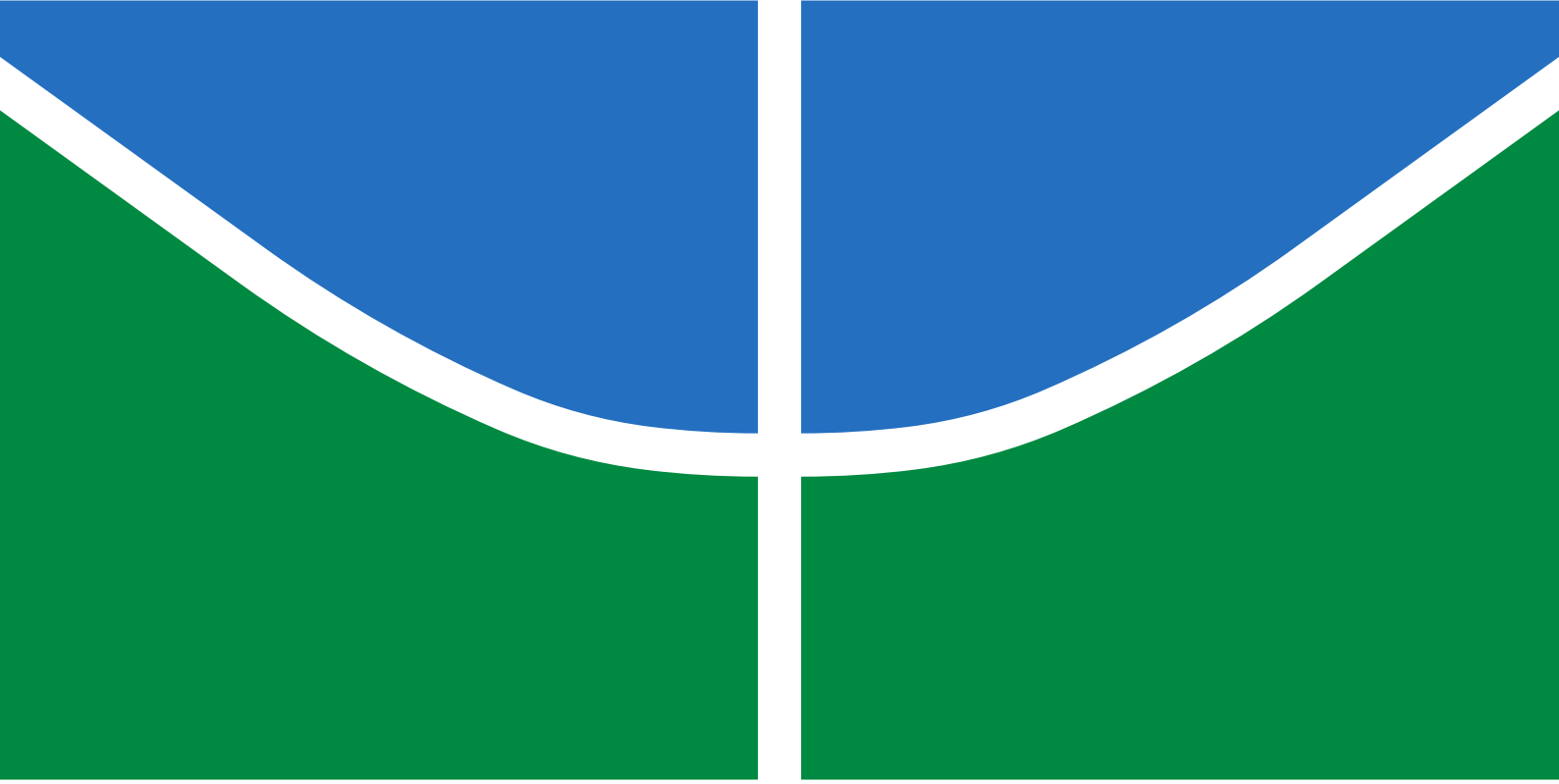

ACELERADOR DE HARDWARE EM FPGA PARA APLICAÇÕES EM APRENDIZADO DE MÁQUINA

HIANDRA FONSECA TOMASI

TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ACELERADOR DE HARDWARE EM FPGA PARA APLICAÇÕES EM APRENDIZADO DE MÁQUINA

HIANDRA FONSECA TOMASI

Orientador: PROF. DR. DANIEL CHAVES CAFÉ, ENE/UNB

TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

BRASÍLIA-DF, 16 DE DEZEMBRO DE 2020.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ACELERADOR DE HARDWARE EM FPGA PARA APLICAÇÕES EM APRENDIZADO DE MÁQUINA

HIANDRA FONSECA TOMASI

TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO SUBMETIDO AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM ENGENHARIA ELÉTRICA.

APROVADA POR:

Prof. Dr. Daniel Chaves Café, ENE/UnB
Orientador

Prof. Dr. Alexandre Ricardo Soares Romariz, ENE/UnB
Examinador interno

Prof. Dr. Alexandre Solon Nery, ENE/UnB
Examinador interno

BRASÍLIA, 16 DE DEZEMBRO DE 2020.

FICHA CATALOGRÁFICA
HIANDRA FONSECA TOMASI
Acelerador de Hardware em FPGA para Aplicações em Aprendizado de Máquina
2020xvi, 54p., 201x297 mm
(ENE/FT/UnB, Bacharel, Engenharia Elétrica, 2020)
Trabalho de Conclusão de Curso de Graduação - Universidade de Brasília
Faculdade de Tecnologia - Departamento de Engenharia Elétrica

REFERÊNCIA BIBLIOGRÁFICA

HIANDRA FONSECA TOMASI (2020) Acelerador de Hardware em FPGA para Aplicações em Aprendizado de Máquina. Trabalho de Conclusão de Curso de Graduação em Engenharia Elétrica, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 54p.

CESSÃO DE DIREITOS

AUTOR: Hiandra Fonseca Tomasi
TÍTULO: Acelerador de Hardware em FPGA para Aplicações em Aprendizado de Máquina.
GRAU: Bacharel ANO: 2020

_____

Hiandra Fonseca Tomasi

Agradecimentos

A realização deste trabalho de conclusão de curso foi apenas possível em decorrência do apoio direto e indireto de diversas pessoas, às quais gostaria de agradecer.

Ao professor Daniel Chaves Café, que aceitou me orientar na realização deste trabalho de conclusão de curso, e que também foi meu orientador do projeto de iniciação científica, além de professor em disciplinas da área de sistemas embarcados. Sou grata pela sua dedicação em esclarecer minhas dúvidas, pela paciência e pela atenção.

Aos demais professores da Universidade de Brasília que contribuíram para a minha formação, especialmente àqueles que me motivaram pela excelente qualidade de suas aulas e ensinamentos. Dentre eles gostaria de agradecer explicitamente ao professor Celius Antônio Magalhães, cujos ensinamentos extrapolaram a sua disciplina, sempre me incentivando a aprender novas ferramentas; e aos professores Eduardo Peixoto Fernandes da Silva, João Carlos Nascimento de Pádua, Cláudia Jacy Barenco Abbas, Lúcio Martins da Silva e Alexandre Ricardo Soares Romariz.

Não é possível citar todos os seus nomes, mas é fundamental agradecer também a todos os professores que tive da pré educação ao ensino médio, visto que cada um contribuiu enormemente para a minha formação compartilhando comigo parte dos seus vastos conhecimentos.

Aos meus pais Maria Ivanir Fonseca Tomasi e Euclides Tomasi, que sempre me deram apoio incondicional em todos os aspectos possíveis e que são minha maior fonte de inspiração e motivação. Sou também eternamente grata por terem me proporcionado acesso à educação de qualidade, que constitui o alicerce da minha formação.

Aos meus avós, em especial ao *nonno* Silverio Orestes Tomasi (*in memorian*) e à minha avó Nair Ribeiro da Fonseca, cujas histórias de vida e de sucessivas superações sempre me incentivaram a seguir em frente independentemente das dificuldades encontradas pelo caminho. Também sou grata por terem me estimulado mesmo à distância e à minha avó por compreender minhas ausências durante o tempo dedicado aos estudos.

À minha tia e madrinha Valdelice Ribeiro Fonseca, que sempre esteve presente demonstrando seu apoio e que jamais hesitou em me ajudar quando precisei.

Ao meu namorado e melhor amigo Rodrigo Andres Rodrigues Fischer, cujo apoio e cumplicidade foram fundamentais não só para a conclusão deste trabalho, como também

para sobrevivermos ao dia a dia na universidade.

Por fim, agradeço aos maravilhosos amigos que pude fazer durante a graduação em engenharia elétrica. A troca de experiências, o suporte e os momentos de diversão com muitos jogos, além da cumplicidade gerada por estarmos no "mesmo barco" foram fundamentais durante os anos de graduação. Dentre eles gostaria de agradecer explicitamente ao meu amigo Diego Neves de Lemos, com quem realizei diversos trabalhos e compartilhei bastantes experiências acadêmicas.

Resumo

A popularização do uso do aprendizado de máquinas no sentido de que sua utilização é crescente nas mais variadas aplicações tecnológicas tem criado uma demanda também crescente por poder computacional e energia. Estes fatores, por serem fundamentais para o treinamento e funcionamento das redes neurais, tornam também limitante a sua utilização quando os recursos necessários não podem ser fornecidos ou quando o tempo de espera de execução dos algoritmos é tão grande que desestimula ou impede o andamento de pesquisas. Esses problemas tem sido acentuados quando se coloca em perspectiva também a questão ambiental associada às altas demandas de energia. Nesse contexto, tornou-se fundamental a busca por alternativas às GPUs e CPUs, as quais além de não poderem ser utilizadas em aplicações de pequeno porte, como *smartphones* e relógios inteligentes, demandam grande quantidade de energia. Uma das alternativas consiste no uso de FPGAs (*Field Programmable Gate Arrays*) para prover um *hardware* dedicado à realização das operações mais custosas dos algoritmos: as matriciais. O motivo por trás disso foi exposto na fundamentação teórica desenvolvida para uma rede MLP genérica utilizando a notação de operações matriciais, o que não foi encontrado nas fontes pesquisadas. Foram apresentadas duas soluções com otimizações diferentes. A primeira tem como foco desempenho e consegue entregar o resultado de um elemento da matriz resultante da multiplicação matricial em apenas $270ns$. Já a segunda implementação resultou em um compromisso entre performance e ocupação da FPGA, ocupando em média 40% a 50% da FPGA da placa Basys 3. O resultado do design 2 pode ser computado em $330ns$. Esse resultado apresenta uma penalidade de apenas 20% com relação ao design 1, porém ocupando $\frac{1}{4}$ do espaço. Considera-se, então, um bom compromisso de *design*. Além disso, o *design* 2 é cerca de 4 vezes mais rápido e 50 vezes mais eficiente que um *script* em Python.

Palavras-chave: Acelerador, *Hardware*, FPGA, Aprendizado de Máquina, Redes Neurais.

Abstract

Machine Learning applications have greatly popularized in the sense that it has been used in a wide range of technological applications. This popularization created an increasing demand for computing power and energy, which are fundamental factors in both the learning and the prediction phases. This dependency limits its use when the needed resources can't be provided or when the algorithms execution times are so long that it even discourages or prevents the research development. The associated problems increase when the environmental problems are also considered, since the energy demand is extremely high. In this context, the search for alternative hardwares is essential. These units can't be used in small applications such as smartphones and smartwatches and also need a lot of power. One of the alternatives are the FPGAs (Field Programmable Gate Arrays), that can be programmed as a hardware dedicated to compute the algorithms hot spots. Those hot spots are mainly the matrix operations. The reason behind that was shown through the text by explaining the MLP's algorithm for a generic net using matrix notation, which wasn't found in the researched resources. Two design solutions were developed with different optimizations. The first focuses on performance and takes just $270ns$ to output one element of the resulting matrix. The second implementation is a trade off between performance and resources utilization. It needs 40% to 50% from Basys 3 area. The same result mentioned for the first design can be achieved in $330ns$, which is a 20% higher penalty when both are compared. However, it has the advantage of using $\frac{1}{4}$ of the first design's space. Therefore, it can be considered that a great design compromise has been achieved. Furthermore, design 2 is approximately 4 times faster and 50 times more efficient than a Python script.

Keywords: Accelerator, Hardware, FPGA, Machine Learning, Neural Networks.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TERMOS E SIGLAS

AFAU	*Activation Function Acceleration Unit*
ASIC	Circuitos Integrados de Aplicação Específica
BP	*Backpropagation*
BRAM	*Block Random Access Memory*
CLBs	*Configurable Logic Blocks*
CNN	Rede Neural Convolucional
CPU	Unidade Central de Processamento
DLAU	*Deep Learning Accelerator Unit*
DMA	*Direct Memory Access*
DNA	Ácido Desoxirribonucléico
DNN	Rede Neural Profunda
DSP	*Digital Signal Processor*
FF	Flip-Flop
FIFO	*First in, First Out*
FPGA	Arranjo de Portas Programáveis em Campo
GPU	Unidade de Processamento Gráfico
HDL	*Hardware Description Language*
I/O	Entrada/Saída
IA	Inteligência Artificial
IOBs	*Input/Output Blocks*
IP	Intellectual Property

LSTM	*Long short-term memory*
LUT	*Look Up Table*
MAC	*Multiplier-Accumulator Units*
MCP	McCullock-Pitts
MLP	Multilayer Perceptron
MLP	*Multi Layer Perceptron*
PSAU	*Part Sum Accumulation Unit*
RBM	*Restricted Boltzmann Machine*
RTL	*Register Transfer Language*
SIMD	*Single Instruction, Multiple Data*
SoC	*System-on-Chip*
TMMU	*Tiled Matrix Multiplication Unit*
TPU	*Tensor Processing Unit*

Capítulo 1

Introdução

O termo Inteligência Artificial (IA) como um campo de pesquisa surgiu na Conferência de Darthmouth no ano de 1956, cuja criação foi impulsionada pelo desenvolvimento dos computadores existentes na época. Nessa década, surgia a segunda geração de computadores, nos quais transistores substituíam os tubos a vácuo e linguagens *assembly* passaram a ser utilizadas pelos programadores ao invés da linguagem de máquina em binário [1]. Nas décadas seguintes, a IA foi considerada por alguns pesquisadores como o caminho para o futuro do desenvolvimento tecnológico e, por outros, como um campo pertencente somente à ficção científica, que jamais se concretizaria. De fato, a implementação de uma inteligência artificial como uma máquina que possui as mesmas características da inteligência de um ser humano ainda não foi realizada.

No entanto, já se consegue implementar algoritmos que têm a capacidade de realizar determinadas tarefas de forma a conseguir resultados por vezes melhores do que um ser humano conseguiria. De acordo com [2], tais algoritmos resultam em tecnologias de IA limitada. O nome advém de que apenas parte da capacidade inteligente de um ser humano é implementada. Nesse contexto, tem-se o aprendizado de máquinas ou *machine learning*, que consiste em uma alternativa para chegar em uma inteligência artificial utilizando dados coletados previamente para treinar o mecanismo de aprendizagem e, posteriormente, realizar predições acerca de novos dados em problemas de classificação ou regressão.

Técnicas convencionais de *machine learning*, como regressão logística, *k-nearest neighbors*, máquinas de vetores de suporte ou *support vector machines* e *random forests*, são limitadas quanto à habilidade de utilizar dados sem pré-processamento. Durante décadas, a implementação desse tipo de sistemas exigia o conhecimento por parte dos pesquisadores de técnicas complexas para extrair as características, *features*, dos dados originais. Essas técnicas transformavam os dados não tratados, *raw data*, em representações internas que possibilitavam a utilização do algoritmo como um sistema de aprendizado.

Nesse contexto, há uma classe de algoritmos denominada aprendizado profundo ou *deep learning*, que consiste em métodos de aprendizado representativo ou *representation-learning*

methods. Em geral, o aprendizado profundo utiliza modelos de redes neurais multi-camadas, cujos métodos mais utilizados atualmente são as redes neurais profundas ou *deep neural networks* (DNNs) e as redes neurais convolucionais ou *convolution neural networks* (CNNs). Tais algoritmos não necessitam que os dados passem por processamentos complexos anteriores ao treinamento das redes. As redes são formadas por camadas compostas por módulos não lineares, as quais transformam a representação original em representações mais abstratas. Dessa forma, o fato de as *features* não serem obtidas por seres humanos, mas diretamente pelo processo de aprendizado, consiste em um diferencial desse tipo de algoritmos, levando a grandes avanços na resolução de problemas que permaneceram estagnados durante anos.

O aprendizado profundo tem sido aplicado, nos últimos anos, nas mais diferenciadas áreas, inclusive no comércio, estando presente em diversos produtos. Dessa forma, o *deep learning* tem-se tornado foco de diversos centros de pesquisa e de grandes empresas. Há diversos exemplos de utilização do aprendizado de máquinas na sociedade moderna, incluindo a identificação de objetos em imagens, transcrição da fala para texto, recomendações de produtos para consumidores de acordo com seu perfil de consumo, seleção de resultados relevantes de pesquisa para cada usuário, e recomendações de conteúdo em plataformas de *streaming* de áudio e vídeo [3].

Também podem ser mencionados os assistentes virtuais inteligentes *Apple Siri*, *Google Now*, *Microsoft Cortana* e *Amazon Echo*, capazes de realizar tarefas ou serviços para os usuários baseando-se nas suas características de uso. Há, inclusive, aplicações no campo da medicina, como, por exemplo, a reconstrução de circuitos cerebrais, a predição de efeitos de mutações em DNA não codificado na expressão dos genes, a análise de radiografias de tórax para detectar crescimento celular anormal, e o reconhecimento de tumores malignos de pele. Ademais, foram obtidos resultados promissores em análise de sentimentos, resposta a perguntas e tradução de idiomas. Um campo relevante de pesquisa em que são utilizados diversos dos exemplos citados anteriormente é o de carros autônomos, em que se verifica a importância do reconhecimento de imagens, gestos e vozes.

Porém, apesar de seu uso haver sido popularizado recentemente, é importante mencionar que os métodos de *deep learning* foram criados há diversas décadas, estando presentes desde o início da IA. Apesar de terem sido criados há vários anos, constituiu fator limitante para sua ampla utilização a existência de problemas em seus algoritmos que foram sendo resolvidos ao longo dos anos. A não utilização de tais métodos na resolução de problemas deveu-se também à alta demanda por poder computacional por parte dos algoritmos profundos, o que não era viável até que as CPUs, ou unidades centrais de processamento, com múltiplos processadores e, posteriormente, as GPUs, ou unidades de processamento gráfico, passaram a ser amplamente utilizadas. Tal alta demanda por poder computacional deve-se a dois fatores principais: o algoritmo utiliza operações matemáticas custosas computacionalmente, principalmente multiplicações matriciais, e é necessária grande quantidade de dados e de parâmetros a serem treinados. Estes aumentam rapidamente conforme o número de camadas utilizadas na rede neural também aumenta, o que também torna maior a quantidade

de operações matemáticas custosas a serem realizadas, assim como o tempo de execução do treinamento.

Um exemplo que mostra o quão extensas podem ser as demandas por dados e poder computacional é o do projeto da Google [4] divulgado em 2012 de reconhecimento de imagens por meio de aprendizado não-supervisionado. O projeto, que consistia em uma rede contendo um bilhão de conexões entre neurônios, utilizou como base de dados para o treinamento *frames* aleatórios de 10 milhões de vídeos do YouTube, sendo que cada vídeo contribuiu com apenas uma imagem para o conjunto de dados ou *dataset*. Além disso, o modelo foi treinado de forma distribuída em um *cluster*, que consiste em diversos computadores trabalhando em conjunto para a realização de uma mesma tarefa, composto por mil máquinas com 16 mil *cores*, que são unidades de processadores de CPUs, durante três dias. A partir desta rede foi possível reconhecer imagens de seres humanos e de gatos sem informar à rede o que são cada um deles, ou seja, o algoritmo foi capaz de obter as *features* relevantes para identificação de forma independente.

Nesse contexto, tem-se que o elevado poder computacional exige alto consumo de energia para o seu funcionamento. De acordo com a fonte [5], divide-se a história do campo de *machine learning* em duas eras em termos do consumo de energia. Desde sua criação até o ano de 2012, a demanda de energia por parte dessa tecnologia dobrava a cada dois anos. Porém, deste ano em diante, a demanda de energia tem dobrado a cada 3,4 meses. Dessa forma, a quantidade de energia demandada pelo aprendizado de máquinas é considerada exorbitante e já há preocupações quanto à capacidade do planeta de suprir tais demandas, visto que a necessidade de crescimento na geração de energia geralmente leva a consequências ambientais sérias que prejudicam o desenvolvimento sustentável.

Surge, portanto, a necessidade de novas soluções para acelerar algoritmos de aprendizado profundo que apresentem alta performance e baixa demanda por energia. Uma das propostas de solução que tem sido amplamente estudada é o uso de FPGA, *field-programmable gate array* ou arranjo de portas lógicas programáveis em campo, visto que o uso de FPGAs como aceleradores de hardware pode atingir níveis moderados de performance com custo energético menor que o de GPUs. Um FPGA consiste em um circuito integrado desenvolvido para ser configurado pelo usuário após a manufatura utilizando uma linguagem de descrição de hardware, de forma que as conexões entre os blocos lógicos são reconfiguráveis, permitindo o uso de uma mesma placa em aplicações distintas.

Assim, verifica-se que a demanda por elevado poder computacional reflete a importância da aceleração de hardware para aplicações de aprendizado de máquinas, especialmente em técnicas de aprendizado profundo. Além disso, há a necessidade de encontrar um acelerador de hardware eficiente, ou seja, que consuma menos energia e apresente performance satisfatória. Isso devido ao apelo pelo desenvolvimento sustentável e também à necessidade de baixo consumo por sistemas embarcados como *smartphones* e robôs, que são, em sua maioria, movidos a baterias. Dessa forma, o objetivo deste trabalho consiste no desenvolvimento de um acelerador de *hardware* para aprendizado de máquinas utilizando FPGAs.

Propõe-se, portanto, a implementação de arquitetura de *hardware* baseada na referência [6] composta por três módulos, responsáveis pelos cálculos mais custosos computacionalmente dos algoritmos. O primeiro é responsável por ler os dados particionados e os pesos, realizar os cálculos de multiplicação e somas parciais. O segundo realiza as operações de acumulação de somas parciais e, o terceiro, implementa a função de ativação. Um diagrama ilustrativo do processo simplificado pode ser visualizado na figura 1.1. Assim, consistem objetivos específicos deste trabalho o estudo do porquê as operações matriciais são predominantes nos algoritmos de redes neurais artificias; o desenvolvimento de uma arquitetura baseada na anteriormente descrita, que possa realizar com eficiência operações matriciais de multiplicação; a realização de testes do acelerador com o intuito de avaliar seu desempenho e a posterior comparação do resultado encontrado com aqueles obtidos por meio de CPU ou GPU em termos de velocidade e consumo de energia.

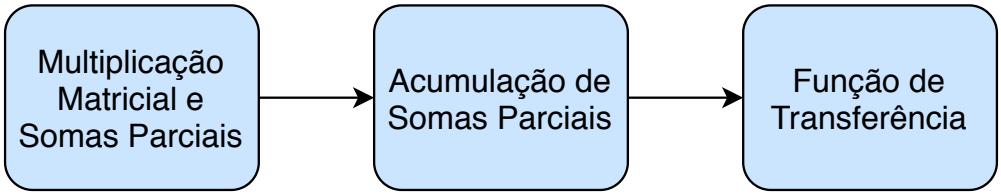

Figura 1.1: Módulos da arquitetura abstrata do acelerador

O trabalho está organizado como segue. O capítulo 2 apresenta uma revisão bibliográfica de artigos que tratam do tema deste trabalho, de forma a apresentar o estado da arte do tema proposto. O capítulo 3 apresenta os conceitos e as definições fundamentais para compreensão do tema e para a implementação do trabalho, assim como analisa de forma detalhada o algoritmo do MLP, com uma explicação do algoritmo para uma rede genérica e utilizando notação de operações matriciais, o que não foi encontrado nas fontes pesquisadas. O capítulo 4 apresenta como foram desenvolvidos e quais são os *designs* de *hardware* propostos. O capítulo 5 apresenta os procedimentos seguidos para obtenção dos resultados, os gráficos das simulações e os recursos demandados para sua implementação física. Há também comparações com a arquitetura tomada como referência. Por fim, o capítulo 6 apresenta as conclusões e as propostas de trabalhos futuros.

Capítulo 2

Estado da Arte

Neste capítulo é feita uma revisão bibliográfica de alguns dos artigos estudados para definição do tema e decisão da arquitetura a ser implementada. Dessa forma, o estado da arte, ou seja, o estado das pesquisas no que tange o tema de implementação de um acelerador de *hardware* em FPGA para aplicações em IA, especificamente em *Deep Learning*, será apresentado.

Até o momento, os métodos desenvolvidos para acelerar algoritmos de aprendizado profundo consistem em *field-programmable gate array* (FPGA) ou arranjo de portas programáveis em campo, em *application specific integrated circuits* (ASIC) ou circuitos integrados de aplicação específica, em *system-on-chip* (SoC) ou sistema em um *chip*, e em *graphic processing unit* (GPU) ou unidade de processamento gráfico. Os três primeiros métodos estão sendo estudados como alternativa para o uso de GPUs, devido ao seu alto consumo de energia, como contextualizado no capítulo 1.

É importante compreender as diferenças entre cada um dos tipos de *hardware* utilizados como aceleradores [7]. Os ASICs são circuitos integrados geralmente projetados para utilização em um sistema específico e oferecem alto desempenho e baixo consumo de energia, mas são consideravelmente caros no sentido de demandarem bastante tempo e recursos para serem desenvolvidos. Já o SoC é basicamente um ASIC que contém um ou mais processadores, além de algumas funções adicionais. Por sua vez, FPGAs se diferenciam dos anteriores devido à capacidade de reconfiguração do seu *hardware* e de implementação de algoritmos de forma paralela.

Os aceleradores de *hardware* FPGA e ASIC apresentam performance moderada com baixo consumo energético quando comparados com GPU. No entanto, ambos possuem recursos computacionais limitados, como memória e largura de banda de *input/output* (I/O) ou entrada/saída. Por esse motivo, é desafiador implementar complexas redes neurais utilizando tais aceleradores de *hardware*. Os ASICs, por serem circuitos projetados para aplicações específicas, possuem um ciclo de desenvolvimento de projeto longo e o resultado final apresenta flexibilidade insatisfatória, visto que não consegue se adaptar a demandas de aplicações

distintas. Em contrapartida, por ser um hardware reconfigurável, já há aceleradores FPGA focados em implementar um algoritmo de forma eficiente com a possibilidade de ajustar o tamanho e a topologia da rede a depender da necessidade da aplicação em arquiteturas de *hardware* escalonáveis e flexíveis.

Verificou-se que há diversos artigos que apresentam implementações em ASICs. Como exemplo, Chen *et al.* [8] desenvolveram um acelerador denominado DianNao, em que, com uma área menor que $1mm^2$, atingiu-se aceleração na velocidade de aproximadamente 118x e redução de energia de aproximadamente 21x com relação a um SIMD, *single instruction, multiple data*, *core* de 128 bits e 2GHz em implementações de CNNs e DNNs. Além disso, Jouppi *et al.* [9] avaliam um acelerador ASIC denominado *tensor processing unit* (TPU) para aplicações em redes neurais do tipo MLPs ou *multi layer perceptrons*, CNNs e LSTMs ou *long short-term memory*. A sua principal característica é a utilização de um *Multiplier-Accumulator Unit* (MAC) de 8 bits que pode promover em média 15 a 30 vezes mais velocidade que a CPU e GPU utilizadas para comparação, a Intel Haswell e a Nvidia K80, respectivamente.

Há também extensa literatura no que tange aceleradores utilizando FPGAs. Por exemplo, Ding *et al.* [10] propuseram um acelerador para CNNs no FPGA Aria 10, em que todas as camadas podem funcionar de forma concorrente em forma de *pipeline* para melhorar o *throughput* do sistema. O processamento de uma imagem 32x32x3 pode atingir aproximadamente 18 vezes a velocidade da CPU Intel Xeon e consumo de energia aproximadamente 30 vezes menor que o da GPU NVIDIA Titan. Além disso, Ma *et al.* [11] propuseram um acelerador FPGA de algoritmos de aprendizado profundo com um compilador RTL ou *register transfer level* modularizado.

As ferramentas denominadas *high-level synthesis* como HLS e OpenCL reduzem o tempo de desenvolvimento da programação das FPGAs consideravelmente, mas são ainda ineficientes quanto à alocação de recursos para maximização de paralelismo e *throughput*. Em contrapartida, o *design* manual de *hardware*, ou seja, RTL, melhora a eficiência e a velocidade, mas exige conhecimento profundo tanto da estrutura do algoritmo de aprendizado profundo quanto da arquitetura do sistema a ser implementado na FPGA, o que aumenta consideravelmente o tempo de projeto. Para intermediar esse *trade-off*, foi proposto o compilador desenvolvido, denominado ALAMO, que analisa a estrutura e os parâmetros do algoritmo e integra automaticamente os módulos primitivos. É um compilador desenvolvido para implementações de CNN considerando os recursos limitados de FPGAs e foi possível obter uma melhora de quase 2 vezes o *throughput* quando comparado ao *design* obtido por meio do OpenCL.

A referência [6] apresenta a proposta de arquitetura em que se baseou a proposta deste trabalho. Portanto, essa será descrita com mais detalhes. É apresentado um acelerador escalonável denominado *deep learning accelerator unit* ou DLAU . Na sua implementação, são utilizadas *tile techniques* ou técnicas de particionamento, FIFO (*first in, first out* ou algoritmo de fila simples) buffers e *pipelines* para minimizar as operações de transferência de memó-

Tabela 2.1: Resultados do *profiling* de *hot spots* em DNN

Algoritmo	Multiplicação Matricial	Função de Ativação	Operações Vetoriais
Feedforward	98,60%	1,40%	-
RBM	98,20%	1,48%	0,30%
BP	99,10%	0,42%	0,48%

ria, e unidades computacionais são reutilizadas para implementar redes neurais profundas. A arquitetura DLAU pode ser configurada para operar com diferentes tamanhos de *tile data*, o que consiste em um *trade-off* entre velocidade e custo de *hardware*. Isso porque quanto maior a partição de dados, maiores são a velocidade e o custo e, quanto menor a partição, menores são a velocidade e o custo (mais lentos e menos custosos serão).

Foram feitos experimentos de *profiling* com o objetivo de verificar os *hot spots* dos algoritmos em que se basou o estudo [6], ou seja, com o objetivo de verificar as regiões dos programas em que ocorre a maior proporção de instruções executadas. O *profiling* consiste em uma forma de análise dinâmica de programas que mede complexidades de memória ou de tempo, o uso de determinadas instruções ou a frequência e duração de chamadas de funções, e geralmente as informações adquiridas são utilizadas para otimização do programa. Os algoritmos em que se efetuou o *profiling* foram o *feed forward*, utilizado na predição; o RBM ou *restricted Boltzmann machine*, que está sendo bastante utilizado para treinar cada camada da rede neural profunda de forma eficiente; e o BP ou *backpropagation*. Os *hot spots* encontrados utilizando o *profiling* foram multiplicação matricial, função de ativação e operação vetorial, cujos percentuais de tempo de execução com relação ao tempo de execução total constam na tabela 2.1, retirada da referência [6]. Da tabela, percebe-se nitidamente uma predominância de multiplicação matricial em todos esses algoritmos.

Com base nos *hot spots* encontrados, implementou-se o DLAU com unidades de processamento para acelerar os dois tipos de operações computacionais mais custosos, as multiplicações matriciais e a função de ativação. Dessa forma, é composto por três unidades de processamento totalmente canalizadas, *fully pipelined*. Tais unidades consistem na *tiled matrix multiplication unit* (TMMU) , na *part sum accumulation unit* (PSAU) e na *activation function acceleration unit* (AFAU) . Além disso, a arquitetura do sistema contém um processador embarcado; um controlador de memória DDR3; um módulo DMA, *direct memory access* ou acesso direto à memória; e o acelerador DLAU, constituído pelas três unidades mencionadas. O processador embarcado fornece a interface de programação aos usuários e a comunicação com o DLAU através de JTAG-UART. Além disso, promove a transferência dos dados de entrada e da matriz de pesos para blocos BRAM, ou *block random access memory*, internos, a ativação do acelerador e o retorno dos resultados ao usuário após a execução.

Na figura 2.1, extraída da referência [6] em análise, é mostrada a arquitetura do acelerador descrito. Durante a execução, o acelerador DLAU lê os dados da memória através de DMA, realiza as operações em cada uma das unidades de forma sequencial e escreve os

resultados encontrados de volta à memória. É importante mencionar que as três unidades do acelerador possuem *buffers* de entrada e de saída para receber e enviar dados por meio de FIFO com objetivo de prevenir perdas de dados que possam ocorrer devido a *throughputs*, ou taxas de transferências efetivas, inconsistentes entre cada unidade. Além disso, a transferência de dados entre unidades adjacentes do acelerador ocorre por meio do mecanismo AXI-Stream.

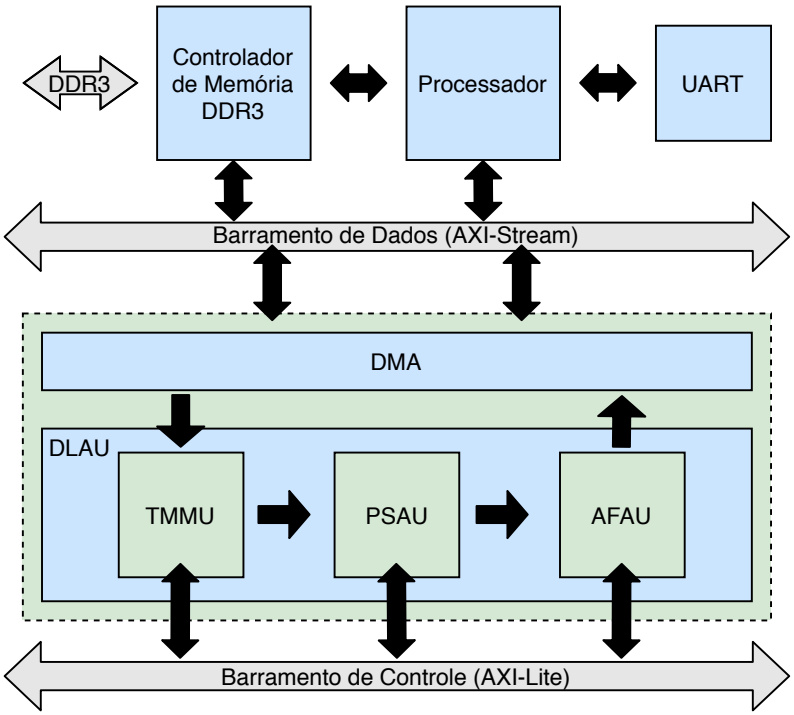

Figura 2.1: Arquitetura do acelerador DLAU

O TMMU é o módulo que lê os dados particionados e os pesos através de DMA, realiza os cálculos de multiplicação e somas parciais, e as transfere ao PSAU. Neste, ocorrem as operações de acumulação das somas parciais. Quando a soma parcial for o resultado final, a unidade escreve tal valor no *buffer* de saída e envia os resultados ao AFAU. Neste, é implementada a função de ativação utilizando interpolação linear por partes. É importante mencionar que o PSAU pode acumular uma soma parcial a cada ciclo de *clock*, sendo seu *throughput* equivalente ao de geração das somas parciais em TMMU, e que o AFAU realiza a operação de uma função de ativação do tipo sigmoide a cada ciclo de *clock*.

O protótipo de *hardware* foi implementado na *Xilinx Zynq Zedboard*, equipada com o processador ARM Cortex-A9 com *clock* de 667 MHz. Foi utilizado o banco de dados Mnist para treinar uma DNN no *Matlab*, sendo a quantidade de camadas variável. Para comparação, tomou-se como referência o processador Intel Core2 com clock de $2,3$ GHz. O clock do DLAU é de 200 MHz e este computa 32 neurônios com 32 pesos a cada ciclo, visto que o tamanho dos dados particionados, *tile data* escolhido foi 32. Foram realizadas análises comparativas de velocidade e de utilização de recursos e de energia. Verificou-se que o acelerador foi capaz de atingir até aproximadamente 36x a velocidade do processador. Por outro lado, quando comparado o consumo de energia do sistema proposto com o de GPUs, tem-se

que a unidade proposta consome 364 vezes menos do que GPUs. Além disso, comparou-se o uso de recursos BRAM, DSPs ou *digital signal processors*, FFs ou *flip-flops*, e LUTs ou *look up tables*, pelo DLAU e por aceleradores baseados em FPGA de outros dois trabalhos, sendo o DLAU mais eficiente.

Capítulo 3

Fundamentação Teórica

Neste capítulo serão abordados os conceitos fundamentais relacionados à implementação de *hardwares* dedicados a algoritmos de aprendizado de máquinas. Em especial, serão discutidos os temas necessários para a compreensão deste trabalho.

3.1 Neurônios Artificiais

O história do desenvolvimento das redes neurais artificiais teve seu início com o desejo de desenvolver máquinas que pudessem imitar a inteligência humana na forma de resolver problemas. Além disso, a peculiaridade dos humanos de conseguirem aprender com as experiências anteriores, tanto em termos de acertos quanto de erros, de forma a aplicar tal aprendizado na resolução de novos problemas foi também propulsora das pesquisas científicas neste campo de estudo.

O desenvolvimento do aprendizado de máquinas por meio das redes neurais artificiais teve seu início no estudo do cérebro humano e, portanto, dos neurônios. Estes consistem em um dos tipos de célula que forma o cérebro e também nas células responsáveis pela transmissão de informações na forma de impulsos nervosos através do corpo.

O neurônio é uma célula eletricamente excitável que se comunica com outras células através de conexões denominadas sinapses. Estas fazem a conexão entre células vizinhas e permitem dar continuidade à propagação do impulso nervoso através da rede neuronal. Nesse contexto, verificou-se que para que um neurônio transmitisse informações, o estímulo elétrico deveria ser maior que um determinado limiar para que tal estímulo fosse propagado para as próximas células. A figura 3.1 mostra o esquemático simplificado de um neurônio com os sinais entrando pelos dendritos, em seguida, sendo integrados ao corpo celular, e, se o limiar for atingido, o sinal de saída é gerado e é passado adiante pelo axônio.

No início dos estudos da neurociência, Warren McCulloch e Walter Pitts publicaram em 1943 o primeiro modelo matemático de um neurônio. Este foi denominado *McCullock-Pitts (MCP) neuron* [12], ou neurônio de M-P, e foi descrito como uma porta lógica simples com

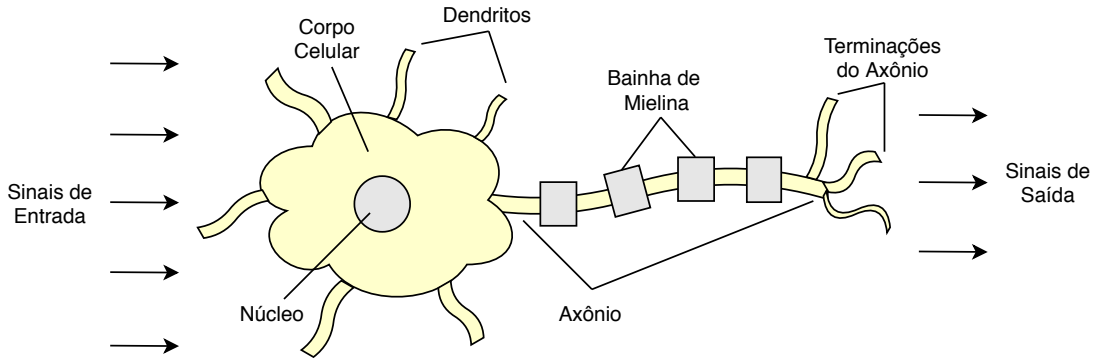

Figura 3.1: Esquema simplificado de um neurônio

múltiplas entradas e uma saída, com cada uma delas binária.

Em 1949, Donald Hebb, em seu livro [13], revolucionou o modo como os neurônios eram percebidos. Isso porque ele propôs a regra de Hebb, em que afirma de forma resumida, que, quando dois neurônios disparam simultaneamente, a conexão entre eles é fortalecida e que esta atividade é uma das principais para o aprendizado e para a memória.

Foi então necessário alterar o MCP, que tinha apenas pesos fixos, para adequar o modelo à nova proposta [14]. Para tanto, adicionaram-se pesos variáveis a cada uma das entradas, com os quais cada entrada tem contribuições com importâncias distintas. Nesse âmbito, Frank Rosenblatt utilizou o modelo do neurônio MCP e as descobertas do Hebb para desenvolver o perceptron, apresentado no seu livro *Principles of Neurodynamics*, em 1962. Neste, foi proposta a regra de aprendizado do perceptron e o seu algoritmo. O perceptron é, portanto, o modelo lógico e matemático do funcionamento de um neurônio.

3.2 Funcionamento do Perceptron

O algoritmo proposto por Rosenblatt é capaz de aprender os pesos ótimos a serem multiplicados pelas entradas, também denominadas *features*, de forma a determinar se o neurônio dispararia ou não. No contexto de aprendizado supervisionado e de classificação, ele poderia ser utilizado para predizer se determinada amostra pertence a uma classe ou outra, visto que o perceptron resolve problemas binários.

O modelo do perceptron pode ser representado de acordo com a figura 3.2 e o seu funcionamento será explicado a seguir [15].

Para compreensão do algoritmo, é necessário apresentar algumas definições. As entradas serão o vetor denominado $\boldsymbol{x}$ e, os pesos, o vetor $\boldsymbol{w}$, representados na equação 3.1, sendo

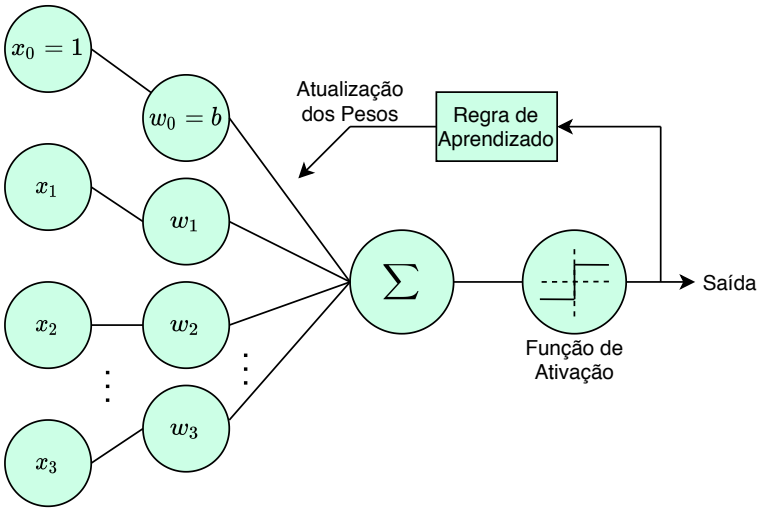

Figura 3.2: Esquemático do funcionamento do perceptron

$m \in \mathbb{N}$ o número de entradas das amostras.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \tag{3.1}$$

Como mencionado, o perceptron resolve problemas de classificação binários. As saídas serão as classes 1 e -1, denominadas positiva e negativa, respectivamente. Há também a função de ativação $\phi(z)$. Esta é a função degrau unitário, representada pela equação 3.2. A função de ativação tem como entrada o produto interno de $\boldsymbol{x}$ e $\boldsymbol{w}$, dado por z. Dessa forma, tem-se que $z = w_1x_1 + ... + w_mx_m$.

$$\phi(z) = \begin{cases} 1, & \text{se } z \geq \theta \\ -1, & \text{caso contrário} \end{cases} \tag{3.2}$$

Além disso, o número de amostras do conjunto de treinamento será denotado por $i \in \mathbb{N}$, de forma que para cada amostra $x^{(i)}$, se a saída da função de ativação $\phi(\cdot)$ for maior que o limiar θ, será predita a classe 1 e, caso seja menor, a classe -1.

Para simplificar a representação da função de ativação, considera-se o limiar θ como o peso zero, de forma que $w_0 = -\theta$ e $x_0 = 1$. O peso da nova entrada fixa em 1 é denominado *bias* e geralmente é representado por b. Tal representação altera o limiar para 0, o que pode ser observado na equação 3.3.

$$z = w_0x_0 + w_1x_1 + ... + w_mx_m = \boldsymbol{w}^T\boldsymbol{x} \quad \text{e} \quad \phi(z) = \begin{cases} 1, & \text{se } z \geq 0 \\ -1, & \text{caso contrário} \end{cases} \tag{3.3}$$

A figura 3.3 contém o gráfico da função de ativação e também um gráfico que exemplifica, para $m = 2$, a separação em duas classes no caso de dados linearmente separáveis. Isto será abordado com mais detalhes no decorrer do texto.

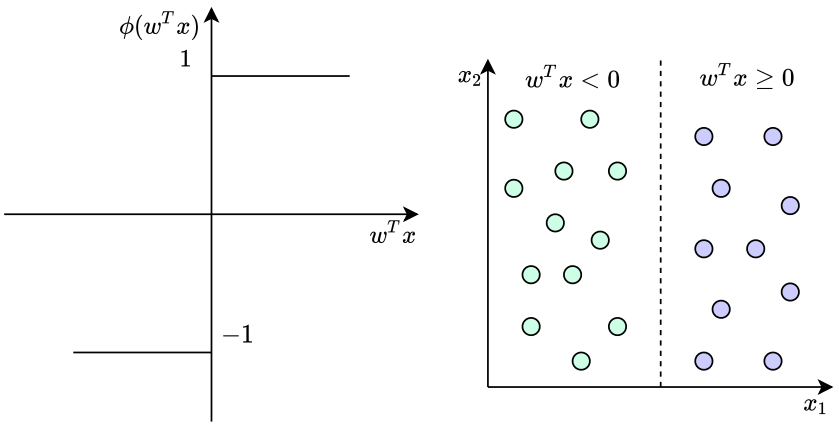

Figura 3.3: Gráfico da função de ativação degrau unitário à esquerda e exemplificação de como ela pode ser usada para classificação binária à direita

A partir das definições anteriores, pode-se apresentar o algoritmo de Rosenblatt para o perceptron. Este, como mencionado, tenta imitar de forma simplificada a célula neuronal, que dispara o impulso ou não. Deve ser seguido, portanto, o procedimento a seguir.

1. Inicialize os pesos em 0 ou em pequenos números aleatórios.
2. Para cada amostra do conjunto de dados de treinamento $x^{(i)}$:
   (a) Calcule o valor de saída da função de ativação $\phi(z) = \hat{y}$.
   (b) Atualize os pesos de acordo com $w_j \coloneqq w_j + \Delta w_j$, sendo $j \in \mathbb{N}$ e $0 \leq j \leq m$.

A regra de aprendizado do perceptron é justamente Δw_j, que consta na equação 3.4, sendo $\eta \in \mathbb{R}^+$ a taxa de aprendizado, com $0.0 < \eta < 1.0$; $y^{(i)}$ a classe verdadeira e, $\hat{y}^{(i)}$, a classe predita para a amostra i do conjunto de treinamento.

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)} \tag{3.4}$$

Para ilustrar o funcionamento da regra de aprendizado do perceptron, serão analisados os cenários em que os pesos permanecem inalterados porque a classe predita foi igual a classe verdadeira e em que os pesos são alterados devido a predições incorretas.

Na equação 3.5, tem-se o caso em que tanto a classe predita quando a verdadeira são iguais a -1, de forma que não é necessário alterar o peso, visto que já houve acerto da predição.

$$\Delta w_j = \eta(-1^{(i)} - (-1)^{(i)})x_j^{(i)} = 0 \tag{3.5}$$

Na equação 3.6, também tem-se uma predição correta, mas com a classe verdadeira igual a 1. Percebe-se que nesse caso os pesos também não serão atualizados, como esperado.

$$\Delta w_j = \eta(1^{(i)} - (1)^{(i)})x_j^{(i)} = 0 \tag{3.6}$$

Já na equação 3.7, verifica-se que a predição foi incorreta, com a classe verdadeira sendo a positiva e, a predita, a negativa. Dessa forma, os pesos serão atualizados de forma a serem direcionados à classe verdadeira, positiva.

$$\Delta w_j = \eta(1^{(i)} - (-1)^{(i)})x_j^{(i)} = \eta(2)x_j^{(i)} \tag{3.7}$$

Já no caso da equação 3.8, considerou-se a situação da predição incorreta com a classe verdadeira sendo a negativa, e, a predita, a positiva. Verifica-se que a atualização dos pesos ocorre de forma a direcionar a predição para a classe negativa.

$$\Delta w_j = \eta(-1^{(i)} - (1)^{(i)})x_j^{(i)} = \eta(-2)x_j^{(i)} \tag{3.8}$$

É importante mencionar que tais atualizações ocorrem de forma proporcional ao valor da entrada $x_j^{(i)}$. Ou seja, como a predição depende da soma dos produtos $x_j^{(i)}w_j^{(i)}$, quanto maior o módulo do valor da entrada, mais significativa é a sua contribuição para direcionar a saída para a predição correta na próxima iteração.

Após as definições apresentadas, pode-se compreender melhor a figura 3.2. Nela, verifica-se que há a multiplicação das entradas de mesmo índice do vetor de entradas $\boldsymbol{x}$ pelo vetor de pesos $\boldsymbol{w}$. Em seguida, ocorre a soma de todos os produtos e a passagem pela função de ativação. Durante o treinamento, a saída do perceptron é utilizada para atualizar os pesos e o processo se repete de forma iterativa.

O algoritmo proposto por Rosenblatt, apesar de eficiente para a resolução de problemas simples, é bastante limitado. Em 1969, Marvin Minsky e Symour Papert mostraram que a convergência do algoritmo do perceptron ocorre somente se as classes forem linearmente separáveis. Caso contrário, o perceptron nunca pararia de atualizar os pesos sem definir um número de épocas ou um limiar para o número tolerado de classificações incorretas. Isso porque classes linearmente inseparáveis não podem ser separadas por uma fronteira linear, como exemplificado na figura 3.4 para o caso de dados de duas dimensões, ou seja, com $m = 2$.

Em particular, o perceptron não funciona para as funções lógicas XOR e XNOR. Além disso, como a maior parte dos problemas reais é linearmente inseparável, tal limitação do perceptron desestimulou as pesquisas em torno do aprendizado de máquinas, que foram retomadas por volta dos anos 1980 com as redes neurais artificiais, ou perceptron multi-camadas.

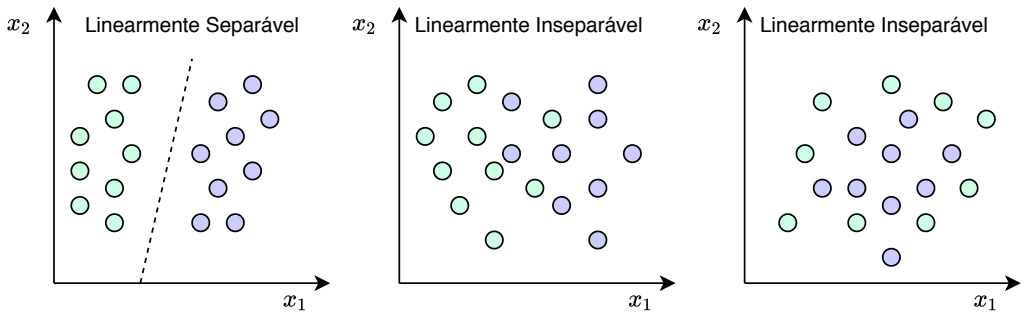

Figura 3.4: Exemplos de dados linearmente separáveis e inseparáveis

3.3 Redes Neurais Artificiais

As redes neurais artificiais utilizam o perceptron (com a função de ativação modificada) como unidade básica e solucionaram o problema da impossibilidade de classificar dados linearmente inseparáveis. Estas conectam as entradas de neurônios artificiais com as saídas de outros neurônios artificiais, criando camadas de neurônios e aumentando a complexidade do algoritmo.

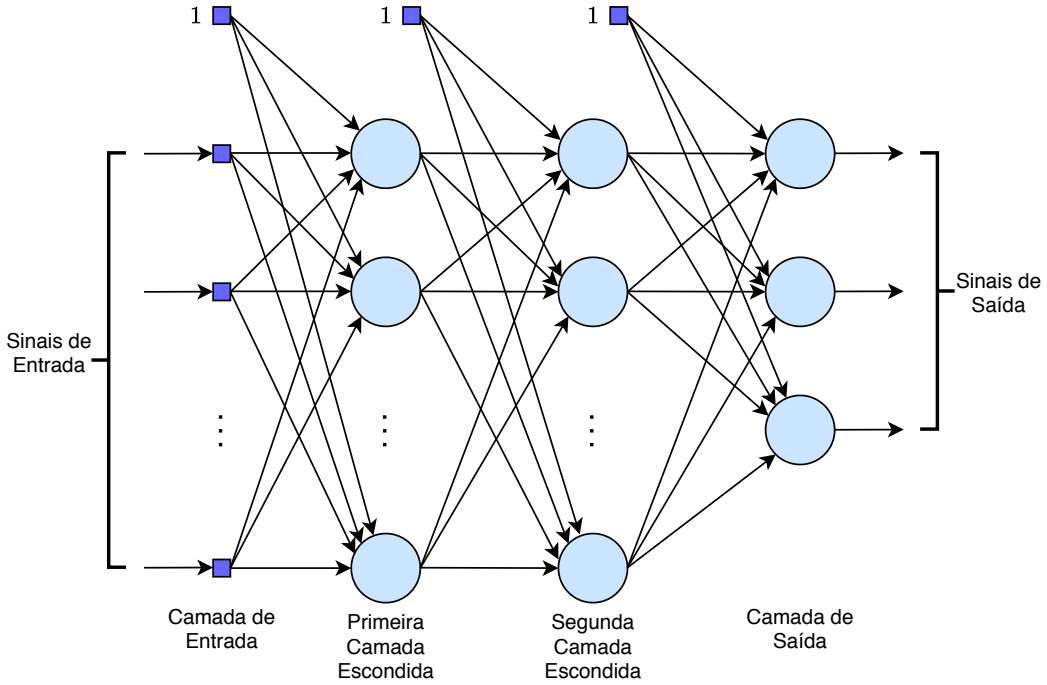

Figura 3.5: Arquitetura de um MLP com duas camadas escondidas

O MLP possui três características principais [16]. O modelo de cada um dos neurônios da rede contém uma função de ativação não linear diferenciável. Essa função de ativação não linear deve estar presente pelo menos na primeira camada. A rede deve conter uma ou mais camadas internas, que são chamadas de ocultas ou escondidas, visto que não são conectadas diretamente nem às entradas nem às saídas. Além disso, há na rede um alto grau de conectividade, cuja intensidade é determinada pelos pesos.

Na figura 3.5, consta a arquitetura de uma rede com uma camada de entrada, duas camadas escondidas e uma de saída. A rede neural mostrada é totalmente conectada, ou seja, cada um dos neurônios está conectado a todos os outros da camada anterior. É necessário notar que a entrada fixa em 1 e o *bias* estão também representados na arquitetura e que eles aparecem em cada uma das camadas de entrada e escondidas.

O aprendizado dessas redes configura-se como um problema de otimização cujo objetivo é encontrar os pesos tais que haja o menor erro nas saídas. Dessa forma, a regra de aprendizado, ou seja, a forma como os pesos são atualizados a cada iteração, é distinta da regra do perceptron. As não linearidades distribuídas ao longo da rede, a alta conectividade entre os neurônios e as camadas escondidas tornam o processo de aprendizado mais difícil de ser visualizado. Nesse processo, devem ser decididas as *features* do conjunto de dados, do *dataset*. Tal decisão é tomada pelos neurônios das camadas escondidas, que agem como detectores de *features*, sendo isso o que distingue o MLP do perceptron de Rosenblatt.

Desse modo, tais redes são também denominadas Perceptron Multicamadas ou Multilayer Perceptron (MLP). Nesse contexto, há também o Aprendizado Profundo ou *Deep Learning*, que pode ser compreendido como um conjunto de algoritmos desenvolvidos para treinar redes neurais artificiais de forma eficiente.

Descrição do algoritmo de aprendizado

O algoritmo mais utilizado para o treinamento do MLP é denominado *Backpropagation*. Ele foi desenvolvido na década de 1980 e colocou um fim ao pessimismo acerca do aprendizado com múltiplos perceptrons que fora evocado por Minsky e Papert em 1969. O treinamento ocorre em duas fases, em que os sinais propagam-se em sentidos contrários em cada uma delas, como mostrado na figura 3.6.

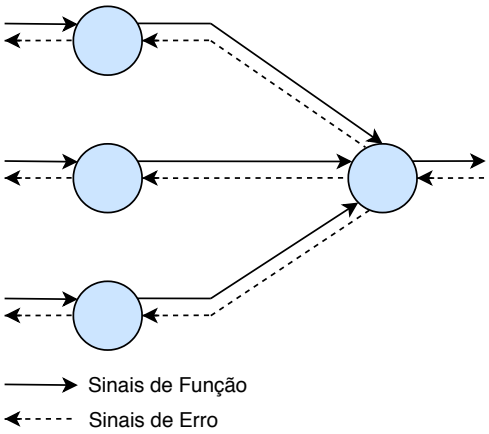

Figura 3.6: *Forward propagation* dos sinais de função e *back propagation* dos sinais de erro

Na primeira fase, denominada *forward phase*, os pesos da rede são fixados e os sinais de entrada, também chamados sinais de função, propagam-se camada por camada até atingirem os sinais de saída. Nesta fase são obtidos os sinais de saída preditos, que serão comparados

com o valor verdadeiro na fase seguinte. A segunda fase consiste na *backward phase*, em que se calculam os erros dos sinais de saída de cada um dos neurônios para que os pesos possam ser atualizados na próxima iteração. Os sinais de erro resultantes são propagados pela rede no sentido das saídas para as entradas. Deve-se a esse motivo o nome do algoritmo ser "propagação para trás", em tradução livre.

Tanto na fase *forward* quanto na *backward* há operações de multiplicação matricial. Como mencionado na seção 2, elas constituem a maior proporção de instruções executadas do algoritmo. Para melhor compreender tal algoritmo e a forte presença de multiplicações matriciais, considere um MLP com uma camada de entrada, uma ou mais camadas escondidas e uma camada de saída com um ou mais neurônios [16].

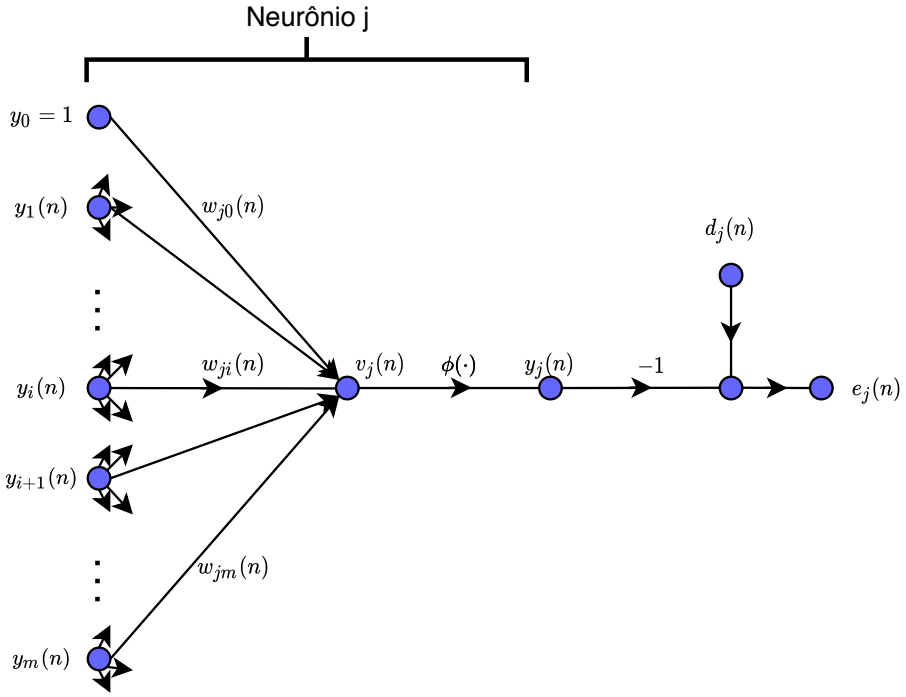

Figura 3.7: Representação de um neurônio j da camada de saída do MLP

A figura 3.7 ilustra um neurônio j da camada de saída e será tomada como referência para a explicação dos sinais. Na equação 3.9, denotam-se as amostras do conjunto dos dados de treinamento supervisionado por $\mathcal{F}$, em que $n, N \in \mathbb{N}$, N é uma constante que denota o número total de amostras de treinamento, n representa uma destas amostras, $\boldsymbol{x}$ é o vetor de entradas e $\boldsymbol{d}$ é o vetor dos valores verdadeiros das saídas.

$$\mathcal{F} = \{\boldsymbol{x}(n), \boldsymbol{d}(n)\}_{n=1}^{N} \tag{3.9}$$

O sinal de erro de um neurônio j da camada de saída é definido por 3.10, sendo $d_j(n)$ o elemento j do vetor de saídas verdadeiras $\boldsymbol{d}(n)$.

$$e_j(n) = d_j(n) - y_j(n) \tag{3.10}$$

O erro $E_j(n)$ de um neurônio j é definido em 3.11.

$$E_j(n) = \frac{1}{2}e_j^2(n) \tag{3.11}$$

Somando-se as contribuições de todos os neurônios da camada de saída, tem-se o erro total da rede, como mostrado em 3.12, em que C é o conjunto de todos os neurônios da camada de saída.

$$\begin{aligned} E(n) =& \sum_{j \in C} E_j(n) \\ =& \frac{1}{2} \sum_{j \in C} e_j^2(n) \end{aligned} \tag{3.12}$$

Há dois estilos de treinamento supervisionado, o *On-line* ou Estocático e o *Batch* ou Batelada. No primeiro, os pesos são atualizados após os cálculos das saídas de todas as N amostras e o erro utilizado no aprendizado é o erro médio entre os obtidos para cada amostra. No segundo, a atualização dos pesos ocorre após a obtenção das saídas de cada uma das amostras. Este é o mais popular para o treinamento do MLP por ser fácil de implementar e proporcionar soluções efetivas para problemas de classificação de padrões difíceis. O erro utilizado no aprendizado *On-line*, é, portanto, o da equação 3.12.

Como já mencionado, o objetivo do algoritmo é a obtenção dos pesos que otimizem o erro da rede para o mais próximo possível de seu valor mínimo. Para tanto, durante o treinamento, a técnica utilizada é denominada Descida de Gradiente ou *Steepest Descent*, em que os pesos são atualizados na direção oposta à do gradiente, visto que este aponta para o máximo da função e deseja-se obter o mínimo. Dessa forma, a regra de aprendizado do MLP é calculada pela equação 3.13, em que $\eta \in \mathbb{R}$ é a taxa de aprendizado, w_{ji} é o peso que multiplica o neurônio i da camada anterior à do neurônio j, com $i, j \in \mathbb{N}$, $i, j \geq 0$ e menores ou iguais aos números totais de neurônios das respectivas camadas.

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} \tag{3.13}$$

Ademais, em 3.14 tem-se o cálculo dos pesos atualizados, que serão usados para a próxima amostra do conjunto de dados de treinamento.

$$w_{ji}(n+1) \coloneqq w_{ji}(n) + \Delta w_{ji}(n) \tag{3.14}$$

O neurônio j tem as suas entradas $y_i(n)$ multiplicadas pelos pesos correspondentes $w_{ji}(n)$. Os produtos são posteriormente somados, resultando em $v_j(n)$, cuja equação consta

em 3.15, sendo m o número de entradas, com exceção do bias, aplicadas a esse neurônio.

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{3.15}$$

Na figura 3.8 também está representado o sinal $v_j(n)$ como argumento da função de ativação $\phi(\cdot)$, cujo resultado é $y_j(n)$. Este é o valor da saída do neurônio j, representado na equação 3.16.

$$y_j(n) = \phi_j(v_j(n)) \tag{3.16}$$

A atualização dos pesos no algoritmo *backpropagation* ocorre de acordo com a equação 3.13. Portanto, para o cálculo de Δw_{ji} é necessário utilizar a regra da cadeia para obter a derivada parcial da função de erro $E(n)$ com relação ao peso w_{ji}, o que pode ser visualizado na equação 3.17. Tal derivada parcial é denominada *sensitivity factor* ou fator de sensibilidade e é o que determina a direção da busca pelo peso w_{ji}.

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{3.17}$$

É então necessário calcular cada uma das derivadas parciais. A partir da derivada parcial da equação 3.12 com relação a $e_j(n)$, obtém-se a equação 3.18.

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{3.18}$$

Por sua vez, a equação 3.19 foi obtida derivando-se ambos os lados da equação 3.10.

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{3.19}$$

Também é necessário derivar ambos os lados da equação 3.16 para obter o termo da equação 3.20.

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi_j'(v_j(n)) \tag{3.20}$$

O último termo é obtido derivando-se ambos os lados da equação 3.15, o qual consta na equação 3.21.

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{3.21}$$

Finalmente, substituindo-se as equações 3.18 a 3.21 em 3.17, obtém-se a equação 3.22.

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\phi'_j(v_j(n))y_i(n) \tag{3.22}$$

É também importante definir o *local gradient*, ou gradiente local, $\delta_j(n)$, que consta na equação 3.23.

$$\begin{aligned}\delta_j(n) =& -\frac{\partial E(n)}{\partial v_j(n)} \\ =& -\frac{\partial E(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\ =& e_j(n)\phi'_j(v_j(n))\end{aligned} \tag{3.23}$$

Substituindo-se 3.22 em 3.13, a expressão para o cálculo da atualização dos pesos para cada amostra do conjunto de dados de treinamento é dada em 3.24.

$$\begin{aligned}\Delta w_{ji}(n) =& \eta e_j(n)\phi'_j(v_j(n))y_i(n) \\ =& \eta\delta_j(n)y_i(n)\end{aligned} \tag{3.24}$$

Nota-se que a expressão da variação dos pesos $\Delta w_{ji}(n)$ depende do sinal de erro. Este pode ser calculado diretamente apenas para neurônios da camada de saída, pois apenas nela tem-se o valor verdadeiro para ser comparado com o valor predito pela rede. Assim, se j é um neurônio de saída, o gradiente local pode ser obtido de forma direta.

No entanto, como explicado anteriormente, todos os pesos contribuem para o erro final e, no caso dos neurônios das camadas escondidas, é necessário determiná-lo de forma recursiva e da última camada para a primeira (*backwards*). Isso porque ele depende dos sinais de erro de todos os neurônios a que está conectado. Para compreender o caso de atualização dos pesos em camadas escondidas, tomou-se como referência a disposição dos neurônios como na figura 3.8, em que o neurônio j pertencente à camada escondida anterior à camada de saída em que se encontra o neurônio k.

Pode-se então reescrever, na equação 3.33 o gradiente local para os neurônios escondidos de forma a não depender diretamente do sinal de erro utilizando a regra da cadeia.

$$\begin{aligned}\delta_j(n) =& -\frac{\partial E(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\ =& -\frac{\partial E(n)}{\partial y_j(n)}\phi'_j(v_j(n))\end{aligned} \tag{3.25}$$

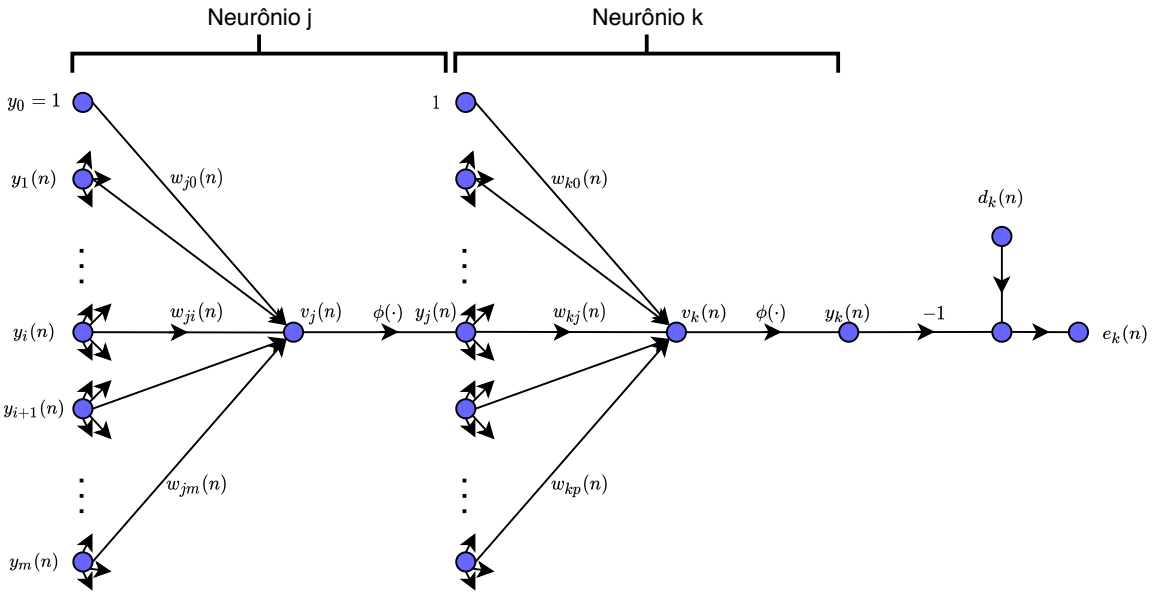

Figura 3.8: Gráfico dos sinais que conectam os neurônios de saída k e escondido j

O erro $E(n)$ para um neurônio da camada de saída pode ser calculado de acordo com a equação 3.12. Como neste caso o neurônio de tal camada denomina-se k, tem-se o erro na equação 3.26.

$$E(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n) \tag{3.26}$$

Procede-se, então, ao cálculo do primeiro termo de $\delta_j(n)$ derivando-se a equação anterior com relação a $y_j(n)$ e obtém-se a expressão a seguir. Esta pode ser reescrita de acordo com a equação 3.27.

$$\begin{aligned} \frac{\partial E(n)}{\partial y_j(n)} &= \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \\ &= \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \end{aligned} \tag{3.27}$$

Como $y_k(n) = \phi_k(v_k(n))$, o erro para o neurônio k pode ser representado de acordo com a equação 3.28.

$$e_k(n) = d_k(n) - \phi_k(v_k(n)) \tag{3.28}$$

Como o valor de d_k é constante para a mesma amostra n do conjunto de dados,

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\phi_k'(v_k(n)) \tag{3.29}$$

Já $v_k(n)$ consta na equação 3.30, em que m é o número de neurônios da camada escondida a que o neurônio j pertence.

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{3.30}$$

A partir da equação acima, pode-se obter a equação 3.31.

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{3.31}$$

Assim, substituindo as equações 3.29 e 3.31 em 3.27, obtém-se a equação 3.32, em que δ_k é o gradiente local de um neurônio de saída, como obtido na equação 3.23.

$$\begin{aligned} \frac{\partial E(n)}{\partial y_j(n)} &= -\sum_k e_k(n) \phi_k'(v_k(n)) w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n) \end{aligned} \tag{3.32}$$

Finalmente, o gradiente local do neurônio da camada escondida pode ser reescrito de acordo com a equação 3.33.

$$\delta_j(n) = \phi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{3.33}$$

Função de Ativação

Há alguns tipos de função de ativação que podem ser utilizados. Dentre eles há as funções logística e tangente hiperbólica, ambas não-lineares. Elas estão representadas, respectivamente, em 3.34 e 3.35.

$$\phi_k(v_k(n)) = \frac{1}{1 + e^{-v_k(n)}} \tag{3.34}$$

$$\phi_k(v_k(n)) = tanh(v_k(n)) \tag{3.35}$$

Representação Matricial do Algoritmo

Há basicamente dois tipos de computações que os neurônios, tanto de camadas escondidas, quanto da camada de saída realizam.

1. O cálculo das saídas de cada neurônio, com propagação direta dos sinais. As saídas são uma função não-linear diferenciável (de ativação) dos sinais de entrada dos neurônios multiplicados pelos respectivos pesos.

2. Obtenção de uma estimativa do vetor de gradientes, necessária para o cálculo das atualizações dos pesos durante o treinamento.

Foi visto anteriormente como ambos os cálculos são feitos para um neurônio de uma camada escondida e para um neurônio de uma camada de saída. Como todas as saídas precisam ser calculadas e, todos os pesos, atualizados, a representação do algoritmo *backpropagation* pode ser feita de forma matricial.

Para tanto, será tomada como referência a rede neural da figura 3.9. Considere, para $n, i, j \in \mathbb{N}^*$ e $i \geq 2$, o vetor $\boldsymbol{X}_{(n+1),1}$ com n entradas mais o *bias* e os vetores $\boldsymbol{Y}^{(i)}_{(j+1),1}$ como sendo os vetores das saídas dos neurônios das camadas i mais o *bias*, sendo j o número de neurônios de cada camada, com j podendo variar entre as camadas.

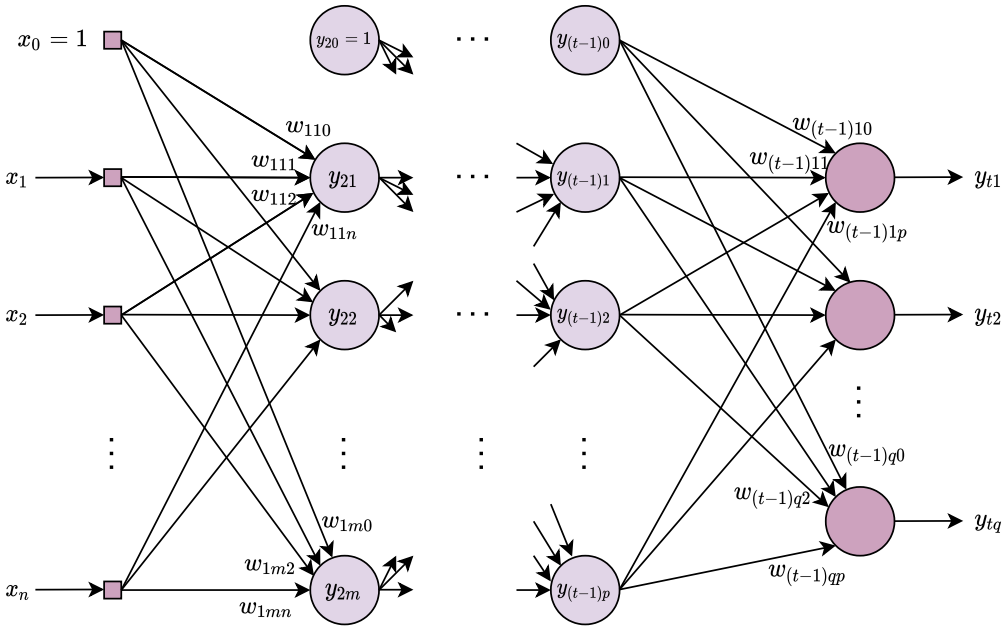

Figura 3.9: Representação de um MLP genérico com t camadas, n entradas e q saídas

Em 3.36, constam as representações dos vetores $\boldsymbol{X}$ e $\boldsymbol{Y}^{(2)}$. Os subíndices de cada elemento dos vetores das saídas indicam a camada a que pertentem e o neurônio de tal camada, respectivamente. O vetor de entradas possui apenas um subíndice, que indica qual é a entrada dentre as n. Além disso, $n, m \in \mathbb{N}^*$.

$$\boldsymbol{X}_{(n+1),1} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad \boldsymbol{Y}^{(2)}_{(m+1),1} = \begin{bmatrix} y_{20} \\ y_{21} \\ \vdots \\ y_{2m} \end{bmatrix} \tag{3.36}$$

Já em 3.37, estão representados os vetores das saídas da última camada escondida $t-1$ e da camada de saída t, em que $t, p, q \in \mathbb{N}^*$.

$$\boldsymbol{Y}^{(t-1)}_{(p+1),1} = \begin{bmatrix} y_{(t-1)0} \\ y_{(t-1)1} \\ \vdots \\ y_{(t-1)p} \end{bmatrix} \qquad \boldsymbol{Y}^{(t)}_{(q+1),1} = \begin{bmatrix} y_{t0} \\ y_{t1} \\ \vdots \\ y_{tq} \end{bmatrix} \tag{3.37}$$

Além disso, é necessário definir os vetores $\boldsymbol{\epsilon}^{(i)}_{j,1}$, que contêm os gradientes locais δ de cada neurônio, com $i, j \in \mathbb{N}^*$, $i \geq 2$ e i sendo a camada e j o número de neurônios da respectiva camada. Os subíndices de δ indicam a camada e o neurônio da camada, respectivamente. Em 3.38, estão representados os vetores dos gradientes locais das camadas 2, que é a primeira camada escondida, e t, a de saída.

$$\boldsymbol{\epsilon}^{(2)}_{m,1} = \begin{bmatrix} \delta_{21} \\ \delta_{22} \\ \vdots \\ \delta_{2m} \end{bmatrix} \qquad \boldsymbol{\epsilon}^{(t)}_{q,1} = \begin{bmatrix} \delta_{t1} \\ \delta_{t2} \\ \vdots \\ \delta_{tq} \end{bmatrix} \tag{3.38}$$

Também é necessário definir as matrizes que contêm os pesos. Para tanto, considere a notação $\boldsymbol{W}^{(i)}_{u,(s+1)}$ como a matriz que contém os pesos que multiplicam as saídas da camada anterior i, que contém s neurônios, e que a camada cujas saídas serão calculadas, $i+1$, contém u neurônios, com $u, s \in \mathbb{N}^*$. Em cada matriz, os pesos possuem três subíndices, sendo o primeiro a camada que propicia as entradas; o segundo, o neurônio da camada cujas saídas serão obtidas; e, o terceiro, a qual saída da camada anterior o peso se refere.

Pode-se visualizar em 3.39 a matriz dos pesos da camada de entrada, ou camada 1.

$$\boldsymbol{W}^{(1)}_{m,(n+1)} = \begin{bmatrix} w_{110} & w_{111} & w_{112} & \cdots & w_{11n} \\ w_{120} & w_{121} & w_{122} & \cdots & w_{12n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{1m0} & w_{1m1} & w_{1m2} & \cdots & w_{1mn} \end{bmatrix} \tag{3.39}$$

Já em 3.40, tem-se a matriz dos pesos da primeira camada escondida, a camada 2. Nesse caso, $r \in \mathbb{N}^*$ é o número de neurônios da camada 3, não representada na figura 3.9.

$$\boldsymbol{W}^{(2)}_{r,(m+1)} = \begin{bmatrix} w_{210} & w_{111} & w_{212} & \cdots & w_{21m} \\ w_{220} & w_{221} & w_{222} & \cdots & w_{22m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{2r0} & w_{2r1} & w_{2r2} & \cdots & w_{2rm} \end{bmatrix} \tag{3.40}$$

De forma análoga às matrizes anteriores, em 3.41, tem-se a matriz da última camada escondida, que é a $t-1$.

$$\boldsymbol{W}^{(t-1)}_{q,(p+1)} = \begin{bmatrix} w_{(t-1)10} & w_{(t-1)11} & w_{(t-1)12} & \cdots & w_{(t-1)1p} \\ w_{(t-1)20} & w_{(t-1)21} & w_{(t-1)22} & \cdots & w_{(t-1)2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{(t-1)q0} & w_{(t-1)q1} & w_{(t-1)q2} & \cdots & w_{(t-1)qp} \end{bmatrix} \tag{3.41}$$

Como a obtenção das saídas dos neurônios ocorre pela multiplicação das entradas dos neurônios pelos pesos e depois pela passagem do resultado pela função de transferência, estas operações podem ser representadas de acordo com a equação 3.42. Nela, $\mathbf{Y}^{(2)}_{m,1}$ é o vetor de saídas da camada 2 sem o *bias*.

$$\mathbf{Y}^{(2)}_{m,1} = \phi(\boldsymbol{W}^{(1)}_{m,(n+1)}\boldsymbol{X}_{(n+1),1}) \tag{3.42}$$

Deve-se então acrescentar o *bias* à primeira entrada de $\mathbf{Y}^{(2)}_{m,1}$, para calcular as saídas da segunda camada escondida, de acordo com a equação 3.43.

$$\mathbf{Y}^{(3)}_{r,1} = \phi(\boldsymbol{W}^{(1)}_{r,(m+1)}\mathbf{Y}^{(2)}_{(m+1),1}) \tag{3.43}$$

São calculadas, sucessivamente, as saídas das camadas escondidas até obter o vetor de saídas da rede, ou seja, da última camada, o que está representado em 3.44.

$$\mathbf{Y}^{(t)}_{(q+1),1} = \phi(\boldsymbol{W}^{(t-1)}_{q,(p+1)}\mathbf{Y}^{(t-1)}_{(p+1),1}) \tag{3.44}$$

Já o cálculo das atualizações dos pesos, como já explicado, ocorre da última camada para a primeira. Inicia-se calculando os gradientes locais da camada de saída, como em 3.45.

$$\Delta\boldsymbol{W}^{(t-1)}_{(p+1),q} = \eta(\boldsymbol{\epsilon}^{(t)}_{q,1}(\mathbf{Y}^{(t-1)}_{(p+1),1})^T)^T \tag{3.45}$$

Os novos pesos são calculados de acordo com 3.46.

$$\boldsymbol{W}^{(t-1)}_{(p+1),q} \coloneqq \boldsymbol{W}^{(t-1)}_{(p+1),q} + \Delta\boldsymbol{W}^{(t-1)}_{(p+1),q} \tag{3.46}$$

São feitos os cálculos de forma análoga para todas as camadas. Em 3.47 e 3.48, tem-se a atualização dos pesos da camada 2.

$$\Delta\boldsymbol{W}^{(2)}_{(m+1),r} = \eta(\boldsymbol{\epsilon}^{(3)}_{r,1}(\boldsymbol{Y}^{(2)}_{(m+1),1})^T)^T \tag{3.47}$$

$$\boldsymbol{W}^{(2)}_{(m+1),r} \coloneqq \boldsymbol{W}^{(2)}_{(m+1),r} + \Delta\boldsymbol{W}^{(2)}_{(m+1),r} \tag{3.48}$$

Finalmente, em 3.49 e 3.50, tem-se a atualização dos pesos da primeira camada.

$$\Delta\boldsymbol{W}^{(1)}_{(n+1),m} = \eta(\boldsymbol{\epsilon}^{(2)}_{m,1}(\boldsymbol{X}^{(1)}_{(n+1),1})^T)^T \tag{3.49}$$

$$\boldsymbol{W}^{(1)}_{(n+1),m} \coloneqq \boldsymbol{W}^{(1)}_{(n+1),m} + \Delta\boldsymbol{W}^{(1)}_{(n+1),m} \tag{3.50}$$

Assim, percebe-se que o algoritmo é basicamente composto de multiplicações matriciais, como mencionado no capítulo 2. Além disso, quanto mais camadas escondidas e quanto mais neurônios por camada, mais custoso é o cálculo computacional do algoritmo, visto que o número de matrizes e suas dimensões aumentam.

3.4 FPGAs

A FPGA consiste em um circuito integrado que pode ser customizado para aplicações específicas em campo. Tal customização ocorre pelo usuário por meio da utilização de uma linguagem de descrição de *hardware*, ou *hardware description language* (HDL). Na figura 3.10 consta um exemplo de FPGA, a Basys 3 da Digilent.

Os fabricantes mais populares são Xilinx, Altera e Lattice. As arquiteturas das FPGAs diferem entre diferentes fabricantes e entre diferentes famílias do mesmo fabricante [17]. Além disso, as HDLs que mais se destacam são o VHDL e o Verilog.

As FPGAs são compostas de um grande *array* de blocos lógicos configuráveis (CLBs), de blocos de processamento de sinais digitais (DSPs), de blocos de RAM (BRAM), e de blocos de entradas e saídas (IOBs). A figura 3.11 mostra um diagrama genérico da arquitetura de uma FPGA, em que os blocos mencionados anteriormente estão dispostos em uma rede de interconexões configuráveis por meio das HDLs [18]. Nela há também um bloco denominado DCM ou *digital clock manager*, que está presente principalmente nas FPGAs

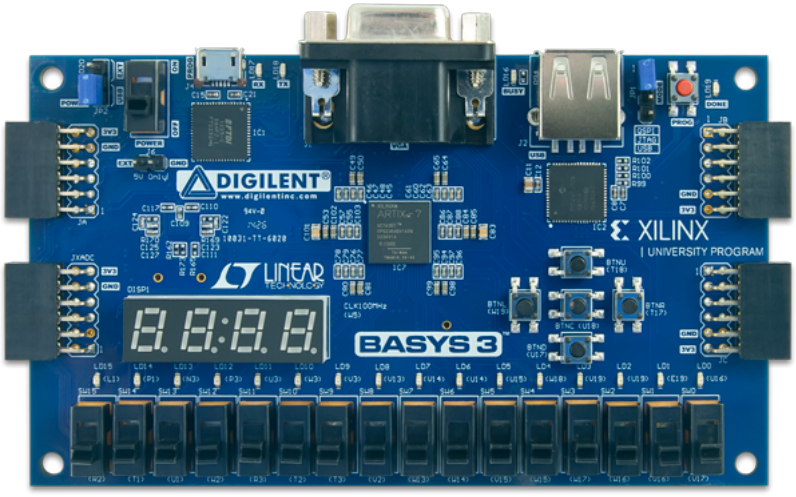

Figura 3.10: Foto da Basys 3 Artix-7 FPGA *Trainer Board* da Digilent

produzidas pela Xilinx e é responsável pelas operações com o *clock*.

Os CLBs são as unidades básicas da arquitetura da FPGA. Estes podem implementar funções complexas, de memória e sincronizar códigos quando ligados entre si por meio das conexões configuráveis. São compostos por unidades menores como flip-flops, look-up tables (LUTs) e multiplexadores. O flip-flop é a menor unidade de memória de uma FPGA e consiste em um registrador binário utilizado para armazenar estados lógicos que podem ser alterados apenas a cada batida de *clock*. Já a LUT armazena, para cada combinação de entradas, as saídas de uma operação lógica, o que consiste em um modo rápido de obter resultados, visto que são referenciados ao invés de calculados. E os multiplexadores selecionam qual das entradas conectadas a ele aparecerá na sua saída.

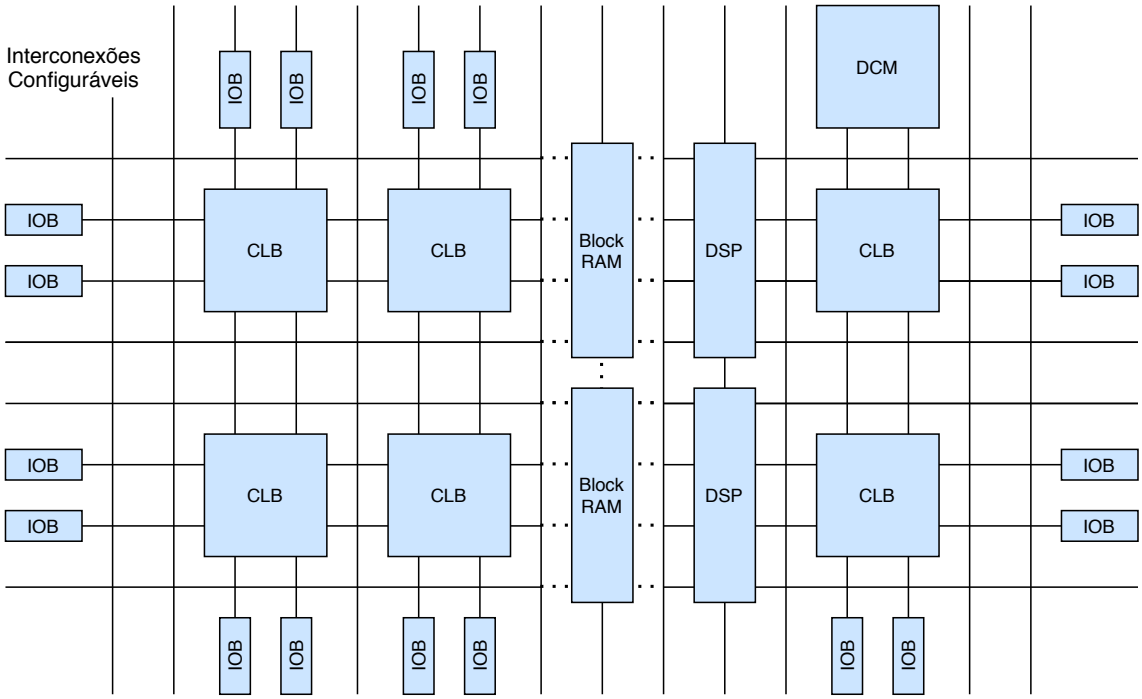

Figura 3.11: Diagrama de blocos da arquitetura de FPGAs

Os DSPs são blocos de processamento digital de sinais e são embarcados em FPGAs com o intuito de melhorar a eficiência no uso de recuros e a performance de operações específicas, como multiplicações de números inteiros e operações de números em ponto flutuante. Já a *Block* RAM é um tipo de *random access memory* presente em FPGAs para armazenamento de dados. Por sua vez, os recursos de entrada e saída são estruturas físicas que permitem a conexão entre uma FPGA a outros dispositivos. Permitem o suporte a, por exemplo, controladores de memórias DDR3 e interfaces de processadores.

O fluxo de *design* de FPGAs começa com a decisão da especificação do circuito em linguagem de *hardware*, seguida da etapa de simulação do circuito por meio de *testbenchs*. Em seguida, ocorre a etapa de síntese, em que o código desenvolvido pelo programador é traduzido para uma descrição das conexões do circuito, que consiste na *netlist*, e que pode ser mapeada para a FPGA. Finalmente, tem-se a etapa de implementação, em que se mapeia o circuito sintetizado nos recursos físicos da FPGA especificada e o *bitstream* é gerado. O fluxo de etapas de desenvolvimento de projetos pode ser melhor compreendido através da figura 3.12.

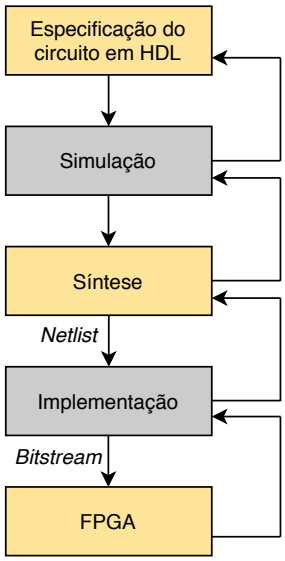

Figura 3.12: Representação das etapas de projeto

3.5 Representação Numérica

Em sistemas digitais, números são representados digitalmente por um conjunto finito de bits. Diferentes aplicações exigem diferentes representações. As mais comuns são: ponto fixo e ponto flutuante. A principal diferença entre os dois consiste na localização do ponto decimal. Números em ponto fixo têm uma quantidade pré-determinada de dígitos destinados à parte fracionária, o que não ocorre para aqueles em ponto flutuante. Por esse motivo, a representação em ponto flutuante oferece um intervalo maior de números que podem ser representados e também maior precisão do que a representação em ponto fixo [19].

No caso da escolha da representação entre ponto fixo e ponto flutuante para o desenvolvimento de *hardware* em FPGA para o aprendizado de máquina enfrenta-se um *trade-off* [20]. Este consiste na área *versus* precisão. A precisão é muito importante para a acurácia da rede e para a velocidade de convergência do algoritmo, mas, quanto maior a precisão, maior o custo em termos de recursos utilizados da FPGA, ou seja, maior a sua área ocupada. A implementação em ponto flutuante é a ideal em termos de precisão e é a utilizada nas simulações de redes neurais nos processadores de PCs. Além disso, a escolha do ponto flutuante torna mais fácil a comparação do desempenho da rede em FPGA com sua versão rodando em computadores pessoais, visto que é a representação utilizada por eles. Entretanto, os recursos em uma FPGA são limitados e representações em 16 ou 32 *bits* em ponto fixo ocupam muito menos área, permitindo a implementação de redes neurais maiores.

O problema de usar representação em ponto fixo em circuitos somadores e acumuladores é o risco de saturação da representação máxima do sistema, o que dificilmente ocorre com a representação em ponto flutuante. Erros de computação devido à saturação podem inclusive impedir a convergência dos algoritmos. Portanto, para este trabalho escolheu-se a representação em ponto flutuante, mesmo que mais custosa. Outro motivo para a escolha foi a possibilidade de comparar os resultados obtidos com aqueles das referências deste trabalho, explicitadas no capítulo 2.

Assim, é importante descrever o formato da representação em ponto flutuante, que tem um padrão descrito pelo IEEE, denominado IEEE 754. O número em ponto flutuante é representado por meio de um sinal s, um expoente E e uma fração $b_0.b_1b_2...b_{w_f-1}$ [21]. Os *bits* b_i da parte fracionária tem peso 2^{-i}, com $i \in \mathbb{N}$, em que o mais significativo b_0 é constante e igual a 1, de modo que o número é dito normalizado pois $1 \leq b_0.b_1b_2...b_{w_f-1} < 2$. A parte E promove um *range* dinâmico, pois consiste em uma potência de 2 que multiplica a parte fracionária. Já o *bit* de sinal indica se o número será negativo, quando $s = 1$, ou positivo, para $s = 0$.

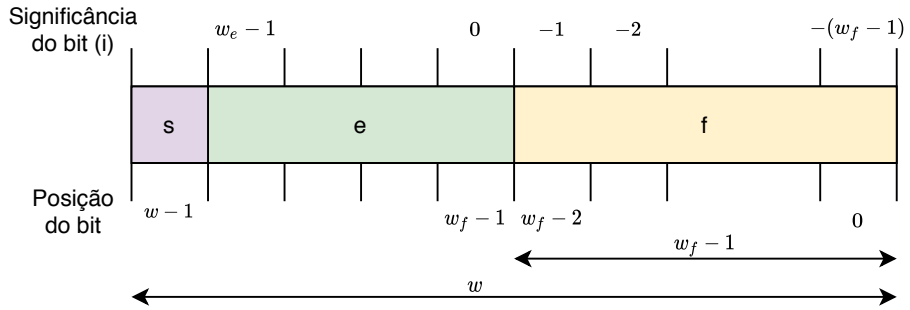

Figura 3.13: Representação dos campos de uma representação em ponto flutuante

O valor do número é dado pela equação 3.51, em que s representa o valor de um *bit*, $E, w_f \in \mathbb{N}^*$ e w_f é o comprimento da fração considerando o *bit* constante b_0.

$$v = (-1)^s 2^E b_0.b_1b_2...b_{w_f-1} \tag{3.51}$$

Para melhor compreender a representação, verifique a figura 3.13, na qual constam os três campos que compõem o número em ponto flutuante. Nela, verificam-se os campos e e f. Como b_0 é constante, $f = b_1 b_2 ... b_{w_f - 1}$. Já o campo e é utilizado para calcular E de acordo com a equação 3.52.

$$E = e - (2^{w_e - 1} - 1) \tag{3.52}$$

Por sua vez, e é calculado de forma análoga a um número inteiro sem sinal, em que $w_e \in \mathbb{N}^*$ é o comprimento do campo de expoente e, representado na equação 3.53. Nela, e_i representa o *bit* da posição i em e.

$$e = \sum_{i=0}^{w_e - 1} e_i 2^i \tag{3.53}$$

3.6 Vivado e IPs

O Vivado *Design Suite* é um *software* produzido pela Xilinx para síntese e análise de *designs* em HDL. Dentre os seus componentes há o Vivado IP *Integrator*. No contexto de FPGAs, IP significa *Intellectual Property*, ou propriedade intelectual, e são bibliotecas de módulos em HDL desenvolvidos por usuários ou empresas para tornar mais rápido o desenvolvimento de um projeto, visto que o programador pode utilizar as partes que já estão prontas ao invés de desenvolver cada parte do *design*. Tais módulos são como uma caixa preta, em que o usuário pode escolher determinadas configurações do seu modo de funcionamento, mas o seu código não pode ser modificado nem visualizado.

Os blocos IP são como peças de lego no sentido de que se pode usar uma peça já pronta para complementar o seu próprio projeto, desde que se respeite que o componente utilizado é propriedade intelectual de outra pessoa. A Xilinx tem uma grande biblioteca de IPs para serem usados em suas FPGAs e também uma biblioteca embutida no próprio Vivado.

Neste trabalho utilizou-se o Vivado e o IP da Xilinx denominado *Floating-Point Operator* [21]. Este realiza diversos tipos de operações em ponto flutuante, como adição, subtração, comparação, divisão, exponenciação, multiplicação e acumulação. Ademais, há várias configurações disponíveis, como a precisão das entradas, ou seja, o número de *bits*; otimizações de velocidade e latência; utilização ou não de DSPs; utilização de sinais de *reset*; e decisão entre os modos de fluxo bloqueante e não-bloqueante. Tais modos serão explicados na seção 3.7, em que serão descritas as entradas e saídas desse IP, assim como o protocolo que tais sinais obedecem.

3.7 Protocolo AXI4-*Stream*

O protocolo AXI4-*Stream* é usado como interface que conecta componentes que trocarão dados entre si. Tal interface pode ser usada para conectar um único mestre, que gera os dados, para um único escravo, que recebe os dados, ou por vários mestres e escravos simultaneamente. Cada canal de entrada ou saída utiliza os mesmos conjuntos de sinais, sendo alguns deles opcionais. O IP *Floating-Point Operator* utiliza este protocolo para todos os sinais, com exceção dos de controle, como o `aclk`, `aclken` e `aresetn`.

Nesta seção serão mostrados os sinais selecionados para utilização neste trabalho pelo *Floating-Point Operator*, assim como o funcionamento do protocolo que promove a comunicação entre os blocos [21]. Isso porque, como será visto no capítulo 4, os IPs configurados para multiplicação, adição e acumulação foram amplamente utilizados para realização deste trabalho.

Um canal consiste sempre em pelo menos dois sinais, o TVALID e o TDATA. As outras portas suportadas pelo *Floating-Point Operator* são o TREADY, o TLAST e o TUSER, sendo que este último não foi utilizado neste trabalho e, por isso, não será abordado. Somente para o acumulador o uso de TLAST é obrigatório para indicar o último dado a ser somado.

Os operandos estão contidos no campo TDATA, tando para as entradas, quanto para as saídas. Os sinais TVALID e TREADY realizam um *handshake* para transferir a mensagem, e o *payload* está contido nos campos TDATA e TLAST. O TVALID é enviado pelo mestre, enquanto o TREADY, pelo escravo. O primeiro indica que os dados enviados no campo de *payload* estão válidos e, o segundo, que o escravo está pronto para receber os dados. A transferência ocorre apenas quando ambos estão em nível lógico alto na subida do *clock*.

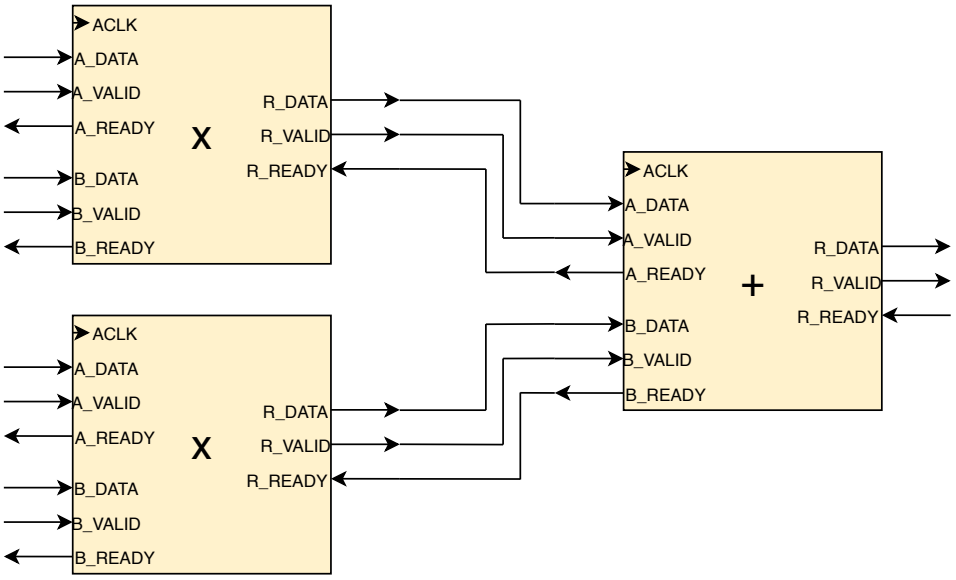

Figura 3.14: Representação dos sinais do *Floating-Point Operator* ao somar dois produtos

Um exemplo de como conectar os blocos consta na figura 3.14. Nela, os dois primeiros blocos foram configurados como multiplicadores e, o terceiro, como somador. Dessa forma,

a saída do circuito é a soma dos produtos das entradas dos multiplicadores. Esta disposição dos blocos é bastante importante para o trabalho.

Além desses dois tipos de blocos também foi utilizado o acumulador, cuja representação dos sinais de entrada e de saída consta na figura 3.15.

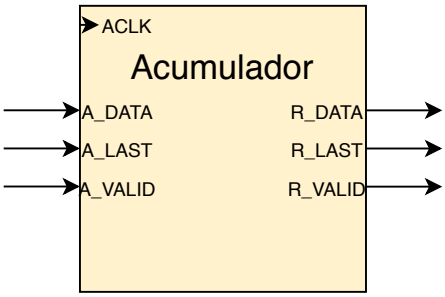

Figura 3.15: Sinais do *Floating-Point Operator* utilizado como acumulador

O modo de operação pode ser definido como não-bloqueante ou bloqueante. No primeiro não há o sinal TREADY em nenhum canal, o que implica que a falta de dados em um canal de entrada não bloqueia a execução de uma operação se o dado é recebido em outro canal de entrada. Nesse modo a implementação é mais simples, ou seja, menos recursos são utilizados. Já no segundo, as operações não ocorrem até que dados novos estejam disponíveis em todos os canais de entrada. Esse bloqueio, em que a *back pressure* (TREADY) é utilizada, previne a perda de dados de forma que o fluxo de dados de saída é apenas propagado quando o bloco seguinte está pronto para processá-los. Quando todos os canais estão recebendo dados válidos, a operação ocorre e o resultado é disponibilizado no canal de saída. Mas, se este canal estiver com seu TREADY em nível lógico baixo, os resultados das operações não são descarregados para o barramento e ficam armazenadas no *buffer* de saída interno ao bloco. Com isso, os *buffers* de entrada não disponibilizam os dados válidos armazenados para novas operações, de modo que os *buffers* de entrada enchem com a chegada de novos dados. Quando isso acontece, os TREADYs de cada canal de entrada são colocados em nível lógico baixo impedindo a chegada de novos dados nos canais de entrada quando os seus TREADYs estão conectados ao TREADY do canal de saída do bloco que disponibiliza os dados das suas entradas. Um exemplo dessa situação foi mostrado na figura 3.14.

Outro tipo de bloqueio é o proporcionado pelo TVALID, em que é necessário que todas as entradas estejam válidas para que haja a operação. Caso contrário, os dados de entrada válidos são armazenados no *buffer* de entrada do canal até que todos os dados estejam válidos e tais *buffers* possam ser descarregados. Mas, se o *buffer* que recebe os dados de entrada encher, o TREADY do respectivo canal é colocado em nível lógico baixo até que os demais canais recebam dados válidos.

O modo bloqueio é útil, portanto, quando são usados os *Floating-Point Operators* em cascata. Neste trabalho utilizou-se o modo bloqueante para multiplicadores e somadores por esse motivo, de modo a prevenir perdas de dados ou operações não sincronizadas devido a atrasos em algum ponto da configuração. Além disso, utilizou-se o modo não-bloqueante

apenas para o IP configurado como acumulador, porque apenas um acumulador é utilizado no respectivo *design*, ao final de toda a cadeia de todas as operações.

Na figura 3.16 podem ser visualizados exemplos dos sinais do acumulador no modo não-bloqueante. A função do acumulador é somar os valores da entrada a cada subida do *clock*, mas apenas quando o seu TVALID estiver em nível lógico alto. Além disso, há também a entrada TLAST, que, quando está em nível lógico alto, indica que o valor da entrada é o último a ser somado antes de zerar a saída para começar outra acumulação. O TVALID da saída indica a validez de cada operação de soma realizada e o seu TLAST em nível lógico alto indica que a acumulação terminou e que aquele é o resultado almejado. O último dado, $A5$, aparece na saída porque, neste modo, o dado de entrada aparece na saída independentemente do TVALID quando a acumulação ainda não começou. O dado $A3$ também aparece na saída, mas o TVALID indica que não é um dado válido.

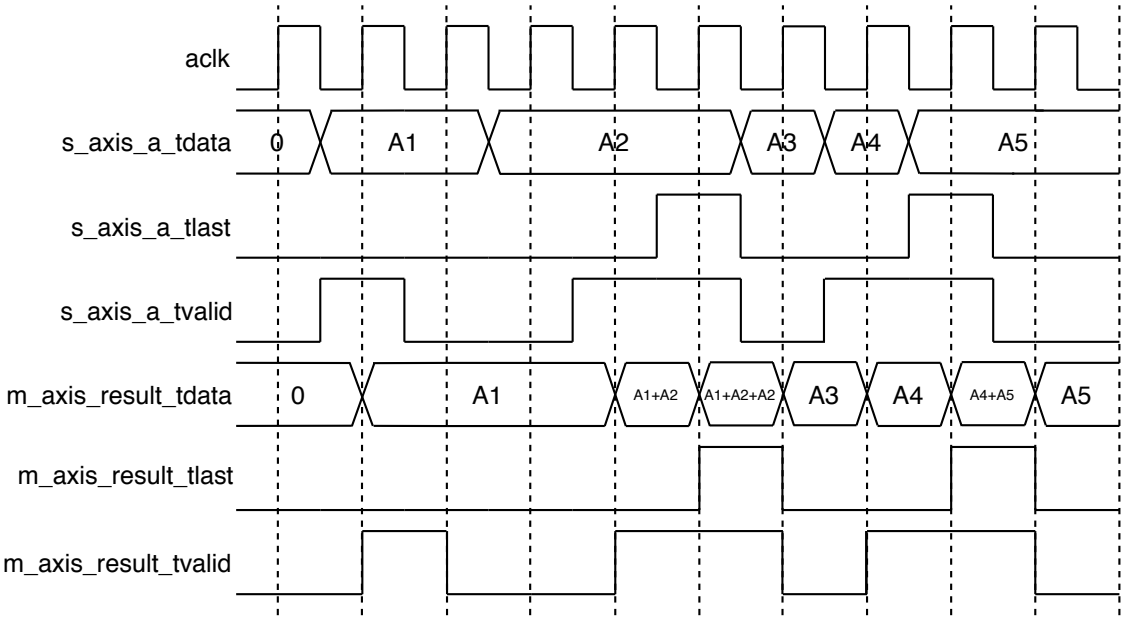

Figura 3.16: Modo não-bloqueante exemplificado com o acumulador

Também foi exemplificado o modo de operação não-bloqueante, que pode ser visualizado na figura 3.17. Os dados com X no campo TDATA representam um *don't care*, visto que os respectivos sinais TVALID não estão setados nesses momentos. Essa figura mostra o comportamento de bloqueio e a *back pressure* no caso de um somador. O primeiro pode ser verificado pelo fato de os dados serem somados em ordem de chegada nos canais de entrada, mas apenas quando ambos os sinais TVALID de A e B estão setados. Já a *back pressure* é verificada quando o TREADY da saída passa para nível lógico baixo e as operações são bloqueadas. Enquanto isso há dados válidos em cada uma das entradas, que acabam levando à saturação dos canais e, portanto, aos respectivos TREADYs passarem também a nível lógico baixo. Deve ser observado que a capacidade dos *buffers* reais é maior que a mostrada no diagrama, que tem a finalidade de ilustrar o conceito. Além disso, na figura, a latência tanto do acumulador quanto do somador mostrados é igual a um ciclo de *clock*, valor que foi alterado durante a realização deste trabalho, como será explicitado na seção 5.

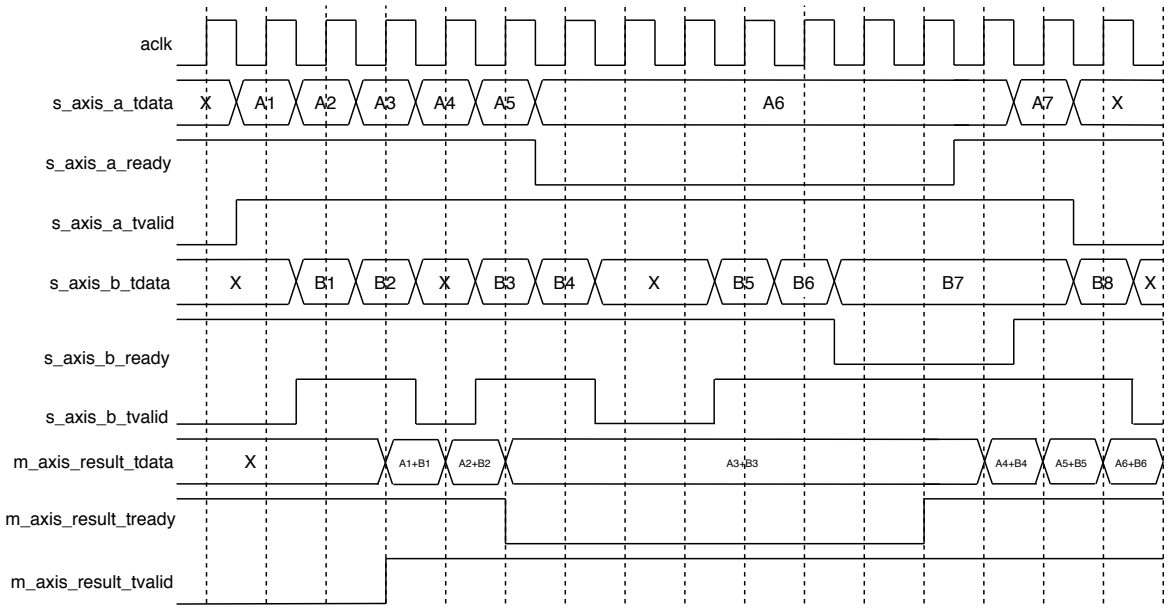

Figura 3.17: Modo bloqueante exemplificado com somador

3.8 Máquina de Estados Finita

Uma máquina de estados finita ou *Finite State Machine* (FSM) é um circuito sequencial utilizado em diversos sistemas digitais para controlar o comportamento de sistemas e de fluxo de dados, tanto em programas de computadores quanto em circuitos lógicos [22]. Há diversos equipamentos que a utilizam como os semáforos que controlam o fluxo de carros, as máquinas automáticas de vendas que liberam o produto após as moedas corretas serem depositadas e os elevadores que sobem até o andar mais elevado cujo botão foi pressionado antes de descer.

Tais máquinas são formadas por diversos estados, sendo que não se pode estar em dois estados ao mesmo tempo. O estado em que a máquina se encontra é chamado de estado atual e a mudança para o próximo estado ocorre apenas quando as condições pré-estabelecidas forem cumpridas. A mudança para o estado seguinte é denominada transição. Dessa forma, uma FSM é definida pela lista dos seus estados e das condições de mudança de estados. Neste trabalho a FSM utilizada consiste em um circuito sequencial síncrono e, portanto, as mudanças ocorrem apenas nas bordas de subida (ou descida) do *clock*.

Além disso, há dois tipos de máquinas de estados: a de Mealy e a de Moore. Na primeira, as saídas dependem tanto do estado atual quanto das entradas. Já na segunda as saídas dependem apenas do estado atual. O modelo da máquina de Mealy está ilustrado na figura 3.18. Este é constituído de um circuito combinacional representado pela "Lógica de Próximo Estado e de Saídas", em que os sinais das saídas e do próximo estado são gerados; e também de um registrador, normalmente modelado por um *flip-flop* do tipo D, que armazena o estado atual até a condição de próximo estado ser atingida e a borda do *clock* ocorrer.

O modelo de Moore também foi representado e consta na figura 3.19. Neste caso o

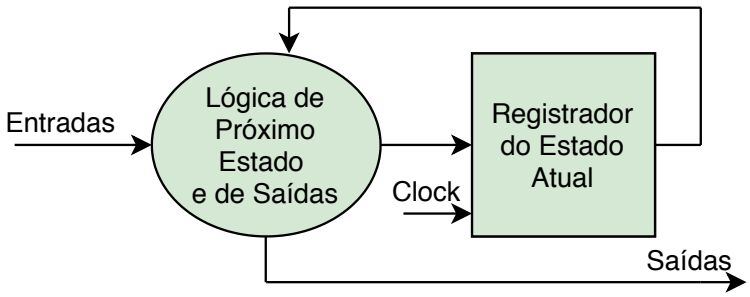

Figura 3.18: Modelo de uma máquina de estados de Mealy

circuito combinacional determina a lógica de próximo estado, que depende tanto do estado atual quanto das entradas. Porém, as saídas dependem apenas do estado atual. O registrador utilizado também é geralmente um *flip-flop* do tipo D e a lógica de saídas é um circuito combinacional.

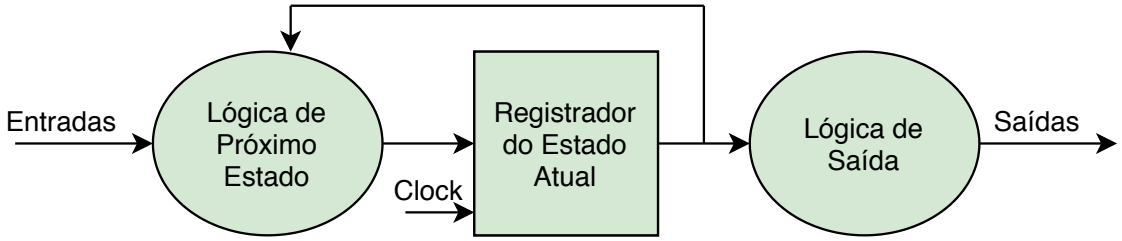

Figura 3.19: Modelo de uma máquina de estados de Moore

O desenvolvimento de uma FSM geralmente é iniciado pelo planejamento em alto nível em forma de esquemático de forma a definir o que será feito em cada estado e qual é a condição de transição. Apenas após essas definições se escreve o *design* em HDL. O VHDL possui uma sintaxe pré-definida para a implementação de máquinas de estados. Já a determinação dos códigos dos estados em nível de *hardware* pode ser de vários tipos, dentre eles há o *one-hot*, o binário e o código de *gray*. Geralmente é a ferramenta de síntese que decide o tipo, como é o caso do Vivado. No entanto, neste *software* a determinação pode ser forçada para um determinado tipo, a depender da necessidade, visto que cada um tem seus prós e contras.

Capítulo 4

Metodologia e Desenvolvimento do Projeto

Neste capítulo serão abordados os procedimentos adotados para o desenvolvimento do acelerador e para a obtenção dos resultados. Além disso, serão mostradas e discutidas as decisões tomadas durante o desenvolvimento do projeto. Dessa forma, serão explicadas as etapas do desenvolvimento do *hardware* projetado.

4.1 Implementação

O *hardware* desenvolvido é dedicado aos cálculos da soma dos produtos das entradas de cada amostra pelos pesos. Ou seja, foi implementada a primeira etapa do treinamento de um neurônio, que corresponde à multiplicação de uma linha por uma coluna de uma matriz. Como já explicitadas na seção 3.2, serão importantes para compreensão do *hardware* as definições do vetor de entradas $\boldsymbol{x}$ e do vetor de pesos $\boldsymbol{w}$, retomadas na equação 4.1, sendo $m \in \mathbb{N}^*$ o número de entradas das amostras, $x_0 = 1$ a entrada fixa em 1 cujo peso w_0 é denominado *bias*.

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix} \tag{4.1}$$

O que o circuito projetado faz é calcular z, sendo esta a entrada da função de ativação. Isso porque esta é a parte mais custosa do algoritmo. Dessa forma, verifica-se o cálculo de z na equação 4.2.

$$z = w_0 + w_1 x_1 + ... + w_m x_m = \boldsymbol{w}^T \boldsymbol{x} \tag{4.2}$$

A base de dados utilizada como referência para o projeto do circuito foi a denominada "Sonar"[23]. Esta contém 208 amostras, com cada uma contendo 60 números de 0.0 a 1.0, que são as *features*. Cada amostra representa um padrão obtido pelo retorno dos sinais de um sonar de um metal ou de uma rocha. Ademais, cada número representa a energia de uma determinada banda de frequência integrada sobre um certo período de tempo. As amostras também são associadas a um *label* contendo a letra "R"ou "M", que indicam se a amostra corresponde a uma rocha ou a um metal, respectivamente. Dessa forma, o problema a ser resolvido através do treinamento supervisionado consiste na classificação de amostras. Assim, a partir da descrição da base de dados utilizada, tem-se que $m = 60$.

Diversas bases de dados utilizam números reais, assim como a deste trabalho. É, portanto, necessário utilizar a representação dos números em ponto fixo ou ponto flutuante. Optou-se por ponto flutuante pelos motivos apresentados na seção 3.5. As arquiteturas que serão mostradas neste trabalho para o cálculo de z foram feitas utilizando o *Floating-Point Operator*, que foi explicado na seção 3.7. Assim, cada bloco de multiplicação ou soma apresentado nas figuras ou no texto correspondem a um desses IPs configurados com a precisão das entradas de dados do tipo *single* (32 bits) e com os tamanhos de expoente e fração padrões, de 8 e 24 bits, respectivamente.

Além disso, as implementações não chegaram a ser testadas em uma FPGA, mas foram simuladas utilizando o Vivado e foram feitas as sínteses de cada uma delas. Com a finalidade de testar as arquiteturas em simulação, considerou-se que seriam admitidas as entradas em forma de dois vetores de 64 ou 16 elementos do tipo std_logic_vector de 32 bits. Não há pinos suficientes para esta quantidade de sinais em FPGAs, então seria necessário transmitir as amostras por meio de um protocolo de comunicação serial de um computador ou de um microcontrolador para a FPGA, o que não foi objetivo deste trabalho.

4.2 Arquiteturas

A forma mais direta de se implementar o *hardware* dedicado a realizar o cálculo de z é utilizar um multiplicador para cada par de elementos dos vetores de entradas e de pesos e utilizar somadores em estrutura de árvore para fazer as adições dos produtos. Para que a estrutura seja em formato de *pipeline*, deve-se utilizar uma potência de 2 para a quantidade de blocos multiplicadores.

Foi feita uma primeira implementação com 61 multiplicadores, que corresponde à equação 4.2 da entrada do perceptron para a base de dados "Sonar". Um dos produtos fica em espera pelos resultados da segunda camada de somadores, a qual resulta em 15 sinais. Um destes era somado com o produto que ficou em espera, de forma que todas as entradas da terceira camada de somadores fossem preenchidas. No entanto, mesmo com o uso do protocolo AXI4-*Stream*, o fato de um dos sinais da terceira camada estar válido antes dos outros atrapalha a estrutura em *pipeline* e não foi possível obter uma configuração que produzisse

os resultados (após o primeiro) a cada batida de *clock*. Além disso, a arquitetura com 64 (2^6) multiplicadores, por não ser específica para uma determinada base de dados, pode ser utilizada como parte de outra implementação.

Dessa maneira, a figura 4.1 mostra a primeira arquitetura proposta, com todos os produtos sendo calculados simultaneamente. Como há também o *bias* a ser computado, são utilizados 61 dos multiplicadores. Por esse motivo, nos três últimos foram colocadas entradas iguais a 0 durante o teste. Na figura, verifica-se a estrutura em forma de árvore e os resultados das operações são indicados nos blocos coloridos, sendo cada camada representada por cores distintas. No código, os resultados são armazenados em vetores cujo número de elementos diminui pela metade a cada camada. Os resultados das multiplicações constam no vetor denominado `mult`, que tem 64 elementos. Já os das somas constam nos vetores denominados `add1`, com 32 elementos, `add2` com 16, `add3` com 8, `add4` com 4 e `add5` com 2. O resultado final z é representado na saída do último somador da figura.

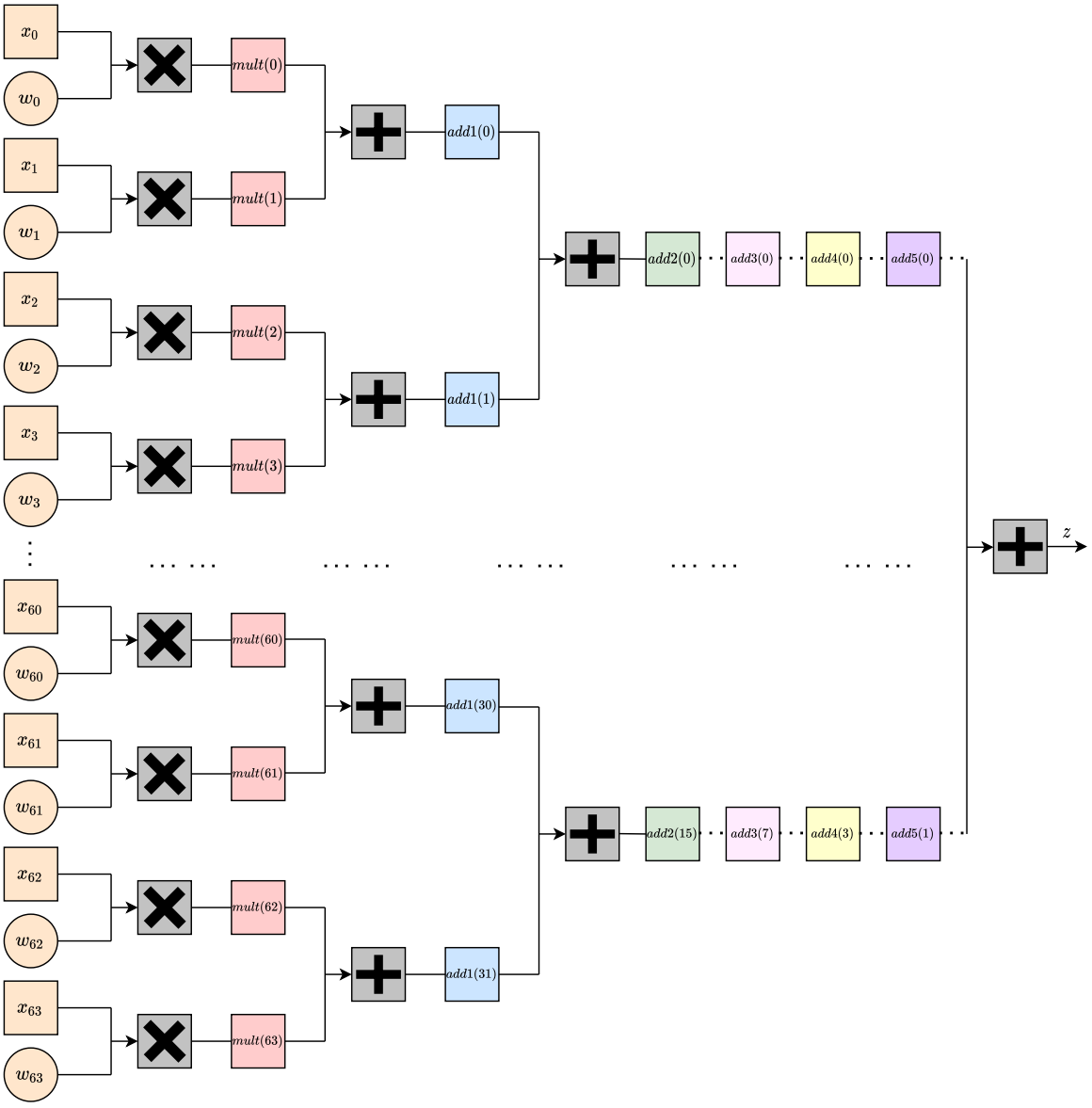

Figura 4.1: Esquemático da implementação para 64 entradas

É importante mencionar que não foi possível representar na figura todas as operações e que, portanto, os pontilhados representam a continuação do diagrama com os números de operadores caindo pela metade a cada camada. Para deixar isso mais claro foram representados os primeiros e últimos resultados das operações de cada camada (a partir da terceira) em blocos quadrados, com cada cor também representando uma camada distinta. Estes foram conectados por uma linha pontilhada para indicar que há operações a serem realizadas entre eles, ou seja, não são diretamente conectados.

Apesar de ser a proposta cujo tempo de obtenção do resultado da computação de z ser o menor, é uma implementação que necessita de muitos recursos, os quais são escassos em FPGAs. Verificou-se no processo de síntese que não seria possível utilizar essa arquitetura para a Basys 3, que foi tomada como referência. Por este motivo, e seguindo a ideia proposta em [6], a segunda arquitetura foi implementada utilizando os dados particionados.

Na figura 4.2 consta o diagrama simplificado que contém os seus módulos. Verifica-se que são recebidas as 64 entradas, como na anterior. Mas, as multiplicações e somas são feitas para cada conjunto de 16 entradas e 16 pesos. Os resultados parciais de cada somatório são somados por um acumulador. O controle dessa estrutura é feito por uma máquina de estados finita, que passa os subconjuntos de entradas para o bloco multiplicador e somador e disponibiliza o resultado da acumulação final na saída.

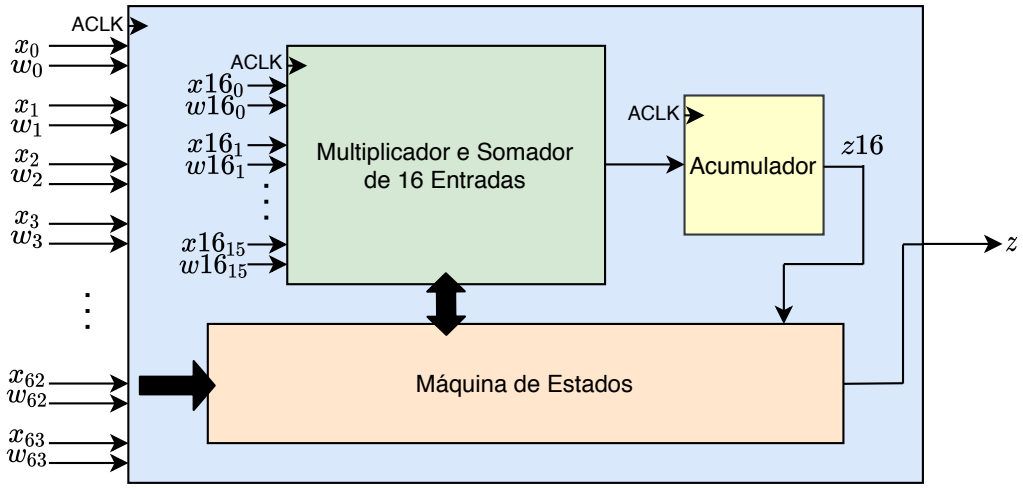

Figura 4.2: Diagrama da arquitetura particionada

O bloco multiplicador e somador de 16 entradas pode ser visualizado na figura 4.3. Este possui a mesma estrutura de árvore que a da arquitetura anterior. Podem ser verificadas as entradas x_j e w_j do sistema com $j \in \mathbb{N}^*$ e $0 \leq j \leq 15$, assim como os sinais resultantes de cada operação de multiplicação e adição. Nesta figura foi suprimido o 16 em $x16$ e $w16$ (da figura 4.2) para não sobrecarregar a notação. Os nomes dos vetores foram mantidos da arquitetura que admite 64 entradas, apenas seus números de elementos são menores. Assim, o vetor `mult` tem 16 elementos; o `add1`, 8; o `add2`, 4, e o `add3`, 2. A saída do último somador, que é a saída do bloco, foi denominada $z16$.

Os códigos foram desenvolvidos em VHDL e como já mencionado, cada um dos blocos

que realiza as multiplicações e adições em estrutura de árvore, tanto de 64 quanto de 16 entradas são formados pelo IP *Floating-Point Operator*. Como as arquiteturas são formadas por um *design* regular e repetitivo de tais operadores, foi utilizado o mecanismo *generate* para descrevê-las em VHDL. Tal mecanismo é utilizado para especificar um grupo de componentes idênticos utilizando apenas uma única especificação. Dessa forma, os operadores de cada camada foram descritos por apenas uma instanciação do respectivo componente, seja ele o multiplicador na primeira, seja ele o somador nas demais.

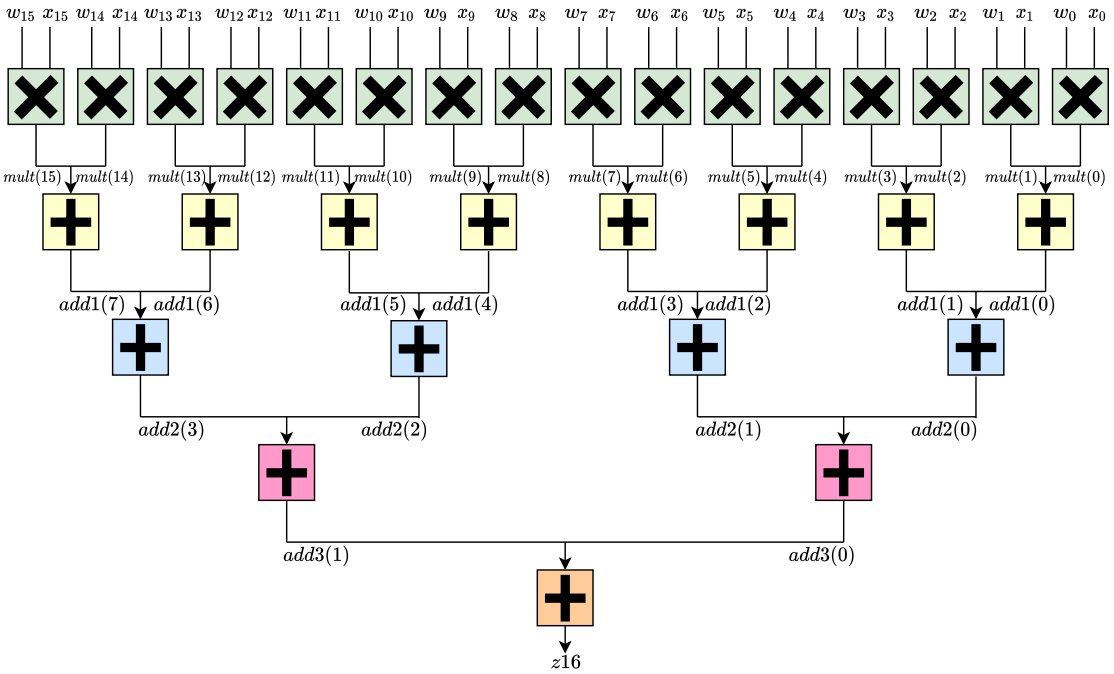

Figura 4.3: Esquemático do multiplicador e somador de 16 entradas

Como mencionado na seção 3.7, cada bloco IP não possui apenas as entradas e saídas de dados, mas também o TVALID, o TREADY e o TLAST, sendo o último apenas encontrado no acumulador. São então 3 os sinais que descrevem cada canal. Para as entradas de cada bloco há a entrada de dados e a TVALID, sendo TREADY uma saída. Já para o canal de saída, o sinal TVALID é uma saída e o TREADY é uma entrada. Ademais, como explicado no capítulo 3, as conexões entre blocos de camadas adjacentes são feitas de acordo com o diagrama da figura 3.14. Por esse motivo, é necessário também especificar a entrada TREADY do canal de saída do último somador e os TVALIDs dos canais de entrada dos multiplicadores, além dos sinais de dados, que são do tipo *float* de 32 bits, modelados como `std_logic_vector`. O TREADY da saída do último somador foi setado como constante e igual a 1 e os TVALIDs das entradas foram utilizados para controlar o fluxo de operações, visto que são responsáveis por permitir a sua realização apenas em caso de validez dos dados em ambos os canais.

Antes de descrever a máquina de estados, é necessário analisar com mais detalhes as entradas, saídas e sinais internos (entradas e saídas dos componentes multiplicador e somador, e acumulador). Isso porque a máquina controla o fluxo de operações por meio dos sinais

internos. Para tanto, os sinais dos componentes internos e os do conjunto são mostrados na figura 4.4. Nesta figura as setinhas cortadas com um número em cima indicam que o sinal corresponde a um vetor e tal número representa a quantidade de elementos do vetor. As setas bidirecionais representam a troca de dados entre os blocos internos e a máquina de estados. Esta é a responsável por passar conjuntos de até 16 das entradas $\boldsymbol{x}$ e $\boldsymbol{w}$ para o componente multiplicador e somador. As operações são controladas por meio dos sinais TVALID que acompanham as entradas de dados.

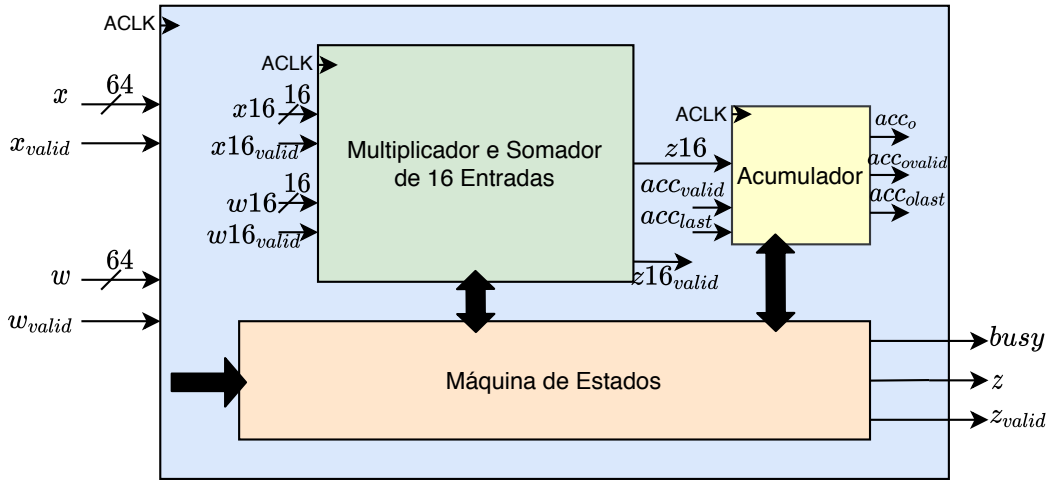

Figura 4.4: Diagrama detalhado da arquitetura particionada

O acumulador tem também o seu fluxo de operações controlado pela máquina de estados. Isso ocorre tanto por meio do TVALID do canal de entrada quanto pelo TLAST, que também é uma entrada. Tais sinais são saídas no canal de saída. O funcionamento e o diagrama do bloco do acumulador foram explicitados na seção 3.7. Ademais, é fundamental o *clock* para o funcionamento do circuito, visto que este é sequencial, de forma que cada operador tem também como entrada o sinal de *clock*, que é igual para todos. A saída z da arquitetura é também acompanhada de um sinal que indica quando o resultado é válido, que será aqui denominado z_{valid}. Há também a saída $busy$ do tipo std_logic, que indica (quando em nível lógico alto) que a máquina está ocupada com o cálculo de z para determinado conjunto de 64 entradas e que se deve aguardar para inserir novos dados.

A partir dessas informações, pode-se descrever o funcionamento da máquina de estados. O seu diagrama pode ser visualizado na figura 4.5. Nela são mostrados os nomes dos estados e também as condições de transição que devem ser obedecidas para a passagem para o próximo estado. Quando não há indicação ao lado da seta que representa a transição, não há nenhuma condição que os sinais devam obedecer para que ela ocorra, de forma que na próxima subida do *clock* ocorrerá a passagem para o estado seguinte.

O início do funcionamento da máquina ocorre no estado denominado "Init", para o qual ela não retorna se não for desligada. Nele são inicializados os sinais internos em 0. Tais sinais são as entradas do multiplicador e somador ($\boldsymbol{x16}$, $x16_{valid}$, $\boldsymbol{w16}$ e $w16_{valid}$) e as entradas do acumulador ($acc = z16$, acc_{valid} e acc_{last}). Além disso, as saídas da arquitetura

particionada também são setadas em 0 ($busy$, z e z_{valid}). Não há condição para a transição, então na subida de *clock* seguinte, já ocorre a passagem para o estado seguinte, denominado "St0 Idle".

O estado "St0 Idle"é aquele em que a máquina permanece até que o circuito receba entradas válidas e para o qual retorna no final da computação de z. Desse modo, é necessário zerar $x16_{valid}$, $w16_{valid}$, z_{valid} e $busy$ no final de cada ciclo. A transição para o próximo estado ocorre se $x_{valid} = 1$ e $w_{valid} = 1$. Começa, então, o cálculo das multiplicações e somas particionadas. Como apenas 16 elementos são computados por vez, são necessárias 4 fases de multiplicações, somas e acumulações. Em cada uma dessas fases há 3 estados distintos. No primeiro, chamado de "Load", os dados são carregados para o componente multiplicador e somador; no segundo, chamado de "Comp"(abreviação de *compute*), é calculado o resultado parcial $z16$; e, no terceiro, chamado de "Save", o resultado parcial é acumulado.

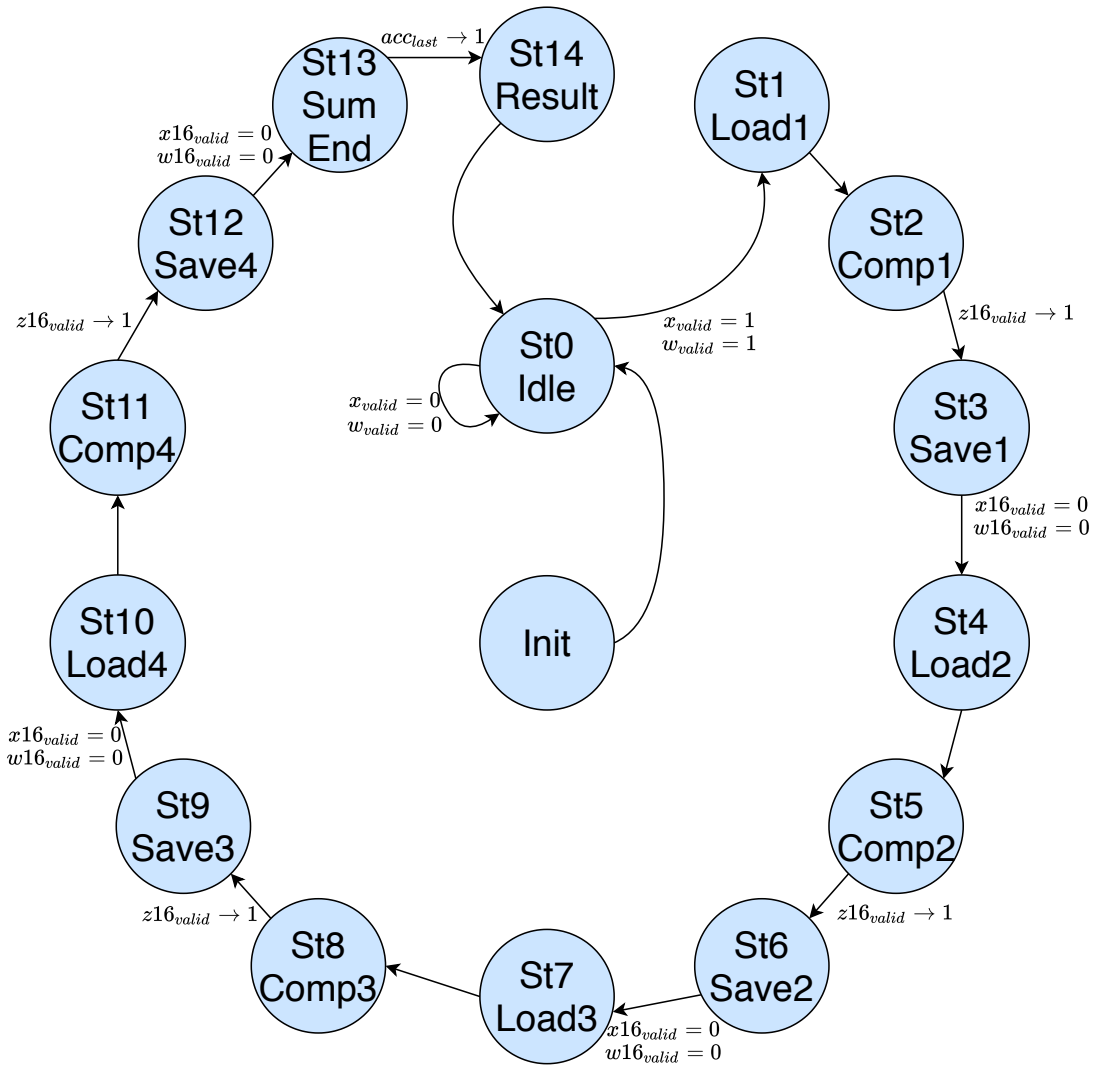

Figura 4.5: Diagrama da máquina de estados da segunda arquitetura

Dessa forma, após o "Idle", há o estado "St1 Load1". Nele a saída $busy$ é alterada para 1 para indicar que a máquina está ocupada e que não devem ser alteradas as entradas do sistema. Além disso, são colocados os primeiros 16 elementos de $\boldsymbol{x}$ e $\boldsymbol{w}$ em $\boldsymbol{x16}$ e $\boldsymbol{w16}$,

respectivamente, ou seja, são carregadas as entradas de dados do multiplicador e somador. Não há condição de transição, então na borda de subida seguinte do *clock*, passa-se para o estado "St2 Comp1". Neste estado, as entradas $x16_{valid}$ e $w16_{valid}$ são colocadas em 1 para dar início aos cálculos.

A transição para o estado "St3 Save1" ocorre apenas quando há uma alteração (evento) em $z16_{valid}$ e seu valor torna-se igual a 1, indicando que o resultado $z16$ está válido. A fase 1 é finalizada no estado seguinte, o "St3 Save1", em que o resultado parcial $z16$ é transferido para a entrada do acumulador e acc_{valid} é colocado em 1. Além disso, faz-se $x16_{valid} = 0$ e $w16_{valid} = 0$ para que não apareça um resultado incorreto na saída quando houver a alteração das entradas. Para que a transição para o próximo estado ocorra, $x16_{valid}$ e $w16_{valid}$ devem ser iguais a 0. A transição ocorre, portanto, no próximo ciclo de *clock*.

O estado seguinte é o St4 Load2, em que é iniciada a fase 2. Nela, há os estados análogos aos da fase 1, mas com as entradas $\boldsymbol{x16}$ e $\boldsymbol{w16}$ do multiplicador e somador configuradas como o segundo bloco de 16 entradas da arquitetura. As fases 3 e 4 ocorrem também de maneira análoga, mas com os terceiro e quarto blocos de 16 entradas, respectivamente. Na última fase, para a base de dados "Sonar" foi necessário atribuir valor nulo para as três últimas entradas.

O final da fase 4 acontece no estado "St12 Save4". Este é um pouco diferente dos demais estados "Save" porque acc_{last} é colocada em nível lógico alto para indicar que é a última soma parcial. O estado seguinte é o "St13 Sum End". Neste as entradas acc_{valid} e acc_{last} são colocadas em 0 e a transição para o estado seguinte ocorre quando há um evento, ou seja, uma alteração no valor de acc_{olast} e quando esta saída do acumulador é igual a 1. Ela indica que o resultado em acc_o é o final. Por fim, há o estado "St14 Result", em que o resultado da acumulação é passado para a saída do sistema, ou seja, tem-se $z = acc_o$. Ademais, z_{valid} é colocado em 1. Não há condição para a transição, então na seguinte subida do *clock* a máquina retorna ao estado "Idle"e o ciclo pode ser recomeçado quando novos valores forem atribuídos a $\boldsymbol{x}$ e $\boldsymbol{w}$.

4.3 *Testbenches*

As simulações em VHDL são feitas por meio da elaboração de códigos denominados *testbenches*. Estes são utilizados apenas para simulação, nunca para síntese. Isso possibilita a utilização de recursos mais robustos do VHDL sem afetar a implementação do circuito e permite testá-lo com mais eficiência.

O código do perceptron foi implementado em Python para que pudessem ser gerados os valores de teste das entradas e dos pesos, os quais foram obtidos durante o aprendizado. Isso para que os aceleradores pudessem ser testados com os dados de uma aplicação real. Os valores a serem simulados foram, então, inicialmente retirados da base de dados "Sonar"; da geração de valores aleatórios para os pesos, que corresponderiam aos da etapa de inicializa-

ção do algoritmo do perceptron; e dos pesos obtidos durante o aprendizado a cada iteração. Foram então calculados os valores z correspondentes utilizando operações em *float32*. Em seguida, tanto as entradas quanto o valor esperado de z foram salvos em um arquivo do tipo *.txt*.

As simulações de ambas arquiteturas descritas na seção anterior foram feitas seguindo a mesma estrutura. Inicialmente os sinais são inicializados em 0. Abre-se então o arquivo que contém os valores das entradas e dos pesos e é iniciada a leitura. Para tanto, é necessário adicionar o pacote $std.textio$. São então atribuídos os valores de cada um dos elementos dos vetores $\boldsymbol{x}$ e $\boldsymbol{w}$. Em seguida, faz-se $x_{valid} = 1$ e $w_{valid} = 1$ durante um ciclo de *clock*. Após este tempo, os sinais que indicam a validez das entradas voltam para 0 e são lidos os próximos valores até que o final do arquivo seja atingido.

É importante mencionar que os valores foram salvos nos arquivos em $float32$. No entanto, os elementos das entradas são do tipo `std_logic_vector(31 downto 0)`. Foi necessário, portanto, converter os dados para que o circuito pudesse ser simulado. Para tanto, utilizou-se a biblioteca VHDL-2008 Support Library, desenvolvida para tornar diversos *packages* compatíveis com o VHDL-93. Os arquivos desta biblioteca estão disponíveis no *github* e é necessário adicionar ao projeto do Vivado os arquivos "fixed_float_types_c.vhdl", "fixed_pkg_c.vhdl"e "float_pkg_c.vhdl"ao projeto na biblioteca "ieee_proposed". Estes arquivos constam na pasta "xilinx_11"do repositório indicado. Após acrescentar os arquivos ao projeto, adiciona-se a biblioteca $ieee\_proposed$ ao início do código da simulação e também o *package* $float\_pkg$. Além disso, é necessário configurar o *testbench* para o tipo VHDL 2008 em *Source File Properties*.

Capítulo 5

Resultados

Neste capítulo serão apresentados os resultados obtidos neste trabalho. Tais resultados foram encontrados a partir das simulações das arquiteturas descritas no capítulo 4. Fazem parte dos resultados os gráficos que mostram como são as saídas do circuito e os tempos necessários para que os cálculos sejam realizados. Além disso, também serão exploradas as quantidades de recursos da FPGA utilizadas para cada arquitetura, que foram obtidas através da síntese de cada *design*.

As arquiteturas mostradas no capítulo 4 serão neste capítulo denominadas *design* 1 e *design* 2, sendo o primeiro a implementação que calcula z de forma direta e, o segundo, a que calcula z particionando as entradas e utilizando uma máquina de estados de Moore.

5.1 Simulações

As simulações dos *designs* foram realizadas com o *clock* em 100 MHz, que é o valor da frequência do oscilador da Basys 3, tomada neste trabalho como referência. Os IPs somador, multiplicador e acumulador foram configurados para 1 ciclo/operação. Além disso, foi configurada a latência de 1 a 4 *clocks* e uso médio de DSPs, sendo 1 para cada somador e multiplicador, e, no caso do *design* 2, 7 para o acumulador. Em outro cenário de simulação, os operadores também foram configurados com uso completo de DSPs, que corresponde a 2 para os somadores e multiplicadores; e a 10 para o acumulador.

O comportamento dos sinais de entradas e saídas do *design* 1 é mostrado na figura 5.1. Foi feito um gráfico em forma de diagrama porque não seria possível mostrar os gráficos das simulações no espaço disponível na mesma escala de tempo. Na figura, as linhas pontilhadas verticais indicam uma diferença de tempo de 1 período de *clock*, com exceção do intervalo da primeira para a segunda, cujo tempo está indicado e corresponde a meio período; e também daquele indicado por três pontos. Estes mostram que todos os sinais foram mantidos na configuração indicada até o instante $285ns$, no qual o primeiro valor calculado aparece na saída z.

É importante mencionar que foi utilizada latência igual a 4 porque foi o valor mínimo para o qual não houve espúrios nas transições dos valores da saída z. Com a latência igual a 1 o tempo de cálculo seria igual a $75,1ns$, mas cuidados maiores deveriam ser tomados durante a verificação dos valores, visto que o tempo de duração dos espúrios após as transições foi de $0,1ns$ e a saída z_{valid} permanecia em 1, assim como no caso correto mostrado no gráfico. Além disso, os valores poderiam sofrer alterações ainda maiores na implementação física, visto que a simulação mostra o cenário ideal. Diz-se que a duração dos espúrios (valores incorretos) foi menor que a de uma batida de *clock* porque foi este o tempo observado entre a transição do valor de saída para um resultado incorreto e a transição para o valor correto, de forma análoga a um atraso.

As entradas do *design* 1 são os vetores $\boldsymbol{x}$ e $\boldsymbol{w}$, sendo cada um composto por 64 elementos de 32 *bits*; e o x_{valid} e o w_{valid}, com cada um desses sinais formado por apenas um *bit*. Já as saídas são z, formado por 32 *bits* e z_{valid}, com apenas 1 *bit*. No diagramas, os valores das entradas formadas por mais de 1 *bits* foram indicados de forma genérica. Dessa forma, as entradas $\boldsymbol{x}$, $\boldsymbol{w}$ e z são indicadas por A_j, B_j e C_j, respectivamente, com $j \in \mathbb{N}^*$ e $1 \leq j \leq 8$.

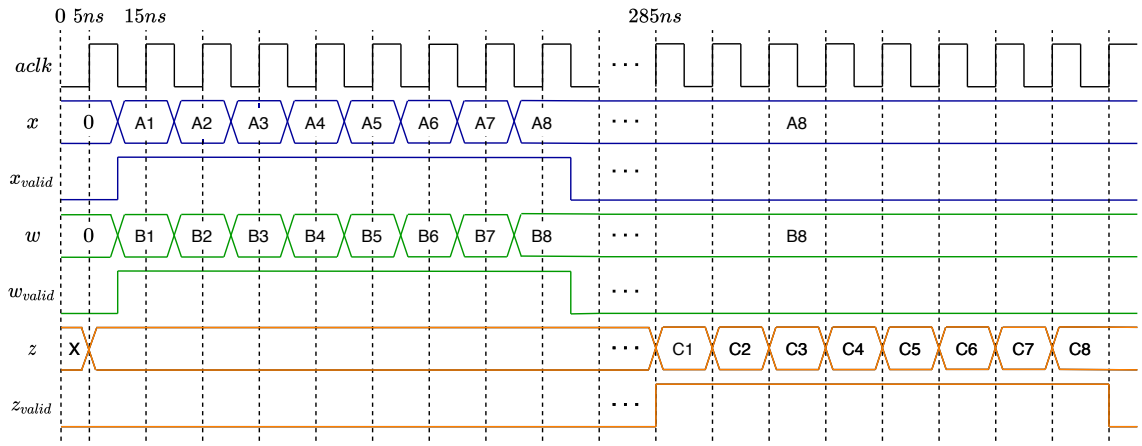

Figura 5.1: Gráfico que mostra o comportamento do *design* 1

Na simulação, as entradas de dados foram mantidas durante um ciclo de *clock*, estando estas estáveis antes da sua borda de subida. Além disso, os sinais que indicam a validade dos dados foram setados em 1 durante o intervalo de tempo correspondente. Tais sinais voltaram para 0 após um período de validez da última alteração dos dados a serem calculados. Verificou-se que o tempo de cálculo total para a computação de z é de $270ns$, ou 27 ciclos de *clock* e que após o primeiro valor aparecer na saída, a estrutura em *pipeline* permitiu que os resultados dos cálculos seguintes aparecessem a cada borda de subida do *clock*. O tempo necessário para a computação da saída foi calculado subtraindo-se do instante de tempo em que a primeira saída é obtida daquele em que a primeira entrada está válida quando o *clock* sobe para 1.

Por sua vez, o *design* 2 tem o seu gráfico ilustrado na figura 5.2. Nela, seguiram-se os mesmos padrões de notação descritos para o *design* 1 para a representação das entradas e das saídas com múltiplos *bits* e para a representação de um intervalo de tempo maior que o

da escala com a utilização dos três pontos. No entanto, há neste caso uma saída a mais, a denominada $busy$. Como explicado no capítulo 4, tal saída indica que a máquina de estados encontra-se ocupada com os cálculos de z e que as entradas não podem ser modificadas para que o resultado não seja prejudicado.

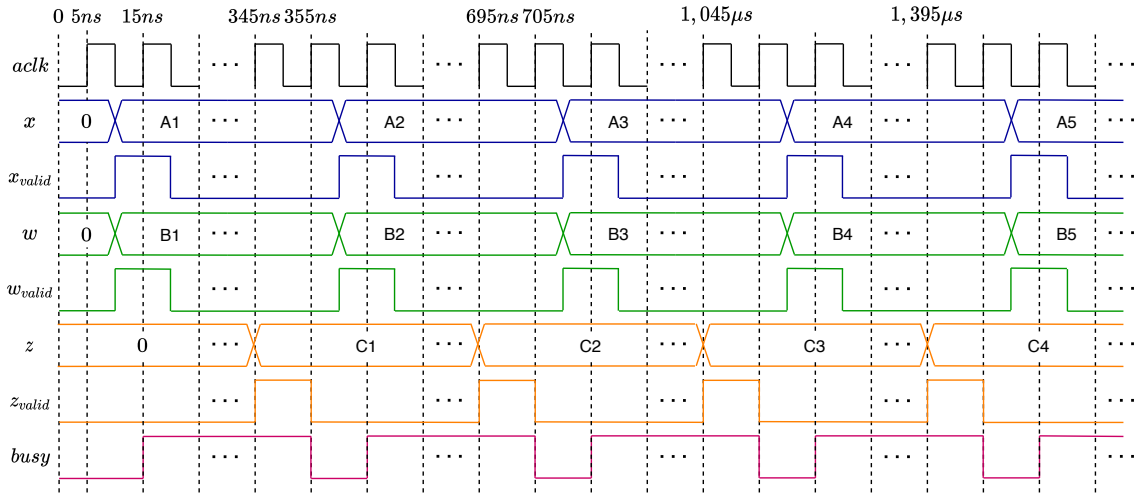

Figura 5.2: Gráfico que mostra o comportamento do *design* 2

Neste novo cenário, verifica-se a maior diferença entre os dois casos: a ausência do *pipeline* no segundo *design*. Ademais, o tempo total de cálculo da saída z é maior do que o do *design* anterior, mesmo que tenha sido utilizada a menor latência, de 1 ciclo de *clock*. O tempo de cálculo obtido foi igual a $330ns$, ou de 33 ciclos. Um aspecto positivo deste *design* foi que mesmo para a latência mínima, não houve valores espúrios na saída. Como não há o *pipeline*, deve-se aguardar $330ns$ para cada saída z ser calculada, mesmo após a primeira. Isso resulta em um grande aumento do tempo total de operações quando é necessário obter diversos valores z.

5.2 Síntese

O processo de síntese permite que seja verificada a utilização dos recursos disponíveis na FPGA. Tal processo foi realizado para ambos os *designs* apresentados. Apesar de o tempo de cálculo do primeiro circuito ser menor do que o do segundo, aquele exige muito mais recursos do que este. A ocupação em termos de utilização de LUT, LUTRAM, FF e DSP com a configuração de uso médio de DSPs pelos IPs consta na tabela 5.1.

Tabela 5.1: Utilização de recursos pelo *design* 1

	Utilização	Disponíveis	Utilização (%)
LUT	38075	20800	183,05
LUTRAM	–	9600	–
FF	17145	41600	41,21
DSP	127	90	141,11

É possível perceber que foram utilizados mais recursos do que aqueles disponíveis na Basys 3, o que impediria a implementação física do circuito na FPGA. A alteração da configuração de uso de DSPs para nenhum ou para o uso máximo não impede que sejam ultrapassados os limites disponíveis, visto que com o uso médio já foram ultrapassadas as quantidades limites tanto de LUTs quanto de DSPs.

Já o *design* 2 resolve este problema, pois a quantidade demandada de recursos é consideravelmente menor. Isso pode ser facilmente compreendido pelo cálculo em etapas da saída z, sendo computadas apenas 0.25 das entradas de dados em cada partição. Tal utilização dos recursos, considerando o uso médio de DSPs, pode ser verificada na tabela 5.2. Observa-se que além de reduzidos, os recursos foram melhor distribuídos dentre as categorias existentes.

Tabela 5.2: Utilização de recursos pelo *design* 2 com uso médio de DSPs

	Utilização	Disponíveis	Utilização (%)
LUT	13743	20800	$47,72$
LUTRAM	659	9600	$6,86$
FF	8390	41600	$20,17$
DSP	38	90	$42,22$

Também foi feita a síntese do segundo *design* considerando o uso completo de DSPs. A utilização de recursos para esse cenário consta na tabela 5.3. Verifica-se que também é possível a sua implementação, mas o funcionamento do circuito não é alterado. Ou seja, os tempos de obtenção das saídas permanecem os mesmos para as diferentes configurações de uso de DSPs.

Tabela 5.3: Utilização de recursos pelo *design* 2 com uso completo de DSPs

	Utilização	Disponíveis	Utilização (%)
LUT	9891	20800	$47,55$
LUTRAM	–	9600	–
FF	5343	41600	$12,84$
DSP	72	90	$80,00$

Ademais, na tabela 5.4, consta a comparação das arquiteturas desenvolvidas (com uso médio de DSPs) com a DLAU, apresentada no capítulo 2. É importante recordar que a arquitetura TMMU, que é parte da DLAU, é responsável pelas multiplicações e somas. Com intuito de diminuir a utilização de recursos, os cálculos na DLAU também são feitos de forma particionada. Daí a inspiração para o *design* 2. Percebe-se que são realizadas na TMMU as mesmas funções dos circuitos desenvolvidos neste trabalho. Foi dito que a implementação seria de um circuito dedicado ao treinamento do perceptron. No entanto, como mostrado no capítulo 3, este forma as redes MLP e o seu treinamento é composto basicamente por multiplicações matriciais.

O cálculo de cada novo elemento da matriz resultante do produto, é o mesmo cálculo de z para o perceptron, sendo necessárias mais repetições caso a matriz tenha mais de 64 entradas.

Tabela 5.4: Comparação de recursos utilizados entre arquiteturas semelhantes

	TMMU	*Design* 1	*Design* 2
LUT	32461	38075	13743
LUTRAM	–	–	659
FF	25356	17145	8390
DSP	158	127	38
BRAM	32	–	–

Dessa forma, o *design* 2 deste trabalho poderia ser utilizado também para o treinamento do MLP em trabalhos futuros. Entretanto, para que ele fosse mais eficiente, seria necessário utilizar um *buffer* que fosse capaz de armazenar os dados recebidos de forma a possibilitar a alteração das entradas da máquina de estados sem interferir no cálculo de z.

A FPGA utilizada pela DLAU, a XC7Z020 da Xilinx, tem mais recursos disponíveis do que a Basys 3 e verificou-se que seria possível a implementação também do *design* 1 em tal FPGA. Além disso, a primeira arquitetura desenvolvida neste trabalho é análoga à DLAU, com a diferença de que esta faz a partição dos cálculos a cada 32 entradas. A comparação da utilização de recursos mostra que são parecidas as ocupações do primeiro e do TMMU, com apenas as LUTs em maior quantidade. Sendo sua partição maior, o tempo de cálculo de z e também dos elementos das matrizes é consequentemente menor, o que se configura como um resultado promissor.

5.3 Comparação entre Tempos de Execução

Foi feita também a comparação entre os tempos de execução do cálculo de z através de um *script* em Python e através dos *designs* desenvolvidos neste trabalho. O *script* em Python cujo tempo de execução do cálculo de z foi medido é o mesmo utilizado para obtenção dos valores de teste dos pesos, que foi mencionado na seção 4.3.

O tempo de execução da parte do algoritmo do perceptron que calcula z foi obtido para o *script* por meio da utilização da função *timer* da biblioteca *timeit*. Tal *script* foi executado em um PC com o *clock* do processador a 5 GHz. O tempo foi salvo para cada execução da computação de z realizada durante o aprendizado. Posteriormente, foi feita a média aritmética desses tempos para obtenção da estimativa do seu tempo de execução. Desse modo, obteve-se uma média de $1,23\mu s$ para o tempo de execução em Python.

Como visto na seção 5.1, o tempo de execução da computação de z para o *design* 1 é de $270ns$ e, para o *design* 2, de $330ns$. Dessa forma, o tempo de cálculo por meio do script é aproximadamente $4,6$ vezes maior que aquele por meio do *design* 1 e, cerca de 4 vezes maior que o do *design* 2.

No entanto, é necessário levar em consideração que o *clock* do processador é 50 vezes mais rápido que o *clock* da FPGA tomada como referência, a Basys 3. O *design* 2, que apre-

sentou melhor compromisso de utilização de recursos *versus* tempo de cálculo, é, portanto, aproximadamente 4 vezes mais rápido e 50 vezes mais eficiente que um *script* em Python.

Capítulo 6

Conclusão

Neste trabalho foram apresentadas duas arquiteturas dedicadas ao cálculo da entrada da função de ativação do perceptron. Também podem ser compreendidas como parte da arquitetura de uma rede do tipo perceptron multicamadas, visto que realizam cálculos de forma particionada dos elementos de uma matriz resultante do produto de outras duas. Essas operações, como apresentado no estado da arte e também na fundamentação teórica, são as principais em algoritmos de redes neurais artificiais. Os *designs* de *hardware* implementados mostraram resultados satisfatórios no quesito tempo de computação e recursos utilizados quando estes foram comparados com a arquitetura estado da arte e quando foram consideradas placas mais robustas que a Basys 3.

Foram apresentadas duas soluções com otimizações diferentes. A primeira, apelidada de design 1, tem como foco desempenho e consegue entregar o resultado de um elemento da matriz resultante da multiplicação matricial em apenas $270ns$. Já a segunda, apelidada de *design* 2, resultou em um compromisso entre performance e ocupação da FPGA, ocupando em média 40% a 50% da FPGA da placa Basys 3. O resultado do design 2 pode ser computado em $330ns$. Esse resultado apresenta uma penalidade de apenas 20% com relação ao design 1, porém ocupando $\frac{1}{4}$ do espaço. Considera-se, então, um bom compromisso de *design*. Além disso, o *design* 2 é cerca de 4 vezes mais rápido e 50 vezes mais eficiente que um *script* em Python.

No entanto, são necessárias diversas melhorias para o funcionamento da implementação do circuito em uma FPGA propriamente dita. Assim, nem todos os objetivos apresentados no capítulo 1 foram cumpridos. Podem ser propostos, portanto, trabalhos futuros para que tais objetivos sejam alcançados. O primeiro seria a utilização de um protocolo de comunicação serial para que os dados possam ser enviados à FPGA, visto que não há pinos de entrada suficientes para que todos os dados sejam admitidos em paralelo; a utilização de *buffers* que permitam que as entradas sejam acessadas sem que a máquina de estados precise ser desocupada, de forma a permitir o funcionamento do *pipeline* para ambas arquiteturas; implementação da acumulação dos resultados obtidos para permitir o cálculo da multiplicação matricial entre matrizes cujas linhas e colunas possam ser maiores que 64; implementação

de um bloco responsável pelo cálculo da função de transferência; e utilização da memória da FPGA para armazenar os novos pesos calculados.

Referências Bibliográficas

[1] BEAL, V. *The Five Generations of Computers*. Acesso em: 6 Dez 2019. Disponível em: <https://www.webopedia.com/DidYouKnow/Hardware_Software/FiveGenerations.asp>.

[2] COPELAND, M. *The Difference Between AI, Machine Learning, and Deep Learning?: NVIDIA Blog*. Nov 2019. Acesso em: 6 Dez 2019. Disponível em: <https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/>.

[3] LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *nature*, Nature Publishing Group, v. 521, n. 7553, p. 436–444, 2015.

[4] LE, Q. V. Building high-level features using large scale unsupervised learning. In: IEEE. *2013 IEEE international conference on acoustics, speech and signal processing*. [S.l.], 2013. p. 8595–8598.

[5] NAUGHTON, J. *Can the planet really afford the exorbitant power demands of machine learning? | John Naughton*. Guardian News and Media, Nov 2019. Acesso em: 6 Dez 2019. Disponível em: <https://www.theguardian.com/commentisfree/2019/nov/16/can-planet-afford-exorbitant-power-demands-of-machine-learning>.

[6] WANG, C. et al. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 36, n. 3, p. 513–517, 2016.

[7] PRADO, A. C. et al. *ASIC, ASSP, SoC, FPGA - Entenda as diferenças - Embarcados - Sua fonte de informações sobre Sistemas Embarcados*. Jan 2018. Acesso em: 11 Dez 2019. Disponível em: <https://www.embarcados.com.br/asic-assp-soc-fpga/>.

[8] CHEN, T. et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ACM. *ACM Sigplan Notices*. [S.l.], 2014. v. 49, n. 4, p. 269–284.

[9] JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In: IEEE. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. [S.l.], 2017. p. 1–12.

[10] DING, W. et al. Designing efficient accelerator of depthwise separable convolutional neural network on fpga. *Journal of Systems Architecture*, Elsevier, v. 97, p. 278–286, 2019.

[11] MA, Y. et al. Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration*, Elsevier, v. 62, p. 14–23, 2018.

[12] MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

[13] HEBB, D. O. *The organization of behavior: A neuropsychological theory*. [S.l.]: Psychology Press, 2005.

[14] CSULB. *History of the Perceptron*. Acesso em: 29 Nov 2020. Disponível em: <https://web.csulb.edu/~cwallis/artificialn/History.htm>.

[15] RASCHKA, S.; MIRJALILI, V. *Python machine learning*. [S.l.]: Packt Publishing Ltd, 2017.

[16] HAYKIN, S. *Neural networks: a comprehensive foundation*. [S.l.]: Prentice-Hall, Inc., 2007.

[17] WAKERLY, J. F. *Digital Design: Principles And Practices, 4/E*. [S.l.]: Pearson Education India, 2008.

[18] SUNDARARAJAN, P. *High performance computing using FPGAs*. [S.l.], 2010.

[19] THORNTON, S. *Fixed point vs Floating point*. Sep 2017. Acesso em: 6 Dez 2020. Disponível em: <https://www.microcontrollertips.com/difference-between-fixed-and-floating-point/>.

[20] MOUSSA, M.; AREIBI, S.; NICHOLS, K. On the arithmetic precision for implementing back-propagation networks on fpga: a case study. In: *FPGA Implementations of Neural Networks*. [S.l.]: Springer, 2006. p. 37–61.

[21] PG060, X. *Floating-Point Operator V7. 1 LogicCore IP Product Guide*. 2017.

[22] XILINX. *Finite State Machines*. 2015. Acesso em: 6 Dez 2020. Disponível em: <https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/lab10.pdf>.

[23] UNKNOWN. *Sonar, Mines vs. Rocks*. Acesso em: 1 Set 2020. Disponível em: <https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/>.