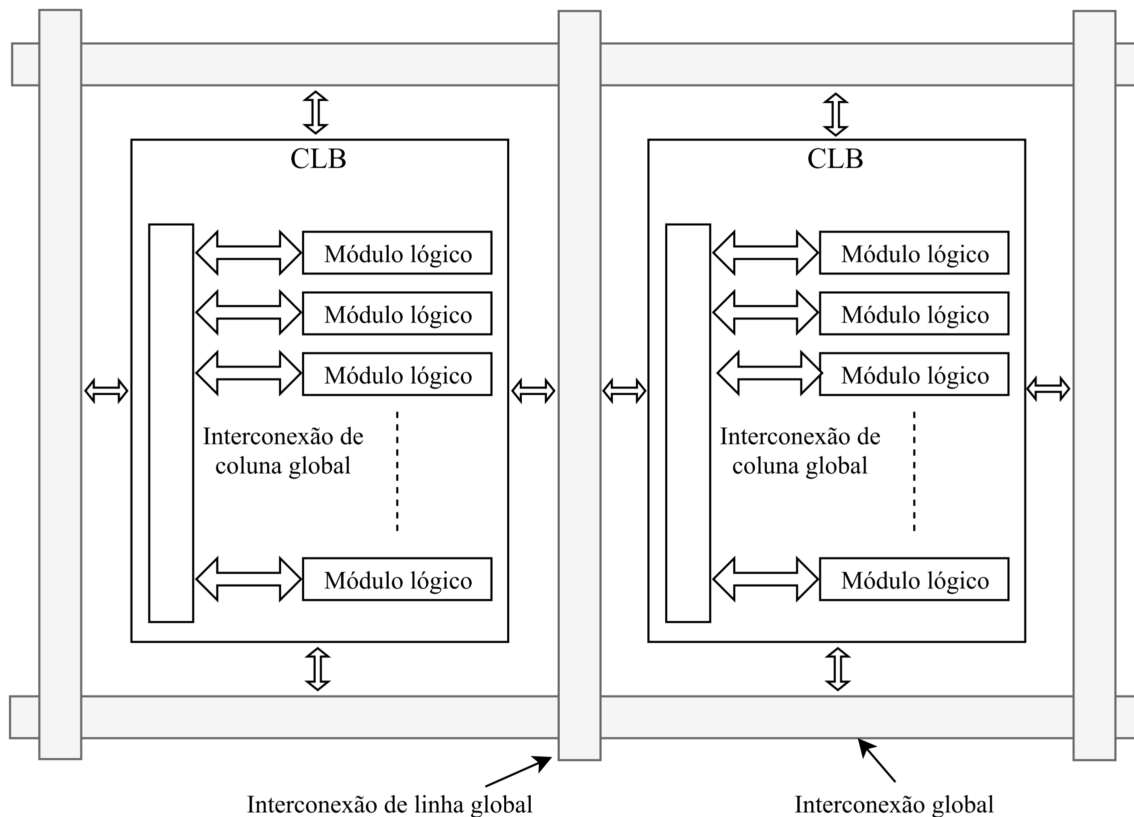

Figura 3.3: Estrutura dos blocos lógicos configuráveis básicos (CLBs) dentro de interconexões programáveis globais em linha/coluna.
Fonte: Floyd (2009).

componentes de *datapath* para construir circuitos digitais. Além disso, há também a possibilidade de empregar ferramentas de *High Level Sysntesis*, que se propõem a converter linguagens de alto nível, como C e Python, em linguagens de descrição de hardware. Essa abordagem vem sendo muito difundida nos últimos anos, por permitir que desenvolvedores de software com pouco ou nenhum conhecimento a nível de hardware, possam projetar circuitos através desta abstração. No entanto, estas ferramentas não conseguem otimizar a utilização dos recursos do FPGA, gerando circuitos com maior número de componentes e menor desempenho (Noronha 2017, Suzuki et al. 2018, Ma et al. 2018).

A segunda fase consiste na implementação em hardware ou na simulação do design, que pode ser muito útil antes de efetuar de fato a implementação no hardware. A Figura 3.6 apresenta as etapas o processo de síntese de um circuito em FPGA. Após a construção do código fonte na fase de especificação, é efetuada a síntese da lógica, que realiza a conversão de construções lógicas de alto nível e código comportamental em portas lógicas. Em seguida, o mapeamento efetua o agrupamento das portas lógicas de acordo com os recursos lógicos do FPGA alvo. Durante o posicionamento, os agrupamentos lógicos são atribuídos a blocos lógicos específicos para que o roteamento determine os recursos de interconexão que transportarão os sinais (Hauck & DeHon 2010). Por fim, efetua-se a geração do *bitstream*, um fluxo de dados em formato binário, que será carregado no FPGA

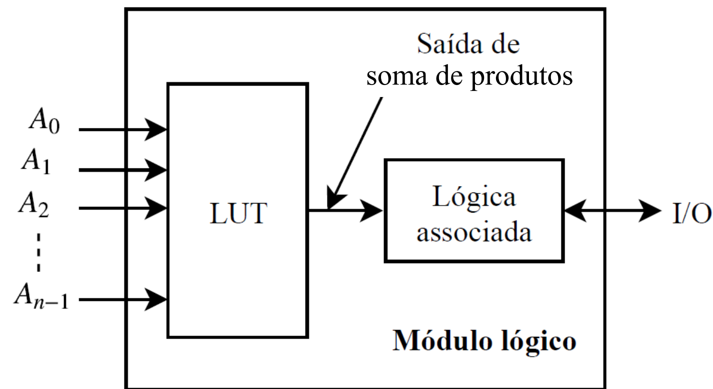

Figura 3.4: Estrutura do módulo lógico.
Fonte: Floyd (2009).

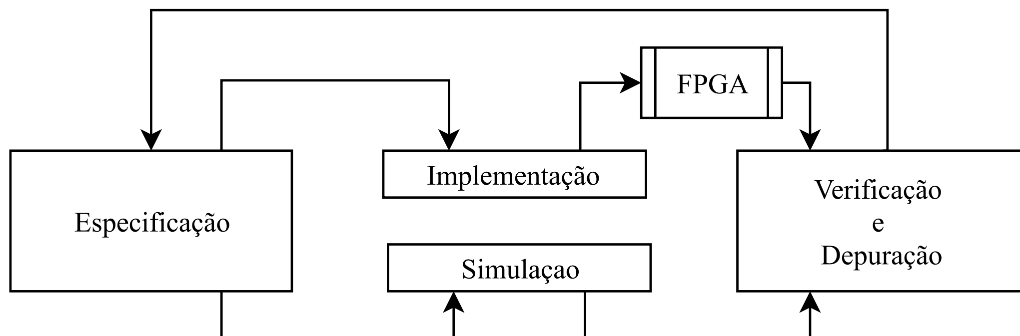

Figura 3.5: Fases do processo de design de um circuito em FPGA.
Fonte: Noronha (2017).

alvo.

Após a especificação e a implementação (ou simulação) do circuito é importante verificar se os requisitos do projeto foram atendidos. Além de validar o funcionamento do sistema, os dados de síntese também devem ser verificados. Geralmente avalia-se a taxa de ocupação dos recursos de hardware do FPGA, o *throughput* do circuito, a potência consumida, entre outros requisitos. Caso seja necessário corrigir erros ou otimizar o circuito construído, deve-se retornar à fase de especificação para modificar o design e implementar ou simular o circuito novamente.

3.2.2 Consumo de Potência

Segundo Shang et al. (2002), existem dois tipos de consumo de potência em FPGAs, que são: potência estática e potência dinâmica. A potência estática é a parcela da potência consumida que independe das atividades de chaveamento dos sinais. Já a potência dinâmica é a potência dissipada pelas transições dos sinais no circuito, consumida devido a carga e descarga das capacitâncias (Shang et al. 2002). É importante destacar que os recursos disponíveis no FPGA não utilizados após a configuração do hardware, não irão

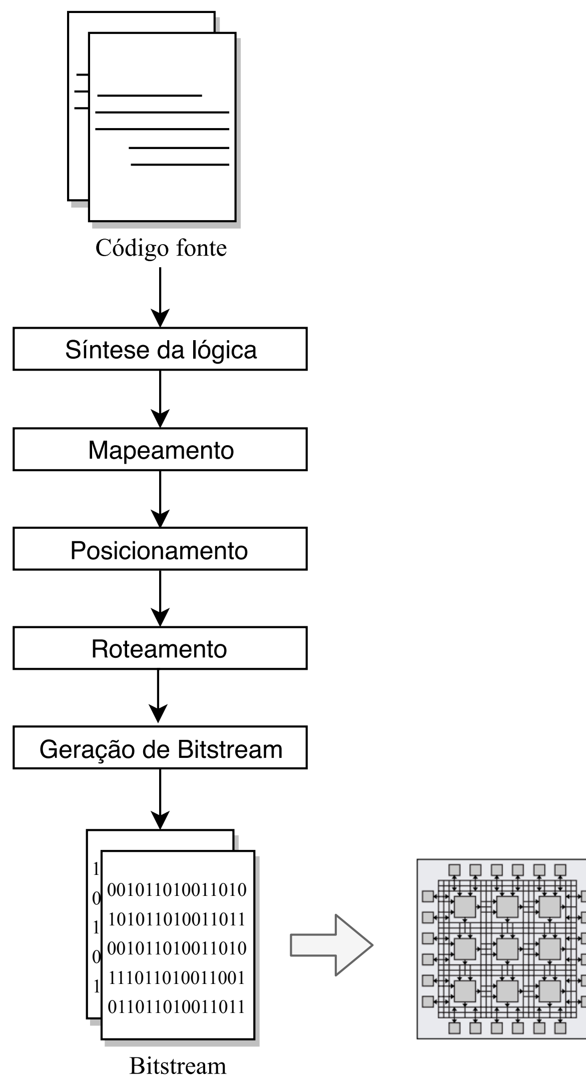

Figura 3.6: Fluxo do processo de síntese do circuito em FPGA.
Fonte: Hauck & DeHon (2010).

consumir potência dinâmica.

Além disso, diferentemente dos processadores que operam em sua frequência máxima, nos FPGAs o *clock* depende do design implementado. Sendo assim, é possível alcançar menor consumo de potência dinâmica ao reduzir a frequência de operação do circuito, fator de extrema importância para aplicações em que o baixo consumo de potência é mais importante que a velocidade de processamento.

Capítulo 4

Aprendizagem Profunda

Neste capítulo será apresentada uma breve contextualização histórica acerca do surgimento e dos avanços na área da Inteligência Artificial até o advento da *Deep Learning*. Em seguida, serão introduzidos os conceitos relacionados às técnicas de aprendizagem profunda, apresentando os *Autoencoders* em sua forma tradicional, e as Redes Neurais Profundas baseadas na técnica *Stacked Sparse Autoencoder*, utilizada neste trabalho.

4.1 Contextualização Histórica

Desde seu surgimento até os dias atuais, a Inteligência Artificial (IA) vêm atraindo pesquisadores das mais variadas áreas, e revolucionando diversos campos da sociedade. Segundo Kurzweil (1990), a Inteligência Artificial consiste na arte de criar máquinas que executam funções que exigem inteligência quando executadas por pessoas. Já Winston (1992) definiu-a como o estudo das computações que tornam possível perceber, raciocinar e agir. Estas e muitas outras definições podem ser aplicadas.

O primeiro trabalho, hoje conhecido como IA, foi desenvolvido no contexto das redes neurais artificiais, por McCulloch e Walter Pitts em 1943, que introduziram a ideia de redes neurais como máquinas de computação. Estes pesquisadores basearam-se no conhecimento da fisiologia básica e da função dos neurônios no cérebro, na análise formal da lógica proposicional e na teoria da computação de Turing, para criar o modelo do neurônio artificial (Norvig & Russell 2014), que serviu como base para muitos avanços posteriores no campo das RNAs.

A inteligência artificial divide-se em diversas subáreas, uma delas é conhecida como Aprendizado de Máquina (*Machine Learning* - ML). Os algoritmos de Aprendizado de Máquina possuem a capacidade de adquirir seu próprio conhecimento através da extração de padrões de dados brutos (Goodfellow et al. 2016).

Segundo Goodfellow et al. (2016), muitas tarefas de IA podem ser resolvidas selecionando o conjunto certo de características a serem extraídas para uma determinada tarefa, e atribuindo-o a um algoritmo simples de ML, no entanto, para outras muitas tarefas, é difícil identificar quais características devem ser extraídas. Nestes casos, a abordagem de Aprendizado por representação (*Representation Learning*), que facilita a extração de informações úteis, se faz necessária.

O Aprendizado por representação busca capturar correlações de alta ordem dos dados

observados, quando nenhuma informação acerca da saída desejada para cada entrada, estiver disponível (Deng et al. 2014). Um dos algoritmos que utilizam aprendizado por representação é o *autoencoder*. Os *autoencoders* são compostos por uma função *encoder*, que converte os dados de entrada em uma representação diferente, e uma função *decoder*, que converte esta nova representação em seu formato original (Goodfellow et al. 2016).

No entanto, em muitas aplicações se faz necessário observar cada pedaço de um determinado dado, o que torna difícil a tarefa de realizar extração de alto nível de características tão abstratas. Problema que pôde ser resolvido com o surgimento da *Deep Learning*, através da utilização de composições de múltiplas transformações não-lineares, para produzir representações mais abstratas e mais úteis (Bengio et al. 2013), ou seja, da utilização de representações que são expressas em termos de outras representações mais simples (Goodfellow et al. 2016).

4.2 Técnicas de Aprendizagem Profunda

As técnicas de Aprendizagem Profunda surgiram com a finalidade de resolver os problemas existentes com as redes neurais artificiais que possuíam arquiteturas rasas (*shallow*), ou seja, com poucas camadas ocultas. Uma rede neural consiste em um processador maciçamente paralelo e distribuído, formado por unidades de processamento simples, chamadas de neurônios artificiais, capazes de armazenar conhecimento experimental e disponibilizá-lo para uso (Haykin 2001). As RNAs, apesar de serem eficazes na solução de muitos problemas, possuíam grandes dificuldades para lidar com aplicações complexas do mundo real, envolvendo sinais naturais, fala humana, cenas visuais, entre outras (Deng et al. 2014).

As técnicas de *Deep Learning* consistem em um tipo particular de *Machine Learning*. Essas técnicas exploram a utilização de muitas camadas de processamento de informações não lineares para extrair características de forma supervisionada ou não supervisionada. A Figura 4.1 ilustra o processo de reconhecimento de objetos em imagens através da utilização da *Deep Learning*.

Neste caso, após o recebimento dos pixels da imagem na camada de entrada (camada visível da rede), as camadas ocultas dispostas em seguida expandem sucessivamente as características da imagem. Dados os pixels, a primeira camada oculta poderá identificar facilmente as bordas, comparando o brilho dos pixels vizinhos, já a segunda camada oculta, a partir das bordas descritas na camada anterior, é capaz de procurar facilmente por cantos e contornos estendidos, que são conhecidos como coleções de bordas, por fim, a partir destes dados, a terceira camada oculta poderá detectar partes inteiras de objetos específicos, encontrando coleções específicas de contornos e cantos (Goodfellow et al. 2016). Assim sendo, a camada de saída poderá identificar efetivamente o objeto.

As redes neurais profundas podem ser consideradas como uma modernização das redes *Multilayer Perceptron* (MLP), tendo em vista que uma das principais diferenças entre as redes MLP e as DNNs consiste na viabilidade do treinamento de redes com muitas camadas ocultas, nas redes de aprendizagem profunda, o que era um grande problema nas MLPs convencionais. Dessa forma, uma MLP com muitas camadas ocultas corresponde a uma DNN (Deng et al. 2014).

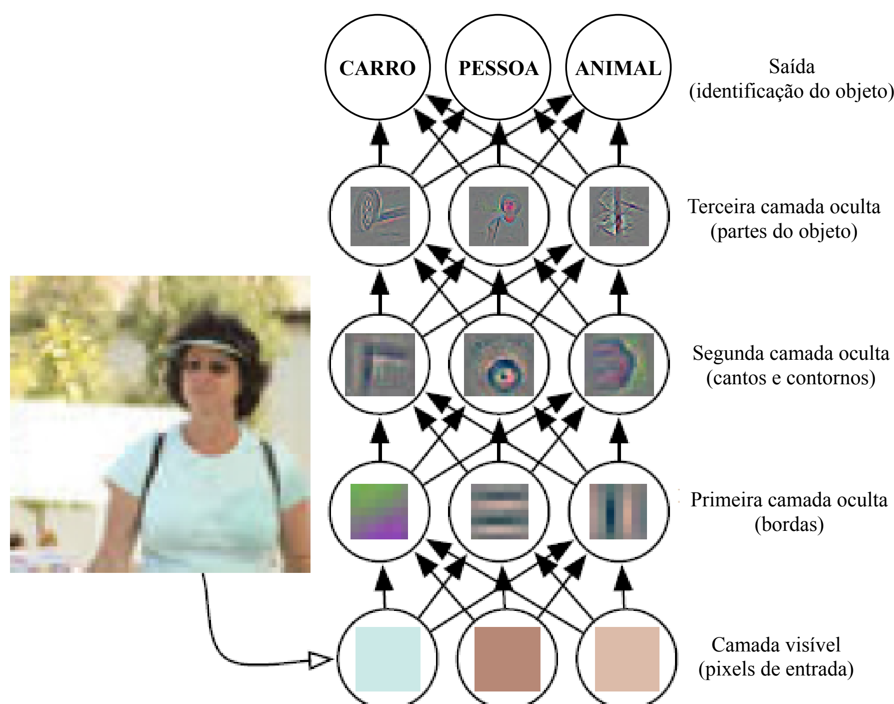

Figura 4.1: Ilustração de um modelo de *Deep Learning*.
Fonte: Goodfellow et al. (2016)

Entre as técnicas de *Deep Learning* presentes na literatura, destacam-se as técnicas baseadas em *Autoencoders*, as Redes Neurais Convolucionais, as Máquinas Restritas de Boltzmman (*Restricted Boltzmann Machines* - RBM), entre outras (Deng et al. 2014, Goodfellow et al. 2016). Estas técnicas abrangem uma ampla possibilidade de atividades, incluindo visão computacional, processamento de áudio e fala, processamento de linguagem natural, robótica, bioinformática, finanças, entre outras (Goodfellow et al. 2016).

4.2.1 *Autoencoder*

O termo *autoencoder* (AE) antecede o advento da *Deep Learning*. Em sua estrutura tradicional, os *autoencoders* eram principalmente aplicados a problemas de redução de dimensionalidade e aprendizagem de características, sendo pertencentes a classe dos algoritmos de aprendizado por representação. Nestas redes o treinamento ocorre para que a camada de saída forneça uma reconstrução da camada de entrada, dessa forma, ambas camadas possuem o mesmo tamanho, além disso é utilizada apenas uma camada oculta, que realiza a extração de características dos dados de entrada.

Sendo assim, a arquitetura do *autoencoder* é composta por três camadas: uma de entrada, uma oculta e uma de saída. Um exemplo de uma estrutura de *autoencoder* é ilustrada na Figura 4.2. As camadas de entrada e oculta formam o *encoder* da rede, e as camadas oculta e de saída, compõem o *decoder* (Goodfellow et al. 2016). Os *autoencoders* têm sido utilizados em muitos problemas de aprendizagem não supervisionada, em

que é possível representar um vetor de entrada de dimensão P (da camada de entrada) em um vetor de dimensão M (da camada oculta), onde $M < P$. Ou seja, os *autoencoders* são capazes de reduzir a dimensionalidade dos dados de entrada extraindo informações essenciais. Com os *autoencoders* é possível também recuperar a informação de entrada utilizando os neurônios da camada de saída. Basicamente, os neurônios da camada oculta reduzem a informação de entrada e os neurônios da camada de saída recuperam esta informação.

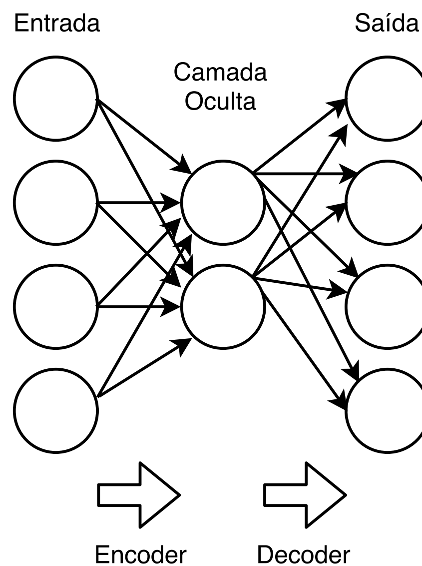

Figura 4.2: Exemplo de uma estrutura de *Autoencoder*.

Os *autoencoders* são treinados para preservar o máximo de informação possível quando uma entrada é executada através do *encoder* e posteriormente, no *decoder*, porém eles também são treinados para fazer com que a nova representação contenha várias propriedades satisfatórias (Goodfellow et al. 2016).

Estas redes são consideradas como um tipo específico de MLP, e seu treinamento pode utilizar os mesmos algoritmos. Geralmente utiliza-se métodos variantes do *backpropagation* (Deng et al. 2014), como o método do Gradiente Descendente Estocástico (*Stochastic Gradient Descent* - SGD) e o método do Gradiente Conjugado Escalonado (*Scaled Conjugate Gradient* - SCG).

Neste trabalho, o treinamento da rede em software utilizou o algoritmo do gradiente conjugado escalonado, proposto por Moller (1993). Este método utiliza uma abordagem de Levenberg-Marquardt, que evita a busca linear (unidimensional) em cada iteração (Fletcher 1975). O SCG utiliza um mecanismo de escalonamento do passe de ajuste α (taxa de aprendizagem), o que torna o algoritmo mais rápido do que outros métodos de segunda ordem (Moller 1993).

Entre as técnicas de *autoencoders* mais utilizadas estão os *Denoising Autoencoders* e os *Sparse Autoencoders*. Os *Denoising Autoencoders* são treinamos para fornecer como saída a regeneração de uma entrada de dados corrompidos (Vincent et al. 2010). Já os *Sparse Autoencoders* são normalmente utilizados para aprender características, de modo a atuar em problemas como classificação de padrões.

Um *sparse autoencoder* é simplesmente um *autoencoder* cujo treinamento envolve uma penalidade de esparsidade, que funciona como um termo regularizador adicionado à função de perda (Goodfellow et al. 2016). *Autoencoders* regularizados estimulam o modelo a obter outras propriedades, como a esparsidade da representação, além da habilidade de copiar sua entrada em sua saída

4.2.2 *Stacked Sparse Autoencoder*

Com o surgimento da *Deep Learning*, os *autoencoders* passaram a ser utilizados de maneira encadeada, formando redes com muitas camadas ocultas, cujo treinamento é realizado previamente em cada *autoencoder*, de forma não supervisionada. Nos *stacked sparse autoencoders* (SSAEs), cada camada oculta é composta pela camada oculta de um *sparse autoencoder* treinado individualmente. Cada *sparse autoencoder* recebe como entrada, a saída da camada oculta do *sparse autoencoder* que o antecede, de modo que as características dos dados de entrada sejam extraídas ao longo das camadas ocultas da rede, possibilitando que a camada de saída seja capaz de efetuar a classificação dos dados, após um treinamento supervisionado. Neste trabalho, primeiramente realizou-se o treinamento da rede por meio desta técnica. A Figura 4.3 apresenta a arquitetura do SSAE proposto com uma camada de entrada (P entradas de dados), duas camadas ocultas (M e N neurônios) e uma camada de saída (H saídas). Esta arquitetura pode ser representada como $P-M-N-H$.

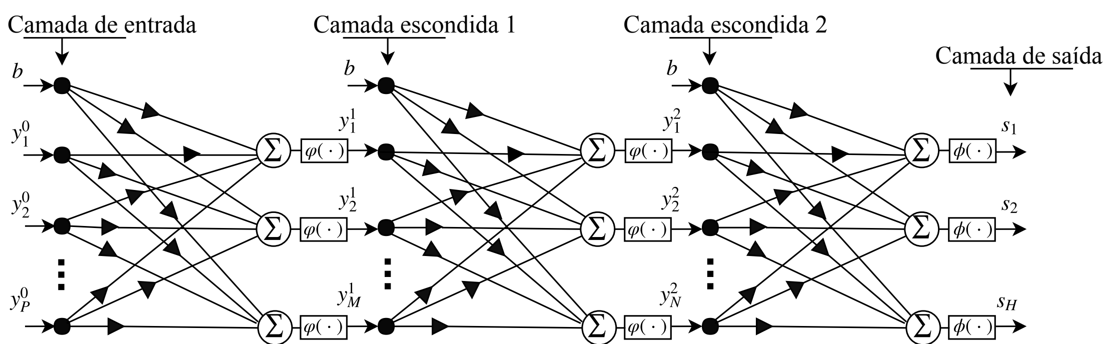

Figura 4.3: Arquitetura do *Stacked Sparse Autoencoder* proposto.

Foi implementada a fase *feedforward* da rede, na qual a equação que define a saída do i-ésimo neurônio da k-ésima camada, $z_i^k(n)$, do n-ésimo instante, pode ser expressa como

$$z_i^k(n) = \sum_{j=1}^{U^l} w_{ij}^k(n) \cdot y_j^l(n) + wb_i^k(n) \cdot b \tag{4.1}$$

em que $w_{ij}^{k}(n)$ é o peso associado a j-ésima entrada do i-ésimo neurônio da k-ésima camada no n-ésimo instante, $y_{j}^{l}(n)$ é a j-ésima entrada da l-ésima camada, em que $l = k-1$, no n-ésimo instante, $wb_{i}^{k}(n)$ é o peso associado ao *bias* do i-ésimo neurônio da k-ésima camada no n-ésimo instante, b é o *bias*, que tem valor de 1, e U^{l} é o número de entradas da l-ésima camada, no qual $U^{0} = P$, $U^{1} = M$ e $U^{2} = N$. A função de ativação utilizada nas camadas ocultas foi a sigmoide, desta forma a saída associada ao i-ésimo neurônio da k-ésima camada, no n-ésimo instante, $v_{i}^{k}(n)$, pode ser caracterizada como

$$v_{i}^{k}(n) = \frac{1}{1 + e^{-z_{i}^{k}(n)}} \tag{4.2}$$

em que $v_{i}^{k}(n)$ será o valor da j-ésima entrada a ser utilizada na camada seguinte, no n-ésimo instante, $y_{j}^{l+1}(n)$, ou seja

$$y_{j}^{l+1}(n) = v_{i}^{k}(n) \tag{4.3}$$

sendo $j = i$.

Na camada de saída, foi utilizada a função de ativação *softmax*, que vem sendo empregada em redes neurais de classificação como apresentado em Maria et al. (2016), e pode ser expressa como

$$s_{i}(n) = \frac{e^{z_{i}^{K}(n)}}{\sum_{h=1}^{H} e^{z_{h}^{K}(n)}} \tag{4.4}$$

onde $s_{i}(n)$ é a i-ésima saída da última camada, K, com H neurônios, no n-ésimo instante. Através desta função a saída da rede é forçada a representar a probabilidade dos dados pertencerem a uma determinada classe. Sendo assim, a quantidade de neurônios desta camada corresponde a quantidade de classes existentes no problema em questão.

Em alguns casos, após o treinamento de todas as camadas da rede de forma individual, é efetuado o ajuste fino dos pesos para agregar maior performance aos resultados da técnica. Dessa forma, treina-se o *stacked sparse autoencoder* como um todo, de maneira supervisionada.

Capítulo 5

Descrição da proposta

Neste capítulo serão apresentadas duas propostas de implementação de SSAE para a fase *feedforward*, utilizando a técnica de matriz sistólica. A principal diferença entre elas consiste na maneira como os pesos sinápticos da rede são inseridos no hardware. Nas seções a seguir serão exibidas as características de cada proposta, bem como os detalhes à respeito do tempo de processamento de cada implementação.

5.1 Proposta 1

Tendo como base a estrutura da rede detalhada na Figura 4.3, a arquitetura geral da primeira implementação proposta neste trabalho é apresentada na Figura 5.1. Esta arquitetura permite o recebimento dos pesos da rede através de *streams* (fluxos) de pesos, sem a necessidade de utilizar recursos de memória para guardá-los, diferentemente da maioria dos trabalhos encontrados na literatura. As variáveis e constantes do projeto estão em ponto fixo, e para cada j-ésima entrada, $y_j^0(n)$, utiliza-se 1 bit na parte inteira (sem sinal) e 12 bits na parte fracionária, tendo em vista que as entradas estão normalizadas entre 0 e 1. Já para os pesos sinápticos dos neurônios das camadas ocultas, um e dois, expressos como $w_{ij}^1(n)$ e $w_{ij}^2(n)$, bem como para os pesos do *bias* de todas as camadas, $wb_i^k(n)$, são utilizados 5 bits na parte inteira (utilizando um para sinal) e 12 bits na parte fracionária. E para os pesos dos neurônios da camada de saída, $w_{ij}^3(n)$, são utilizados 7 bits na parte inteira (utilizando um para sinal) e 12 bits na parte fracionária.

Para a implementação da proposta foi utilizada a técnica de matriz sistólica (*systolic array*) (Kung & Leiserson 1978), que funciona como uma abordagem intermediária entre a metodologia completamente paralela e a completamente serial. Esta técnica permite que os dados sejam recebidos de forma serial e os elementos de processamento (*Processing Elements* - PEs) executem suas operações de forma paralela (Kung & Leiserson 1978).

A rede implementada neste trabalho utilizou duas camadas ocultas, consistindo na arquitetura $784-100-50-10$. No entanto, para acrescentar mais camadas ocultas à rede é necessário apenas replicar o bloco intermediário presente na Figura 5.1, que representa a segunda camada oculta da rede, e realizar algumas adaptações em cada camada para a quantidade de neurônios desejada, como será apresentado a seguir.

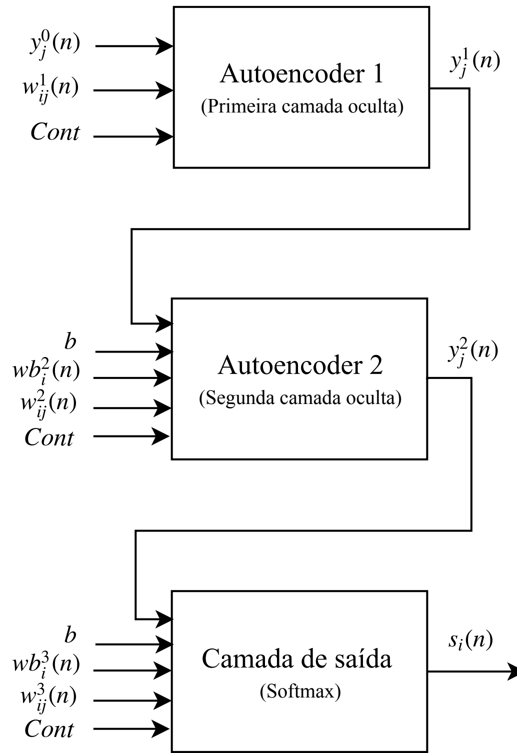

Figura 5.1: Arquitetura geral da proposta 1.

5.1.1 Camadas do SSAE

A arquitetura da k-ésima camada oculta, chamada de *autoencoder k* na Figura 5.1, é apresentada na Figura 5.2 e ela implementa a técnica de matriz sistólica. Cada neurônio da k-ésima camada é representado por um i-ésimo PE_i^k e a quantidade de PEs é definida pelo valor de V^k, em que $V^1 = M$, $V^2 = N$ e $V^3 = H$. Dessa forma, cada i-ésimo PE_i^k implementa a Equação 4.1. Acrescentar mais neurônios à camada consiste apenas em replicar o bloco do PE quantas vezes for desejado. Os valores computados em cada i-ésimo PE_i^k da k-ésima camada oculta passam por uma função de ativação, chamada aqui de FA^k (ver Equação 4.2), e são utilizados como entrada da camada seguinte (ver Equação 4.3). Já os valores computados nos PEs da camada de saída passam pela função de ativação definida na Equação 4.4, para gerar a saída da rede. O sinal *sel* corresponde ao seletor do multiplexador, Mux^k, utilizado para selecionar as saídas de cada i-ésimo PE_i^k para a entrada da função de ativação da k-ésima camada, FA^k. Além disso, nesta primeira proposta, utilizou-se um contador em anel em cada k-ésima camada para habilitar o recebimento dos pesos em cada i-ésimo PE_i^k.

Através da matriz sistólica, os valores das entradas e dos pesos da camada fluem entre os PEs de modo que cada PE inicie suas operações no instante seguinte ao início das operações no PE que o antecede, fazendo com que estes módulos passem a operar de forma paralela. Vale ressaltar que a partir da segunda camada oculta, ou seja, quando $k > 1$, o *bias*, b, e os pesos do *bias*, $wb_i^k(n)$, são inseridos como entrada da k-ésima camada, seguindo a estrutura apresentada na Figura 5.1. Apenas na primeira camada

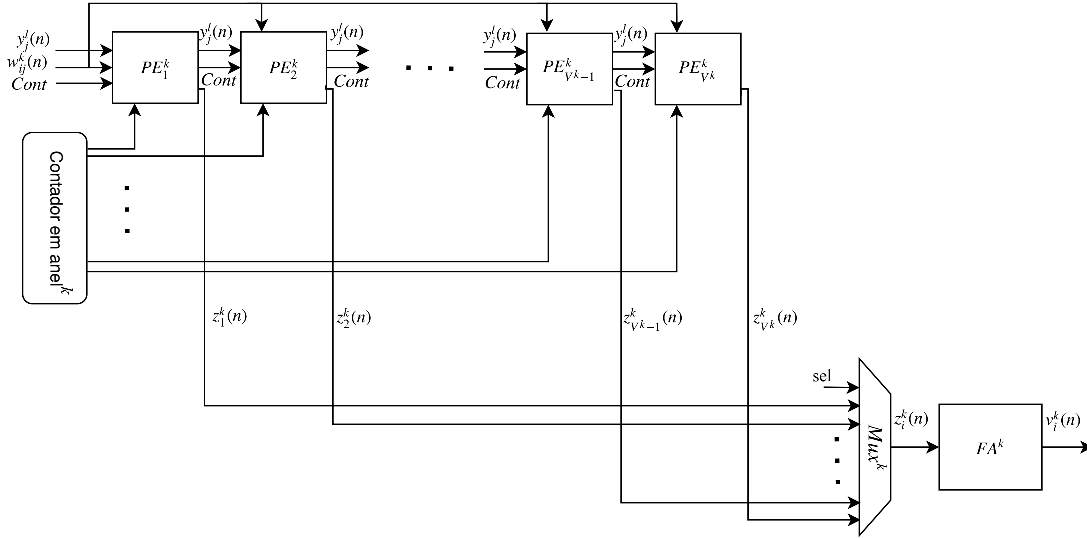

Figura 5.2: Arquitetura da k-ésima camada oculta do SSAE da proposta 1.

oculta, ou seja, quando $k = 1$, estes valores são inseridos em conjunto com as entradas da rede e com os pesos dos neurônios da camada, respectivamente.

Uma vantagem desta primeira proposta é permitir a inserção dos pesos de cada camada de forma serial através de um único *stream* de pesos, na primeira camada oculta, e de dois *streams* de pesos, nas demais camadas da rede. Esta seria a menor quantidade de *streams* de pesos necessária para o funcionamento deste circuito. No entanto, esta arquitetura é facilmente escalonável, permitindo a paralelização dos *streams* de pesos de cada camada. A quantidade de *streams* de pesos por camada pode variar de um até a quantidade de neurônios da camada, V^1, na primeira camada oculta da rede, e de dois até $2 \times V^k$, nas demais camadas da rede, devido aos pesos do *bias*.

Vale destacar que possibilitar o recebimento dos pesos por *streams*, viabiliza a utilização de um mesmo hardware para problemas distintos, já que os pesos treinados para um novo problema poderão ser inseridos na rede, bem como as entradas do problema em questão, sem a necessidade de reconfigurar o hardware.

5.1.2 Unidades de Processamento

As unidades de processamento da DNN correspondem aos elementos de processamento (PEs) da matriz sistólica. A arquitetura dos PEs da primeira camada oculta difere da arquitetura dos PEs das demais camadas, em decorrência de na primeira camada oculta o *bias*, b, e os pesos do *bias*, $wb_i^1(n)$, serem inseridos juntamente com as entradas da rede e os pesos sinápticos dos neurônios da camada, respectivamente. Sendo assim, a arquitetura de cada i-ésimo PE_i^1, da proposta 1, é detalhada na Figura 5.3 e é formada por um multiplicador, um somador e quatro registradores, R. Já a arquitetura de cada i-ésimo PE_i^k, para $k > 1$, da proposta 1, é detalhada na Figura 5.4 e é formada por dois multiplicadores, dois somadores e seis registradores, R. Cada i-ésimo PE_i^k gera uma saída após Z amostras, onde para a primeira camada oculta, $Z = P$ e para as demais, a variável Z

deverá ser maior ou igual a quantidade de entradas ($Z \geq P$) e múltiplo da quantidade de neurônios da primeira camada oculta (mod(Z,M) = 0).

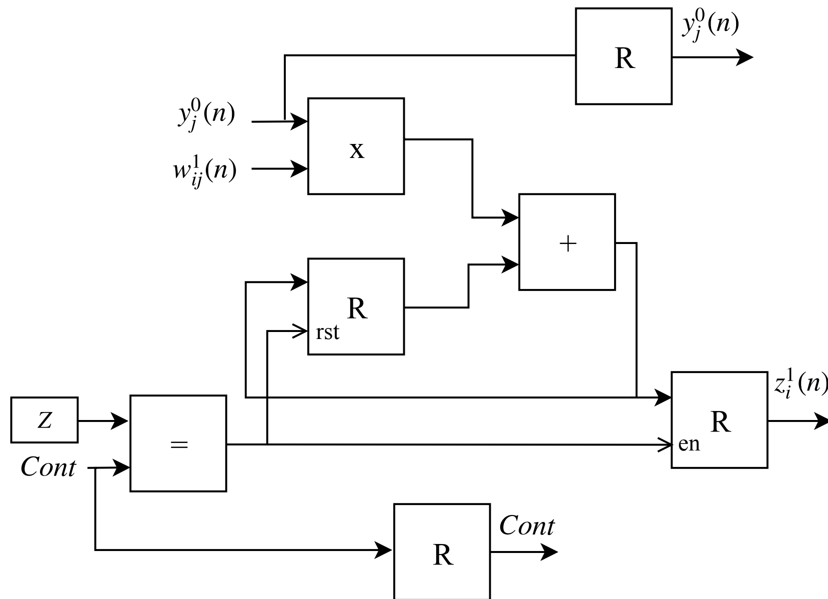

Figura 5.3: Arquitetura do i-ésimo PE_i^1 da proposta 1.

5.1.3 Funções de Ativação

A implementação de cada k-ésima função de ativação, FA^k, se deu com a utilização da técnica de *Lookup Table* (LUT), ilustrada na Figura 5.5, que permite aproximar funções por meio de uma tabela de L valores. Para implementar a função de ativação de cada camada oculta (Equação 4.2) foi utilizada uma memória ROM com profundidade de 16 bits, em que $L = 2^{16}$, armazenando palavras de 13 bits. Vale destacar que para alterar a função de ativação utilizada nestas camadas, é necessário apenas gravar os valores da nova função de ativação desejada na *lookup table*. Já para a implementação da função de ativação da camada de saída, a *softmax*, definida pela Equação 4.4, foi utilizada uma LUT para fazer a aproximação da função exponencial, e só adiante, computar a divisão da Equação 4.4. A LUT da camada de saída, FA^3, foi configurada com uma profundidade de $L = 2^{16}$, armazenando palavras de 57 bits.

Um ponto importante associado a esta estratégia de implementação é a utilização de apenas uma única função de ativação, ou LUT, por camada. Esta característica reduz de forma significativa o espaço ocupado no FPGA (ver de Souza & Fernandes (2014)).

5.1.4 Tempo de Processamento

Na proposta 1, um contador em anel em cada k-ésima camada, mostrado na Figura 5.2, é utilizado para habilitar o recebimento dos pesos em cada i-ésimo PE_i^k da k-ésima camada. Como está sendo utilizada a menor quantidade possível de *streams* de pesos

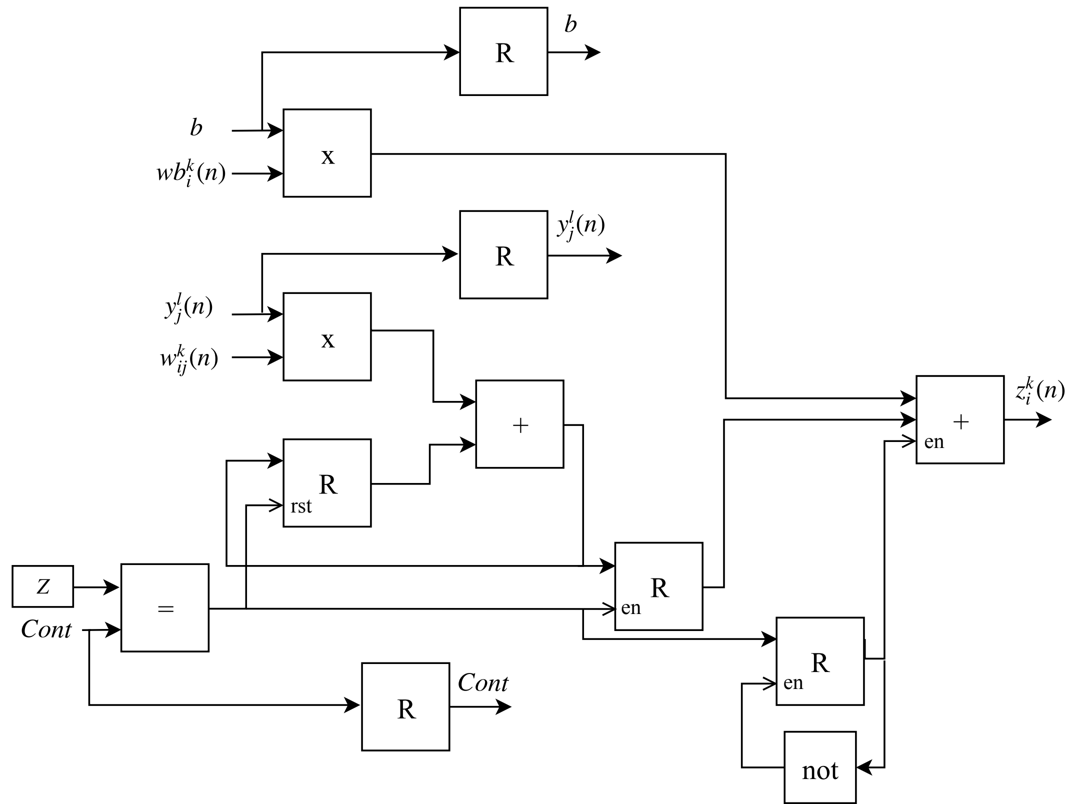

Figura 5.4: Arquitetura do i-ésimo PE_i^k, para $k > 1$, da proposta 1.

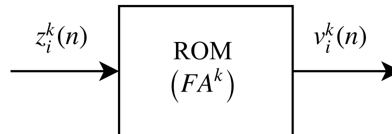

Figura 5.5: Arquitetura da k-ésima função de ativação, FA^k, associada as camadas ocultas.

por camada, a operação do contador em anel da k-ésima camada deve ser V^k vezes mais rápida do que a operação de cada PE_i^k. Com a finalidade de manter o sincronismo da implementação, considera-se $V^k > V^{k+1}$. Com isso, o tempo de execução de cada PE do circuito é determinado pelo tempo de execução do contador em anel da primeira camada oculta, t_{ca}. Sendo assim, o tempo do PE, t_{PE}, pode ser definido como

$$t_{PE} = V^1 \times t_{ca} \tag{5.1}$$

onde V^1 é a quantidade de neurônios na primeira camada oculta, $(V^1 \times t_{ca}) \geq t_c$ e t_c é o tempo do caminho crítico do sistema.

A arquitetura proposta possui um atraso inicial que pode ser expresso como

$$d = (Q \times K + D) \times t_{PE} \tag{5.2}$$

no qual Q consiste na quantidade de amostras necessárias para que o primeiro neurônio da primeira camada oculta comece a retornar valores de saída, que corresponde a um número maior ou igual a quantidade de entradas da rede ($Q \geq P$) e múltiplo da quantidade de neurônios da primeira camada oculta ($\text{mod}(Q,M) = 0$), K corresponde a quantidade de camadas com neurônios, D é o atraso, em número de amostras, provocado pelas funções de ativação das camadas ocultas e de saída, e t_{PE} é o tempo do PE. Já o *throughput* (th_{ff}), em *frames* por segundo (FPS), pode ser expresso como

$$th_{ff} = \frac{1}{Q \times t_{PE}}. \tag{5.3}$$

É importante destacar que o *throughput* da implementação independe da quantidade de camadas da rede. Sendo dependente apenas da quantidade de entradas e de neurônios utilizados na primeira camada oculta. A quantidade de camadas com neurônios, K, impactará apenas no delay inicial do sistema. No entanto, nesta primeira proposta, ao utilizar a quantidade mínima de *streams* de pesos por camada, atingiremos apenas o valor *lower bound* do *throughput*, que consiste em seu limite inferior. O crescimento do *throughput* da implementação será impulsionado pela paralelização dos *streams* de pesos utilizados.

Com base na Equação 5.3, o tempo de execução do SSAE proposto (chamado aqui de tempo de *feedforward* (t_{ff})), após o atraso inicial de d segundos (Equação 5.2), pode ser expresso por

$$t_{ff} = Q \times t_{PE} = \frac{1}{th_{ff}} \tag{5.4}$$

sendo assim, em cada t_{ff} é possível obter a saída de todos os H neurônios da última camada, ou seja, a saída da rede referente a uma determinada entrada.

5.2 Proposta 2

A arquitetura geral da segunda proposta é apresentada na Figura 5.6 e ela tem como base a estrutura apresentada na Figura 4.3. As principais diferenças com relação a primeira proposta estão relacionadas ao modo como os pesos sinápticos são inseridos na rede. Nesta segunda proposta, optou-se por guardar os valores dos pesos em memórias ROM, e não mais recebê-los de forma serial, como ocorre na primeira implementação. As variáveis e constantes do projeto estão em ponto fixo, utilizando as resoluções de bits apresentadas na Subsecão 5.1.1, para a proposta 1.

5.2.1 Camadas do SSAE

A Figura 5.2 apresenta a arquitetura geral da k-ésima camada oculta, chamada de *autoencoder* k (ver Figura 5.1), da proposta 2. Nesta segunda proposta, não há *streams* para os pesos sinápticos, já que estes valores encontram-se guardados em LUTs, através de memórias ROM. Isto é possível em decorrência do treinamento da rede ser feito previamente e não existir a necessidade de alterar os pesos, uma vez que a rede já tiver sido treinada para um problema específico. No entanto, ao desejar utilizar a rede para um novo

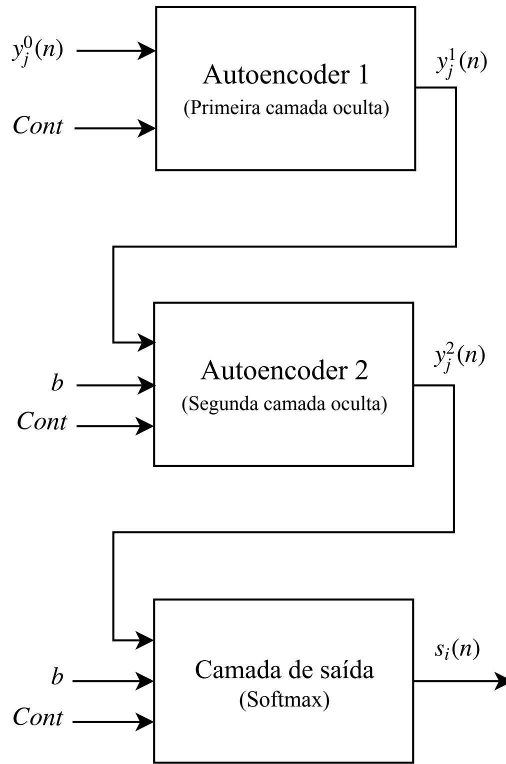

Figura 5.6: Arquitetura geral da proposta 2.

problema, o hardware deve ser reconfigurado, gravando os novos pesos, obtidos após o treinamento prévio da rede para este novo problema, nas *lookup tables*.

A implementação de cada k-ésima função de ativação, FA^k, das camadas ocultas, bem como da função de ativação da camada de saída da proposta 2, corresponde ao que foi apresentado para a proposta 1 na Subseção 5.1.3.

5.2.2 Unidades de Processamento

Como na proposta anterior, cada PE implementa uma unidade de processamento. Nesta segunda proposta, a arquitetura dos PEs da primeira camada oculta também difere da arquitetura dos PEs das demais camadas. A arquitetura de cada i-ésimo PE_i^1, da proposta 2, é detalhada na Figura 5.8 e é formada por um multiplicador, um somador, quatro registradores, R, e uma LUT para guardar os pesos do i-ésimo PE_i^1, chamada de W_i^1. Já a arquitetura de cada i-ésimo PE_i^k, para $k > 1$, da proposta 2, é detalhada na Figura 5.9 e é formada por dois multiplicadores, dois somadores, seis registradores, R, uma LUT para guardar os pesos do i-ésimo PE_i^k, chamada de W_i^k, e uma constante para o peso do *bias*, wb_i^k. Cada i-ésimo PE_i^k gera uma saída após Z amostras, como exposto na Subseção 5.1.2 para a proposta 1. As LUTs de cada PE_i^k, W_i^k, utilizam uma memória ROM com uma profundidade de $L = Z$, armazenando palavras de 17 bits nas camadas ocultas, e 19 bits na camada de saída. Além disso, as constantes utilizadas para o peso do *bias* de cada i-ésimo neurônio, wb_i^k, foram configuradas com 5 bits na parte inteira (utilizando um para

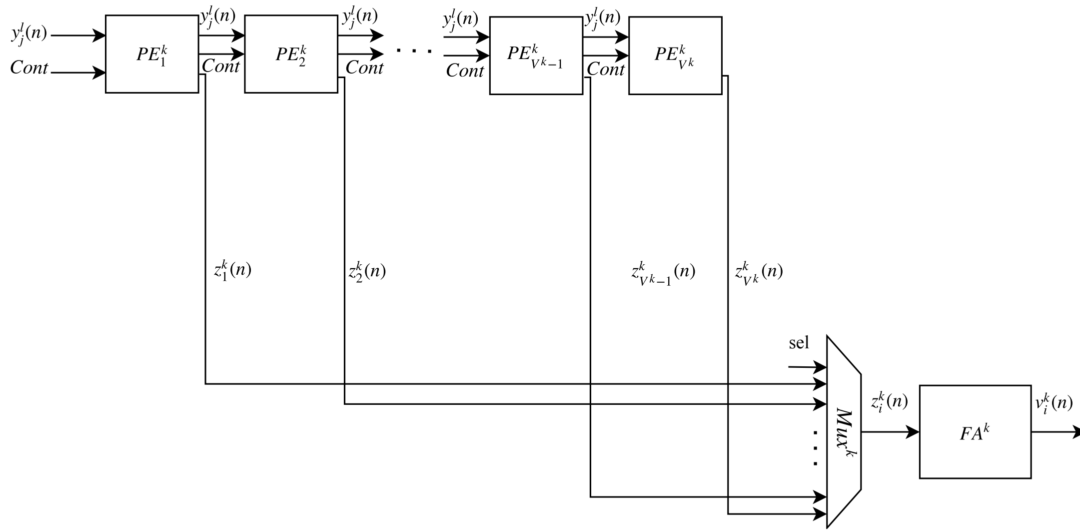

Figura 5.7: Arquitetura da k-ésima camada oculta do SSAE da proposta 2.

sinal) e 12 bits na parte fracionária.

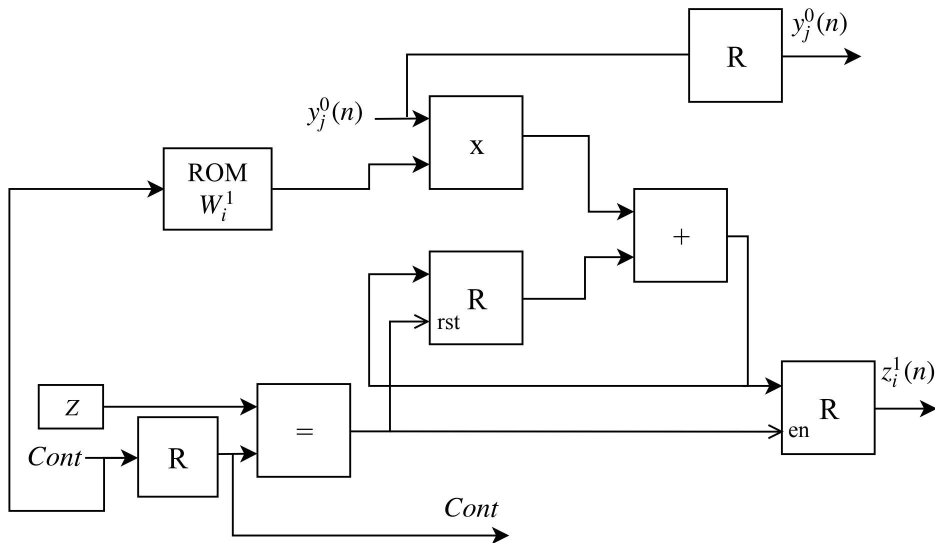

Figura 5.8: Arquitetura do i-ésimo PE_i^1 da proposta 2.

5.2.3 Tempo de Processamento

Nesta segunda proposta, o tempo de execução do PE corresponde apenas ao tempo do caminho crítico do sistema, t_c, ou seja

$$t_{PE} = t_c. \tag{5.5}$$

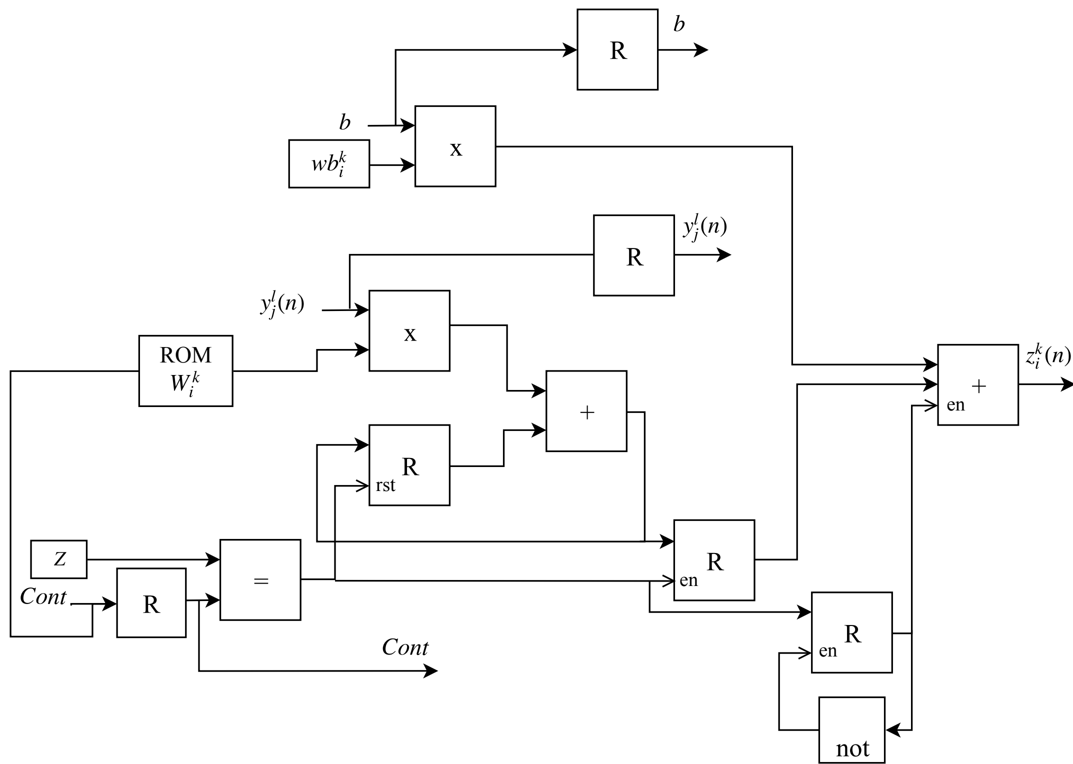

Figura 5.9: Arquitetura do i-ésimo PE_i^k, para $k > 1$, da proposta 2.

No entanto, as demais Equações da Subseção 5.1.4, podem ser consideradas para o cálculo do tempo de processamento da proposta 2, levando em consideração a Equação 5.5. O atraso inicial, d, pode ser definido pela Equação 5.2. Além disso, o *throughput*, th_{ff}, pode ser expresso pela Equação 5.3, bem como o tempo de execução do SSAE, t_{ff}, pode ser expresso pela Equação 5.4.

Capítulo 6

Resultados

Nesta capítulo será apresentada uma análise da ocupação em hardware associada a quantidade de PEs utilizados no FPGA. Também será exposta a validação da implementação em hardware, que incluem os resultados de síntese para cada proposta aqui apresentada. Os resultados de síntese mostram os dados associados a taxa de ocupação, ao tempo de processamento e ao consumo de potência de cada circuito. Também serão apresentadas comparações com um trabalho do estado da arte, que evidenciam as vantagens da proposta apresentada neste trabalho.

6.1 Análise de Ocupação em Hardware

Os resultados de síntese para a análise da ocupação em hardware associada a quantidade de PEs utilizados, foram obtidos para um FPGA Virtex 6 xc6vlx240t-1ff1156. A Tabela 6.1 apresenta a ocupação em hardware (células lógicas e multiplicadores) por PE (n_{PE}) para a proposta 1. Como cada PE_i^k representa o i-ésimo neurônio da k-ésima camada, a Tabela 6.1 apresenta a ocupação por neurônio na k-ésima camada.

Tabela 6.1: Ocupação em hardware (células lógicas e multiplicadores) por PE para a proposta 1.

Camada	n_{PE}	n_{CL}	n_{Mult}
$k = 1$	1	141 (0,06%)	1 (0,13%)
	5	853 (0,35%)	5 (0,65%)
	10	1664 (0,69%)	10 (1,30%)
	25	4200 (1,74%)	25 (3,26%)
	50	9397 (3,90%)	50 (6,51%)
	100	22162 (9,19%)	100 (13,02%)
$k \geq 2$	1	269 (0,11%)	2 (0,26%)
	2	599 (0,25%)	4 (0,52%)
	5	1349 (0,56%)	10 (1,30%)
	10	2610 (1,08%)	20 (2,60%)
	25	6511 (2,70%)	50 (6,51%)
	50	14621 (6,06%)	100 (13,02%)

As Figuras 6.1 e 6.2 apresentam a curva da regressão linear para a primeira camada ($k = 1$) e para as demais camadas da rede ($k \geq 2$), respectivamente. Os pontos observados foram obtidos da Tabela 6.1 e a equação associada à análise da regressão pode ser expressa como

$$n_{CL} = \begin{cases} 212{,}2n_{PE} & \text{para } k = 1 \\ 285{,}1n_{PE} & \text{para } k \geq 2, \end{cases} \tag{6.1}$$

onde k representa a camada específica e n_{PE}, a quantidade de PEs. Para os multiplicadores, a análise de regressão não é necessária, sendo assim, a equação que define a quantidade de multiplicadores por camada pode ser expressa como

$$n_{Mult} = \begin{cases} n_{PE} & \text{para } k = 1 \\ 2n_{PE} & \text{para } k \geq 2. \end{cases} \tag{6.2}$$

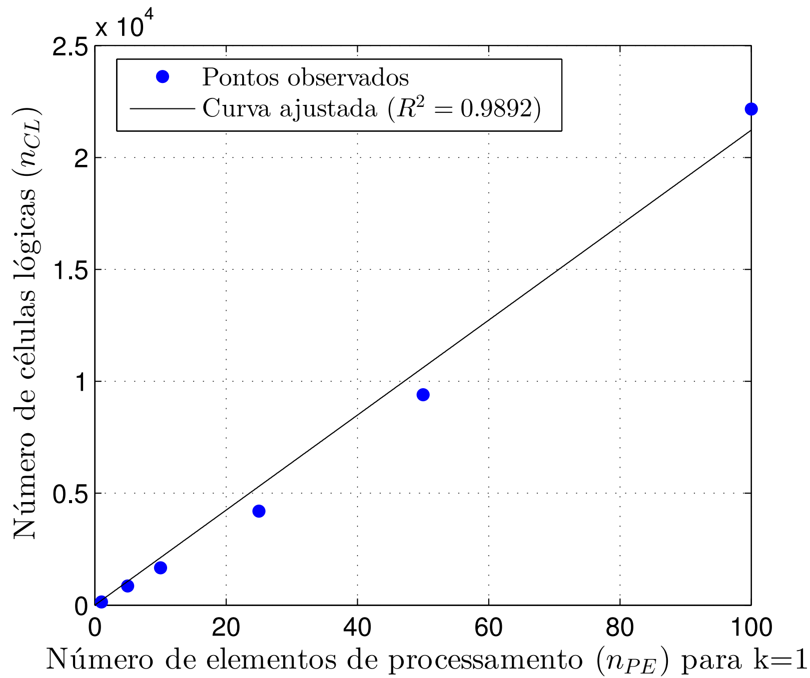

Figura 6.1: Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 1 ($k = 1$).

A ocupação em hardware relacionada à proposta 2 é apresentada na Tabela 6.2. As Figuras 6.1 e 6.2 apresentam a análise de regressão para a primeira e para as demais camadas da rede, respectivamente. A equação da análise de regressão para as células lógicas pode ser expressa por

$$n_{CL} = \begin{cases} 133{,}8n_{PE} + 34{,}17 & \text{para } k = 1 \\ 191{,}7n_{PE} + 129{,}1 & \text{para } k \geq 2, \end{cases} \tag{6.3}$$

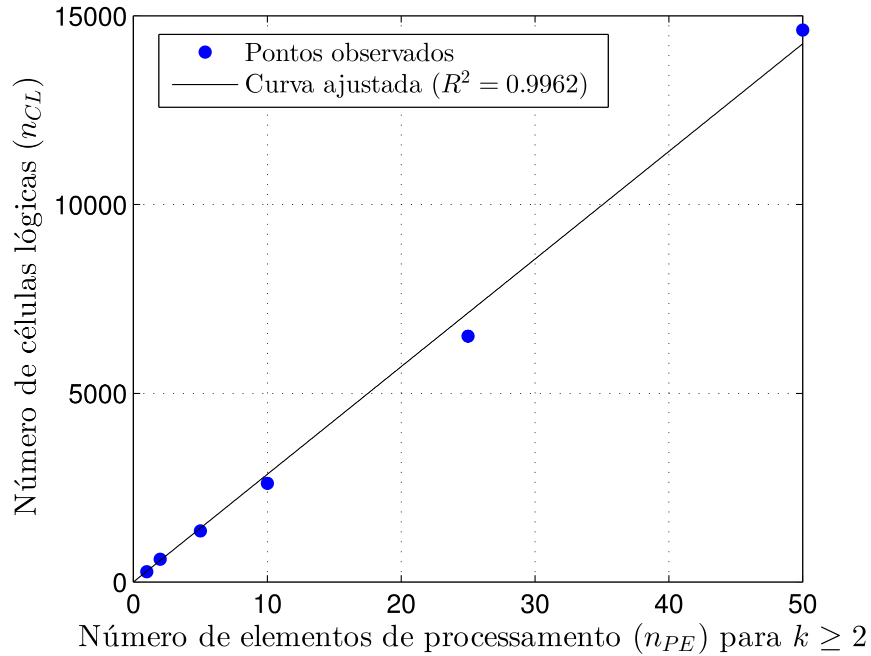

Figura 6.2: Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 1 ($k \geq 2$).

e a ocupação dos multiplicadores pode ser caracterizada pela Equação 6.2.

É importante observar que as Equações 6.1 e 6.3 levam em consideração que a quantidade máxima de multiplicadores DSP, n_{Mult}^{max}, é suficiente para todos os PEs da rede, caso contrário ($n_{Mult} > n_{Mult}^{max}$), a Equação 6.1 pode ser reescrita como

$$n_{CL} = \begin{cases} (212{,}2 + N_{CLM})\alpha + 212{,}2 n_{Mult}^{max} & \text{para } k = 1 \\ (285{,}1 + 2N_{CLM})\alpha + 285{,}1 \frac{n_{Mult}^{max}}{2} & \text{para } k \geq 2, \end{cases} \tag{6.4}$$

em que

$$\alpha = \begin{cases} n_{PE} - n_{Mult}^{max} & \text{para } k = 1 \\ n_{PE} - \frac{n_{Mult}^{max}}{2} & \text{para } k \geq 2, \end{cases} \tag{6.5}$$

e N_{CLM} representa o número de células lógicas necessárias para construir um multiplicador. Seguindo esta mesma ideia, a Equação 6.3 (para $n_{Mult} > n_{Mult}^{max}$) pode ser reescrita como

$$n_{CL} = \begin{cases} (133{,}8 + N_{CLM})\alpha + \beta & \text{para } k = 1 \\ (191{,}7 + 2N_{CLM})\alpha + \beta & \text{para } k \geq 2, \end{cases} \tag{6.6}$$

onde

$$\beta = \begin{cases} 34{,}17 + 133{,}8 n_{Mult}^{max} & \text{para } k = 1 \\ 129{,}1 + 191{,}7 \frac{n_{Mult}^{max}}{2} & \text{para } k \geq 2. \end{cases} \tag{6.7}$$

As Figuras 6.5 e 6.6 exibem uma estimativa de PEs, n_{PE}, para alguns FPGAs comer-

Tabela 6.2: Ocupação em hardware (células lógicas e multiplicadores) por PE para a proposta 2.

Camada	n_{PE}	n_{CL}	n_{Mult}
$k = 1$	1	141 (0,06%)	1 (0,13%)
	5	746 (0,31%)	5 (0,65%)
	10	1384 (0,57%)	10 (1,30%)
	25	3375 (1,40%)	25 (3,26%)
	50	6682 (2,77%)	50 (6,51%)
	100	13432 (5,57%)	100 (13,02%)
$k \geq 2$	1	229 (0,09%)	2 (0,26%)
	2	570 (0,24%)	4 (0,52%)
	5	1151 (0,48%)	10 (1,30%)
	10	2056 (0,85%)	20 (2,60%)
	25	4860 (2,02%)	50 (6,51%)
	50	9735 (4,04%)	100 (13,02%)

ciais (XILINX Virtex-6 2018, XILINX Virtex-7 2018, XILINX Virtex Ultra 2018). Para a estimativa, as Equações 6.1 e 6.4 foram utilizadas para a proposta 1, e as Equações 6.3 e 6.6, para a proposta 2. A partir do datasheet do IP Core do Multiplicador Xilinx (XILINX IP Core 2018*a*, XILINX IP Core 2018*b*), considerou-se a variável $N_{CLM} = 200$. Para todos os valores, considerou-se que 20% dos PEs estão na primeira camada da rede ($k = 1$) e o restante (80% dos PEs) estão nas demais camadas. A Tabela 6.3 apresenta os valores acerca da quantidade máxima de células lógicas e PEs, n_{PE}, para cada proposta.

Tabela 6.3: Estimativa de n_{PE} para alguns FPGAs comerciais.

FPGA	n_{CL}^{max}	n_{Mult}^{max}	n_{PE}	
			Proposta 1	Proposta 2
Virtex 6 XC6VLX240T	241152	768	620	720
Virtex 6 XC6VLX760	758784	864	1470	1720
Virtex 7 XC7V2000T	1954560	2160	3780	4410
Virtex UltraScale VU440	5540850	2880	9700	10000

Os resultados apresentados nas Figuras 6.5 e 6.6 mostram a viabilidade da utilização desta proposta em problemas reais com muitas camadas e neurônios (cada PE_i^k representa o i-ésimo neurônio associado à k-ésima camada).

6.2 Validação

Com a finalidade de validar a implementação proposta neste trabalho, utilizou-se para os experimentos, um banco de imagens de dígitos manuscritos, chamado de MNIST, que contém 60.000 imagens para o conjunto treinamento e 10.000 imagens para o conjunto de testes, disponível em LeCun et al. (2018). Este problema de classificação é um dos mais

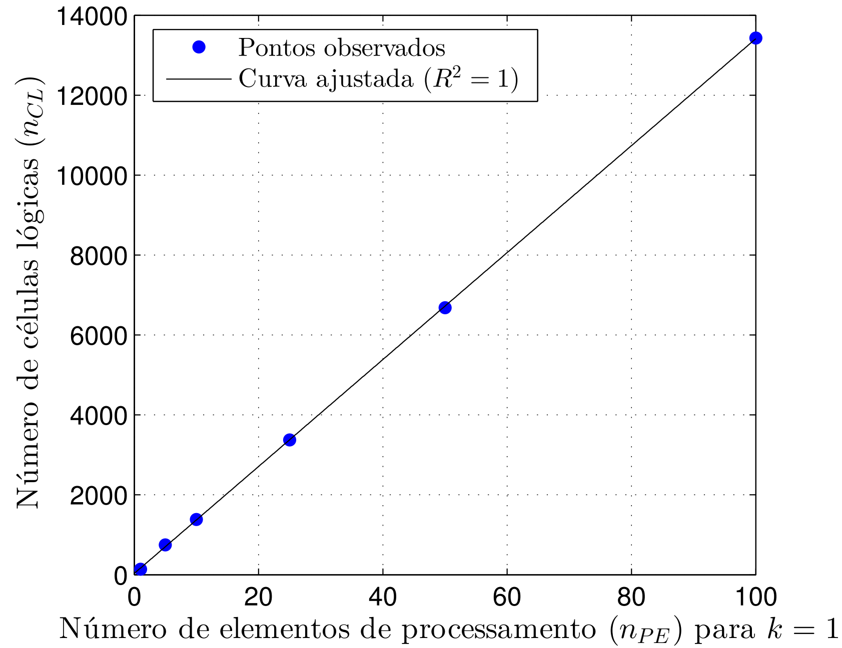

Figura 6.3: Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 2 ($k = 1$).

utilizados nos testes de pesquisas em *Deep Learning* (Goodfellow et al. 2016). A Figura 6.7 apresenta algumas imagens da base da dados MNIST. Cada imagem possui 28×28 pixels, totalizando 784 entradas para o SSAE. Os experimentos foram realizados com as 10.000 imagens do conjunto de testes do MNIST. O treinamento da rede foi realizado previamente utilizando as 60.000 imagens do conjunto de testes na plataforma de simulação Matlab/Simulink[1] (The MathWorks 2018), utilizando o algoritmo do Gradiente Conjugado Escalonado (*Scaled Conjugate Gradient* - SCG) (Moller 1993), e o FPGA Virtex 6 xc6vlx240t-1ff1156.

6.2.1 Validação da Proposta 1

Para a validação do hardware foi realizada uma comparação entre os resultados obtidos pela implementação na plataforma Matlab/Simulink e os resultados obtidos pela implementação em hardware. Sendo assim, foi calculado o erro quadrático médio (*Mean Square Error* - MSE) entre a saída da implementação em Matlab, $s_i^{ref}(n)$, e a saída da implementação em hardware. O cálculo do MSE, para este experimento, é definido como

$$MSE = \frac{1}{H \times 10.000} \sum_{i=1}^{H} \sum_{n=1}^{10000} \left(s_i^{ref}(n) - s_i(n) \right)^2. \tag{6.8}$$

Com isso, verificou-se que o MSE entre os resultados da implementação em Matlab,

[1] *License number* 1080073

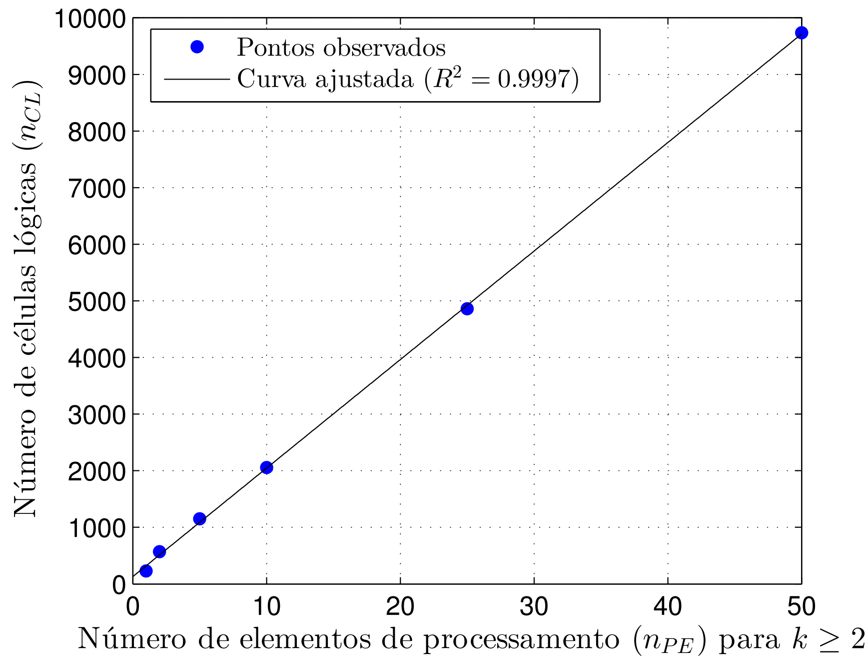

Figura 6.4: Curva de regressão linear relacionada a ocupação em hardware (células lógicas) por PE para a proposta 2 ($k \geq 2$).

que utiliza ponto flutuante (64 bits), e da implementação em hardware, em ponto fixo, foi de apenas $2,1 \times 10^{-6}$, que é um resultado bastante aceitável, já que os pesos associados a implementação em hardware utilizam apenas 12 bits na parte fracionária. Entre as 10.000 imagens utilizadas na validação, a implementação em Matlab obteve uma quantidade de acertos de 9.340 imagens, enquanto a implementação em hardware aqui proposta conseguiu acertar 9.338 imagens. Sendo assim, apenas 2 imagens foram classificadas incorretamente. A pequena porcentagem de erro na classificação da implementação em FPGA com relação à implementação em Matlab, está relacionada a resolução de bits utilizada para os pesos sinápticos da rede. No entanto, este resultado ainda é bastante relevante pois indica que não é necessária uma alta resolução em bits (como 64 bits, por exemplo) para alcançar resultados significativos, mostrando que ao utilizar apenas 12 bits na parte fracionária, em ponto fixo, é possível garantir a redução na ocupação de área de hardware, além de promover o aumento do *throughput* (de Souza & Fernandes 2014, Jiang et al. 2016, Finnerty & Ratigner 2017).

Após a validação da implementação em hardware com o Matlab, foi realizada a síntese para obter o relatório de ocupação dos recursos do FPGA para a arquitetura de rede implementada ($784-100-50-10$). A Tabela 6.4 detalha os dados relacionados a ocupação de área de hardware do circuito implementado no FPGA. A primeira coluna apresenta a quantidade de multiplicadores utilizados. Os multiplicadores são consumidos nos PEs de cada k-ésima camada. A segunda coluna exibe a quantidade de registradores e a terceira coluna, a quantidade de células lógicas utilizadas como LUT, n_{LUT}, em todo o circuito.

Os dados apresentados na Tabela 6.4 evidenciam a viabilidade da implementação da

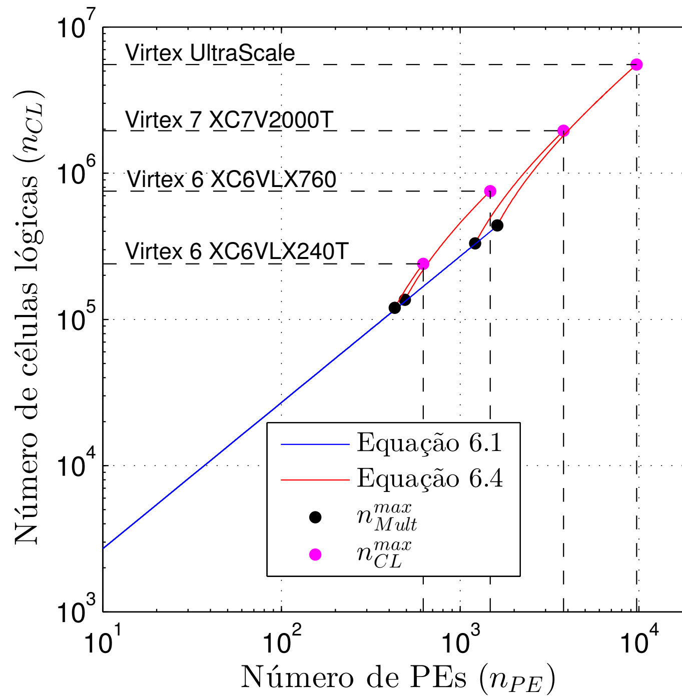

Figura 6.5: Escalabilidade da proposta 1 com relação ao número de PEs (n_{PE}).

Tabela 6.4: Síntese da proposta 1 - Taxa de ocupação

Multiplicadores	Registradores	n_{LUT}
220 (28%)	29.445 (9%)	23.623 (15%)

proposta 1 deste trabalho. Verifica-se que foram ocupados apenas 9% dos registradores e 15% das células lógicas utilizadas como LUT do FPGA alvo, mostrando que ainda existe bastante espaço para adição de novas camadas e neurônios. Um dos elementos que foram mais utilizados foram os multiplicadores, em torno de 28%, dado que cada PE da primeira camada consome um, e os PEs das outras camadas consomem dois, devido ao *bias*. Apesar disso, constata-se que ainda é possível aumentar bastante o número de camadas e neurônios do SSAE.

A Tabela 6.5 apresenta informações à respeito do tempo de processamento do circuito da proposta 1, considerando a menor quantidade de *streams* de pesos necessária. A primeira coluna expõe o tempo do contador em anel da primeira camada oculta da rede, t_{ca}. A segunda coluna exibe o atraso inicial da arquitetura, d, definido pela Equação 5.2. Na terceira coluna é mostrado o tempo de execução do *feedforward* da rede, t_{ff}, expresso pela Equação 5.4, após o atraso inicial (Equação 5.2), a terceira coluna apresenta o *throughput* da rede, th_{ff}, determinado pela Equação 5.3, que neste trabalho consiste na quantidade de imagens classificadas por segundo. A última coluna apresenta a potência dinâmica consumida, P_d, pelo circuito da proposta 1. A potência foi obtida utilizando a ferramenta XPower Analyzer da empresa Xilinx. Vale ressaltar que a potência estática do FPGA Virtex 6xc6vlx240t-1ff1156, utilizado neste trabalho, corresponde a cerca de

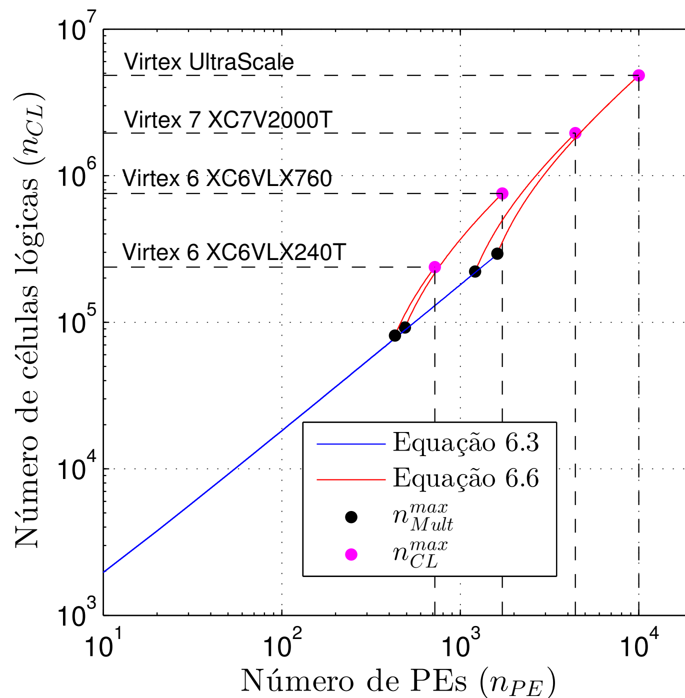

Figura 6.6: Escalabilidade da proposta 2 com relação ao número de PEs (n_{PE}).

3,42 W.

Tabela 6.5: Síntese da proposta 1 - Associação ao tempo de processamento

t_{ca}	Delay (d)	Tempo de *feedforward* (t_{ff})	*Throughput* (th_{ff})	Potência consumida (P_d)
$10\,ns$	$\leq 2{,}4\,ms$	$\leq 0{,}8\,ms$	$\geq 1{,}25$ KFPS	0,003 W

Os dados apresentados na Tabela 6.5 são bastante significativos. Considerando-se a estrutura que utiliza a menor quantidade de *streams* de pesos necessária, após a síntese obteve-se um tempo de $10ns$ para a operação do contador em anel da primeira camada oculta da rede. Com isso, após o atraso inicial de apenas $2,4ms$ é possível obter as saídas de todos os neurônios da última camada, que neste caso é a classificação de uma imagem, a cada $0,8ms$. Estes valores permitem alcançar um *throughput* de pelo menos 1.250 imagens classificadas por segundo. Este é o valor do *throughput* mínimo atingido por esta proposta, ou seja, seu *lower bound*, já que a implementação utilizou a menor quantidade de *streams* de pesos necessária, o que implica no aumento do tempo de amostragem do circuito, e consequentemente, no aumento do delay, d, e do tempo de *feedforward*, t_{ff}, da implementação. Ao paralelizar estes *streams* é possível atingir resultados ainda mais expressivos. Além disso, a potência consumida pelo circuito foi de apenas 0,003 W. De todo modo, os resultados apresentados revelam a possibilidade da utilização desta proposta de DNN em problemas de dados massivos.

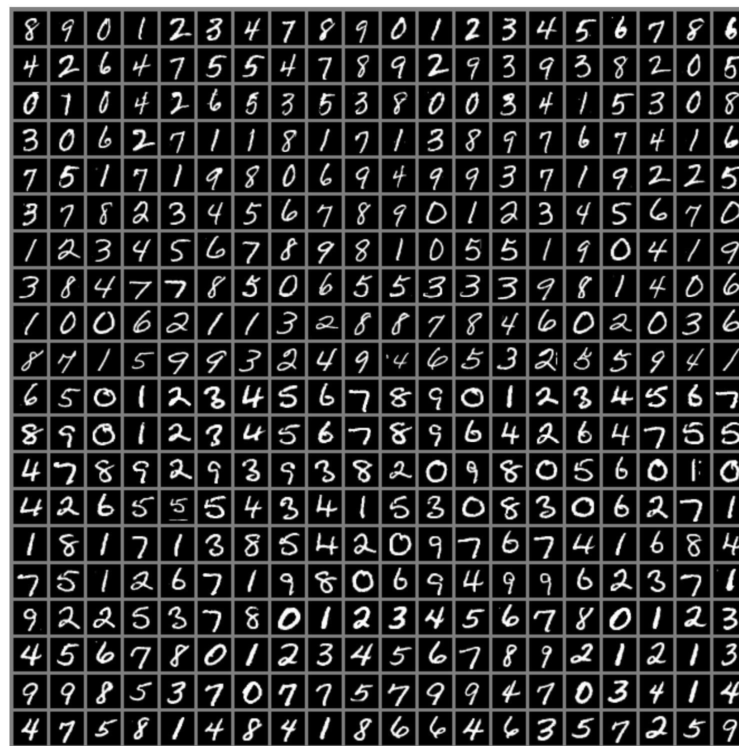

Figura 6.7: Exemplos de imagens da base de dados MNIST.
Fonte: Goodfellow et al. (2016).

6.2.2 Validação da Proposta 2

Os resultados da validação do hardware com o Matlab para a proposta 1, apresentados na Subseção 6.2.1 para um FPGA Virtex 6 xc6vlx240t-1ff1156, também se aplicam para a proposta 2. No entanto, foram obtidos novos relatórios referentes a taxa de ocupação dos recursos de hardware e ao tempo de processamento, além da potência consumida, após a realização da síntese da implementação da segunda proposta, que serão apresentados a seguir.

A Tabela 6.6 apresenta os dados de síntese relacionados a ocupação de área da proposta 2 no FPGA. As colunas desta tabela estão organizadas de acordo com o que foi apresentado para a Tabela 6.4 na Subseção 6.2.1, referente a proposta 1.

Tabela 6.6: Síntese da proposta 2 - Taxa de ocupação

Multiplicadores	Registradores	n_{LUT}
220 (28%)	11.898 (3%)	15.249 (10%)

Os dados apresentados na Tabela 6.6 mostram que a proposta 2 conseguiu otimizar ainda mais a utilização dos recursos de hardware do FPGA alvo em comparação com a proposta 1. Isto foi possível devido a utilização das LUTs destinadas a guardar os pesos

sinápticos em cada *i*-ésimo PE_i^k, dispensando a implementação de alguns circuitos utilizados na proposta 1. Verifica-se que foram ocupados apenas 3% dos registradores e 10% das células lógicas utilizadas como LUT. A quantidade de multiplicadores utilizados permaneceu a mesma com relação a proposta anterior, em torno de 28%, dado que não houve alteração na maneira como as computações são feitas em cada PE_i^k. Contudo, os dados apresentados para a proposta 2 reafirmam a possibilidade de aumentar significativamente o número de camadas e neurônios da rede, mostrando resultados ainda mais satisfatórios que a proposta anterior.

A Tabela 6.7 apresenta informações à respeito do tempo de processamento do circuito da segunda proposta. A primeira coluna desta tabela expõe o tempo de processamento do PE, t_{PE}, e as demais colunas estão organizadas de acordo com o que foi apresentado para a Tabela 6.5 na Subseção 6.2.1, referente a proposta 1.

Tabela 6.7: Síntese da proposta 2 - Associação ao tempo de processamento

t_{PE}	Delay (d)	Tempo de *feedforward* (t_{ff})	*Throughput* (th_{ff})	Potência consumida (P_d)
$48\,ns$	$0{,}1\,ms$	$0{,}03\,ms$	26 KFPS	0,112 W

Os dados apresentados na Tabela 6.7, para a proposta 2, são bastante expressivos. O tempo de execução do PE obtido foi de apenas $48\,ns$. Além disso, nesta proposta o atraso inicial é de apenas $0{,}1\,ms$, e o tempo do *feedforward* de apenas $0{,}03\,ms$. Com isso, foi possível alcançar um *throughput* de 26.000 imagens classificadas por segundo, que é um valor aproximadamente 20 vezes superior ao valor obtido na proposta 1, utilizando a menor quantidade possível de *streams* de pesos. Além disso, a potência consumida pelo circuito da proposta 2 foi de apenas 0,112 W. O circuita da proposta 1 obteve uma potência inferior devido a utilização dos *streams* de pesos, que obriga o circuito a operar em uma frequência menor. No entanto, a frequência do circuito da proposta 2 poderia ser diminuída para reduzir o consumo de potência. Por fim, constatou-se que a utilização das LUTs para armazenar os pesos de cada neurônio da rede proporcionou o aumento do *throughput* obtido em comparação com a proposta anterior, em contrapartida, provocou o aumento da potência consumida. Entretanto, ao utilizar o limite máximo de *streams* de pesos por camada, a proposta 1 tende a atingir os resultados da proposta 2. Sendo assim, ambas propostas mostraram-se viáveis de serem utilizadas em problemas de dados massivos, como problemas envolvendo dados de vídeo.

6.3 Comparação com o estado da arte

A partir dos resultados apresentados nas seções anteriores, esta seção destina-se a fazer comparações dos resultados obtidos pela implementação de cada proposta aqui apresentada com os resultados de um trabalho do estado da arte, que também implementou uma proposta de SSAE em FPGA. O trabalho relacionado também é aplicado a um problema de classificação, utiliza um banco de imagens RGB de 32×32 pixels para validação e faz uso de um FPGA equivalente em termos de processamento.

Do ponto de vista da proposta 2, que mais se assemelha ao trabalho da literatura por utilizar recursos de memória para armazenar os pesos da rede, foi possível obter os resultados apresentados na Tabela 6.8. A primeira coluna desta tabela apresenta a referência comparada e na segunda coluna é informado o dispositivo alvo do trabalho. Na terceira coluna é mostrada a arquitetura da rede implementada. Na quarta coluna é apresentado o *throughput* obtido pelo trabalho relacionado, th_R, e nas duas últimas colunas são exibidos o *throughput*, th_{P2} (ou th_{ff} da proposta 1), e o *speedup* que seriam alcançados pela segunda proposta aqui apresentada, caso uma arquitetura com as mesmas configurações do trabalho relacionado fosse implementada.

Tabela 6.8: Comparação da proposta 2 com um trabalho relacionado

Referência	Dispositivo	Arquitetura da rede	th_R (FPS)	th_{P2} (FPS)	*Speedup*
(Maria et al. 2016)	Altera Stratix V D5	3072 – 2000 – 750 – 10	45	5208	115×

A partir da Tabela 6.8, constatou-se que a implementação da proposta 2 deste trabalho conseguiria trabalhar cerca de 115 vezes mais rápido do que a arquitetura apresentada no trabalho de Maria et al. (2016). É importante destacar que a arquitetura implementada aqui é menor que a arquitetura apresentada em Maria et al. (2016), apesar disso, esta proposta de implementação conseguiria atingir um *throughput* elevado caso a arquitetura de Maria et al. (2016) fosse implementada, o que reafirma a possibilidade de acrescentar mais camadas e neurônios à rede implementada neste trabalho.

Todavia, não faz sentido comparar a proposta 1 com este trabalho relacionado, já que nele não se deixa claro propor a utilização de *streams* para as entradas dos pesos na rede. Porém, se esta comparação for realizada, serão obtidos como resultados para a proposta com a quantidade mínima de *streams* por camada, os valores apresentados na Tabela 6.9, que tem suas quatro primeiras colunas similares as da Tabela 6.8, e em sua última coluna é apresentado o *throughput* que seria obtido pela proposta 1, th_{P1}, caso a arquitetura do trabalho relacionado fosse implementada.

Tabela 6.9: Comparação da proposta 1 com um trabalho relacionado

Referência	Dispositivo	Arquitetura da rede	th_R (FPS)	th_{P1} (FPS)
(Maria et al. 2016)	Altera Stratix V D5	3072 – 2000 – 750 – 10	45	$\geq$ 12,5

Os dados apresentados na Tabela 6.9 devem ser analisados por outra perspectiva. O menor valor de *throughput* obtido pela primeira proposta aqui apresentada foi de 12,5 FPS. Apesar do valor do *throughput* obtido parecer inferior ao *throughput* de referência, este valor trata-se do limite inferior alcançado (*lower bound*), devido a utilização da quantidade mínima de *streams* de pesos por camada. De modo que, ao paralelizar os *streams* de pesos das camadas ocultas em apenas 4 vezes, já seria possível ultrapassar o valor do *throughput* atingido pelo trabalho relacionado, tendo em vista que $12{,}5 \times 4 = 50$ FPS,

com a vantagem de permitir facilmente a alteração dos valores dos pesos sinápticos da rede.

Capítulo 7

Considerações Finais

Neste capítulo são apresentadas as considerações finais do trabalho, evidenciando as principais conclusões à respeito de cada implementação realizada. Também serão expostas as perspectivas futuras com a finalização deste trabalho.

7.1 Conclusões

Este trabalho apresentou uma proposta de implementação em hardware da técnica de *Deep Learning*, *Stacked Sparse Autoencoder*. Foi implementada a fase *feedforward* da DNN, utilizando ponto fixo e design em RTL. Em toda a implementação foi aplicada a técnica de matriz sistólica, que permitiu utilizar muitos neurônios nas várias camadas da rede, além de possibilitar a obtenção dos resultados em baixo tempo de processamento. Todos os detalhes da implementação foram apresentados bem como resultados relativos às sínteses para ocupação em hardware, tempo de processamento e consumo de potência de duas implementações distintas, para um FPGA Virtex 6 xc6vlx240t-1ff1156. Além disso, foi apresentada uma análise de ocupação em hardware associada a quantidade de PEs utilizados no FPGA.

A validação do hardware com o Matlab indicou que não é necessária um resolução em ponto flutuante para atingir resultados satisfatórios, evidenciando as vantagens da utilização de resolução em ponto fixo para redução da ocupação em hardware e aumento do *throughput*. Os resultados da análise de ocupação em hardware mostraram a possibilidade de adicionar arquiteturas de redes com muitas camadas e muitos neurônios em diferentes FPGAs comerciais, possibilitando sua aplicação em problemas reais.

Além disso, os resultados mostraram que ambas implementações conseguem atingir *throughputs* elevados. No entanto, a utilização de LUTs para armazenar os pesos sinápticos de cada neurônio na proposta 2, possibilitou a redução da área ocupada no FPGA, além de garantir o aumento do *throughput* obtido, em comparação com o valor *lower bound* do *throughput* da proposta 1. Todavia, o valor do *throughput* atingido pela proposta 1 pode ser aumentado em consonância com o aumento da quantidade de *streams* de pesos por camada da rede, podendo tender ao valor do *throughput* obtido pela proposta 2, com a vantagem de permitir que em aplicações práticas, um mesmo hardware possa ser utilizado para problemas distintos, ao possibilitar que novos pesos sejam adicionados à rede através de *streams* de pesos, após a realização de um treinamento prévio da rede.

Sendo assim, ambas propostas mostraram-se viáveis de serem utilizadas em problemas de dados massivos.

Ademais, os circuitos implementados neste trabalho obtiveram um baixo consumo de potência, que pode ser um requisito importante para diversas aplicações. Por fim, as comparações com um trabalho relacionado também reafirmaram as contribuições das implementações propostas neste trabalho, mostrando a possibilidade de atingir *speedups* elevados.

7.2 Perspectivas Futuras

Com a finalização deste trabalho, algumas perspectivas futuras podem ser elencadas. A primeira delas consiste em investigar a implicação da quantidade de bits utilizados na parte fracionária dos pesos da rede com relação a acurácia da classificação no hardware e a ocupação de área no FPGA. Também podem ser investigadas alternativas de otimização do circuito para elevar ainda mais os ganhos alcançados, principalmente com relação a proposta 1.

Outra perspectiva futura consiste em analisar as possibilidades da implementação do treinamento do SSAE utilizando computação reconfigurável ou computação heterogênea. Além disso, a arquitetura da matriz sistólica aqui proposta poderá ser utilizada para a implementação em hardware de outras técnicas de aprendizagem profunda. Ademais, objetiva-se também trabalhar com a técnica *Deep Q-learning*, que consiste na combinação de técnicas de *Deep Learning* com a técnica de *Q-learning*.

Referências Bibliográficas

Azarian, Ali & Mahmood Ahmadi (2009), Reconfigurable computing architecture survey and introduction, *em* 'Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on', IEEE, pp. 269–274.

Baldi, Pierre (2012), Autoencoders, unsupervised learning, and deep architectures, *em* 'Proceedings of ICML Workshop on Unsupervised and Transfer Learning', pp. 37–49.

Bengio, Yoshua, Aaron Courville & Pascal Vincent (2013), 'Representation learning: A review and new perspectives', *IEEE transactions on pattern analysis and machine intelligence* **35**(8), 1798–1828.

Bettoni, M., G. Urgese, Y. Kobayashi, E. Macii & A. Acquaviva (2017), A convolutional neural network fully implemented on fpga for embedded platforms, *em* '2017 New Generation of CAS (NGCAS)', pp. 49–52.

Costa, Cesar da (2009), *Projetos de circuitos digitais com FPGA*, São Paulo: Editora Érica.

de Souza, Alisson C. D. & Marcelo A. C. Fernandes (2014), 'Parallel fixed point implementation of a radial basis function network in an fpga', *Sensors* **14**(10), 18223–18243.

Deng, Li, Dong Yu et al. (2014), 'Deep learning: methods and applications', *Foundations and Trends® in Signal Processing* **7**(3–4), 197–387.

Deotale, Prashant D & Lalit Dole (2014), Design of fpga based general purpose neural network, *em* 'Information Communication and Embedded Systems (ICICES), 2014 International Conference on', IEEE, pp. 1–5.

Finnerty, Ambrose & Hervé Ratigner (2017), Reduce power and cost by converting from floating point to fixed point, *em* 'WP491 (v1. 0)'.

Fletcher, Roger (1975), *Practical methods of optimization*, John Wiley & Sons.

Floyd, Thomas (2009), *Sistemas digitais: fundamentos e aplicações*, Bookman Editora.

Goodfellow, Ian, Yoshua Bengio & Aaron Courville (2016), *Deep Learning*, MIT press.

Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan & Pritish Narayanan (2015), Deep learning with limited numerical precision, *em* 'International Conference on Machine Learning', pp. 1737–1746.

Hauck, Scott & Andre DeHon (2010), *Reconfigurable computing: the theory and practice of FPGA-based computation*, Vol. 1, Elsevier.

Haykin, Simon (2001), *Redes neurais: princípios e prática*, Bookman Editora.

Huynh, T. V. (2017), Deep neural network accelerator based on fpga, *em* '2017 4th NAFOSTED Conference on Information and Computer Science', pp. 254–257.

Jiang, J, R Hu, D Wang, J Xu & Y Dou (2016), 'Performance of the fixed-point autoencoder', **23**, 77–82.

Jin, Youngjae & Daeshik Kim (2014), 'Unsupervised feature learning by pre-route simulation of auto-encoder behavior model', *International Journal of Computer, Electrical, Automation, Control and Information Engineering* **8**(5), 706 – 710.

Kung, H. T. & Charles E. Leiserson (1978), *Systolic arrays (for VLSI)*, I.S. Duff and C.G. Stewart. Eds., Proceedings Symposium on Sparse Matrix Computations.

Kuon, Ian & Jonathan Rose (2007), 'Measuring the gap between fpgas and asics', *IEEE Transactions on computer-aided design of integrated circuits and systems* **26**(2), 203–215.

Kurzweil, R. (1990), *The age of intelligent machines*, Vol. 579, MIT press Cambridge, MA.

LeCun, Yann, Corinna Cortes & Christopher J.C. Burges (2018), 'Yann LeCun's Home Page', http://yann.lecun.com/exdb/mnist/.

Liu, Yunfan, Xueshi Hou, Jiansheng Chen, Chang Yang, Guangda Su & Weibei Dou (2014), Facial expression recognition and generation using sparse autoencoder, *em* '2014 International Conference on Smart Computing (SMARTCOMP)', IEEE, pp. 125–130.

Ma, Yufei, Naveen Suda, Yu Cao, Sarma Vrudhula & Jae sun Seo (2018), 'Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler', *Integration* **62**, 14 – 23.

Maria, Joao, Joao Amaro, Gabriel Falcao & Luís A. Alexandre (2016), 'Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems', *Neural Process. Lett.* **43**(2), 445–458.

Moller, Martin Fodslette (1993), 'A scaled conjugate gradient algorithm for fast supervised learning', *Neural networks* **6**(4), 525–533.

Moss, D. J. M., D. Boland, P. Pourbeik & P. H. W. Leong (2018), Real-time fpga-based anomaly detection for radio frequency signals, *em* '2018 IEEE International Symposium on Circuits and Systems (ISCAS)', pp. 1–5.

Noronha, Daniel Holanda (2017), Proposta de implementação em fpga de máquina de vetores de suporte (svm) utilizando otimização sequencial mínima (smo), Dissertação de mestrado, Brasil.

Norvig, Peter & Stuart Russell (2014), *Inteligência Artificial: Tradução da 3a Edição*, Vol. 1, Elsevier Brasil.

Ponti, Moacir Antonelli & Gabriel B Paranhos da Costa (2018), 'Como funciona o deep learning', *arXiv preprint arXiv:1806.07908* .

Pratiher, Sawon, Subhankar Chattoraj & Karan Vishwakarma (2018), Application of stacked sparse autoencoder in automated detection of glaucoma in fundus images, *em* 'Unconventional Optical Imaging', Vol. 10677, International Society for Optics and Photonics, p. 106772X.

Qi, Yumei, Changqing Shen, Dong Wang, Juanjuan Shi, Xingxing Jiang & Zhongkui Zhu (2017), 'Stacked sparse autoencoder-based deep network for fault diagnosis of rotating machinery', *IEEE Access* **5**, 15066–15079.

Schmidhuber, Jürgen (2015), 'Deep learning in neural networks: An overview', *Neural networks* **61**, 85–117.

Shang, Li, Alireza S Kaviani & Kusuma Bathala (2002), Dynamic power consumption in virtex^{TM}-ii fpga family, *em* 'Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays', ACM, pp. 157–164.

Sharma, Hardik, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra & Hadi Esmaeilzadeh (2016), From high-level deep neural models to fpgas, *em* 'The 49th Annual IEEE/ACM International Symposium on Microarchitecture', IEEE Press, p. 17.

Silva, Carlos Alberto de Albuquerque (2015), Implementação paralela de uma matriz de neurônios dinamicamente reconfiguravel para descrição de topologias de redes neurais artificiais Multilayer Perceptrons, Tese de doutorado, Universidade Federal do Rio Grande do Norte, UFRN, Natal, RN.

Suzuki, Akihiro, Takashi Morie & Hakaru Tamukoh (2018), 'A shared synapse architecture for efficient fpga implementation of autoencoders', *PLOS ONE* **13**(3), 1–22.

The MathWorks (2018), 'Matlab/Simlink', https://www.mathworks.com/.

Todman, Timothy J, George A Constantinides, Steven JE Wilton, Oscar Mencer, Wayne Luk & Peter YK Cheung (2005), 'Reconfigurable computing: architectures and design methods', *IEE Proceedings-Computers and Digital Techniques* **152**(2), 193–207.

Torquato, Matheus Fernandes (2017), Proposta de implementação paralela de algoritmo genético em fpga, Dissertação de mestrado, Universidade Federal do Rio Grande do Norte, UFRN, Natal, RN.

Vincent, Pascal, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio & Pierre-Antoine Manzagol (2010), 'Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion', *J. Mach. Learn. Res.* **11**, 3371–3408.

Wang, Chao, Lei Gong, Qi Yu, Xi Li, Yuan Xie & Xuehai Zhou (2017), 'Dlau: A scalable deep learning accelerator unit on fpga', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **36**(3), 513–517.

Winston, P. H. (1992), *Artificial Intelligence*, Addison-Wesley (3rd Edition).

XILINX IP Core (2018*a*), 'LogiCORE IP Multiplier v11.2', https://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf.

XILINX IP Core (2018*b*), 'LogiCORE IP Multiplier v12.0', https://www.xilinx.com/support/documentation/ip_documentation/mult_gen/v12_0/pg108-mult-gen.pdf.

XILINX Virtex-6 (2018), 'Virtex-6 Family Overview', https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

XILINX Virtex-7 (2018), '7 Series FPGAs Data Sheet: Overview', https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

XILINX Virtex Ultra (2018), 'UltraScale Architecture and Product Data Sheet: Overview', https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

Xu, Jun, Lei Xiang, Qingshan Liu, Hannah Gilmore, Jianzhong Wu, Jinghai Tang & Anant Madabhushi (2016), 'Stacked sparse autoencoder (ssae) for nuclei detection on breast cancer histopathology images', *IEEE transactions on medical imaging* **35**(1), 119–130.

Zhang, Chen, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan & Jason Cong (2018), 'Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* .

Zhang, Chen, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao & Jason Cong (2015), Optimizing fpga-based accelerator design for deep convolutional neural networks, *em* 'Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays', ACM, pp. 161–170.

Zhang, Xiaofan, Hang Dou, Tao Ju, Jun Xu & Shaoting Zhang (2016), 'Fusing heterogeneous features from stacked sparse autoencoder for histopathological image analysis', *IEEE journal of biomedical and health informatics* **20**(5), 1377–1383.

Zhao, Wei, Zuchen Jia, Xiaosong Wei & Hai Wang (2018), 'An fpga implementation of a convolutional auto-encoder', *Applied Sciences* **8**(4), 504.

Zhou, Yongmei & Jingfei Jiang (2015), *em* '2015 4th International Conference on Computer Science and Network Technology (ICCSNT)', Vol. 01, pp. 829–832.