Pós-Graduação em Ciência da Computação

João Gabriel Machado da Silva

Projeto e Desenvolvimento de uma Arquitetura para mapeamento de Redes Neurais Artificiais em FPGAs

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/ posgraduacao

Recife

2019

João Gabriel Machado da Silva

Projeto e Desenvolvimento de uma Arquitetura para mapeamento de Redes Neurais Artificiais em FPGAs

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Sistemas Embarcados

Orientador(a): Edna Natividade da Silva Barros

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S586p Silva, João Gabriel Machado da
Projeto e desenvolvimento de uma arquitetura para mapeamento de redes neurais artificiais em FPGAs / João Gabriel Machado da Silva. – 2019.
92 f.: il., fig., tab.

Orientadora: Edna Natividade da Silva Barros.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências.

1. Inteligência artificial. 2. Redes neurais artificias. 3. Aprendizagem profunda. I. Barros, Edna Natividade da Silva (orientadora). II. Título.

006.3 CDD (23. ed.) UFPE- MEI 2019-109

João Gabriel Machado da Silva

"Projeto e Desenvolvimento de uma Arquitetura para mapeamento de Redes Neurais Artificiais em FPGAs"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 15/03/2019.

BANCA EXAMINADORA

Prof. Dr. Manoel Eusébio de Lima
Centro de Informática/UFPE

Prof. Dr. Elmar Uwe Kurt Melcher
Centro de Engenharia Elétrica e Informática/UFCG

Profa. Dra. Edna Natividade da Silva Barros
Centro de Informática/UFPE
(Orientadora)

Dedico esta dissertação a minha mãe, minha esposa e aos meus amigos que foram porto seguro perante as dificuldades durante este percurso.

AGRADECIMENTOS

Agradeço a Deus, porque d'Ele e para Ele são todas as coisas. Agradeço à minha linda e amada esposa Marcelle Machado por toda sua dedicação, por sonhar comigo os meus sonhos que acabam se tornando nossos sonhos e também por todo suporte e amor que foram a mim dedicados, durante todo o meu desenvolvimento acadêmico.

Agradeço também a minha "Mainha"Helena por sua contribuição em minha formação pessoal, acadêmica e por toda educação que me foi dada. Agradeço a minha família, em especial as minhas tias, pelo apoio e por me dá suporte.

Agradeço a professora Edna Barros por sempre acreditar em meu trabalho, por me orientar e por me dar a oportunidade de desenvolver meus conhecimentos. Agradeço também por servir como um grande modelo profissional, por todos os ensinamentos que me transmitiu durante esses anos de convivência e por ter me dado à oportunidade de participar de grandes desafios ao seu lado.

Quero agradecer especialmente ao meu amigo e chefe João Paulo Fernandes, por toda ajuda e apoio no desenvolvimento deste trabalho, sem seu apoio e motivação em determinados momento a construção deste trabalho seria bem mais árdua. Como também os amigos de trabalho Henrique Figueirôa e Rafael Nunes pela motivação e apoio sempre que precisei.

Quero agradecer ao meu amigo Antonyus Pyetro por toda ajuda e apoio no desenvolvimento deste trabalho e no meu desenvolvimento acadêmico, especialmente pela confiança e acreditar na minha capacidade. Agradeço a meu colega de trabalho e amigo Heitor Rapela, que me ajudou diretamente no desenvolvimento deste trabalho.

Gostaria de agradecer ao meu amigo Jefferson Ramos, que me acompanha dês do início da minha jornada acadêmica. Aos meus grandes amigos Guilherme Praciano e Djeefther Souza por compartilharem comigo minhas angustias e frustrações, mas sempre incentivando e enaltecendo minha capacidade.

Quero agradecer a todos que fazem parte da família RobôCIn, que sempre me acolheram e apoiaram durante o desenvolvimento desta dissertação: Lucas Cavalcanti, Roberto Fernandes, Juliana Damurie, Renato Souza, Cristiano Santos, Camila Oliveria, Cecília Virgínia, Raphael Brito, Felipe Martins, Victor Sabino, Walber Rodrigues, Mateus Gonçalves e tantos outros. #GoRoboCIn

Agradeço aos amigos do LINCS/CETENE que de alguma forma contribuíram para que fosse possível a conclusão desta dissertação: Lucas Cambuim, Vanessa Ogg, Pedro Ishimaru, Igor Barbosa, Severino José, Vanine Sabino, Reniê Delgado, Mavson Allan, Diogo Izidio, Jorge Dias, Rogério Rosa.

Por fim, agradeço aos professores e funcionário do Centro de Informática que contribuíram com toda minha formação acadêmica.

"Eu acredito, que às vezes são as pessoas que ninguém espera nada que fazem as coisas que ninguém consegue imaginar."(HODGES, 1983).

RESUMO

As técnicas de aprendizagem profunda tiveram um grande impacto em áreas como reconhecimento de padrões, robótica, visão computacional, processamento de linguagem natural, entre outras, tornando-se o estado da arte para diversos problemas nessas áreas. Esta grande quantidade de aplicações estão sendo implementadas em equipamentos portáteis o que leva à necessidade de soluções que tenham bom desempenho e baixo consumo de energia. Este trabalho apresenta um fluxo de desenvolvimento de Redes Neurais Artificiais (RNAs) Perceptron Multicamadas (MLP) Profundas, baseado em FPGAs, visando o aumento de desempenho em aplicações como aquelas descritas acima. Para isto foi definida uma arquitetura para uma RNA-MLP profunda configurável e de fácil instanciação, com geração automática dos módulos de processamento. A arquitetura foi implementada na linguagem de hardware SystemVerilog e permite o processamento de aplicações de alto desempenho através de um pipeline eficiente. Adicionalmente, a solução proposta foi prototipada em uma plataforma que suporta a aceleração da RNA-MLP em FPGA integrada com aplicações de aprendizagem. Para isso foi preciso atender aos requisitos de tempo, grande quantidade de entradas, quantidade de blocos lógicos, utilização de pinos. A arquitetura proposta foi comparada com algumas arquiteturas propostas na literatura de forma a mostrar seu diferencial. Comparações de qualidade e tempo de execução com outros trabalhos foram feitas usando conjuntos de dados de imagens como MNIST, Fashion-MNIST comumente usadas na literatura para comparações em aprendizagem profunda. O tratamento de consumo de energia não é abordado neste trabalho. Além disso, o desempenho da solução prototipada em FPGA foi comparado com uma solução em GPU utilizando Keras, uma biblioteca de rede neural de código aberto escrita em Python, e alcançou uma aceleração média de 59x no subconjunto de testes do conjunto de dados MNIST e Fashion-MNIST.

Palavras-chaves: Aprendizagem Profunda. Perceptron Multicamadas. Redes Neurais Artificiais. Inteligência Artificial. FPGA.

ABSTRACT

Deep learning techniques imposed a significant impact in areas such as pattern recognition, robotics, computer vision, natural language processing, among others, making it state of the art for various problems in these areas. Besides, this large number of applications are being implemented in portable equipment which leads to the need for solutions that have good performance and low power consumption. This work presents a flow of development of Artificial Neural Networks (ANNs) Multilayer Perceptron (MLP) Deep, based on FPGAs, aiming the increase of performance in applications like those described above. For this, we are proposing architecture for a deep configurable ANN-MLP with easy instantiation with the automatic generation of the processing modules was defined. The architecture was implemented in the SystemVerilog hardware language and allowed the processing of high-performance applications through an efficient pipeline. In addition, the proposed solution was prototyped in a platform that supports the acceleration of an ANN-MLP in FPGAs in an integrated way with learning applications running in software. For this, it was necessary to meet the requirements of time, a large number of inputs, a number of logical blocks, and use of pins. A comparison between the proposed architecture and some architectures proposed in the literature showed the differential of our design. Quality and time-to-execution comparisons with other works were done using public data sets of images such as MNIST, Fashion-MNIST commonly used in the research for comparisons in deep learning. The treatment of energy consumption is not approached in this work. Also, the performance of the solution prototyped in FPGA was compared to a GPU solution using Keras, a Python open source neural network library, and achieved a mean acceleration of 59.2x in the test using subsets of the MNIST and Fashion-MNIST datasets.

Keywords: Deep Learning. Multilayer Perceptron. Artificial Neural Networks. Artificial Intelligence. FPGA.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ALB	Array of Logical Blocks
AM	Aprendizagem de Máquina
ASIC	Circuito Integrado de Aplicação Específico
BRAM	Blocos de Memória RAM
CI	Circuito Integrado
CLB	Configurable Logical Block
CPU	Unidade Central de Processamento
FPGA	Field Programmable Gate Array
GPU	Unidade de Processamento Gráfico
IA	Inteligência Artificial
IEEE	Instituto dos Engenheiros Elétricos e Eletrônicos
LUT	Look Up Table
MLP	Perceptron Multicamadas
MNIST	Modified National Institute of Standards and Technology
NIST	National Institute of Standards and Technology
RIFFA	A Reusable Integration Framework For FPGA Accelerators
RNA	Rede Neural Artificial
RNA-MLP	Rede Neural Artificial Perceptron Multicamadas
SIMD	Single instruction, Multiple Data

SUMÁRIO

1 INTRODUÇÃO

A Aprendizagem de Máquina (AM) é um método da análise de dados em Inteligência Artificial (IA) que automatiza a construção de modelos analíticos, baseado na ideia de que sistemas podem aprender com dados, identificar padrões e tomar decisões com o mínimo de intervenção humana (HOSCH, 2016).

Aprendizagem de Máquina desempenha um papel fundamental na solução de muitos problemas científicos importantes. Tem sido aplicado em domínios tão variados quanto sistemas de processamento de sinais (YU; DENG, 2011), análise de dados médicos (LITJENS et al., 2017; XU et al., 2014), previsão de séries temporais (KUREMOTO et al., 2014; QIU et al., 2014), robótica (YU et al., 2013), e muitos outros. Em algumas dessas aplicações, a quantidade de dados a serem processados é enorme ou o tempo de resposta necessário é muito reduzido, o que pode tornar impraticável que soluções por software sejam implementadas em alguns sistemas embarcados, por assim dizer também em sistemas ciber-físicos integrados (Khaitan; McCalley, 2015).

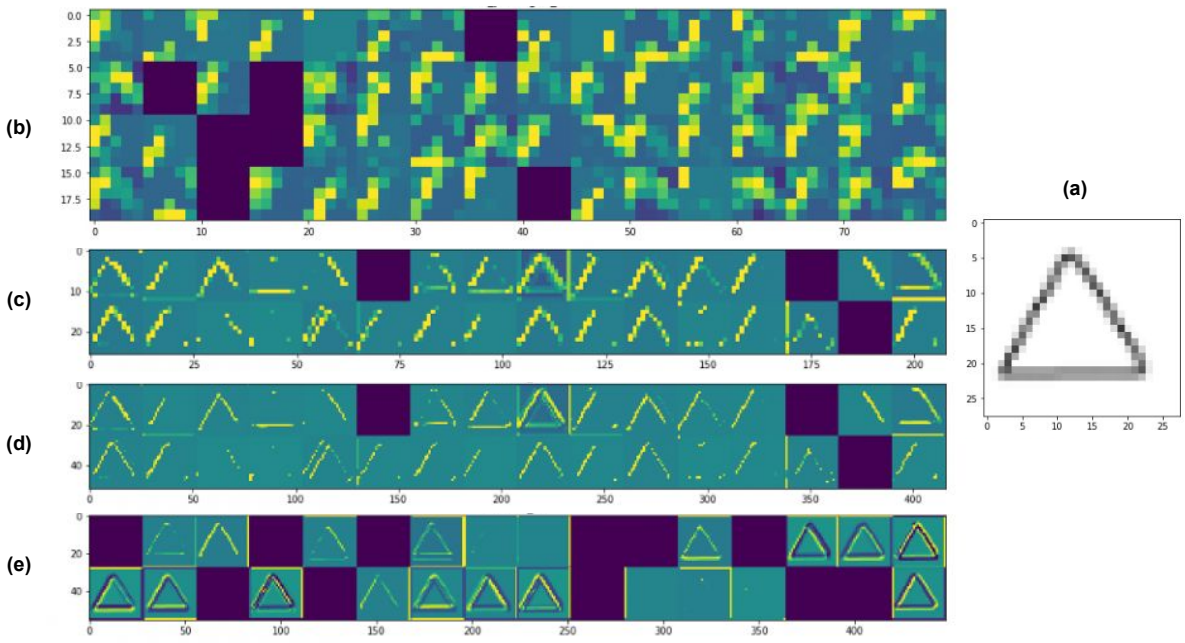
Os sistemas ciber-físicos são sistemas interligados fisicamente, cujas operações são monitoradas, coordenadas, controladas e integradas por um núcleo de computação e comunicação. Assim como a internet transformou o modo como os humanos interagem uns com os outros, os sistemas ciber-físicos irão transformar a forma como interagimos com o mundo físico ao nosso redor (Rajkumar et al., 2010). Avanços recentes na indústria abriram caminho para a implantação de sistemas ciber-físicos, dentro dos quais as informações são monitoradas de perto e sincronizadas entre o espaço físico e o espaço computacional. Além disso, ao utilizar a análise avançada de informações, os dispositivos e maquinários interligados poderão funcionar de forma mais eficiente, colaborativa e resiliente. Essa tendência está transformando a indústria para a próxima geração, ou seja, a Indústria 4.0 (LEE; BAGHERI; KAO, 2015). Desta forma, técnicas de AM podem ser empregadas na análise e aplicação de todas essas informações.

As abordagens convencionais de AM são incapazes de processar dados naturais brutos (como os valores de pixel de uma imagem). Durante décadas, a construção de um sistema de reconhecimento de padrões ou de AM exigia um conhecimento prévio especializado da aplicação para se projetar um extrator de características que transformasse os dados brutos em uma representação interna adequada, geralmente um classificador, podendo detectar ou classificar padrões na entrada(LECUN; BENGIO; HINTON, 2015). Dessa forma, aprendizado por representação consiste na técnica de AM onde, um conjunto de métodos permite que uma máquina seja alimentada com dados brutos e descubra automaticamente as representações necessárias para detecção ou classificação.

Aprendizagem profunda consiste em métodos de aprendizagem por representação com múltiplos níveis de representações internas, obtidas da composição de módulos simples

(funções), mas não lineares, que transformam a representação da informação a cada nível. Com a composição de tais transformações, funções muito complexas podem ser aprendidas para tarefas de classificação. Na Figura 1 a imagem de um triangulo (a) é apresentada como exemplo. A imagem é representada na forma de uma matriz de valores de pixel, e os recursos aprendidos na primeira camada (b) de representação da rede, normalmente representam a presença ou ausência de bordas em locais e orientações específicas da imagem. A segunda camada (c) geralmente detecta sinais de composições das bordas, independentemente de pequenas variações nas posições. A terceira camada (d) pode agrupar tais composições em conjuntos maiores que correspondem a partes de objetos, e camadas subsequentes (e) detectariam objetos como a combinações dessas partes. A característica predominante das técnicas de aprendizagem profunda é que essas camadas não são projetadas a partir de conhecimento prévio, elas são aprendidas a partir de dados usando um procedimento de aprendizado (LECUN; BENGIO; HINTON, 2015).

Figura 1 – Exemplos da representação interna obtidos a partir da imagem de entrada de um triangulo **(a)**, ao longo das camadas **(b,c,d,e)** de uma técnica de aprendizagem profunda



Fonte: Google Imagens

Em uma linha do tempo desde 2012, a pesquisa em aprendizado profundo trouxe novamente as redes neurais para o foco, resultando na revitalização e construção de técnicas existentes e em várias novas aplicações em software. Os resultados destas pesquisas estão impactando significativamente áreas como reconhecimento de fala, visão computacional e processamento de linguagem natural (SCHMIDHUBER, 2015). Uma das técnicas que foram revitalizadas nesse contexto foram as Redes Neurais Artificiais (RNAs).

Redes Neurais Artificiais são modelos bio-inspirados em estruturas observadas no cérebro, que é composto por bilhões de neurônios interconectados em uma vasta rede cerebral. Este modelo herda os recursos paralelos e distribuídos do modelo biológico. As RNAs são empregadas em muitas aplicações relacionadas a AM e aprendizagem profunda, já citadas anteriormente. Algumas propriedades desejadas das RNAs incluem aprendizado com exemplos rotulados (aprendizado supervisionado), adaptabilidade, generalização e tolerância a falhas. No próximo capítulo as RNAs serão melhor discutidas em detalhe.

Uma RNA Perceptron Multicamadas (MLP) é um tipo de rede composta por camadas formadas por neurônios artificiais, estes, totalmente conectados entre uma camada para a outra à frente da rede. Normalmente uma RNA-MLP padrão possuía de uma a três camadas no máximo. Uma Rede Neural Artificial Perceptron Multicamada Profunda, variação que surgiu com a aprendizagem profunda, tipicamente pode incluir de uma a seis camadas, mas nada impede de serem dezenas de camadas (Huynh, 2017). Essa característica permite uma sequência mais profunda de transformações onde cada camada realiza uma inferência.

A grande maioria das técnicas de aprendizagem profunda tiveram êxito devido às aplicações serem executadas em servidores reais, com a capacidade de grande armazenamento e poder de processamento massivo, disponível em várias processadores de aplicação modernos, como as Unidades de Processamento Gráfico (GPU). A transposição de todos esses avanços para sistemas embarcados e ciber-físicos traz uma ampla gama de funcionalidades, mas surge o problema de como lidar com a quantidade massiva de dados em dispositivos com limite de espaço físico, baixa capacidade de processamento e de armazenamento. Outros desafios envolvem a correspondência da representação numérica e a precisão dos resultados.

Para tais sistemas, implementações mais rápidas de técnicas de AM são necessárias para atender às restrições de tempo e tamanho cada vez mais críticas. Nesses casos, oferecer desempenho em métodos incorporados significa que uma nova gama de soluções pode estar disponível para os usuários. Enquanto as RNA-MLP Profundas podem ser facilmente implementadas por software em uma Unidade Central de Processamento (CPU) ou em GPU, implementações em hardware ainda expõem muitos desafios, como implementar o cálculo do estado de ativação, junto com a função de ativação que compõe a equação de computação de um impulso nervoso do neurônio artificial.

Uma RNA-MLP Profunda implementada em hardware permite o uso do paralelismo intrínseco da rede, obtendo um sistema mais rápido, com um desempenho superior a outras plataformas e eficiente em termos de área(DIAS; ANTUNES; MOTA, 2004). Por outro lado, o desenvolvimento de hardware é mais lento e complexo que o desenvolvimento de software(DIAS; ANTUNES; MOTA, 2004). Uma maneira de contornar este problema é permitir a criação de arquiteturas de hardware para RNA-MLP e realizar a prototipação mais rápida e fácil, através do uso das plataformas baseadas em *Field Programmable Gate*

Array (FPGA), que provou ser eficiente em aplicações de redes neurais, diminuindo seu tempo de desenvolvimento (Huynh, 2017).

Na literatura, diferentes implementações em hardware de RNA podem ser encontradas. No capítulo 3 deste trabalho, serão apresentados quatro trabalhos relevantes de desenvolvimento de arquiteturas de hardware em FPGA para RNAs. Pode-se destacar a dificuldade de se encontrar uma arquitetura que seja configurável para diferentes topologias de uma rede neural artificial, sendo aplicáveis no contexto de problemas de aprendizagem profunda, e que apresentem um bom compromisso entre desempenho e uso de recursos do FPGA.

Este trabalho propõe o projeto e desenvolvimento de uma arquitetura configurável e flexível de uma RNA-MLP para aprendizagem profunda. Neste contexto, é explorado a adequação de dispositivos FPGAs para implementar o paralelismo intrínseco das RNA-MLPs, visando melhorar a construção de aplicações de sistemas embarcados e ciber-físicos. Diferentemente dos trabalhos relacionados, a arquitetura proposta explora o paralelismo temporal e espacial através de um caminho de dados sequencial das entradas em um pipeline fim a fim da RNA-MLP com a topologia da rede sendo totalmente parametrizável e com representação numérica em 32 *bits* de ponto flutuante.

O trabalho proposto foi validado utilizando um conjunto de dados padrão de validação em AM para aprendizagem profunda. Sendo ele um conjunto de dados de dígitos manuscritos MNIST e um outro conjunto de dados de vestuário Fashion-MNIST. Os resultados de desempenho da rede foram comparados com trabalhos relacionados que fizeram uso do mesmo conjunto de dados. Também foram feitas comparações com a mesma RNA-MLP implementada em software em CPU, GPU e em FPGA. Podemos adiantar que os resultados de tempos de execução da arquitetura proposta em FPGA foi 59.2 vezes mais rápido que em GPU e 366 vezes mais rápido que em CPU.

1.1 Estrutura do Trabalho

O restante do trabalho é organizado da seguinte forma: o Capítulo 2 apresenta uma revisão de todos os conceitos básicos referentes as RNA-MLP, neurônio artificial, estrutura em camadas e aprendizado. Também neste capítulo é apresentada uma motivação para o uso das plataformas baseadas em FPGA. O capítulo 3 descreve alguns trabalhos relacionados envolvendo desenvolvimento de RNA em várias plataformas de hardware baseado em FPGA. A descrição da arquitetura proposta neste trabalho e a organização dos dados, cálculo do estado de ativação, função de ativação são apresentadas no capítulo 4. O capítulo 5 apresenta toda a metodologia de desenvolvimento deste trabalho, dividido em três conjuntos de etapas; software, hardware, hardware e software. A descrição da implementação dos módulos da arquitetura proposta é demonstrada nesta parte do texto. Em seguida, o capítulo 6 apresenta os resultados de desempenho da arquitetura proposta com prototipação em FPGA, bem como uma comparação com resultados de outras abordagens

existentes das RNAs prototipadas em FPGAs. Finalmente no capítulo 7 são apresentadas as conclusões obtidas neste trabalho, bem como trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

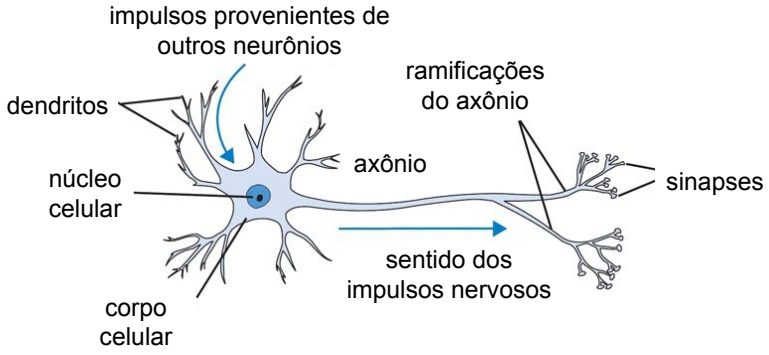
Este capítulo apresenta conceitos e definições necessárias para se entender os demais capítulos desta dissertação. A seção 2.1 apresenta alguns conceitos básicos de redes neurais artificiais; e a seção 2.2 descreve o princípio de funcionamento e principais vantagens das plataformas baseadas em FPGAs para implementação da arquitetura de rede neural proposta neste trabalho.

2.1 Visão Geral das Redes Neurais Artificiais

A estrutura das redes neurais artificiais (RNAs) têm inspiração nos neurônios biológicos e nos sistemas nervosos. Podemos dizer que redes neurais artificiais consistem em um mecanismo de implementar soluções de problemas de inteligência artificial(BARRETO, 1997). Assim, invés de se tentar programar um sistema computacional de modo a fazê-lo imitar um comportamento inteligente (por exemplo, jogar xadrez, compreender caracteres, fazer cálculos, etc), buscamos construir um sistema computacional que implementa componentes (funções) que possam ser conectadas por modelos de ligações que imitam as ligações cerebrais. Desta forma, espera-se que este sistema computacional tenha comportamento inteligente, com capacidade de aprender novas tarefas, de errar, fazer generalizações e consequentemente de ultrapassar seu conhecimento inicial.

O cérebro humano é composto basicamente por duas grandes classes de células, neurônios e células da glia. É comum atribuir aos neurônios as principais funções cerebrais, já que com relação à função da glia ainda tem muito a ser descoberto(BARRETO, 1997). Existem bilhões de células interconectadas no cérebro humano formando uma rede gigantesca, onde aproximadamente 10^{11} são neurônios, os quais podem ser de diversos tipos. Um neurônio típico é apresentado na Figura 2.

Figura 2 – Representação de um neurônio biológico



Fonte: extraído e adaptado de (STANFORD, 2019)

Um neurônio é composto de um corpo celular, chamado também de *soma* (onde estão o núcleo, o citoplasma e o citoesqueleto) e de seus prolongamentos celulares finos, as ramificações, que podem ser subdivididas em *dendritos* e *axônios*. O corpo celular é caracterizado por não se dividir depois da fase embrionária, assim como, desenvolver vários dendritos e um único axônio a partir de sua estrutura, podendo se ramificar e ficar recoberto por *sinapses*. Juntamente com os dendritos, o corpo celular possui mecanismos para prover energia à célula e sintetizar proteínas.

As ramificações conhecidas como dendritos, conduzem sinais das extremidades para o corpo celular. São estruturas tubulares que se ramificam para formar um ramo de uma árvore ao redor do corpo celular. Os dendritos constituem a superfície física, através da qual um neurônio recebe sinais de entrada de muitas outras células, caracterizando-se como terminal de entrada dos neurônios.

Existe também uma ramificação, geralmente única, chamada axônio, que transmite um sinal do corpo celular para suas extremidades. Eles são extensões do corpo celular nos quais os sinais trafegam a partir do corpo por longas distâncias. O axônio difere dos dendritos estruturalmente e nas propriedades de sua membrana externa. Os axônios são mais finos e longos que os dendritos e em geral ramificam-se na parte terminal, formando as sinapses e permitindo que a informação de uma célula atinja muitas outras. A membrana externa dos axônios propaga impulsos elétricos e libera neurotransmissores (substâncias químicas produzidas pelos neurônios) em seus terminais enquanto que a membrana nos dendritos responde aos neurotransmissores.

As sinapses têm um papel fundamental na memorização da informação e são principalmente as sinapses do córtex cerebral e de partes mais profundas do cérebro que armazenam informação. Pode-se imaginar que em cada sinapse a quantidade de neurotransmissores que podem ser liberados em uma mesma frequência de pulsos do axônio representa a informação armazenada nesta sinapse(BARRETO, 1997).

Cada vez que a sinapse é ativada e encontra ativado ou consegue ativar outro neurônio, o número de neurotransmissores liberados aumenta na próxima vez que o neurônio for ativado. Isto representa um aumento da conexão entre os dois neurônios. Este processo chama-se *facilitação*. Um neurônio tem de 10^3 a 10^4 sinapses e pode receber informação de até 10^3 outros neurônios(BARRETO, 1997). Tal mecanismo de facilitação, inspirou a conhecida Lei de Hebb, que neste contexto pode ser interpretada como: *A intensidade de uma conexão sináptica entre dois neurônios aumenta quando os dois neurônios estão excitados simultaneamente.* A lei de Hebb é um conceito básico utilizado em muitos algoritmos de aprendizagem baseada em RNAs(BARRETO, 1997).

Um conjunto de sinais vindo dos dendritos são processados no corpo celular de um neurônio. Se ele atingir uma certa diferença de potencial entre o meio interno e o meio externo, um impulso (ou um conjunto deles) é propagado pelo axônio. Nas extremidades sinápticas do axônio, o sinal é transmitido para os dendritos de um ou mais neurônios, assim sucessivamente entre milhares de neurônios conectados.

A criação das redes neurais artificiais (RNAs) têm construção e fundamentação inspirada nos neurônios biológicos e nos sistemas nervosos. Nosso cérebro realiza processamento paralelo e distribuído, as informações são guardadas nas conexões de neurônios e aprende-se através de exemplos e repetições. Tudo isso realizado por unidades simples, porém numerosas que isoladamente não fazem diferença para o todo. Mas é bem verdade que se cria e também se usa o conhecimento prévio para adaptar-se a situações nunca antes passadas(FERREIRA, 2011). Usando o paralelo biológico, nas RNAs um grande número de neurônios trabalha de forma conexa para processar a informação e fornecer um resultado(BRAGA; CARVALHO; LUDERMIR, 2007).

Inspirado no modelo biológico, foi definido para as RNAs um modelo de neurônio artificial e as conexões que simulam as sinapses. Esse ramo da computação teve seu primeiro apogeu com a criação das redes Perceptron(BRAGA; CARVALHO; LUDERMIR, 2007). As RNAs são utilizadas na solução de diversas áreas do conhecimento, como processamento de sinais, reconhecimento de caracteres, análise e reconhecimento de imagens, sistemas de diagnóstico, previsões de séries temporais, etc. Um problema tipicamente solucionável com RNAs deve ser descrito como um problema de reconhecimento de padrões ou de aproximação de uma função. Para os casos de reconhecimento de padrões, uma RNA deve classificar um padrão de entrada dentre os de saída, mesmo sem nunca ter visto o referido padrão.

As RNAs apresentam as seguintes características desejáveis (FERREIRA, 2011):

1. **Aprendem através de exemplos** – Vários exemplos são mostrados à rede e na medida em que se ajustam os pesos a rede vai aprendendo.

2. **Adaptabilidade** – Alguns modelos de RNAs podem aprender continuamente, sem que se precise uma nova rodada de treinamento.

3. **Capacidade de generalização** – A rede após treinamento deve responder bem a exemplos nunca antes vistos por ela.

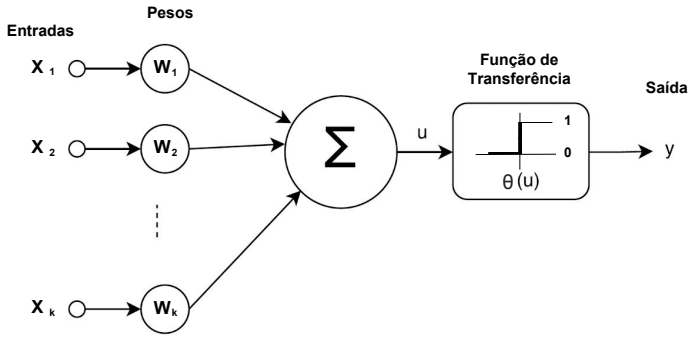
4. **Tolerância a falhas** – Se retirados poucos neurônios a rede ainda assim responde da mesma forma (conhecimento descentralizado). E mesmo que os dados de treinamento contenham um certo nível de ruído, a rede ainda responde corretamente.

5. **Implementação rápida** – As únicas etapas de desenvolvimento são treinamento, validação e teste. Elas são repetidas até que uma configuração candidata responda satisfatoriamente ao conjunto de testes.

2.1.1 Surgimento do Neurônio Artificial

O modelo de neurônio foi proposto pelo neurofisiologista americano Warren Sturgis McCulloch e pelo logístico Walter Pitts em 1943(MCCULLOCH; PITTS, 1943). Este é um modelo simples e foi criado com o objetivo de representar matematicamente um neurônio biológico. Assim, conhecendo o potencial de ação das membranas dos neurônios biológicos, eles interpretaram o funcionamento dos neurônios como sendo um circuito binário. O modelo binário pode ser observado na Figura 3.

Figura 3 – Modelo do neurônio de McCulloch e Pitts



A entrada do neurônio é também binária, pois adveio de um neurônio, e as várias entradas são combinadas por uma soma ponderada, produzindo a *entrada efetiva* do neurônio:

$$u = \sum_{i=1}^{k} x_i w_i \tag{2.1}$$

O resultado da entrada efetiva u serve de parâmetro para a *função de transferência*, neste caso, uma função binária que terá uma saída ativa considerando o limiar(*threshold*):

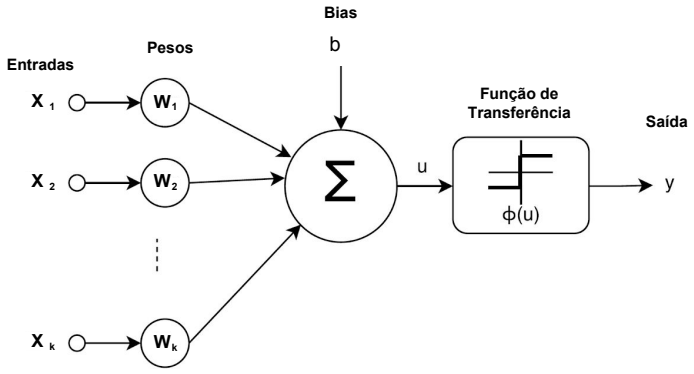
$$u \geq \theta \tag{2.2}$$

Em 1958, Frank Rosenblatt, um neurocientista americano, propôs a arquitetura de mapeamento de padrões chamada de *Perceptron*, com uma camada única como classificador de conjunto de padrões com valores contínuos em uma de duas classes, através de treinamento supervisionado(ROSENBLATT, 1958). Logo, tal arquitetura não permite aprender ou solucionar problemas que não sejam linearmente separáveis, mas foi pioneira em apresentar a primeira rede neural artificial fazendo o uso do modelo geral de neurônio artificial, com pesos e *bias* ajustáveis.

2.1.2 Modelo Geral do Neurônio Artificial

O modelo geral de um neurônio artificial é mostrado na Figura 4, sendo uma generalização do modelo de McCulloch e Pitts.

Figura 4 – Modelo geral de um neurônio artificial



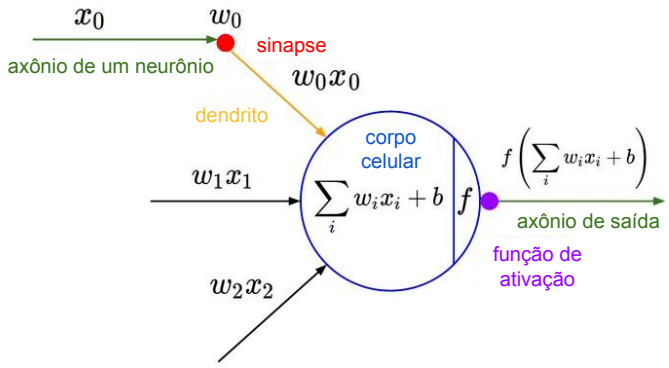
Neste modelo podemos observar que os sinais de entrada, representados por X_1, X_2, $\cdots$,X_k representam os estímulos eletroquímicos, os impulsos, gerados pelo corpo celular sendo transmitidos pelo axônio do neurônio biológico. Para cada entrada X_k existe seu respectivo peso sináptico W_k que será multiplicado.

Os pesos sinápticos representam variações nas ligações sinápticas entre os neurônios e determinam a influência dessa entrada como um estimulo para o neurônio através de seus dendritos. Os valores dos pesos sinápticos podem ser positivos ou negativos, dependendo de as sinapses correspondentes serem inibitórias ou excitatórias (BRAGA; CARVALHO; LUDERMIR, 2007).

Os estímulos multiplicados pelos seus respectivos pesos são somados e acrescidos de um *bias*, representado por b, que é o responsável por determinar o limiar de ativação para o neurônio e geralmente usado para representar uma polarização(BARRETO, 1997). O resultado desta soma, é chamado de *estado de ativação* do neurônio e é utilizada como parâmetro de entrada de uma função de ativação ϕ, responsável por apresentar o comportamento não linear encontrado em um neurônio biológico (NIELSEN, 2015).

A Figura 5 mostra o modelo matemático geral de um neurônio artificial em consonância com o modelo biológico.

Figura 5 – Modelo matemático geral de um neurônio artificial e sua representação biológica



Fonte: extraído e adaptado de (STANFORD, 2019)

Podemos resumir matematicamente um neurônio artificial à seguinte equação:

$$y = f(\sum_{i=1}^{k}(x_i \cdot w_i) + b) \tag{2.3}$$

Onde y é o impulso nervoso de saída do neurônio que será transmitido para os próximos neurônios. Podemos separar a equação 2.3 em duas equações:

1. Equação do estado de ativação do neurônio:

$$u = \sum_{i=1}^{k}(x_i \cdot w_i) + b \tag{2.4}$$

2. Equação da função de ativação que fornece a saída do neurônio:

$$y = f(u) \tag{2.5}$$

Podemos assumir, que a soma de produtos realizada na computação do estado de ativação é equivalente ao produto interno entre o vetor de pesos pelo vetor de entradas mais o *bias* b. Desta maneira, podemos reescrever a equação 2.4 da seguinte forma:

$$u = \sum_{i=1}^{k}(x_i \cdot w_i) + b = \begin{bmatrix} w_1 & \cdots & w_k \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} + b \tag{2.6}$$

Considerando o *bias* b um peso a mais, ficamos com $k+1$ pesos sinápticos e precisamos ajustar o vetor de entrada para podermos efetuar o cálculo do estado de ativação. Para

isso, adicionamos uma entrada correspondente ao *bias* de valor fixo em 1 resultando na seguinte equação:

$$u = \sum_{i=1}^{k+1}(x_i \cdot w_i) = \begin{bmatrix} w_1 & \cdots & w_{k+1} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_{k+1} \end{bmatrix} \tag{2.7}$$

Podemos concluir que para implementar uma rede neural artificial é preciso computar um número de produtos internos, que cresce exponencialmente com a quantidade de conexões entre os neurônios da rede. Nas próximas seções detalharemos as Redes Neurais Artificiais (RNA) Perceptron Multicamadas (MLP), onde esse fato ficará mais evidente. Depois de efetuada a computação do estado de ativação u, esse resultado é aplicado na função de ativação que tem como objetivo introduzir a não linearidade do neurônio artificial, ser diferenciável e contínua (BARRETO, 1997).

Historicamente, uma escolha comum da função de ativação é a função sigmoide, pois ela recebe um número real de entrada (estado de ativação) e o ajusta entre 0 e 1 de forma suave. Desta forma, uma suavização da função de ativação do tipo degrau é realizada (STANFORD, 2019). Ambas as funções podem ser visualizadas na Figura 6 e neste contexto podem ser representadas pelas equações:

1. Função degrau:

$$f(u) = \begin{cases} 0, & se\ u < 0 \\ 1, & se\ u \geq 0 \end{cases} \tag{2.8}$$

2. Função sigmoide:

$$f(u) = \frac{1}{1 + e^{-u}} \tag{2.9}$$

Outros exemplos de funções de ativação amplamente usadas são a Tangente Hiperbólica, Unidade Linear Retificada, Unidade Linear Exponencial, que são detalhadas nas Equações:

1. Tangente Hiperbólica:

$$f(u) = \frac{e^{2u} - 1}{e^{2u} + 1} \tag{2.10}$$

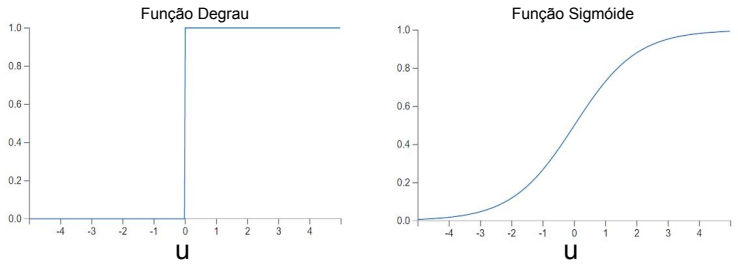
2. Unidade Linear Retificada (relu)

$$f(u) = \begin{cases} 0, & if\ u < 0 \\ u, & if\ u \geq 0 \end{cases} \tag{2.11}$$

3. Unidade Linear Exponencial (elu)

$$f(u) = \begin{cases} \alpha\,(e^u - 1), & if\ u < 0 \\ u, & if\ u \geq 0 \end{cases} \tag{2.12}$$

Figura 6 – Funções de ativação degrau e sigmoide, historicamente muito utilizadas em neurônios artificiais



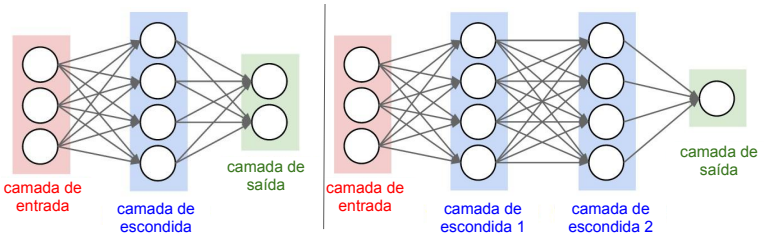
Fonte: extraído e adaptado de (NIELSEN, 2015)

2.1.3 Organização em camadas: Perceptron Multicamadas

As Redes Neurais Artificiais são modeladas como conjuntos de neurônios conectados em um gráfico acíclico. Em outras palavras, as saídas de alguns ou geralmente todos os neurônios podem se tornar entradas para outros neurônios. Ciclos não são permitidos, pois isso implicaria em um *loop* infinito no sentido da rede, que geralmente tem sentido único para frente(em direção a camada de saída) da rede (STANFORD, 2019). Assim, os modelos de rede neural são frequentemente organizados em camadas distintas de neurônios. Para as RNAs-MLP, o mais comum é as camadas serem totalmente conectadas no qual os neurônios entre duas camadas adjacentes são totalmente conectados para o sentido da frente da rede, mas os neurônios dentro de uma única camada não compartilham conexões.

Ao contrário de todas as camadas de uma rede neural, os neurônios da camada de saída geralmente não têm uma função de ativação (STANFORD, 2019). Isso ocorre porque a última camada de saída é geralmente usada para representar as pontuações da classe, representada pelo neurônio de saída. Na Figura 7 estão duas topologias de rede neural artificial de perceptron multicamada totalmente conectadas:

Figura 7 – **A esquerda:** Uma MLP de 2 camadas, uma camada oculta de 4 neurônios e uma camada de saída com 2 neurônios e três entradas. **A Direita:** Uma rede neural de 3 camadas com três entradas, duas camadas ocultas de 4 neurônios cada e uma camada de saída. Observe que, em ambos os casos, há conexões (sinapses) entre os neurônios através das camadas, mas não dentro da mesma camada



Fonte: extraído e adaptado de (STANFORD, 2019)

É importante destacar que quando se fala em rede neural de m camadas , não contamos a camada de entrada. Portanto, uma rede neural de camada única descreve uma rede sem camadas ocultas (entrada diretamente mapeada para saída). Além disso, a quantidade de parâmetros (pesos e *bias*) da rede é comumente usada para descrever o tamanho da rede. Podemos usar as duas redes apresentadas na Figura 7 para exemplificar:

- A primeira rede (esquerda) tem (3 x 4) + (4 x 2) = 20 pesos e (4 + 2) = 6 *bias*, um total de 26 parâmetros que podem ser aprendidos.
- A segunda rede (direita) tem (3 x 4) + (4 x 4) + (4 x 1) = 12 + 16 + 4 = 32 pesos e (4 + 4 + 1) = 9 *bias*, para um total de 41 parâmetros aprendíveis.

Devido a organização das redes neurais artificiais ser em camadas, torna muito simples e eficiente computar as redes neurais artificiais utilizando operações vetoriais matriciais, como observamos na Equação 2.7, onde a computação do estado de ativação de um neurônio pode ser feita através do produto interno do vetor de pesos daquele neurônio pelo vetor de entradas. Assim podemos estender tal operação como sendo uma multiplicação de matrizes, se consideramos todos os neurônios e parâmetros de um camada.

Podemos utilizar como exemplo a rede neural artificia de três camadas (direita) da Figura 7. A entrada seria um vetor x, de tamanho (3x1) mais uma entrada de valor fixo em 1 correspondente ao *bias*, ficando com o tamanho (4x1). Todos os pesos de uma camada podem ser armazenadas em uma única matriz. Por exemplo, os pesos da primeira camada oculta $w1$, seriam do tamanho (4x3) mais a adição do vetor coluna (concatenação) do

bias para todas os nerônios da camada, de tamanho (4x1), ficando com uma matriz de parâmetros de tamanho (4x4) . Assim, cada neurônio da camada oculta tem seus pesos e o *bias* correspondente às conexões das entradas em uma linha da matriz $w1$. Ao realizar a multiplicação das matrizes $w1 \times x$ computamos o estado de ativação de todos os neurônios desta camada. Da mesma forma, isto pode ser feito para as conexões da segunda camada oculta e para a última camada (saída). A computação total desta rede neural artificial de 3 camadas é simplesmente três multiplicações de matriz, entrelaçadas com a aplicação da função de ativação. A Equação 2.13 exemplifica matematicamente a computação do estado de ativação de uma camada com n neurônios e k entradas.

$$\begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,k+1} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,k+1} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_{k+1} \end{bmatrix} \tag{2.13}$$

2.1.4 Aquisição do Conhecimento das Redes Neurais Artificiais

Redes Neurais Artificiais possuem a capacidade de aprender por exemplos e fazer interpolações e extrapolações do que aprenderam. Por aprendizado entende-se a capacidade de aprender a partir da interação com o ambiente de forma a melhorar seu desempenho no decorrer do tempo. Aprender pode ser considerado como atributo fundamental de ter um comportamento inteligente. No aprendizado conexionista, procura-se determinar a intensidade de conexões entre neurônios. Como o conhecimento é armazenado nas conexões, o uso das redes neurais artificiais está intimamente ligado ao que se chama de conexionismo (BRAGA; CARVALHO; LUDERMIR, 2007).

A utilização de uma RNA na solução de uma tarefa passa inicialmente por uma fase de aprendizagem, quando a rede extrai informações relevantes de padrões (conjunto de entradas) de informação apresentados para elas, criando assim uma representação própria para o problema. A etapa de aprendizagem, também chamada de treinamento da rede, consiste em um processo iterativo de ajuste dos parâmetros da rede, os pesos das conexões entre os neurônios, que guardam, ao final do processo, o conhecimento que a rede adquiriu do ambiente.

Diversos métodos para treinamento de redes foram desenvolvidos, podendo ser agrupados em dois paradigmas principais (BRAGA; CARVALHO; LUDERMIR, 2007):

- **Aprendizado Supervisionado**

  Este tipo de aprendizado é o mais comum no treinamento das redes neurais artificiais. É chamado de aprendizado supervisionado porque a entrada e saída desejadas para a rede são fornecidas por um supervisor (professor) externo. O objetivo é ajustar os parâmetros da rede, de forma a encontrar uma ligação entre os pares de entrada e saída fornecidos. O professor indica explicitamente um comportamento bom ou ruim para a rede. visando direcionar o processo de treinamento.

A rede tem sua saída calculada comparada com a saída desejada, recebendo do supervisor o erro da resposta atual. A cada padrão de entrada inserido na rede compara-se a resposta obtida com a desejada, ajustando-se os pesos das conexões para minimizar o erro. A minimização do erro é incremental, já que pequenos ajustes são feitos nos pesos a cada etapa de treinamento, buscando uma resposta correta. A soma dos erros quadráticos de todas as saídas é normalmente utilizada como medida de precisão da rede e também como função de custo a ser minimizada pelo algoritmo de treinamento.

A desvantagem do aprendizado supervisionado é que, na ausência do supervisor a rede não conseguirá aprender novas estratégias para situações não conhecidas pelos exemplos do treinamento da rede. O algoritmo mais conhecido para aprendizado supervisionado é a regra delta e a sua generalização para rede neurais artificiais perceptron multicamadas, o algoritmo *backpropagation*.

- **Aprendizado não Supervisionado**

  No aprendizado não-supervisionado, não há um supervisor (professor) para acompanhar o processo de aprendizado. Para este método de treinamento somente os padrões de entrada estão disponíveis para a rede, ao contrário do aprendizado supervisionado, onde as entradas e saídas desejadas estão disponíveis. Este tipo de aprendizado só se torna possível quando existe redundância nos dados de entrada. Sem essa redundância seria impossível encontrar quaisquer padrões ou características dos dados de entrada. A partir do momento em que a rede estabelece uma concordância ou regularidades estatísticas das entradas, desenvolve-se nela uma habilidade de classificar automaticamente as entradas, extraindo estatisticamente suas características mais relevantes criando novas classes ou grupos automaticamente. O algoritmo mais relevante é o *K-means*, que consiste em um método de *Clustering* onde os dados semelhantes são agrupados automaticamente.

De uma maneira geral os algoritmos de aprendizado são responsáveis por adaptar os parâmetros de uma rede neural artificial para que a mesma possa aprender uma determinada ação ou classificação. Como era de se esperar, não há um único algoritmo de aprendizado. O que temos é um conjunto de ferramentas representadas por diversos algoritmos, cada qual com suas vantagens e desvantagens. Esses algoritmos basicamente diferem pela maneira pela qual o ajuste dos pesos é feito.

Independentemente, a etapa de aprendizagem de uma rede neural artificial é caracterizada por ser um esforço de projeto de inteligencia artificial, que é realizado apenas durante o projeto de construção da topologia de configuração da rede neural artificial e aquisição dos pesos ideais para a determinada aplicação. Sendo assim, não é uma etapa que causa um impacto na execução de uma aplicação de redes neurais artificiais.

Como este trabalho propõe uma arquitetura configurável para aplicações de aprendizagem profunda, apenas a etapa de execução de uma rede neural artificial será considerada no projeto da arquitetura. Assim sendo, não é preciso aprofundar ainda mais no entendimento, nas técnicas e algoritmos da etapa de aprendizagem de uma rede neural artificial.

2.2 Conceitos básicos em FPGAs

Alcançar resultados ótimos em termos de velocidade, precisão, consumo de recursos depende da natureza do algoritmo, da estratégia de processamento, e também da plataforma para o processamento. Como vimos na seção anterior, para computar apenas o estado de ativação de um neurônio artificial é preciso fazer um produto interno. Quando estendemos esta operação a uma camada da rede neural artificial, precisamos fazer uma multiplicação de matrizes que se constitui de operações de multiplicação e somas. A quantidade deste tipo de operação cresce de maneira exponencial com o aumento do número de neurônios da rede em cada camada. Desta forma, uma grande quantidade de operações são necessárias. Assim, processar redes neurais artificiais perceptron multicamadas profundas, constituídas de camadas com centenas, milhares de neurônios certamente não alcançará desempenho aceitável em aplicações que precisem de classificação em tempo real (por exemplo, classificação de imagens a uma taxa de 30fps - *frames* por segundo), utilizando processamento sequencial. Dessa forma, uma maneira bastante promissora para se alcançar desempenho de tempo real seria explorar o paralelismo intrínseco de uma multiplicação de matrizes. Existem quatro tipos de plataformas frequentemente usadas para lidar com paralelismo: as abordagens baseadas em processadores de propósito geral (CPUs), as abordagens baseados em unidades de processamento gráfico (GPUs) e abordagens baseados em hardware tais com os ASICs e os FPGAs.

As CPUs, embora seja uma arquitetura de processamento inerentemente sequencial, com os avanços das tecnologias de arquitetura de processadores, esta passou a oferecer algumas estratégias de obtenção de paralelismo tais como instruções vetoriais que operam em um espaço de dados de 128 bits, os chamados instruções SIMD (*Single instruction, Multiple Data*), e os vários núcleos (*cores*) para processamento paralelo. Contudo, esta quantidade de paralelismo ainda pode ser insuficiente para a quantidade de operações possíveis em determinadas topologias de rede. As CPUs são processadores de propósito geral que oferecem um tipo de programação em forma execução de instruções. A escolha das instruções determina a funcionalidade desejada. Através de linguagens de código amigáveis, o desenvolvimento de uma determinada funcionalidade neste tipo de plataforma é bem rápido.

Ao contrário das CPUs que oferecem uma quantidade limitada de paralelismo, os processadores GPUs oferecem uma melhor infra-estrutura de processamento para exploração do paralelismo dos algoritmos devido ao uma grande quantidade de núcleos (mais de 1000 núcleos) de processamento disponíveis para computação independente dos dados.

Assim, é possível processar uma quantidade bem maior de operações em paralelo e desta forma, aumentar drasticamente o ganho de desempenho em relação a implementações em CPU. Contudo, as GPUs são processadores que trabalham em altas frequências e como consequência consomem e dissipam muita potência (por exemplo, uma GPU NVIDIA GTX 650 Ti dissipa mais de 110 Watts de potência) não sendo adequadas para aplicações embarcadas (BAILEY, 2011).

Os Circuitos Integrados de Aplicação Específica (ASICs) são abordagens baseados em hardware onde a o processamento das operações é feito através de uma implementação que utiliza componentes lógicos básicos que formarão um circuito integrado (CI). Os ASICs permitem o desenvolvimento de sistemas totalmente customizados de acordo com a aplicação possibilitando desenvolver produtos de alta tecnologia com baixo custo e consumo reduzido de energia. Para isso, é necessário aperfeiçoar e otimizar a lógica de acordo com a sua função. O consumo de energia também é reduzido devido a esta otimização pois o processador pode operar em uma frequência menor, já que possui alto desempenho. Contudo, dois problemas residem no desenvolvimento de abordagens com ASICs. O primeiro problema é a inflexibilidade para mudança de funcionalidade. Uma vez que o circuito foi configurado, o circuito integrado se torna difícil ou até impossível de ser reconfigurado caso a sua funcionalidade precise ser modificada. No caso das RNAs seria preciso fabricar um circuito para cada topologia de rede desejada, podendo ser parametrizável até um certo limite. O segundo problema é o alto custo de fabricação e desenvolvimento para produzir um circuito integrado com a funcionalidade desejada, além do grande tempo para desenvolver tal circuito. O custo e o tempo de desenvolvimento alto dificultam projetos que precisam de um retorno rápido do produto para o mercado.

Em alternativa, a tecnologia de lógica reconfigurável baseada em Field Programmable Gate Array (FPGA) combina as características dos ASICs com a flexibilidade do desenvolvimento de software das CPUs. Nesta tecnologia a funcionalidade pode ser reprogramada ou reconfigurada a qualquer momento. Os FPGAs são dispositivos em forma de chip programável que permite desenvolver um circuito de hardware totalmente customizado para a aplicação desejada tal como os ASICs. A grande vantagem no desenvolvimento de projeto usando FPGAs é a possibilidade de obtenção de um protótipo em fases iniciais do projeto. Isto é possível, porque estes dispositivos, juntamente com mecanismos de especificação baseados em linguagens amigáveis com abstrações em alto nível e as ferramentas de síntese e simulação, oferecem uma estratégia de desenvolvimento e verificação da implementação que combinam ao mesmo tempo a facilidade de simulação e a obtenção do protótipo para validação Courtoy (1998). Uma vez que o custo de desenvolvimento dos circuitos integrados específicos de aplicação (ASICs) ainda permanece alto, a utilização dos FPGAs têm sido largamente adotada tanto na comunidade acadêmica quanto na indústria. Com a crescente evolução tecnológica destes dispositivos que permite uma grande quantidade de unidades lógicas em um único circuito integrado, torna-se possível criar dispositivos

altamente complexos em um único chip de FPGAs. O uso da computação configurável através de FPGAs possibilita constante melhoria das técnicas de processamento de uma determinada função, uma vez que a reprogramação de um FPGA consiste na modificação de sua especificação feita em uma linguagem de descrição de hardware (HDL), sem que sejam necessárias alterações físicas no próprio hardware, resultando em sistemas eficientes, compactos, de baixo consumo de energia e de custo reduzido.

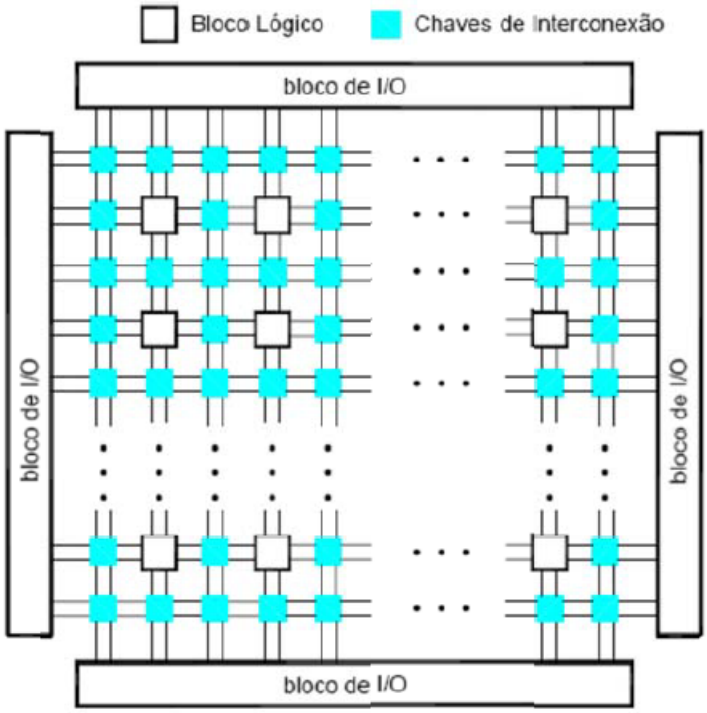
2.2.1 Estrutura Interna de um FPGA

A estrutura geral de uma FPGA é apresentada na Figura 8, na qual se pode visualizar três tipos de recursos: blocos I/O para conectar os pinos de entrada e saída do dispositivo, blocos lógicos (ou *slices*, em inglês) dispostos num arranjo de matriz e um conjunto de chaves de interconexão organizadas como canais de roteamento horizontal e vertical entre as linhas e colunas dos blocos lógicos.

Array of Logical Blocks (ALBs) e o *Configurable Logical Block* (CLB) são os nomes dados aos blocos lógicos que compõem um FPGA do fabricante Altera e o fabricante Xilinx, respectivamente. Os blocos lógicos, de uma forma geral, são constituídos por uma unidade lógica combinacional programável e um conjunto de somadores, registradores, flip-flops e latches.

Cada fabricante determina a organização interna do seu bloco lógico e seu comportamento, mas na maioria das vezes a estrutura interna dos blocos lógicos pode ser representada através da Figura 9. Existem diferentes formas de implementar a unidade lógica combinacional. O método mais comum consiste em usar *Look-Up Tables* (LUTs), que são células de memórias conectadas a um multiplexador que seleciona qual das células será saída da unidade lógica combinacional.

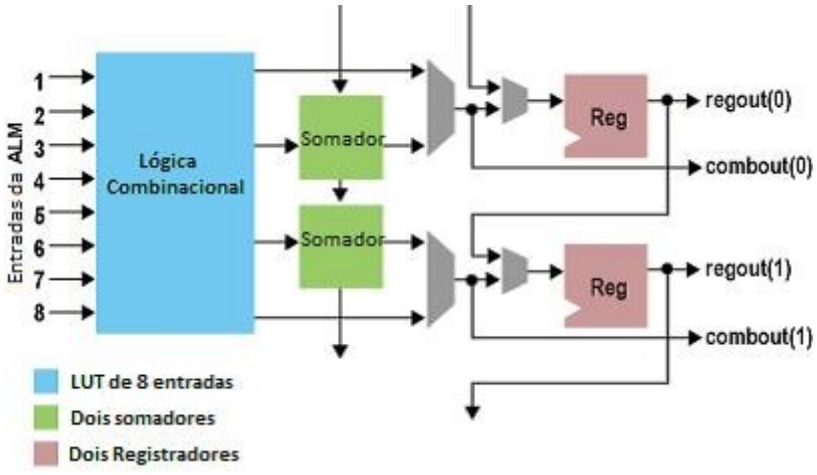
Figura 8 – Estrutura interna de uma FPGA



Fonte: extraído de (BROWN; VRANESIC, 2000)

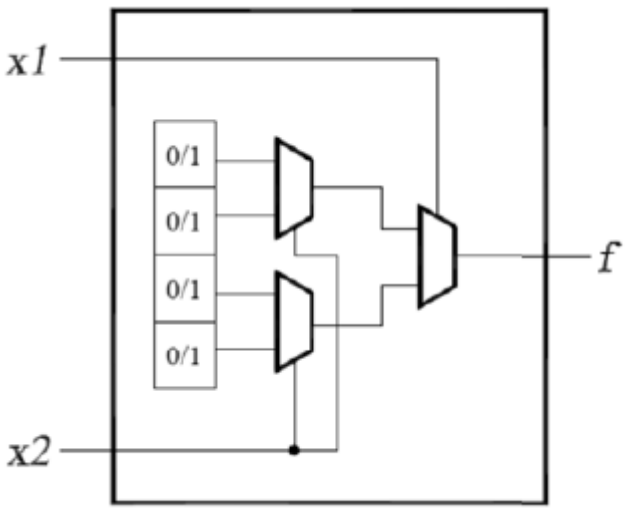
Na LUT ficam armazenadas possíveis respostas de partes de um sistema a um conjunto de estímulos. O sinal com tais estímulos é utilizado como o seletor do multiplexador que fica localizado na saída da unidade lógica combinacional. Assim, de acordo com o estímulo, é selecionada a resposta correta a ser encaminhada para a saída da unidade lógica combinacional.

Figura 9 – Organização interna de um bloco lógico de FPGA



Fonte: extraído de (DUTRA, 2010)

Figura 10 – Estrutura de uma LUT de duas entradas



Fonte: extraído de (BROWN; VRANESIC, 2000)

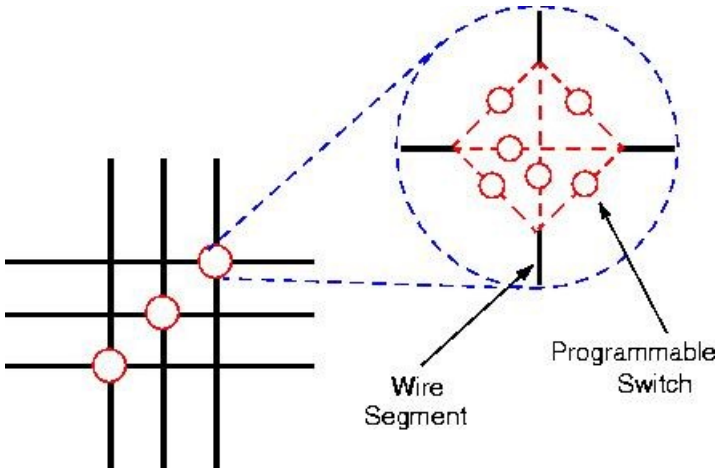
A Figura 10 apresenta a estrutura de uma pequena LUT com duas entradas, x_1 e x_2, e uma saída f, a qual é capaz de implementar qualquer função de duas variáveis. Devida à tabela verdade de duas variáveis possuir quatro combinações possíveis, esta LUT possui quatro células de armazenamento. As variáveis de entrada x_1 e x_2 são usadas com seletores de entrada dos três multiplexadores, que dependendo dos valores destas variáveis selecionam o conteúdo de uma das quatro células como saída da LUT. As FPGAs disponíveis no mercado geralmente possuem LUTs de quatro a oito entradas.

Quando um circuito lógico é implementado em um FPGA, os blocos lógicos são programados para realizar as funções necessárias e as chaves de interconexão são programadas para definir o canal de comunicação entre estes blocos lógicos. A interconexão entre os blocos lógicos permite que sejam construídos circuitos complexos, circuitos paralelos, tal interconexão é provida pelos elementos de roteamento. Estes elementos são distribuídos por todo o FPGA e são programados, na maioria das vezes, por ferramentas de software, que são capazes de encontrar os melhores caminhos e conexões a serem ativados para que os recursos disponíveis no FPGA sejam utilizados da forma mais otimizada possível.

A programação dos elementos de roteamento é feita através da configuração de uma estrutura conhecida como matriz de chaveadores (ou do inglês *Switch Matrix*), que é apresentada na Figura 11. Com o uso de tal estrutura, para ativar uma conexão entre dois elementos lógicos basta ativar um ou mais caminhos que levam sinais de um elemento para o outro.

Os FPGAs possuem blocos de memória embarcados (BRAM) que ficam conectadas aos blocos lógicos e que podem ser configurados para funcionar como SRAM ou ROM como mostrado na Figura 12. Tais unidades de memória servem para prover o armazenamento de

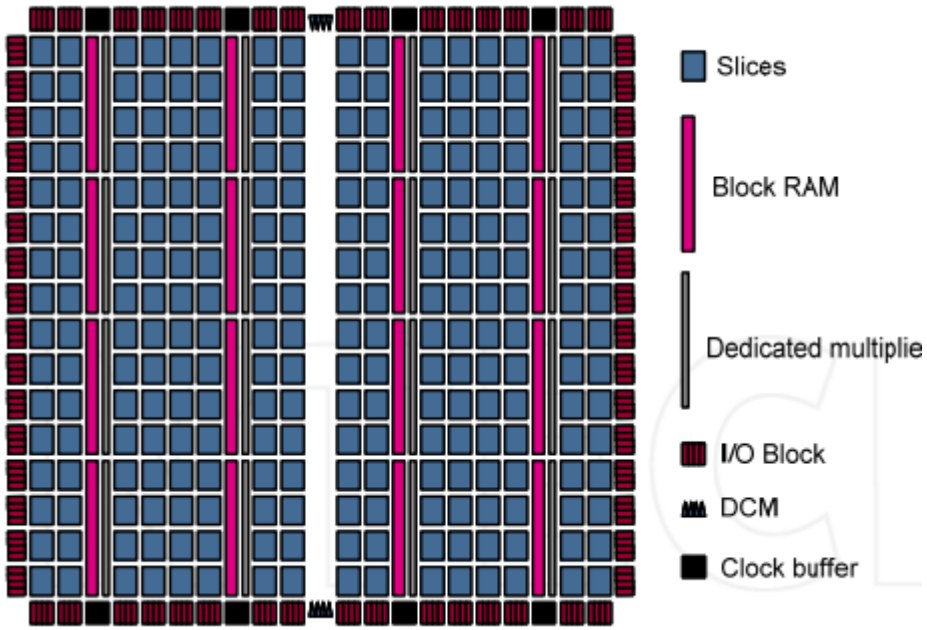
Figura 11 – Estrutura de uma matriz de chaveadores (*Switch Matrix*)



Fonte: extraído de (ROCHA, 2010)

dados no FPGA. Isso é importante, pois, como é descrito no capítulo 4, o armazenamento interno dos resultados intermediários do processamento realizado no FPGA elimina a necessidade de que tais resultados sejam escritos e lidos constantemente da memória externa que é conectada ao FPGA. Assim, todo o dado necessário para o processamento é lido apenas uma vez e o resultado é escrito apenas uma vez na memória externa conectada ao FPGA, o que resulta numa redução da necessidade de largura de banda de comunicação entre FPGA e memória.

Figura 12 – Ilustração da memória (BRAM) na estrutura de um FPGA genérico



Fonte: extraído de (MAGDALENO; RODRÍGUEZ, 2012)

Os blocos de entrada e saída implementam a comunicação do FPGA com elementos externos. Através dos blocos de entrada e saída é possível enviar sinais para serem processados dentro do FPGA e receber as respostas geradas por tal processamento. Além das estruturas básicas discutidas anteriormente, os FPGAs atuais possuem ainda em sua estrutura, um conjunto maior de tipos de elementos para aplicações especiais, como, por exemplo: unidades DSP, para processamento rápido de sinais digitais; unidades de memória, para armazenamento interno de dados DUTRA (2010); canais de comunicação de alta velocidade para a troca de dados entre FPGAs; e núcleos de CPU, que podem ser utilizados para executar programas no dispositivo reconfigurável. Todos estes são fatores que têm contribuído para que este tipo de tecnologia se torne cada vez mais uma opção atraente para o desenvolvimento de projetos que seguem a linha da computação reconfigurável.

2.3 Conclusão

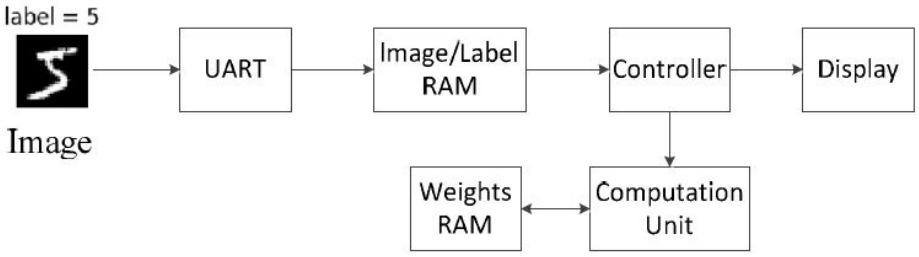
Embora as redes neurais artificiais ofereçam inúmeras possibilidades de aplicações, fazer o uso destas exige uma capacidade computacional massiva, devido ao crescimento exponencial de operações de produto interno (multiplicação de matrizes) com o aumento de conexões entre os neurônios nas camadas da rede. Estas características resultam numa grande dependência de dados levando à necessidade de grandes espaços de armazenamentos. Desta forma, para que esta abordagem possa ser utilizada em aplicações de tempo real e sobretudo implementadas em plataformas reconfiguráveis de FPGAs, a implementação de algoritmos de RNAs precisa resolver todos os desafios que a implementação das redes neurais artificiais impõe de maneira eficiente.

3 TRABALHOS RELACIONADOS

Na literatura podemos encontrar diversos trabalhos que tratam do desenvolvimento de arquiteturas para redes neurais artificiais perceptron multicamadas. Neste capítulo são apresentados alguns desses trabalhos que consideramos relevantes para serem relacionados com as técnicas propostas nesta dissertação, considerando alguns pontos que serão relacionados ao final do capítulo, após uma breve revisão dos trabalho.

No trabalho de Harris (Si; Harris, 2018) é apresentado o projeto de uma rede neural artificial perceptron multicamada em FPGA, que realiza o treinamento (aprendizado) e execução do benchmarck MNIST, um conjunto de dados de dígitos manuscritos. A implementação foi realizada em SystemVerilog, e o diagrama da arquitetura de hardware pode ser visualizada na Figura 13.

Figura 13 – Arquitetura do projeto em FPGA



Fonte: extraído de (Si; Harris, 2018)

A arquitetura consiste de um módulo de comunicação UART com um computador pessoal, uma memória RAM para armazenamento das imagens e seus respectivos rótulos (*labels*, utilizados na realização do treinamento supervisionado). Um módulo de controle transmite os dados para a unidade de computação. O controle também envia resultados para exibição em uma unidade de visualização externa (*display*). Quando uma única imagem e rótulo são transferidos para o FPGA, o módulo de controle pode ser acionado para iniciar o treinamento ou execução da rede neural utilizando o módulo da unidade de computação da rede. A Unidade de Computação da rede lê os pesos da rede na memória RAM, no modo de execução. No modo de treinamento é feita a leitura e atualização dos pesos. O trabalho não explica o detalhamento de como a unidade de computação foi

implementada na arquitetura, para executar todas as equações descritas no trabalho.

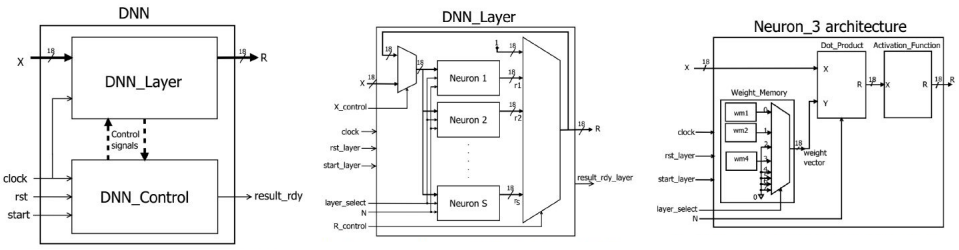
A arquitetura trabalha com uma representação numérica de 8 bits dos dados da rede neural, o que diminui a precisão, impactando na acurácia da rede em execução nesta arquitetura. Além disso, a arquitetura não é escalável em relação a topologia da rede, sendo uma implementação fixa de uma camada, onde são 784 entradas na primeira camada (28x28 *pixels* da imagem numérica manuscrita, contida no conjunto de dados MNIST), e 10 neurônios na camada de saída, cada um responsável por uma classe (dígitos de 0 a 9). Resultando em um total de 7850 parâmetros da rede neural.

Os autores reportaram que a arquitetura atingiu uma frequência de operação de 25 Mhz em um FPGA Cyclone IV-E e um tempo de execução de 3.8 segundos para a execução do conjunto de dados de teste do MNIST (10.000 imagens 28x28 *pixels*), sem considerar o tempo de comunicação com o computador pessoal. Em relação ao consumo de recursos do FPGA, a única informação disponibilizada é o número de elementos lógicos do FPGA, para 8 bits de precisão numérica, a arquitetura consumiu 34 mil elementos lógicos.

Podemos concluir que mesmo sendo um trabalho bem atual, o trabalho não buscou implementar uma arquitetura reconfigurável, escalável para atender a outras aplicações de redes neurais. Considerando as aplicações atuais de aprendizagem profunda em inteligencia artificial onde as redes neurais são medidas em centenas de milhares de parâmetros da rede, este trabalho possui algumas desvantagens sendo uma delas o fato de apresentar uma rede fixa para a solução de um conjunto de dados bem consolidado. Além disso, a implementação numérica com 8 *bits* de precisão, não corresponde, com a proposta de realizar treinamento da rede, pois a pouca precisão irá prolongar a iterações do algoritmo de aprendizagem, podendo até perder a capacidade de generalização da rede.

Diferentemente, o trabalho de Huynh (Huynh, 2017) propõe uma arquitetura personalizável em número de camadas, número de neurônios e entradas por camadas. A arquitetura de hardware apresentada usa apenas uma única camada de computação física para executar toda a estrutura computacional de uma rede neural artificial perceptron multicamada. A topologia da rede pode ser facilmente personalizada, mas com um limite de até 8 camadas, com número arbitrário de neurônios por camada e número de entradas. As entradas, pesos e saídas da rede proposta são representados em formato de número de ponto flutuante de meia precisão simples de $16 bits$. A arquitetura de hardware foi implementada usando a linguagem de descrição de hardware Verilog, com a reconfiguração da implementação sendo feita por um *script* em linguagem de MATLAB (MATHWORKS, 2019).

Figura 14 – **Esquerda:** visão geral da arquitetura do trabalho **Centro:** arquitetura do módulo único de computação da rede neural **Direita:** arquitetura de um neurônio da arquitetura



Fonte: extraído e adaptado de (Huynh, 2017)

A arquitetura detalhada na Figura 14 consiste em uma única camada de computação física e uma unidade de controle para alternar entre diferentes camadas da rede neural que serão computadas na camada única implementada. Podemos observar que a largura de dados dos sinais de entrada e saída no formato de ponto flutuante de meia precisão simples é de 18 bits, onde 2 bits, servem para informar o tipo de número (00 para o valor zero, 01 para valores normais, 10 para infinitos e 11 para não válido). A camada de computação é composta por S neurônios, sendo S a quantidade de neurônios da maior camada da rede, além de multiplexadores controlados pelo módulo de controle para seleção dos dados de entrada e saída da camada.

Cada neurônio, contem uma memória de peso com todos os vetores de peso correspondentes a diferentes camadas da rede. Durante o cálculo de um neurônio, o controle seleciona o vetor de peso correto daquele neurônio correspondente à camada em execução. Ao computar a multiplicação da matriz de pesos e o vetor de entradas em cada camada, todos os neurônios de hardware nessa camada executam em paralelo e a entrada é alimentada para todos os neurônios sequencialmente, ou seja, a entrada é compartilhada com todos os neurônios de hardware.

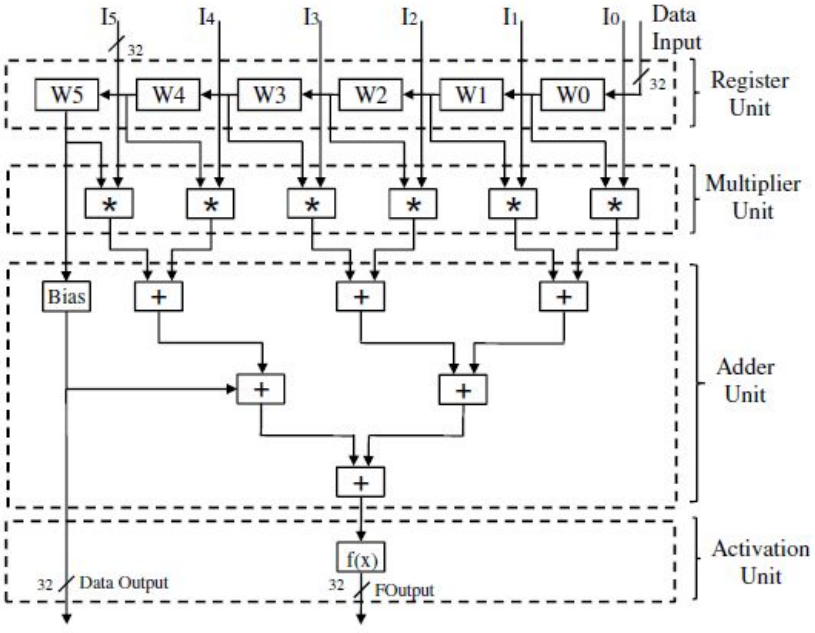
Os autores relataram que validaram a eficiência da arquitetura utilizando varias topologias de rede neural para a solução do conjunto de dados de dígitos manuscritos MNIST. A maior delas foi uma rede com 4 camadas, totalizando 132.184 parâmetros da rede, sendo executada em um FPGA Virtex-5 ZynQ-7000 7Z045 com 219 mil elementos lógicos, correspondente a 99% de uso do FPGA e uma frequência de operação de 219 Mhz. Os autores não explicam como os dados de entrada e saída da rede são comunicados com

o FPGA. O desempenho relatado pelos autores é de 15.9 mil imagens do conjunto de dados por segundo, o que corresponde a executar as 10.000 imagens de teste do conjunto em 0.63 segundos, mas não é relatado se esse tempo leva em consideração a comunicação com um computador pessoal ou dispositivo de alto nível, o que, naturalmente diminuiria o desempenho. Apesar de não informado explicitamente, o texto sugere que esse tempo é tempo de simulação e não da prototipação final da rede.

Concluímos que o trabalho é de extrema relevância comparativa com o trabalho proposto nesta dissertação. Infelizmente os autores omitiram informações a respeito da integração com o FPGA, o que prejudica a comparação em relação ao desempenho de processamento da rede para o conjunto de dados MNIST. Podemos destacar também o uso massivo de recursos do FPGA para a implementação de uma rede com 4 camadas com 132.184 parâmetros.

No trabalho publicado em (KOYUNCU et al., 2017), os autores propuseram uma biblioteca de neurônios para a rápida execução de uma rede neural artificial em plataformas baseadas em FPGA. A biblioteca contem um total de 60 neurônios distintos, obtidos a partir da definição de como será a configuração do neurônio, que pode realizar a multiplicação de uma até seis entradas de 32 *bits* em ponto flutuante de precisão simples, com pesos e *bias*. Sendo possível escolher entre dez diferentes funções de ativação implementadas, entre elas, sigmoide e tangente hiperbólica. A Figura 15 ilustra o detalhamento interno de um neurônio.

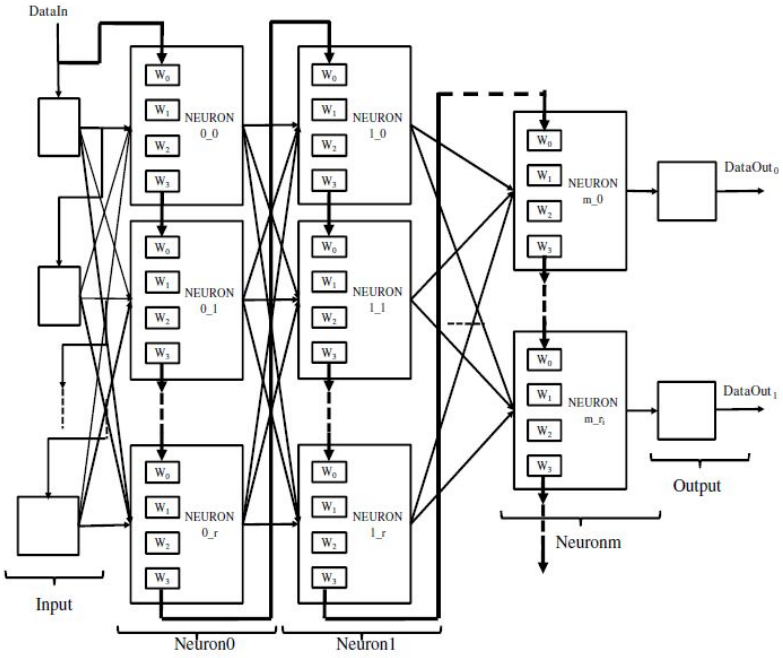
Figura 15 – Exemplo da arquitetura interna de um neurônio com 6 entradas e função de ativação



Fonte: extraído de (KOYUNCU et al., 2017)

Os neurônios foram projetados para serem conectados em sequencia uns aos outros formando as camadas da rede neural. A Figura 16 ilustra uma visão geral de uma rede neral com i entradas, m camadas, e r nerônios. Os autores não deixam claro como é feita a configuração individual das camadas, ou se todas as camadas precisam ser iguais.

Figura 16 – Visão geral das ligações entre os neurônios formando uma rede neural com até m camadas, r nerônios e i entradas



Fonte: extraído de (KOYUNCU et al., 2017)

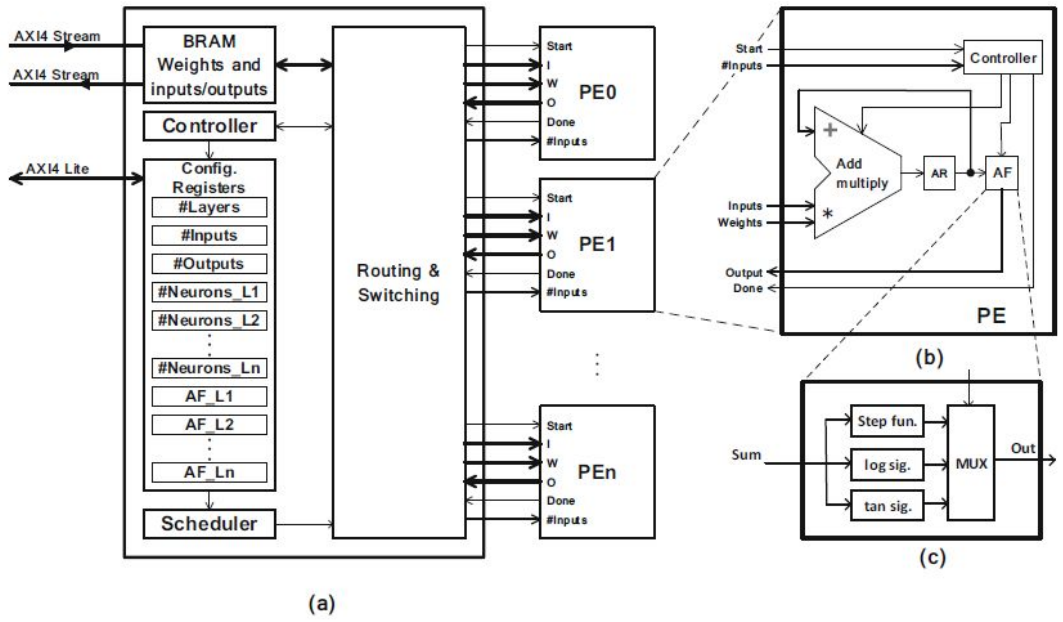
Os autores usaram a arquitetura desenvolvida para a construção de uma rede neural de 3 camadas com apenas 51 parâmetros para a aproximação da função de um sistema caótico. A rede foi treinada em MATLAB. A rede para execução em hardware foi validada, utilizando um conjunto de teste com 100 dados em um FPGA Xilinx's Virtex-6 (XC6VLX240T) a 304 Mhz e ocupando 66% da área do chip. Os autores não relatam o tempo de processamento em segundos, apenas uma comparação em velocidade com um computador pessoal.

Podemos destacar neste trabalho a utilização das redes neurais artificias em FPGAs numa aplicação diferente do tradicional. Infelizmente os autores não validaram com um conjunto de dados comumente utilizado o que dificulta uma comparação direta com a arquitetura proposta nesta dissertação. Mas como se trata de um trabalho recente, achamos relevante incluir esta referência na dissertação, pois consiste na construção de uma arquitetura configurável para redes neurais artificiais.

Em (AKLAH; ANDREWS, 2015) os autores apresentam o desenvolvimento de um co-processador de rede neural artificial perceptron multicamada configurável prototipado em FPGAs. O trabalho destaca que a arquitetura é ajustável via parâmetros de hardware,

para implementar qualquer topologia de rede (até um máximo definido) e usar diferentes tipos de funções de ativação (três disponíveis). Isso fornece a flexibilidade aos projetistas em vários domínios de aplicações.

Figura 17 – **(a):** Arquitetura geral do co-processador **(b):** Estrutura do elemento de processamento **(c):** Estrutura da função de ativação



Fonte: extraído de (AKLAH; ANDREWS, 2015)

A Figura 17 mostra a estrutura genérica de um núcleo do co-processador, que consiste em uma matriz linear de elementos de processamento, gerenciadas por um controlador e um escalonador dos elementos de processamento. O escalonador é responsável por gerenciar o número de elementos de processamento para executar uma rede com um número maior de neurônios, organizando eles em grupos. Cada elemento de processamento corresponde a um neurônio artificial e pode realizar até três funções de ativação implementadas.

O trabalho foi implementado em C/C++ e o hardware foi gerado utilizando a ferramenta de geração de projeto de hardware, Vivado Design Suite da Xilinx(XILINX, 2019). A arquitetura foi configurada e sintetizada para uma plataforma da FPGA Xilinx Zedboard, sendo avaliada em diversas configurações e aplicações relacionadas pelos autores. Podemos destacar duas configurações de rede interessantes, em uma delas a arquitetura é configurada para ter a quantidade de elementos de processamento igual a quantidade de neurônios da rede completa, o que segundo os autores é semelhante a uma abordagem em pipeline. Na outra configuração o número de elementos será igual a quantidade de neurônios da maior camada. Os autores relatam que a abordagem em pipeline apresenta um desempenho muito melhor, mas devido a instanciação de um elemento de processamento para cada neurônio a arquitetura consome muito recursos do FPGA, não sendo

sintetizável para um número de 64 elementos de processamento, o que é muito pouco.

A conclusão que se tem, é que a arquitetura ser configurável via parâmetros de implementação de hardware é algo relevante e muito vantajoso na praticidade de reutilização da arquitetura pelo projetista de inteligencia artificial. No entanto é preciso fazer um balanceamento entre o quão flexível e configurável é a arquitetura, pois a criação de um elemento de processamento com multiplicadores, somadores, com três funções de ativação implementadas, pode elevar a utilização de recursos do FPGA impossibilitando a instanciação redes maiores.

3.1 Conclusões

Na Tabela 2, nós comparamos os trabalhos relacionados apresentados com relação a algumas características criticas, citadas anteriormente, para o desenvolvimento da arquitetura. Este tipo de comparação permite visualizar os pontos que precisam ser melhorados e que motivaram o desenvolvimento de uma arquitetura mais flexível e eficiente para redes neurais artificiais, a qual está sendo proposta neste trabalho de dissertação.

O uso de representação em ponto flutuante de 32 *bits*, como o que acontece nos trabalhos (KOYUNCU et al., 2017) e (AKLAH; ANDREWS, 2015), trás uma precisão melhor para as redes neurais artificiais na sua capacidade de resolução de problemas. Contudo, se faz necessário módulos de operações matemáticas exclusivos para operações em ponto flutuante, o que aumenta a dificuldade em se utilizar essa representação numérica dos números reais em hardware. Somado a isso, fazer a implementação de uma nova arquitetura utilizando uma linguagem de descrição de hardware, permite ao projetista um domínio maior sobre as operações lógicas, controle dos dados e sinais em baixo nível de implementação.

A construção de uma arquitetura parametrizável, totalmente flexível e configurável, como acontece nos trabalho (Huynh, 2017), (KOYUNCU et al., 2017) e (AKLAH; ANDREWS, 2015) possibilita o uso da arquitetura em outras aplicações de rede neurais, com conjuntos de dados diferentes. Contudo, pudemos observar nestes trabalhos, mesmo os autores tendo afirmado que embora desenvolvidas arquiteturas configuráveis através de parâmetros, estes possuem certos limites de flexibilidade. Observa-se principalmente o consumo excessivo de recursos de hardware do FPGA em seus exemplos de validação, mesmo que em aplicações de pequenas redes neurais(poucos parâmetros). Construir uma arquitetura configurável significa ter uma topologia de rede neural flexível, em número de entradas, camadas e neurônios na rede, com um consumo de recursos reduzido, possibilitando a utilização da arquitetura em aplicações de aprendizagem profunda por exemplo, com a quantidade de parâmetros enormes.

A função de ativação de uma rede neural pode ser diferente de uma camada para outra, como discutimos no capítulo 2. Os trabalhos (KOYUNCU et al., 2017) e (AKLAH; ANDREWS, 2015) possibilitaram o uso de mais de uma função de ativação, mas é preciso ter cautela, devido a alta complexidade do hardware que resolve algumas dessas funções normalmente

Tabela 2 – Comparativo dos trabalhos relacionados considerando algumas características relevantes

Característica	Referência			
	(Si; Harris, 2018)	(Huynh, 2017)	(KOYUNCU et al., 2017)	(AKLAH; ANDREWS, 2015)
Representação Numérica	Representação linear em 8 *bits*	18 *bits*(onde, 2 bits de representação e 16 bits ponto flutuante meia precisão)	32 *bits*(ponto flutuante precisão simples)	32 *bits*(ponto flutuante ou ponto fixo)
Implementação	SystemVerilog	Verilog	Não Informado	C/C++
Arquitetura Configurável	Não	Sim (Scrit em MATLAB)	Sim	Sim
Quantidade de Entradas	Fixo	Parametrizável	Parametrizável	Parametrizável
Quantidade de Neurônios	Fixo	Parametrizável	Parametrizável	Parametrizável
Quantidade de Camadas	Fixo	Parametrizável (até 8, limite citado)	Parametrizável	Parametrizável
Função de Ativação Configurável	Fixa	Não informado	Sim (até 10 Funções)	Sim (até 3 Funções)
Validação utilizando Conjunto de Dados	Sim(MNIST)	Sim(MNIST)	Não (caso de uso)	Não (caso de uso)
Quantidade de Parâmetros da Rede	7.850	132.184(maior rede citada)	51	1.096 (maior rede citada)
Integração Hardware e Software	Não Informado	Não Informado	Não Informado	Não Informado

utilizadas em redes neurais artificias. Por isso é interessante buscarmos alternativas que possibilitem o uso de várias funções de ativação, comumente utilizadas em redes neurais artificiais.

A implementação em hardware de uma rede neural artificial precisa apresentar uma precisão igual a sua implementação em software. Para validarmos essa correspondência da arquitetura de hardware faz-se necessário a aplicação de um conjunto de dados de referência, a ser aplicado nas versões da rede em software e em hardware. Vários destes conjuntos de dados são disponibilizados para fins de comparação e são bem consolidados na área de aprendizagem de máquina (LECUN; BENGIO; HINTON, 2015). Os trabalhos apresentados em (Si; Harris, 2018) e (Huynh, 2017) validaram suas arquiteturas utilizando um conjunto de dados comumente utilizando em aprendizagem de máquina. Por outro lado, os trabalhos de (KOYUNCU et al., 2017) e (AKLAH; ANDREWS, 2015) validaram seus trabalhos

utilizando apenas casos de uso específicos com um conjunto de dados pequeno, sendo até gerados pelos próprios autores. No trabalho (KOYUNCU et al., 2017), por exemplo, os dados foram gerados pelos próprios autores.

Por fim, para a construção de uma arquitetura configurável e flexível que suporte uma quantidade maior de aplicações, é preciso apresentar resultados de integração dessa arquitetura de hardware a uma aplicação de software para uma validação real do uso da arquitetura e aferição do desempenho esperado em tempo de execução. Verificando assim a real aplicação da arquitetura em um contexto pratico das aplicações de redes neurais artificias em software. Nenhum dos trabalhos citados realizou tal integração hardware e software, o que nós deixa receosos da real aplicabilidade dos seus resultados.

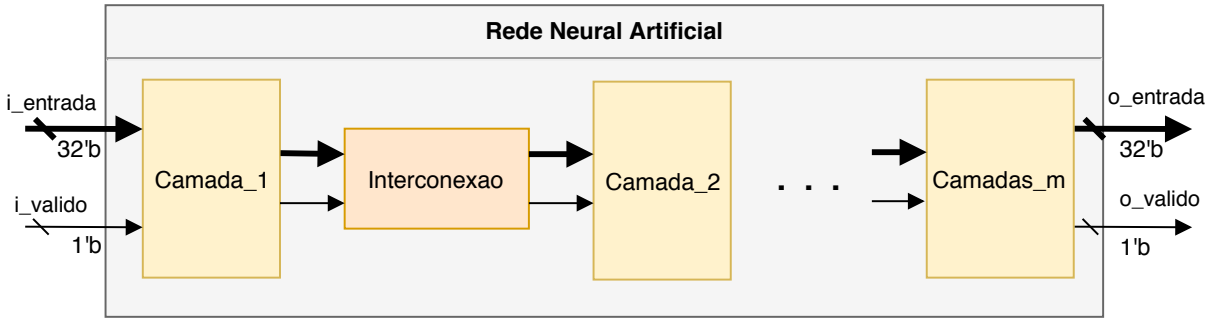
No próximo capítulo, será apresentada a proposta de uma arquitetura de redes neurais artificiais, a qual considera objetivos importantes, tais como; a construção de uma arquitetura configurável em hardware, com quantidade entradas, camadas e neurônios das camadas flexíveis; a implementação de um módulo função de ativação com complexidade reduzida, que possibilite a aplicação de várias funções de ativação; validação da arquitetura com uma topologia de rede neural com muitos parâmetros(centenas de milhares) no contexto de aprendizagem profunda; realizar a integração hardware e software para verificar o bom desempenho de execução da arquitetura proposta em um contexto prático real.

4 PROPOSTA DE UMA ARQUITETURA CONFIGURÁVEL PARA REDES NEURAIS ARTIFICIAIS

Neste capítulo será apresentada a arquitetura proposta para Redes Neurais Artificiais Perceptron Multicamadas, com o objetivo de auxiliar o fluxo de projeto de redes neurais artificiais em soluções heterogêneas, constituída pela junção de um processador de propósito geral, interligado a um FPGA. Esta arquitetura pode ser utilizada em aplicações voltadas para aprendizagem profunda. Tal objetivo ficará mais claro ao decorrer deste capítulo.

A Figura 18 ilustra uma visão geral da arquitetura da rede neural artificial perceptron multicamadas proposta. Podemos observar a semelhança com a que vimos na subseção 2.1.3 no capítulo 2. Tal semelhança é proposital, tendo como objetivo ser mais simples e amigável aos engenheiros de sistemas no projeto de uma solução de rede neural artificial em FPGA. Pois, um dos objetivos desta arquitetura é facilitar a reconfiguração e geração da mesma para a topologia da rede necessária na aplicação de aprendizagem requerida pelo projetista de inteligência artificial.

Figura 18 – Visão geral da arquitetura de redes neurais artificiais perceptron multicamadas



A arquitetura proposta foi dividida em dois módulos estruturais principais. O módulo *Camada* onde se encontra o cerne da arquitetura proposta neste trabalho, responsável pela computação do estado de ativação e da função de ativação de uma camada da rede. Este módulo foi projetado para ser genérico e parametrizado em sua instanciação, levando em consideração entre outros fatores: a quantidade de neurônios da camada e o tamanho do padrão de entrada vindo da camada anterior.

Entre as camadas, encontra-se o módulo *Interconexão*, de maneira semelhante a todas as conexões entre os neurônios de uma camada para outra. Este módulo de interconexão entre duas camadas garante que os dados que são emitidos na saída de uma dada camada obedeça o mesmo padrão dos dados de entradas que serão entregues a próxima camada. Tornando a arquitetura ainda mais genérica, de modo que o módulo de *Camada* pode ter uma interface regular dos dados de entradas, independente da localização na topologia da

rede. Assim sendo, é um módulo simples de controle e tem o objetivo de manter o mesmo formato dos dados geradas por uma camada, igual ao que é recebido na primeira camada da rede.

A interface de entrada e comunicação entre os módulos da arquitetura tem como entrada um único dado do tipo real em ponto flutuante com 32 *bits* e um *bit* de controle, denominado válido, o qual é utilizado para sinalizar que o dado é válido a partir do instante em que o conjunto de entradas começou a ser recebido. Uma camada é configurada para receber um conjunto de k entradas em sequência, onde, cada entrada deve ficar estável/fixa por n ciclos de *clock*, que é o número de neurônios da camada. Portanto, este módulo recebe uma entrada proveniente de uma camada anterior e a mantém estável por n ciclos de *clock* este procedimento ocorre para as k entradas de dados da rede. Tal fato será melhor explicado e exemplificado na seção 4.2.

Para explicar o módulo de *Camada* é preciso detalhar como é feita a computação do estado de ativação, apresentando todo fluxo dos dados e *pipeline* proposto. Em seguida, será descrita a aproximação das funções de ativação utilizadas no contexto desta proposta.

4.1 Computação do Estado de Ativação

Cada neurônio artificial realiza uma soma de produtos entre os pesos sinápticos e suas respectivas entradas no neurônio, ao final sendo somado o *bias*. Tal operação é chamada de computação do estado de ativação do neurônio e foi discutida no capítulo 2. Podemos assumir que a soma de produtos realizada na computação do estado de ativação é equivalente ao produto interno entre o vetor de pesos com a inclusão do *bias* pelo vetor de entradas com a inclusão de uma entrada equivalente ao *bias*. Tal reformulação da equação do estado de ativação do neurônio artificial foi melhor exemplificada na subseção 2.1.2. Assim sendo, a equação do estado de ativação u pelo produto interno entres os vetores de pesos, e entradas de tamanho $k+1$ onde k é a quantidade de entradas do neurônio, mais o *bias* conforme a equação 4.1.

$$u = \begin{bmatrix} w_1 & \cdots & w_{k+1} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_{k+1} \end{bmatrix} \tag{4.1}$$

Como mencionado no capítulo 2, podemos assumir que um camada com n neurônios realiza uma operação de multiplicação entre a matriz $W_{(n,k+1)}$, onde cada linha desta matriz corresponde aos pesos de um neurônio mais o *bias* e a matriz coluna $X_{(k+1,1)}$, onde cada linha corresponde a uma entrada da camada que é conectada a todos os neurônios daquela camada, mais as entradas de valor fixo em 1 equivalente ao *bias* e aos pesos. O resultante da multiplicação de matrizes U é dado pela equação 4.2. Pode-se observar que

cada elemento da matriz U tem resultado equivalente ao da equação 4.1 isoladamente.

$$U \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} = W \begin{bmatrix} w_{1,1} & \cdots & w_{1,k+1} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,k+1} \end{bmatrix} \times X \begin{bmatrix} x_1 \\ \vdots \\ x_{k+1} \end{bmatrix} \tag{4.2}$$

Para realizar uma multiplicação de matrizes em cada camada m, seria preciso ter k somadores e $k+1$ multiplicadores independentes para cada neurônio, ou seja, cada linha n da matriz de pesos W. Portanto, se pudéssemos instanciar $k+1$ multiplicadores e k somadores em um FPGA, o resultado seria computado em $n \times k$ ciclos de *clock* (após o *pipeline* estar cheio).

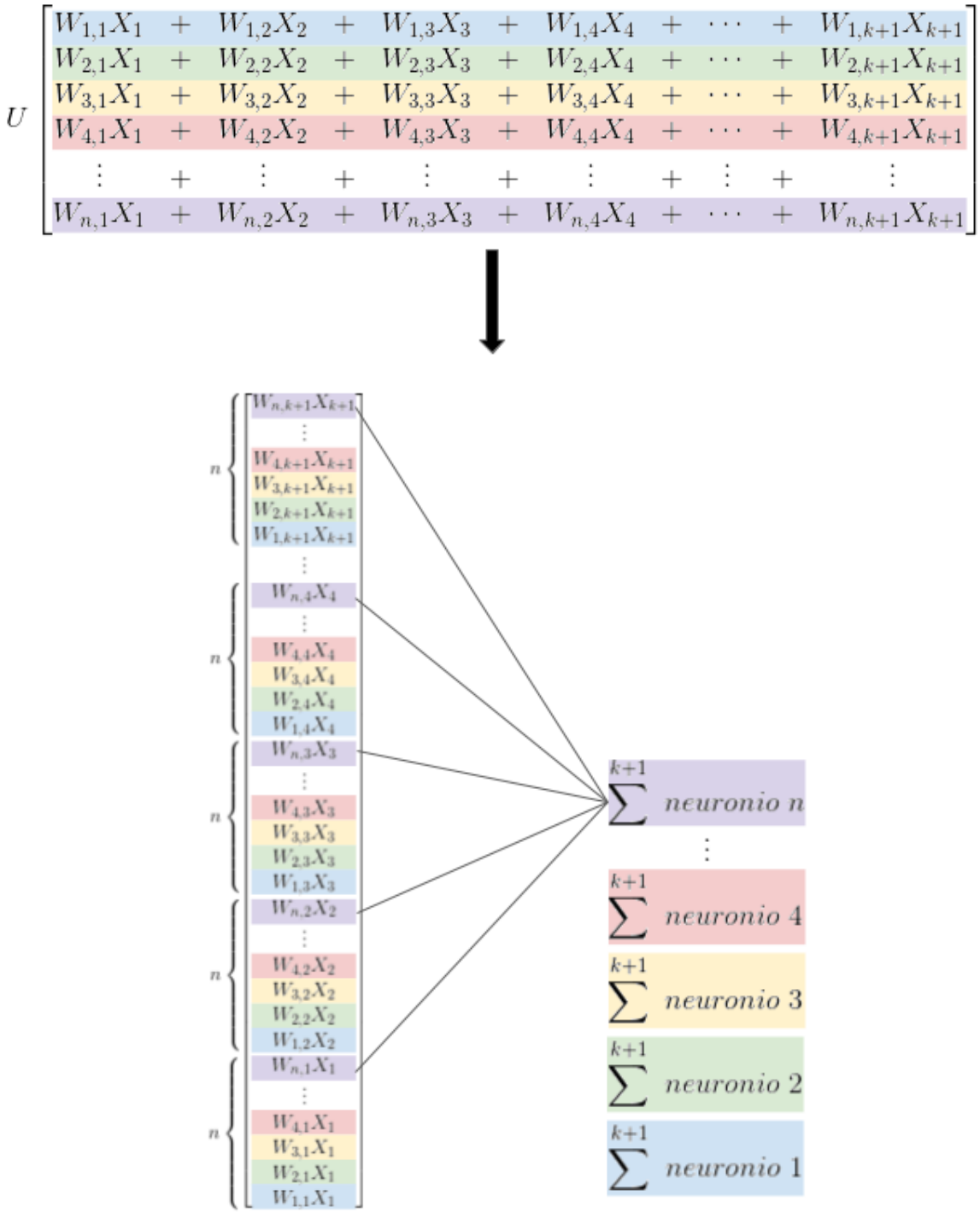
Baseado em uma solução de multiplicação de matrizes de alto desempenho proposta por (SOUZA, 2008), estamos propondo uma organização dos dados para a computação do estado de ativação que realiza o produto matricial utilizando uma coluna da matriz W por um único elemento da matriz X. Desta forma, precisamos uma única vez do valor de entrada da matriz X. Portanto, calculamos primeiro os produtos independentes entre todos os elementos de W que necessitam da entrada da matriz X, por exemplo $w_{(1,1)} \times x_{(1,1)}, \cdots, w_{(n,1)} \times x_{(1,1)}$ e em seguida serão realizadas as somas em uma organização que será explicada mais adiante.

Essa abordagem na organização dos dados possibilita que as entradas não precisem ficar armazenadas internamente no FPGA, o que proporciona uma redução do uso de memória interna do FPGA e facilita a implementação. Além disso, os dados de entrada da matriz X não precisam ser enviados todos de uma vez para o FPGA, o que reduz o número de pinos de entrada e saída do FPGA, bem como a largura de banda na comunicação com o FPGA.

A Figura 19 ilustra os dados da matriz U equacionada da equação 4.1 e como esses mesmos dados seriam organizados para realizar o *pipeline* da organização proposta para a multiplicação entre as matrizes de pesos W e de entradas X.

Observando a Figura 19, podemos notar que todas as operações de multiplicação necessárias para realizar a multiplicação entre as matrizes ficaram em uma sequência de entrada permitindo aplicar-se tal organização em um *pipeline*, reduzindo a necessidade de $k+1$ multiplicadores por camada, para um único módulo de multiplicação de ponto flutuante. Neste módulo os pesos da matriz W serão multiplicados pela entrada correspondente, desde que esta fique estável na entrada no multiplicador por n ciclos de (clock). Isto corresponde à quantidade de neurônios na camada e à quantidade de linhas da matriz W.

Figura 19 – Proposta de organização dos dados para a realização do pipeline na multiplicação de matrizes



Após realizar a multiplicação de cada elemento independente, é preciso somar-lo ao próximo elemento correspondente àquele neurônio e assim realizar a soma dos produtos de cada neurônio da camada. Na Figura 19, os elementos de mesma cor correspondem a um mesmo neurônio na camada que precisam ser somados, resultando finalmente na multiplicação de matrizes $U = W \times X$, como vimos na equação 4.2.

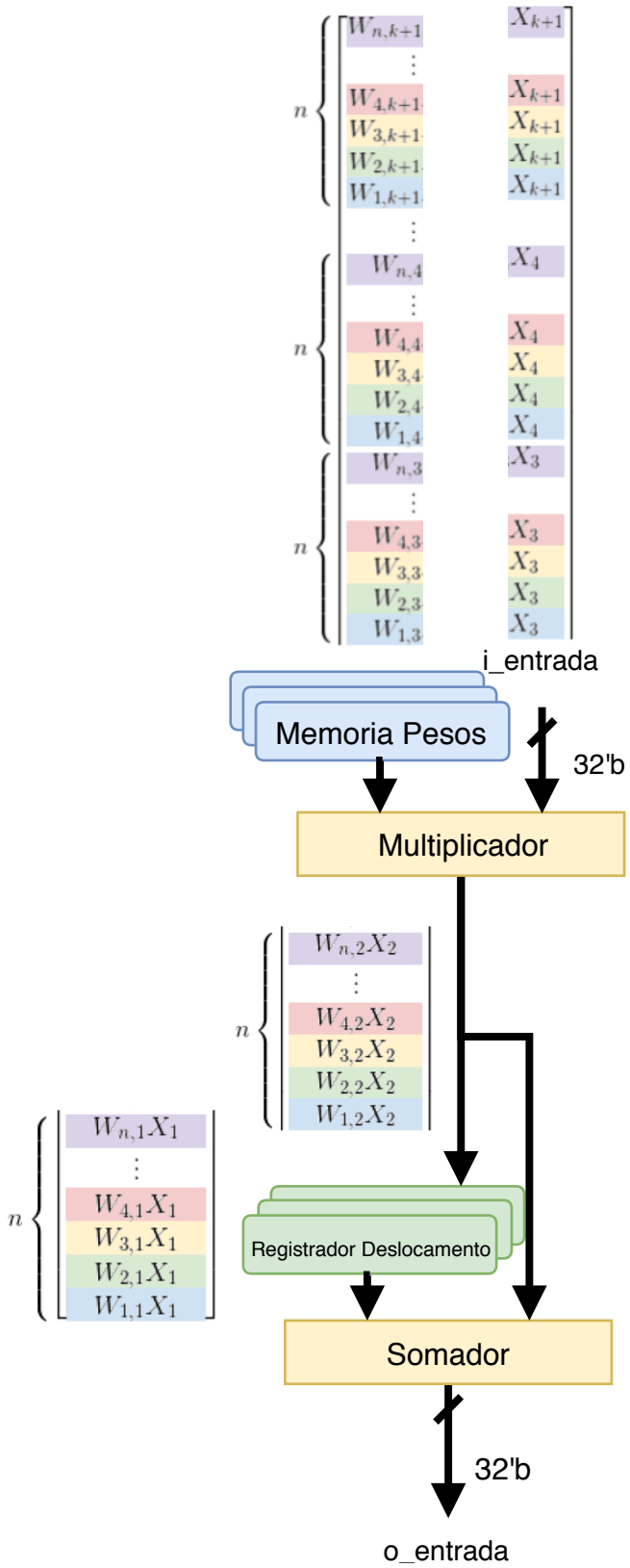
Para realizar a soma das multiplicações seguindo a organização proposta na Figura 19, é preciso criar uma estrutura de alocação e retardo do elemento a ser somado até o módulo de multiplicação gerar o próximo elemento. O leitor deve lembrar que os dados são organizados em sequência e são multiplicados em *pipeline* no módulo de multiplicação. Assim, o objetivo é manter esses dados em sequência, aproveitando a natureza do pipeline formado e realizar as somas. Para atender a essas necessidades, podemos criar um registrador de deslocamento para acumular esses dados e alinhar com o próximo a ser somado. A Figura 20 ilustra como a organização dos dados, ficariam com o pipeline cheio na entrada do primeiro somador.

A Figura 20, mostra que a quantidade de elementos que é preciso guardar e manter em sequência no registrador de descolamento é igual a n, quantidade de neurônios da camada. Nesta figura, o registrador de descolamento guarda o dado por n ciclos de *clock* até o próximo elemento a ser somado com ele ficar pronto. Mas esse tempo n de ciclos de *clock* só é possível para o primeiro estágio de soma da arquitetura. Esse tempo tende a crescer exponencialmente à medida que aumenta-se a quantidade de estágio de somas, necessários para somar todos os elementos de entrada. Isto ocorre porque a quantidade de somadores depende da quantidade de entradas k da camada.

A quantidade de somadores que será necessária para computar o estado de ativação de um neurônio artificial seguindo a abordagem proposta em *pipeline* é dado pela equação 4.3. Partindo do princípio que um somador faz a soma de elementos dois a dois, a cada estágio de soma a quantidade de elementos de entrada no estágio de soma é dividido por 2. Temos então que p será a quantidade de somadores necessários que satisfaz $2^p >= k$ para k entradas na camada que serão computadas pelos neurônios.

$$p = \lfloor log_2(k) \rfloor + 1 \tag{4.3}$$

Figura 20 – Organização dos dados proposto com o pipeline cheio na entrada do primeiro somador

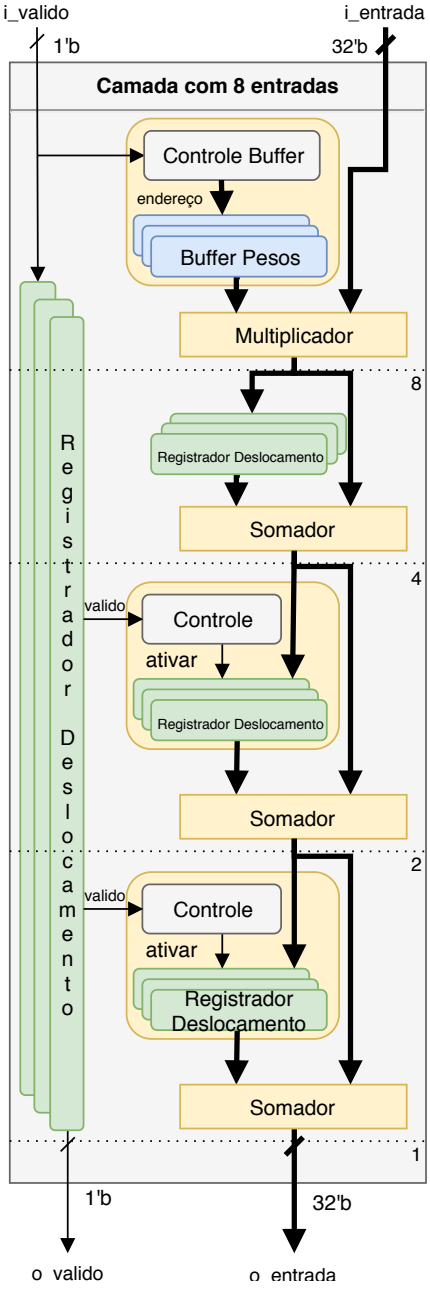


Na Figura 21 é apresentado um exemplo da arquitetura configurável para computar o

estado de ativação de uma camada com 8 entradas independente do número de neurônios. Considerando a organização dos dados proposta e seu processamento em *pipeline*, ao aplicarmos a equação 4.3 com $k = 8$ serão necessários $p = 4$ somadores.

Pode-se observar na Figura 21, que cada estágio da organização é separado por uma linha pontilhada. Ao lado temos a quantidade de entradas ou os elementos válidos daquele estágio, provenientes do estágio anterior. O objetivo é obter apenas um valor ao final que será o estado de ativação.

Figura 21 – Exemplo da arquitetura configurável do módulo Camada para a computação do estado de ativação com 8 entradas

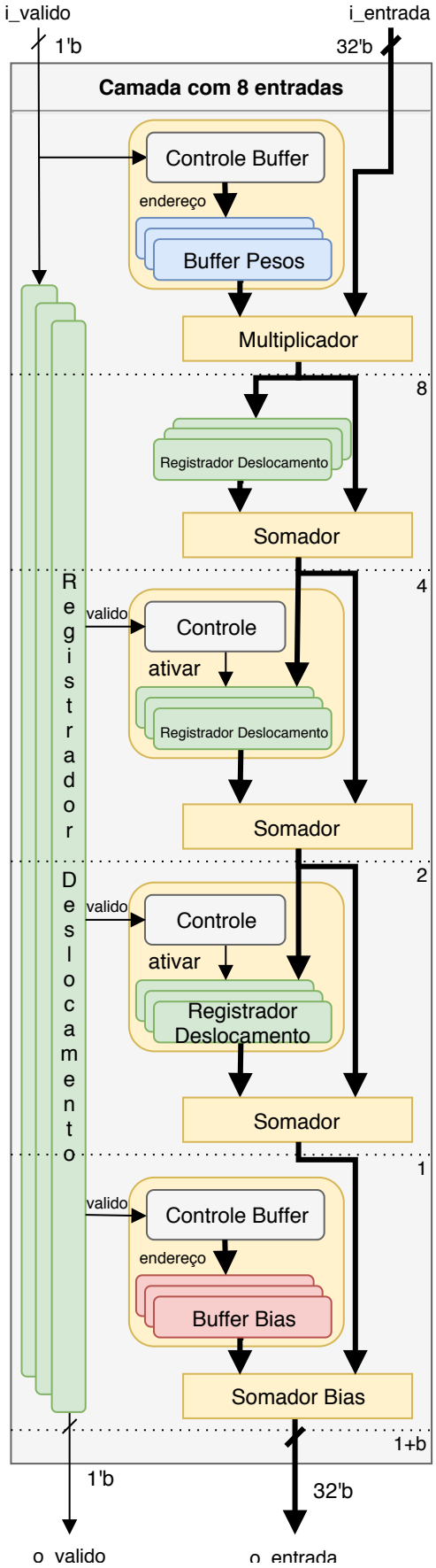


Acompanhando o fluxo de dados na Figura 21 temos que 8 entradas válidas serão multiplicadas pelos seus pesos no multiplicador de ponto flutuante, resultando, assim,

em 8 elementos válidos para o primeiro somador de ponto flutuante e para o registrador de deslocamento. Esses 8 elementos quando sincronizados serão somados dois a dois e resultarão em 4 elementos válidos. Essa organização de somadores se repete até o final do pipeline para a computação do estado de ativação. A equação 4.3 resultou em $p = 4$ somadores, enquanto que a Figura 21 só apresenta três somadores. Isso aconteceu pois neste exemplo o quarto somador será o somador do *bias*, que será exemplificado na Figura 22.

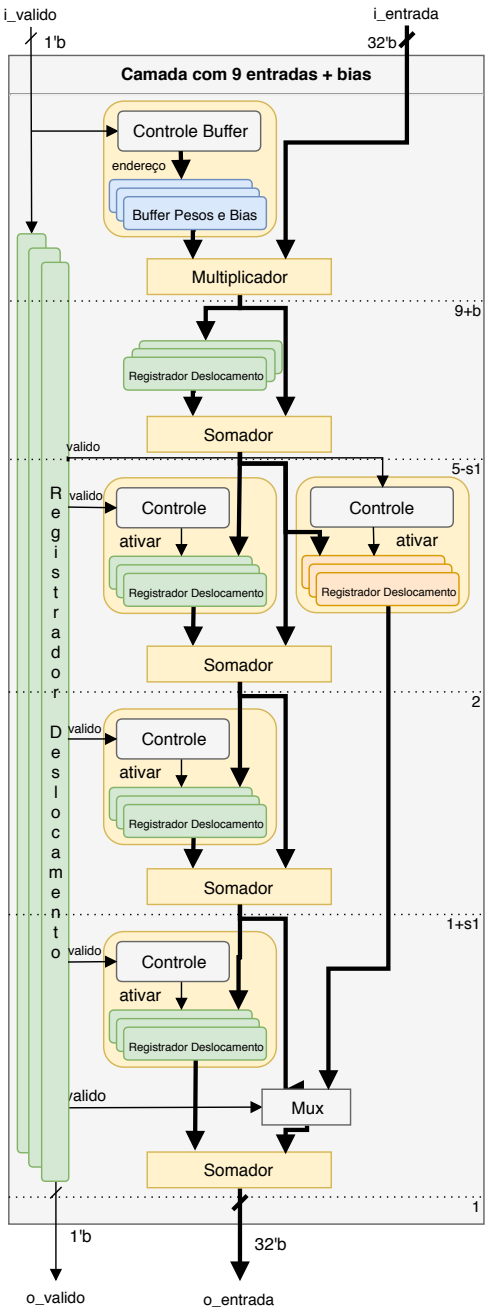
Como pode-se observar na Figura 22, o *bias* é um parâmetro fixo da arquitetura, que deve ser acrescentado aos pesos. Eles ficam armazenados em um *buffer* (memória interna do FPGA), sendo liberados quando o bit válido sinaliza que os pesos/*bias* estão alinhados com a entrada no estágio de multiplicação (no caso dos pesos/*bias*) e em qualquer outro estágio de soma da arquitetura, no caso do *bias*. A inserção do *bias* desta maneira ocorre para otimizar o tempo de execução da arquitetura em ciclos de *clock*, pois não será preciso adicionar um novo estágio de registrador de descolamento e somador. Neste caso, será apenas uma soma direta, no caso o quarto somador do exemplo com 8 entradas.

Figura 22 – Exemplo da arquitetura configurável do módulo Camada para a computação do estado de ativação com 8 entradas mais o *bias*



A arquitetura proposta neste trabalho tem seus estágios de soma configurados com base na quantidade de entradas k de uma camada, como explicado anteriormente. Como observado no exemplo anterior, tal configuração tem casos especiais com o objetivo de melhorar o desempenho geral da arquitetura. Além do caso já citado, outros casos possíveis serão exemplificados mais adiante.

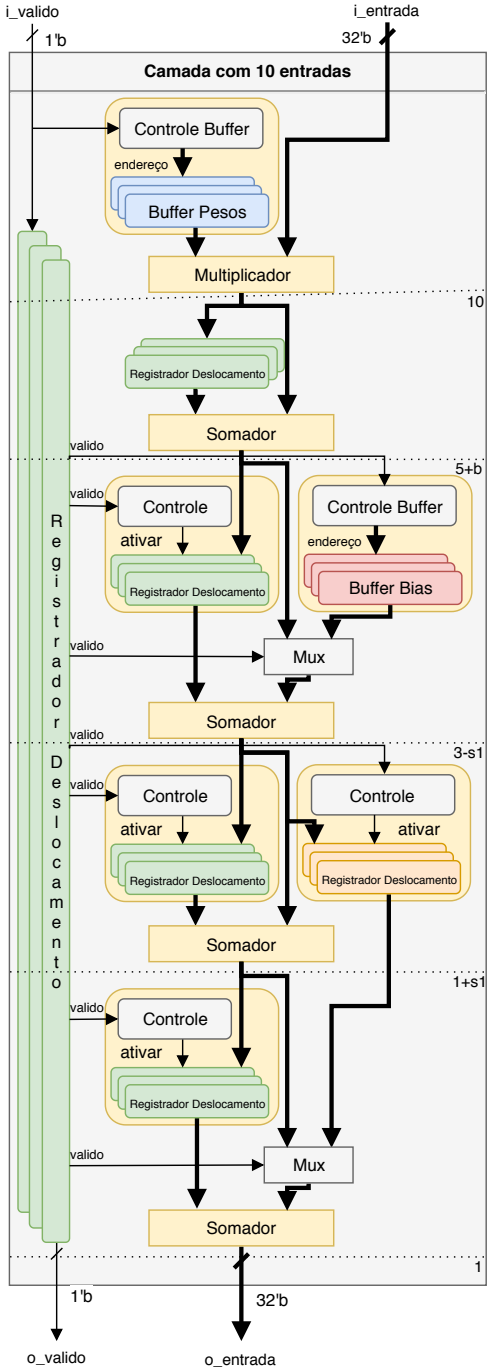
Figura 23 – Exemplo da arquitetura configurável do módulo Camada para a computação do estado de ativação com 9 entradas mais o *bias*



Uma informação importante que foi levada em consideração na definição dos estágios da arquitetura é se a camada tem a adição do *bias* ou não. Em algumas topologias de rede neural artificial não é necessário o *bias*. Geralmente isso é associado à função de ativação que será utilizada.

Para a definição de onde será inserido o *bias* na arquitetura, inicialmente é preciso verificar se a quantidade de entradas da camada k é impar ou par. Pois, como as somas serão feitas dois a dois, se a quantidade de entradas for impar, podemos inserir o *bias* já na entrada, totalizando $k+1$, como já foi discutido anteriormente. O *bias* terá o valor fixo em 1 na entrada para ser multiplicado pelo respectivo peso armazenado no *buffer* interno junto com os pesos já no estágio de multiplicação. Este caso pode ser observado na Figura 23.

Figura 24 – Exemplo da arquitetura configurável do módulo Camada para a computação do estado de ativação com 10 entradas



Se a quantidade de entradas k for par, existem duas possibilidades. Uma delas já foi exemplificada na Figura 22, onde o *bias* é somado ao final do processamento com um estágio de soma direta. A outra possibilidade é a inserção do *bias* em um estágio de soma, como pode-se observar na Figura 24. Neste modelo, a cada estágio da arquitetura verifica-se a quantidade de dados válidos para o próximo estágio é par ou ímpar; se for par, nada precisa ser feito e o fluxo continua igual ao fluxo exemplificado na Figura 22. Se for ímpar, este estágio será o local de inserção do *bias* no fluxo de dados, tornando a quantidade par novamente. Para isso um *mux* precisou ser incorporado ao caminho de dados para selecionar quando o *bias* deve ser inserido na entrada de um somador.

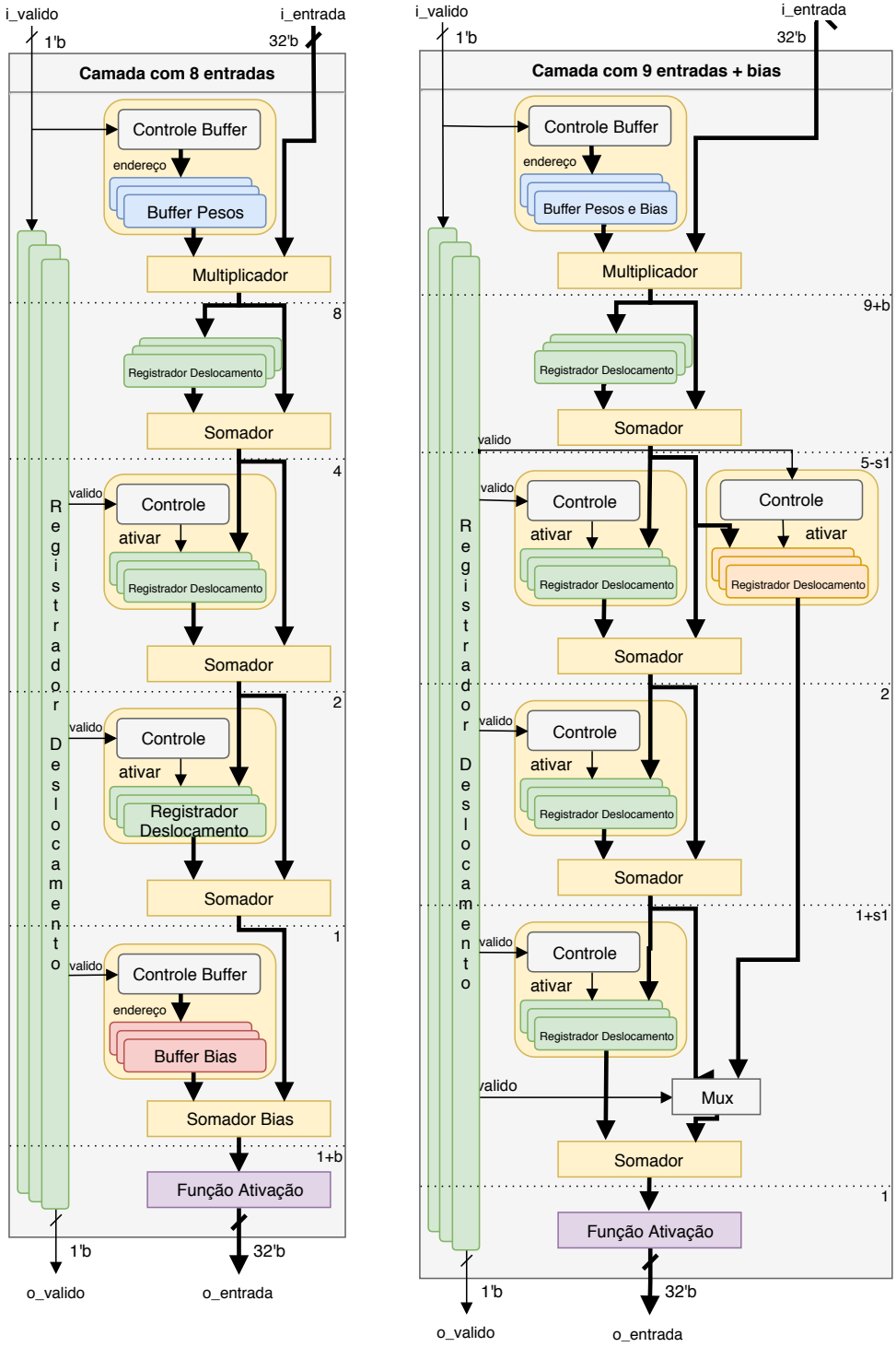
Quando a quantidade de dados válidos para o próximo estágio é ímpar, e o *bias* já foi inserido em algum momento anterior, então será preciso remover um elemento s da soma, tornando a quantidade par novamente. Este elemento removido será armazenado em um registrador de deslocamento e irá esperar um próximo estágio de soma que tenha quantidade de elementos ímpar novamente, para devolver este elemento s no processamento. Este processo é chamado de *salto* ou *bypass*. Para fazer isso, de maneira similar ao *bias* inserido em um estágio de soma, um *mux* é incorporado no caminho de dados para selecionar quando for o elemento deve ser inserido no estágio de soma. Pode-se observar esses saltos nos exemplos das Figuas 23 e 24. Dependendo da quantidade de entradas k da arquitetura, vários saltos podem acontecer.

4.2 Função de Ativação

Com o objetivo de desenvolver uma arquitetura configurável, mas com desempenho eficiente em FPGA, manteve-se a organização dos dados e arquitetura propostos na seção anterior, ou seja a função de ativação foi projetada para ser mais um módulo do *pipeline*, seguindo o fluxo sequencial dos dados ao final da computação do estado de ativação. Desta forma foi possível manter a arquitetura simples e configurável para diferentes topologias de redes neurais artificiais.

A Figura 25 mostra dois exemplos já apresentados na seção anterior, mas agora com a inserção do módulo de *Função Ativação* no final do pipeline, recebendo como entrada a computação do estado de ativação do neurônio. A saída do módulo de *Função Ativação* será o resultado final do impulso nervoso dos neurônios da camada, acompanhado do *bit* de válido. O *bit* de válido percorre todo o pipeline através de um registrado de deslocamento de tamanho igual a quantidade de ciclos de *clock* que toda camada leva para realizar a computação dos neurônios.

Figura 25 – Exemplos da arquitetura configurável do módulo Camada para a computação do estado de ativação com 8 e 9 entradas mais a adição da função de ativação na organização proposta



Como mencionado na subseção 2.1.2, várias funções podem ser usadas como função de ativação do neurônio artificial. Durante o projeto da topologia das RNAs-MLP Profundas que serão utilizadas na solução dos conjuntos de dados MNIST e Fashin-MINST, os melhores resultados foram obtidos com as funções não-linear, Sigmoide e Tangente Hiperbólica. Portando elas serão a base da arquitetura apresentada nesta seção. Algu-

mas implementações dessas funções complexas foram propostas por (ZHANG; VASSILIADIS; DELGADO-FRIAS, 1996), (BASTERRETXEA; TARELA; CAMPO, 2004), (AMIN; CURTIS; HAYES-GILL, 1997), onde a implementação direta em hardware destas funções requerem muitos recursos de hardware do FPGA.

Baseado no trabalho de (FERREIRA; BARROS, 2010), que verificou quatro possibilidades de implementação de um módulo Sigmoide ou Tangente Hiperbólica, este trabalho irá se basear em um dos métodos propostos que consiste em uma tabela de consulta armazenada na memória interna do FPGA. Sendo assim, é proposto um método de aproximação de função de ativação que possibilite fazer uma distribuição de seus valores possíveis dentro de um intervalo pré-definido em uma tabela de consulta. Nesse caso, a precisão da solução pode ser melhorada com o aumento dos valores armazenados na tabela de consulta. Portanto, a precisão pode ser dimensionada dependendo dos recursos de hardware disponíveis no FPGA. Como o tempo de acesso à tabela de consulta é uma possível desvantagem dessa solução, para superar esse gargalo de acesso foi projetada uma estratégia em *pipeline*, assim como toda a arquitetura, para acessar a tabela de consulta mais rapidamente.

O método proposto consiste em armazenar os valores da função de ativação (podendo ser qualquer função, no caso, Sigmoide ou Tangente Hiperbólica) em uma tabela. A tabela terá indexação escalável considerando os valores de u, onde u é a computação do estado de ativação de um neurônio. Apesar deste método ser bem simples, ele não parece ser eficiente no consumo de recursos. No entanto, em nossa proposta, a precisão da representação da função pode ser ajustável pelo tamanho da tabela, onde, mais precisão necessita de mais memória para armazenar mais valores. Desta forma, oferecemos ao projetista uma precisão customizável.

Como citado anteriormente, a organização dos dados da arquitetura proposta de uma camada de neurônios possibilita computar o estado de ativação u de todos os neurônios da camada em sequência, permitindo usar um único módulo de *Função Ativação*. Este módulo implementa a função com o método baseado em tabela de consulta. Neste caso a tabela é usada para calcular a função de ativação de uma camada inteira, em vez de usar uma tabela de consulta para cada um dos neurônios, o que inviabilizaria o uso do método pelo consumo excessivo de memória interna do FPGA, como já citado no capítulo 3. Os trabalhos lá relacionados tiveram problemas no consumo de recursos excessivo do FPGA, e um dos motivos observados, foi a implementação da função de ativação para cada neurônio individualmente.

Para realizar a consulta na tabela mais rapidamente, considerando o método proposto, temos que a computação do estado de ativação de um neurônio tem valor u, e y é o valor em ponto flutuante correspondente na função de ativação, isto é, dada uma entrada u no eixo x do gráfico da função, obtemos y. Então, o acesso rápido à tabela de consulta foi obtido representando u em um intervalo fechado $[a, b]$, no eixo x com $a < b$. Usando

Tabela 3 – Exemplo de aplicação de acesso a tabela de consulta

índice	1	2	3	4	5	6	7	8	9
valor	-1	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1

um valor de incremento de $\frac{1}{i}$, onde i corresponde ao grau deslocamento entre um valor e outro representado da tabela de consulta. Considera-se então a Equação 4.4, para calcular quantos l valores da função de ativação serão armazenados.

$$l = (b - a) \cdot i + 1 \tag{4.4}$$

Os valores, são armazenados em um vetor $Y_{(1,l)}$ que representa a tabela de consulta, e são obtidos fazendo-se a distribuição matemática de y no intervalo $[a, b]$ em l valores para cada de posição q do vetor Y.

A posição q na tabela de consultas é obtida a partir da equação 4.5. Onde, para um dado valor u, o índice correspondente q da tabela de consulta Y é obtido.

$$q = \lfloor (u - a) \cdot i + 1 \rfloor \tag{4.5}$$

Para fins de ilustração, um exemplo dessa estratégia pode ser visto na Tabela 3, onde o intervalo fechado $[-1, 1]$ foi definido com o deslocamento $\frac{1}{i} = \frac{1}{4}$, aplicando a Equação 4.4. Observa-se que a tabela de consultas tem $l = (1 - (-1)) \cdot 4 + 1 = 9$ posições.

Desta mesma forma, dado um valor $u = 0$ no eixo x, aplicando a Equação 4.5, temos $q = \lfloor (0 - (-1)) \cdot 4 + 1 \rfloor = 5$; Igualmente, para um $u = 0.3$ no eixo x, obtemos o índice $q = \lfloor (0.3 - (-1)).4 + 1 \rfloor = \lfloor 6.2 \rfloor = 6$.

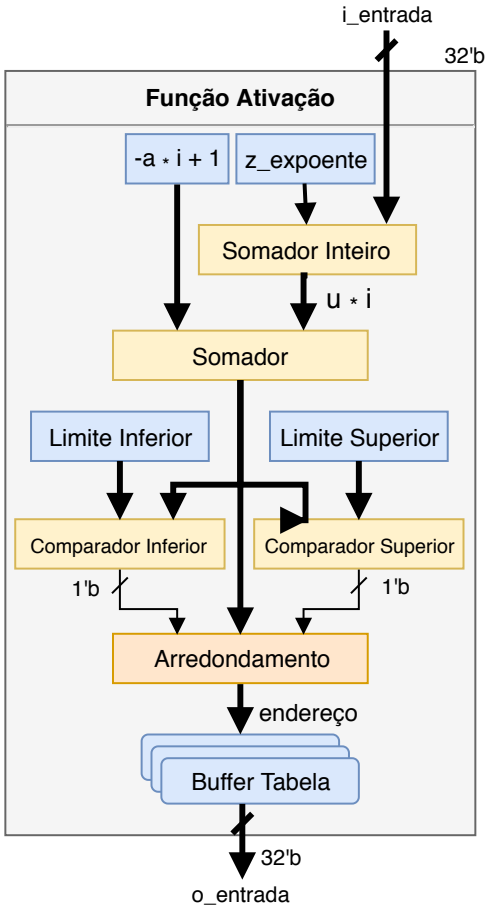
Para reduzir a complexidade de hardware para o cálculo da Equação 4.5, podemos reescrever como a Equação 4.6. O termo $-a \cdot i + 1$ é uma constante e pode ser pré-computado em software e utilizado como parâmetro, juntamente com todos os valores que irão compor a tabela de consultas y. Além disso, pode-se utilizar o valor de deslocamento i como sendo uma potência de 2, onde $i = 2^z$. Desta forma é possível concluir que o produto $u \cdot i$ pode ser computado em hardware através da soma do inteiro z no expoente de ponto flutuante de u.

$$q = \lfloor -a \cdot i + 1 + u \cdot i \rfloor \tag{4.6}$$

A Figura 26 apresenta o *pipeline* proposto para o módulo da *Função Ativação*, baseado no método proposto, explicado anteriormente. Este módulo realiza inicialmente uma soma de inteiros de 8*bits*, ou seja, a soma do expoente do ponto flutuante u de entrada com o parâmetro constante *z_expoente*. Em seguida, o resultado desta operação, correspondente ao produto $u \cdot i$ precisa ser somado, em ponto flutuante, com o parâmetro constante $-a \cdot i + 1$. O resultado desta soma, precisa ser comparado para verificar se o valor resultando passou do limite superior ou inferior da tabela de consultas Y. Ao final

é feito o arrendondamento para baixo do valor resultante, obtendo assim a posição q do índice da tabela. Tal índice é o endereço do *buffer* de memória, onde a tabela de consulta foi armazenada.

Figura 26 – Exemplo da arquitetura parametrizável do módulo Função Ativação para a computação do impulso nervoso do neurônio artificial, utilizando a organização de dados e método proposto



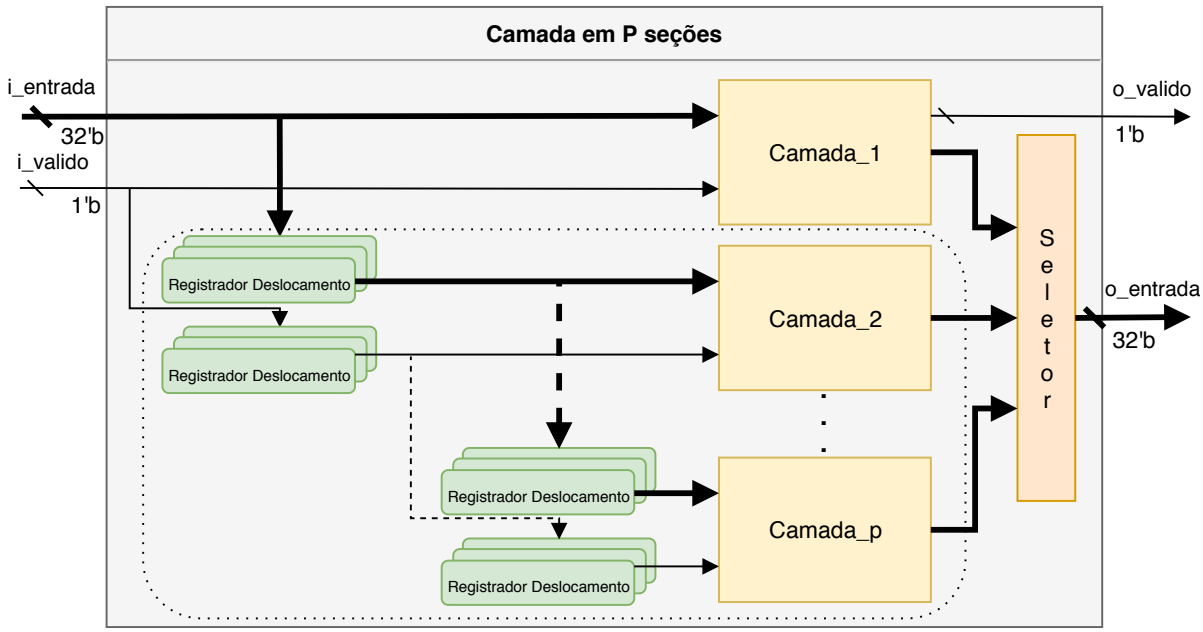
4.3 Otimização do Desempenho por Camada

Como citado no início deste capítulo, uma camada é configurada para receber um conjunto de k entradas em sequência, onde, cada entrada deve ficar estável/fixa por n ciclos de *clock*, que é o número de neurônios da camada. Assim sendo, uma camada recebe uma entrada proveniente de uma camada anterior e mantém essa entrada estável por n ciclos de *clock*, fazendo isso para as k entradas, o que resulta em um total de $n \times k$ entradas sequenciais. Ou seja, seriam necessários, no mínimo, $n \times k + t$ ciclos de *clock* para processar esses dados, considerando t como o tempo de execução da camada no pipeline. Vale lembrar que todos os estágios de alinhamento interno dos dados da computação da camada consideram a quantidade de neurônios n(por exemplo a profundidade dos registradores de deslocamento entre as somas).

Pode-se então otimizar o tempo de processamento (em ciclos de *clock*) de uma camada, dividindo a mesma em p seções de camadas, onde cada camada seccionada ficaria responsável por computar $\frac{n}{p}$ neurônios. A Figura 27 mostra uma visão geral de como ficaria o caminho de dados da arquitetura em pipeline desta distribuição da camada em p seções. Observa-se que para cada seção p da camada, existe um registrador de deslocamento para retardar, por $\frac{n}{p}$ ciclos de *clock*, os dados de entrada e *bit* de válido proveniente da interconexão com a camada anterior da rede. Isso precisa ser feito para que a camada distribuída em p seções apresente o resultado da computação ao final em sequência, através de um seletor que seleciona qual será a saída da camada para as suas p seções, por $\frac{n}{p}$ ciclos. Tal ação se faz necessária, pois se todas as seções iniciassem seu processamento ao mesmo tempo, o que seria possível, todas apresentariam o resultado ao mesmo tempo, o que iria impactar na interconexão com a próxima camada.

A possibilidade de distribuir uma camada em p seções, resulta em uma otimização de $\frac{n \times k + t}{p}$ ciclos de *clock* de processamento de uma camada, menos o tempo de alinhamento das seções. O projetista precisa considerar o compromisso entre desempenho em tempo de execução e o uso de recursos do FPGA, pois ao dividir a camada em p seções estamos multiplicando por p a quantidade de recursos consumidos naquela camada. Essa distribuição da camada em p seções é configurável na arquitetura para todas as camadas se assim for preciso, o que facilita ao projetista realizar testes em busca do melhor compromisso tempo de execução e área do FPGA.

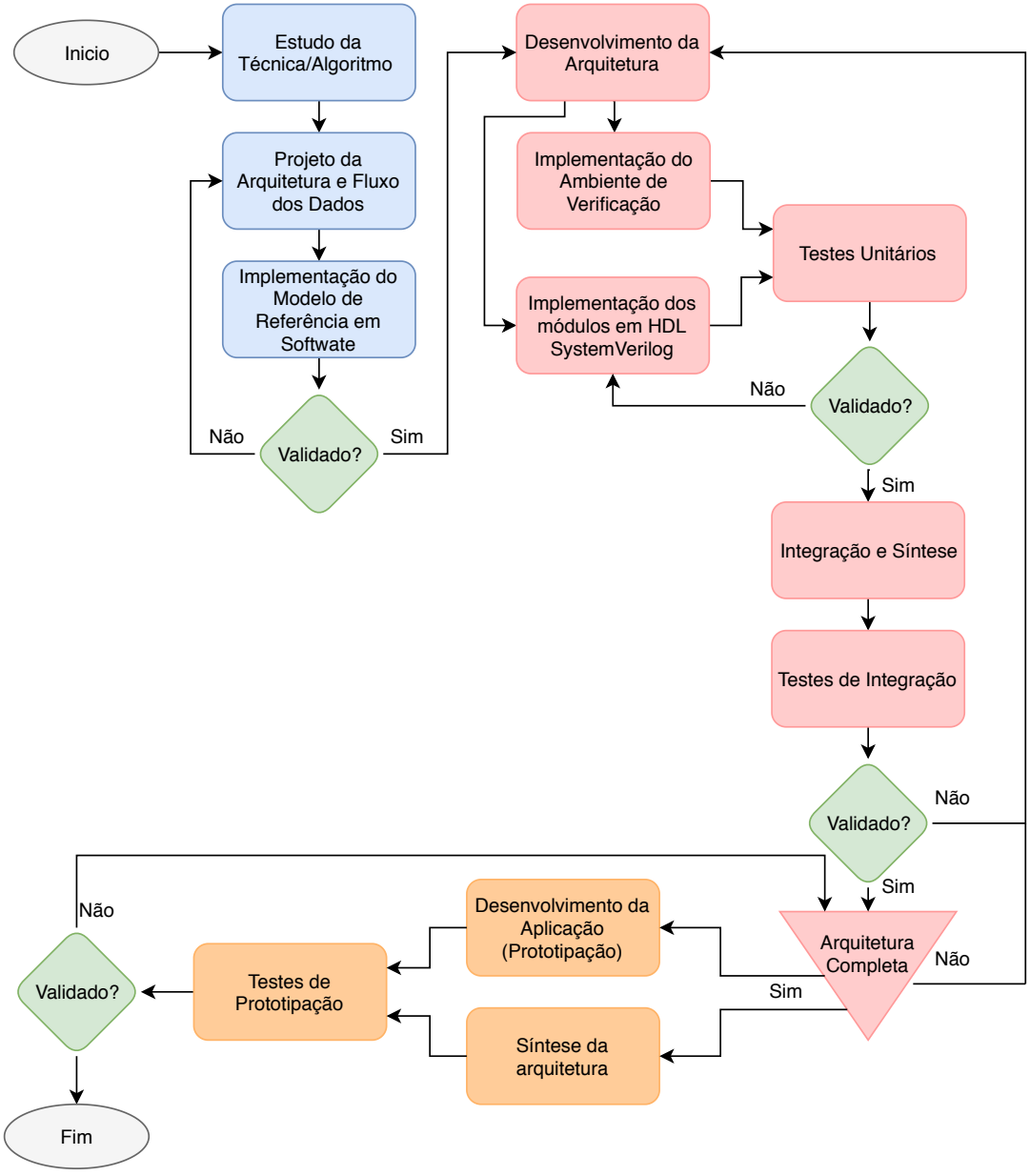
Figura 27 – Exemplo do módulo camada dividido em p seções com n/p neurônios



5 IMPLEMENTAÇÃO

Para a construção da arquitetura proposta neste trabalho e apresentada no Capítulo 4, foi usado um fluxo de desenvolvimento e implementação afim de garantir a prototipação do módulo em FPGA e sua validação com conjunto de dados públicos usados pela comunidade científica. A Figura 28 ilustra o diagrama de blocos do fluxo de desenvolvimento. O fluxo foi planejado considerando três conjuntos de etapas importantes: a etapa do desenvolvimento de software (blocos em azul), etapa do desenvolvimento do hardware (blocos em vermelho) e a etapa de integração de hardware e software (blocos em laranja). Estas três etapas, foram seguidas ao longo deste trabalho de mestrado.

Figura 28 – Fluxo de desenvolvimento e validação do trabalho proposto



5.1 Desenvolvimento em Software

Inicialmente, pode-se observar no fluxo adotado neste trabalho, foram realizados estudos e pesquisas a respeito das Redes Neurais Artificiais. Tal estudo, fundamentou a escrita da seção 2.1 no Capitulo 2.

Ainda na etapa de estudo, com o conhecimento teórico em mente, foi implementada uma rede neural artificial, utilizando-se uma biblioteca bem consolidada nos dias atuais, a biblioteca Keras.

O ambiente de desenvolvimento Keras é uma biblioteca de rede neural de código aberto, escrita na linguagem de programação Python. O Keras foi projetado para permitir a rápida experimentação de redes neurais artificiais profundas, facilitando a modelagem e execução das mesmas em plataformas baseadas em processadores de propósito geral e GPUs. O Keras oferece um conjunto de abstrações em um nível mais intuitivo que facilita o desenvolvimento de modelos de aprendizagem, independentemente do mecanismo de baixo nível computacional usado (KERAS, 2019).

O objetivo na implementação em alto nível foi entender na prática o funcionamento de uma rede neural artificial, suas equações e algoritmos, criando assim um conhecimento sólido para o projeto da arquitetura. Além disso, o trabalho proposto nesta dissertação foi desenvolver uma arquitetura configurável para execução de redes neurais em dispositivos FPGAs, para as mais diversas aplicações, o que não envolve fazer o treinamento (aprendizagem da rede, aquisição de pesos e *bias*) e definição de qual a melhor topologia de rede para a aplicação. Essas tarefas devem ser realizadas pelo projetista de Inteligencia artificial, utilizando bibliotecas e ambientes como o Keras.

Assim sendo, para se projetar uma arquitetura configurável em hardware, foi preciso definir qual seria uma rede neural artificial perceptron de multicamadas de referência para validar-se o funcionamento da arquitetura proposta. Foi realizado, então, um levantamento dos conjuntos de dados públicos e comumente usados para se validar uma rede neural artificial profunda. Os conjuntos de dados públicos escolhidos para este trabalho foram o MNIST(dígitos manuscritos) e o Fashion-MNIST(itens de vestuário). Estes conjuntos de dados serão explicados no capítulo 6.

Uma vez escolhidos os conjuntos de dados, então foram projetas redes neurais artificiais perceptron multicamadas em Python usando o Keras para cada conjunto de dados. Para isto foram realizados os treinamentos para a aquisição dos pesos e *bias* das redes. Estes modelos e seus parâmetros foram utilizados como referência para realizar as etapas de projeto da arquitetura proposta e seu fluxo de dado, seguido da implementação do modelo de referência da arquitetura em software.

A arquitetura e seu fluxo de dados, organização dos dados internos, foram projetados primeiramente como diagrama de blocos, diagramas esses apresentados nos exemplo do capítulo 4. Um dos requisitos básicos para a definição da arquitetura foi ter uma largura de banda pequena e interface de comunicação simples, possibilitando seu uso em diversas

plataformas de FPGAs. Outro ponto importante levado em consideração, também explicado várias vezes no capítulo 4, foi tentar otimizar o uso de recursos no FPGA, para suportar a implementação nas diversas plataformas de FPGAs no mercado.

A arquitetura projetada foi implementada em C/C++, seguindo a organização e fluxo de dados proposto neste trabalho. Tal implementação foi validada em comparação com os resultados esperados da mesma rede neural desenvolvida em Python com o Keras. Caso algum problema/erro fosse detectado na validação, retornava-se para o projeto e implementação do modelo de referência para corrigir o erro.

Ao final do desenvolvimento em software obtivemos o projeto de arquitetura reconfigurável para execução de uma rede neural artificial perceptron multicamadas, bem como uma implementação de referência em C/C++ da arquitetura. Além disso, tivemos o projeto e implementação de redes neurais artificiais para dois conjuntos de dados distintos em CPU e GPU que nortearam os resultados deste trabalho, que será apresentado no próximo capitulo.

5.2 Desenvolvimento em Hardware

Após a conclusão do desenvolvimento em software com a validação do projeto da arquitetura usando o modelo de referência desenvolvido, foi iniciada a implementação em hardware. As etapas do desenvolvimento em hardware podem ser vistas no fluxo da Figura 28.

O desenvolvimento da arquitetura foi dividido em duas etapas: implementação dos módulos que compõe a arquitetura usando a linguagem de descrição de hardware SystemVerilog (SPEAR, 2008), e a implementação do ambiente de verificação destes respectivos módulos, também em SystemVerilog. A escolha de qual módulo seria implementado primeiro seguiu uma abordagem *bottom-up*, começando pelos módulos mais internos até a conclusão do módulo principal, chamado de *Rede*, que instancia as camadas e interconexões da rede neural artificial. A implementação de todos os módulos individualmente será explicada ao final desta seção, na subseção 5.2.1.

Para validar a implementação dos módulos da arquitetura foi preciso implementar um ambiente de validação para realizar os testes de cada módulo individualmente. Este ambiente de testes recebe um conjunto de entradas em ponto flutuante, pré-processados em forma de arquivo, e os envia para a entrada do módulo sequencialmente. Esta abordagem permitiu encontrar erros cruciais na sincronização do pipeline da arquitetura. A fim de cobrir os diversos tipos de cenários, além de utilizar entradas reais geradas nas etapas da seção anterior, foram introduzidos dados em um certo grau de aleatoriedade nas entradas a serem verificadas.

O resultados do módulo implementado foram armazenados durante os testes em arquivos. Os dados de cada arquivo de saída foram comparados por meio de um comparador de arquivos, com os arquivos gerados pelo modelo de referência em C++ do respectivo

módulo. Se todos os arquivos de saída estivessem iguais aos arquivos do modelo então a implementação estaria validada. O módulo validado pode ser integrado e sintetizado à arquitetura. Testes de integração foram feitos e as etapas se repetiram até a conclusão do desenvolvimento da arquitetura.

A ferramenta utilizada para avaliar, debugar e validar a implementação do ambiente de verificação foi o ModelSim (GRAPHICS, 2007). O ModelSim oferece uma maneira de gerar sinais de entrada a partir de linguagens de alto nível tal como SystemVerilog e permite visualizar todos os sinais internos e externos de todos os módulos implementados através de formas de onda. Através desta ferramenta a implementação foi avaliada primeiramente por meio de simulação funcional, na qual o módulo é executado e avaliado apenas através da funcionalidade descrita em SystemVerilog. Dessa forma, diversos erros puramente de funcionalidade puderam ser rapidamente detectados e corrigidos neste tipo de simulação.

A simulação funcional não leva em consideração atrasos na propagação dos sinais elétricos no FPGA. Para isso, existe a simulação temporal, que leva em consideração o comportamento, o tempo, além da propagação de todos os atrasos permitindo uma simulação mais próxima do que acontece quando o módulo é executado no FPGA, revelando vários problemas de implementação que não foram capturados em simulação funcional. Uma especificação pode ser validada em simulação funcional, mas devido a atrasos de propagação dos sinais no FPGA, pode não funcionar corretamente em simulação temporal.

Para realizar a simulação temporal é necessário que o código seja primeiramente verificado e sintetizado, utilizando uma ferramenta de síntese para geração de de uma implementação a nível de portas lógicas e arquivos de reprogramação do FPGA. Ao longo do desenvolvimento deste trabalho, foi utilizada a ferramenta Intel Quartus Prime, versão 18.1 Standard Edition. Ao final da verificação e síntese dois arquivos foram gerados: o primeiro com extensão .svo é a implementação do módulo a nível de portas lógicas; o segundo arquivo tem extensão .sdo e define o atraso de propagação de cada porta lógica de acordo com o FPGA escolhido. Estes arquivos são importados na ferramenta ModelSim e integrados em um ambiente de verificação para realização dos testes.

Ao longo deste trabalho o FPGA adotado como referência foi o Stratix IV Intel FPGA[1] (EP4SGX530KH40C2). Este FPGA é integrado na plataforma acadêmica Terasic DE4(TERASIC, 2019), no formato de uma placa com conexão PCI-Express 2.0, podendo ser conectada a qualquer computador pessoal com suporte a PCI-Express 2.0 em sua placa mãe.

Esta plataforma acadêmica possibilita a criação de uma arquitetura heterogênea que inclui processador e FPGA. Desta forma é possível ter um ambiente onde aplicações com redes neurais artificiais sejam implementadas no processador usando linguagens de alto

[1] especificação pode ser encontrada em https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literat
iv-product-table.pdf

nível como C++, Python e sejam interligadas, através de um protocolo de comunicação com o barramento PCI-Express, com o módulo de hardware que compõe a arquitetura proposta neste trabalho. Este tipo de plataforma pode reduzir o tempo de desenvolvimento do projeto, além de possibilitar o projeto de sistemas com melhor desempenho para aplicações com redes neurais de tempo real, por exemplo, aplicações de processamento de vídeo que exigem taxa de processamento acima de 30 FPS.

5.2.1 Módulos de hardware implementados

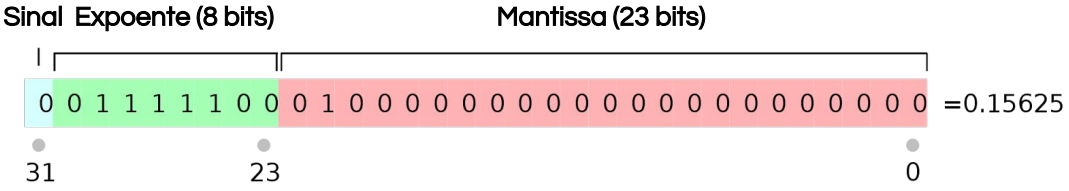
Nesta seção, os detalhes a respeito da implementação da arquitetura proposta no capítulo 4 serão apresentados. Todos os módulos serão explicados seguindo a mesma abordagem de implementação *bottom-up*, começando pelos módulos mais internos até a conclusão do módulo principal, chamado de *Rede*, que instancia as camadas e interconexões da RNA-MLP. Toda a implementação foi feita na linguagem de descrição de hardware SystemVerilog.

5.2.1.1 Multiplicador e Somador de Ponto Flutuante 32*bits*

Devido à presença dos números reais na computação de uma RNA, principalmente na computação da função de ativação dos neurônios, foi necessário decidir se seria usada a representação de números reais em ponto fixo ou ponto flutuante. A definição da representação numérica em hardware adotada pela arquitetura é de extrema importância para um boa precisão na computação dos impulsos nervosos que compõe a RNA. Desta forma buscou-se encontrar uma representação que tivesse uma maior fidelidade com o modelo das RNAs definido em software. Na implementação da arquitetura proposta optou-se pelo uso do ponto flutuante de 32*bits* de precisão simples.

Brevemente, o ponto flutuante é um padrão de representação numérica definido pelo Instituto dos Engenheiros Elétricos e Eletrônicos (IEEE) sob o padrão IEEE 754. A representação em ponto flutuante é similar à notação científica em que existe um número multiplicado por sua base elevada a um expoente. O maior benefício desta representação é que ela provê vários graus de precisão baseados na escala dos números que se está usando. O padrão IEEE 754 define representações tanto para ponto flutuante de precisão simples (32 bits) como de dupla precisão (64 bits), bem como para suas versões estendidas. Conforme mostrado na Figura 29, um número representado em precisão simples tem o bit de sinal, um campo de expoente, e a mantissa.

Figura 29 – Representação em ponto flutuante de precisão simples 32 *bits*



O expoente é representado como um número de 8 bits resultando em um intervalo de -126 até 127. Na verdade, o expoente não está na representação típica de complemento a 2, ao invés disso, ele é enviesado, adicionando o valor 127 ao expoente desejado. Isto permite representar números com expoente negativos. Neste padrão, a mantissa está na representação binária normalizada do número a ser multiplicado pela base 2 elevado a potência definida no expoente.

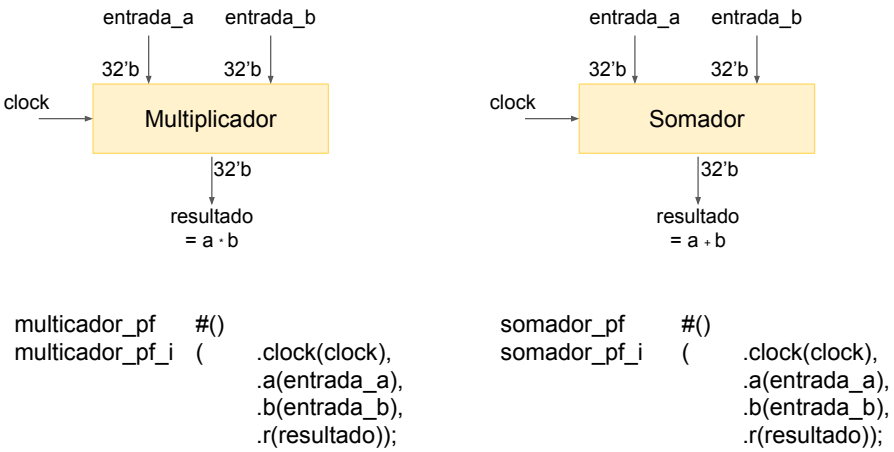
Além da escolha da representação numérica foi preciso definir como seriam feitas as duas operações aritméticas de ponto flutuante necessárias na arquitetura apresentada no capítulo 4: Multiplicação e Soma de dois números em ponto flutuante.

A Intel FPGA dispõe de componentes de aritmética de ponto flutuante em sua biblioteca de Módulos de Propriedade Intelectual (IP). Atualmente estão disponíveis componentes para precisão simples, precisão dupla e para precisão simples estendida. O uso do componente disponibilizado pela fabricante do FPGA utilizado agregou maior confiabilidade ao fluxo de desenvolvimento da arquitetura visto que esses componentes já estão validados. Desta forma usamos os módulos somador/subtrator (Intel ALTFP_ADD_SUB), multiplicador (Intel ALTFP_MULT) ambos de precisão simples e do padrão IEEE 754 da Intel FPGA. A Figura 30 apresenta uma ilustração dos módulos e suas interfaces de conexão, além do código exemplo de instanciação dos módulos.

A profundidade do pipeline dos componentes pode ser selecionada. O pipeline do somador possui 14 ciclos, enquanto que a do multiplicador em 11 ciclos, ambas no máximo disponível. A escolha do tamanho máximo do pipeline para os componentes se reflete no aumento da frequência máxima de operação desses módulos. Este fato se deve a quebra das operações em mais estágios de pipeline internos do módulo reduzindo o caminho crítico em cada estágio. Assim, nessa configuração a frequência de operação pode atingir

o seu máximo adequado para cada dispositivo FPGA.

Figura 30 – Módulos de Multiplicador e Somador de ponto flutuante de precisão simples e exemplo de instanciação em SystemVerilog




```
multicador_pf     #()
multicador_pf_i   (      .clock(clock),
                         .a(entrada_a),
                         .b(entrada_b),
                         .r(resultado));
```

```
somador_pf     #()
somador_pf_i   (      .clock(clock),
                      .a(entrada_a),
                      .b(entrada_b),
                      .r(resultado));
```

5.2.1.2 Registrador de Deslocamento e Controle

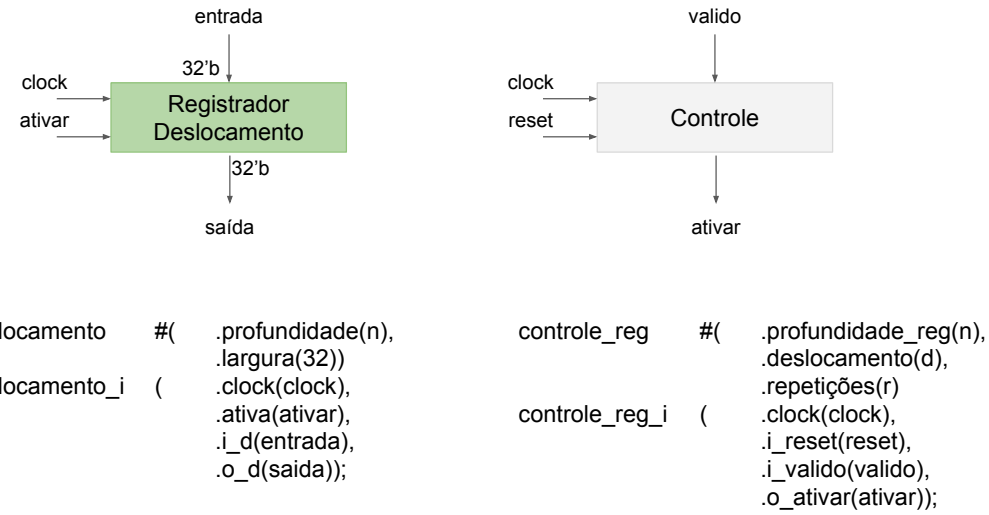
O módulo registrador de deslocamento pode ser considerado como uma sequência de registradores interligados um ao outro em forma de fila. Esta estrutura reflete a estrutura de uma rede neural, onde um dado de entrada precisa percorrer todos os n neurônios da camada. Neste caso, os dados percorrem os registradores da fila até a saída do módulo, enquanto o sinal de ativar o módulo estiver positivo, validando, assim, este deslocamento dos dados.

O registrador foi implementado de forma genérica e parametrizável na largura do dado a ser registrado e na profundidade n da sequência ou fila de registradores. A Figura 31 ilustra as conexões e instanciação do módulo, que pode ter n dados de largura fixa de 32 bits, tamanho dos dados em ponto flutuante.

O registrador de deslocamento foi implementado sem nenhum controle, apenas uma fila de registradores interligado, onde um dado de entrada é registrado no início da fila e esse dado é deslocado para o próximo registrador e assim sucessivamente entre todos os registradores, enquanto o sinal de ativo estiver positivo (geralmente quando tem uma nova entrada válida). Este tipo de implementação foi feita para permitir que o módulo pudesse ser ou não controlado por um módulo de controle externo. Se o sinal de ativar estiver com o valor 1, o módulo irá fazer o deslocamento dos registradores continuamente

a cada ciclo de *clock*, o que é importante para a arquitetura proposta em alguns estágios da organização dos dados em pipeline, como discutido no capítulo anterior.

Figura 31 – Módulo de Registrador de Deslocamento e módulo de Controle e exemplo da instanciação em SystemVerilog

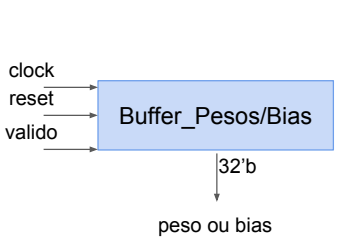


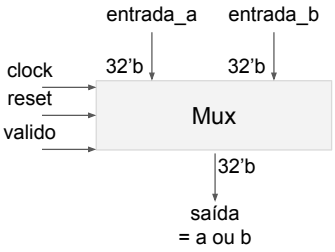
O módulo registrador de deslocamento precisa ser ativado quando dados intermediários dos somadores da arquitetura são válidos, como vimos no capítulo 4. Quando esses dados estão desalinhados no pipeline, o registrador precisa ser desativado para evitar a perda dos dados válidos alocados no registrador até o momento em que a sincronização para a realização da soma é valida. Para isso existe um módulo de controle que ao recebe um sinal de válido, começa a contagem de d ciclos de deslocamento entre um conjunto com n dados válidos e um outro conjunto. Essa contagem é realizada r vezes.

5.2.1.3 Buffer de Pesos ou *Bias* e Mux

Na Figura 32 podemos visualizar o módulo mux, que é basicamente um seletor entre dois caminhos de dados. Este mux foi implementado para manter a saída fixa na entrada a, caso um sinal de válido seja positivo ele seleciona a saída como sendo a entrada b, por n (neurônios da camada) dados em sequencia. No capítulo 4 foi explicado que esse módulo é importante para inserir dados a serem somados no meio do caminho de dados do pipeline. Por exemplo, quando um estágio precisa somar uma quantidade ímpar de dados, é inserido o *bias* através da seleção do mux pelo bit de válido.

Figura 32 – Módulo Buffer de Pesos ou *Bias* e Módulo de Mux e exemplo instanciação em SystemVerilog






```
buffer_pesos    #(    .entradas(k),
                      .neuronios(n),
                      .pesos(pesos))
buffer_pesos_i  (     .clock(clock),
                      .i_reset(reset),
                      .i_valido(valido),
                      .o_r(peso ou bias));
```

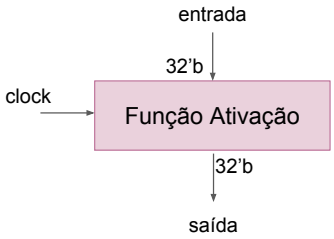
```
mux       #(    .profundidade(n))
mux_i     (     .clock(clock),
                .i_reset(reset),
                .i_a(entrada_a),
                .i_b(entrada_b),
                .i_valido(valido),
                .o_r(saida));
```

O módulo de buffer de pesos ou *bias* é implementado em uma memória RAM, inicializada com os pesos ou *bias* parametrizáveis da instanciação do módulo, como podemos ver na Figura 32. Essa memória RAM é integrada a um controle que libera na saída um peso ou *bias* de acordo com um endereço interno da memória, endereçável por um registrador manipulado pelo controle a partir do sinal positivo de válido. Um único buffer de pesos e um buffer de *bias* é necessário por camada da RNA-MLP da arquitetura proposta, como vimos no capítulo 4. O tamanho do buffer é definido pelos parâmetros quantidade de entradas k e quantidade n de neurônios. No caso de um buffer de *bias*, a quantidade de entradas é igual a 1, pois é necessário apenas um *bias* por neurônio.

5.2.1.4 Função de ativação

Na seção 4.2 do capítulo 4 o módulo função de ativação teve sua arquitetura apresentada e implementação descrita através de seus módulos e parâmetros internos, devido a sua complexidade. A implementação da função de ativação é caracterizada pela presença de uma memória RAM que armazena os valores de aproximação da função de ativação definida pelo projetista de IA como parâmetro do módulo, de forma semelhante aos pesos do módulo buffer. Neste trabalho foram implementadas aproximações das funções sigmoide e tangente hiperbólica, com uma tabela de consulta com 1280 posições. O caminho de dados proposto para implementação da função de ativação utilizando a tabela de consultas resultou em um pipeline com 23 ciclos de *clock*.

Figura 33 – Módulo Função Ativação e exemplo instanciação em SystemVerilog

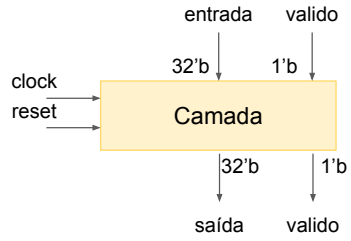



```
funcao_ativacao     #(    .tabela(valores))
funcao_ativacao_i   (     .clock(clock),
                          .i_d(entrada),
                          .o_d(saida);
```

5.2.1.5 Camadas, Interconexões e Rede

O módulo camada é o módulo principal que compõe a arquitetura proposta. Como vimos no capítulo 4 este módulo incorpora todos os parâmetros de configuração da rede, pois cada camada precisa ser configurada individualmente dentro do módulo Rede onde a MLP-RNA Profunda será instanciada por completo de acordo com a topologia da rede definida pelo projetista de IA. A Figura 34 apresenta o módulo camada e seus parâmetros de instanciação. O leitor pode observar que uma camada é configurável em k entradas, n neurônios, conjunto de pesos e *bias*, valores da tabela de consulta da função de ativação, quantos somadores na organização do pipeline são necessários (depende de k) e em quantas seções p o projetista quer distribuir a camada(como vimos no capitulo 4), e se a camada tem *bias*.

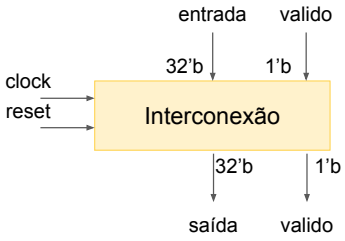
Figura 34 – Módulo Camada e exemplo instanciação em SystemVerilog




```
camada   #(   .entradas(k),
               .neuronios(n),
               .pesos(pesos),
               .bias(bias),
               .tabela(valores),
               .secoes(secoes)
               .tem_bias(tem_bias),
               .quantidade_somadores(somadores),
camada_i  (    .clock(clock),
               .i_reset(reset),
               .i_valido(valido),
               .i_data(entrada),
               .o_valid(valido),
               .o_data(saida));
```

O módulo interconexão é responsável por fazer as conexões entre as camadas no módulo de rede. O módulo conexão recebe k saídas sequencialmente de uma camada como entrada, armazena quando necessário e prepara esses dados para serem enviados para a próxima camada. Considerando a interface de comunicação e organização dos dados propostos, serão enviadas as k entradas para os n neurônios da próxima camada. Um *bit* válido de entrada sinaliza que as entradas provenientes de uma camada anterior estão sendo recebidas, e o módulo precisa enviar o *bit* de válido para sinalizar para a próxima camada que os dados que estão sendo transmitidos são válidos. A Figura 35 apresenta o módulo interconexão e um exemplo de sua instanciação em SystemVerilog.

Figura 35 – Módulo Interconexão e exemplo instanciação em SystemVerilog




```
interconexao    #(    .entradas(k),
                      .neuronios(n))
interconexao_i  (     .clock(clock)
                      .i_reset(i_reset),
                      .i_valido(valido),
                      .i_d(entrada),
                      .o_valido(valido),
                      .o_d(saida));
```

5.3 Integração Hardware e Software

Uma vez que a implementação em hardware da arquitetura proposta foi concluída, as próximas etapas do fluxo de desenvolvimento proposto consistem na integração dos módulos de hardware implementados na plataforma descritos na seção anterior, com uma aplicação de rede neural artificial em software. Isso significa que foi preciso desenvolver um software, no caso, em C/C++ que realize a comunicação através do barramento PCI-Express com a plataforma DE4(TERASIC, 2019).

Um desafio importante, neste contexto, foi identificar uma maneira eficiente em utilizar o barramento PCI-Express, pois, o uso incorreto impactaria diretamente no desempenho. Neste sentido foi usada a biblioteca de comunicação RIFFA, proposta em (JACOBSEN et al., 2015), que consegue atingir a máxima eficiência do barramento disponível. O RIFFA foi disponibilizado em sua totalidade, o que diminuiu o esforço de ter que replicar sua implementação.

O RIFFA dispõe de um *driver* que foi instalado no computador e possibilitou a comunicação software e hardware via barramento PCI-Express. Com o *driver* instalado, foi possível fazer o desenvolvimento do software da aplicação de redes neurais artificiais utilizando uma biblioteca de funções de alto nível em C/C++, com funções de recebimento e envio de dados via barramento, também fornecida junto com o *driver*. Usando a biblioteca de estruturas de comunicação RIFFA, a largura de banda estabelecida foi de 64 *bits*. Quando este vetor é enviado para a FPGA os dados passam a ser representados como 64 bits, o que representa duas entradas de ponto flutuante de 32 *bits* concatenados. Portando, o software da aplicação, envia os dados de entrada em ponto flutuante da rede

neural artificial que foi configurada utilizando a arquitetura proposta.

No FPGA, o RIFFA disponibiliza todas as estruturas de funcionamento que implementam um protocolo de comunicação que deve ser respeitado para o recebimento e envio dos dados do FPGA para a CPU. Para possibilitar que a arquitetura proposta nesse trabalho fosse executada utilizando o RIFFA, foi preciso implementar um módulo de controle que recebe os dados no protocolo RIFFA e os converte para a interface de dados de entrada da arquitetura proposta. Este controle envia o *bit* de válido e todos os dados para o módulo principal da arquitetura.

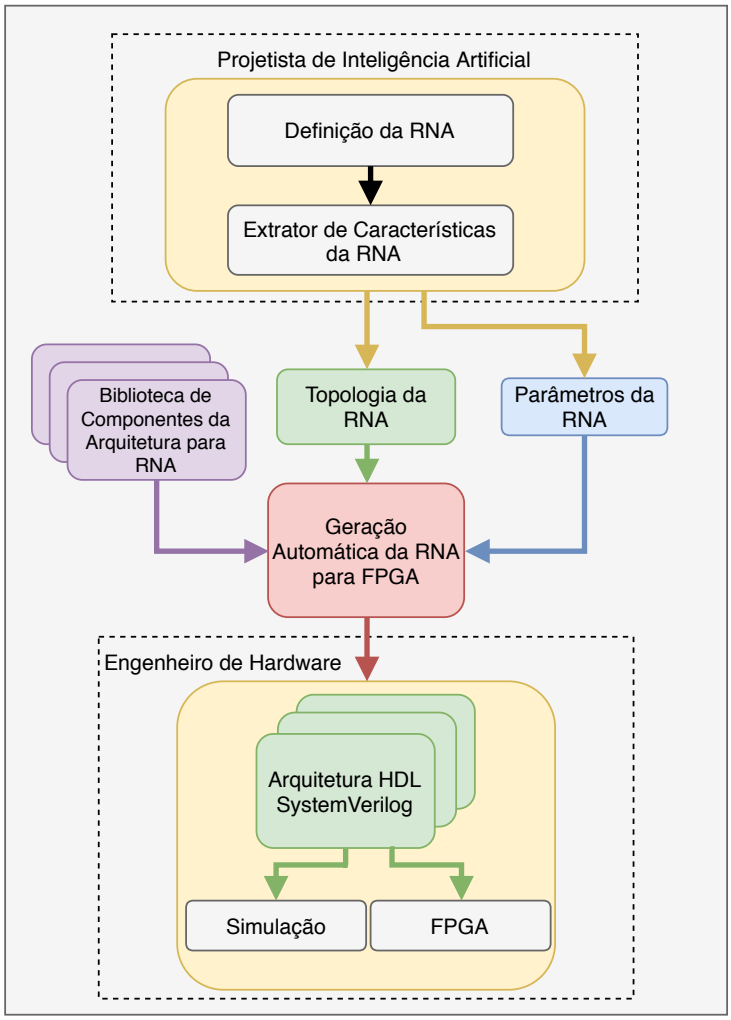
Quando o módulo principal da arquitetura libera a resposta da rede neural artificial, o módulo de controle empacota os dados dois a dois, formando os 64 *bits* de largura de banda e o envia para a CPU, respeitando o protocolo RIFFA. É importante lembrar que os dados de saída são compostos pelos valores em ponto flutuante dos impulsos nervosos da última camada da rede neural (camada de saída). No software, quando o processador solicitar a função de recebimento, os dados enviados do FPGA são recebidos. A aplicação de redes neurais artificiais irá receber estes dados e utilizar como definido.

5.3.1 Arquitetura Reconfigurável

A arquitetura proposta neste trabalho de dissertação foi projetada para ser de fácil reconfiguração através dos parâmetros dos módulos, para que dada a configuração de uma RNA-MLP, toda a arquitetura se configure automaticamente durante a síntese pelos mecanismos que foram implementados, como explicado na subseção 5.2.1. Desta forma o uso da arquitetura é facilitado, possibilitando o uso de diversas configurações topológicas de uma RNA-MLP. A facilidade de se reconfigurar a arquitetura possibilita que a arquitetura proposta contribua para que projetistas de inteligencia artificial em parceria com um engenheiro de hardware possam utilizar uma rede neural em um dispositivo FPGA, sem a necessidade de implementar a rede neural do início. Dependendo da plataforma utilizada será necessário apenas realizar a integração de hardware e software.

A Figura 36 apresenta o fluxo de projeto de uma aplicação baseada em rede neural artificial utilizando a arquitetura proposta neste trabalho.

Figura 36 – Fluxo de projeto para o uso da arquitetura reconfigurável proposta



Inicialmente o projetista de inteligência artificial, utilizando um ambiente de projeto e desenvolvimento de redes neurais artificiais, em Keras por exemplo, irá fazer o projeto da arquitetura da rede e seu treinamento, definindo assim como a rede irá ser configurada e executada.

Em seguida é preciso executar um software de extração de todas as características da rede neural artificial. Este software foi implementado também como parte deste projeto. Que irá extrair as informações de configuração topológica da rede e os pesos e *bias* das camadas (Parâmetros da RNA). Considerando o formato de dados estabelecido para a arquitetura proposta.

A geração automática da rede é feita em uma linguagem de descrição de hardware, utilizando os módulos em SystemVerilog. Além disso, é preciso que o engenheiro de hardware, munido das informações extraídas pelo software e com a biblioteca dos módulos da arquitetura proposta, realize a instanciação do módulo principal da arquitetura e das camadas da rede neural com suas interconexões, informando seus respectivos parâmetros.

Dessa forma, a arquitetura de hardware é formada a partir de todos os componentes e configurações necessárias para a rede neural projetada, podendo então ser utilizada na

plataforma estabelecida. Tal fluxo, apresentado em conjunto com a arquitetura proposta e software de extração, reduz significativamente a dificuldade e tempo de projeto para se implementar uma aplicação baseada em rede neural artificial perceptron de multicamada em FPGA.

6 RESULTADOS

Como discutimos no capítulo 5, a arquitetura de hardware proposta para execução de uma RNA-MLP profunda em FPGA foi validada por simulação funcional, simulação temporal e pela prototipação e integração entre o hardware e o software. Todas as etapas de validação foram realizadas considerando a plataforma educacional Terasic DE4 (TERASIC, 2019) com uma Intel FPGA Stratix IV[1] (EP4SGX530KH40C2), interligado a um computador pessoal através de barramento PCI-Express 2.0. A biblioteca de estruturas de comunicação PCI-Express, RIFFA 2.2.2 (JACOBSEN et al., 2015), possibilitou a prototipação deste trabalho.

Experimentos foram conduzidos para avaliar a arquitetura proposta e comparar o desempenho das RNAs geradas para o FPGA com as mesmas RNAs executas em CPU e GPU. Além disso os experimentos possibilitaram a comparação direta com alguns trabalhos relevantes para esta proposta. Dois conjuntos de dados amplamente utilizados em aprendizagem profunda foram utilizados, o conjuntos de dados MNIST e Fashion-MNIST.

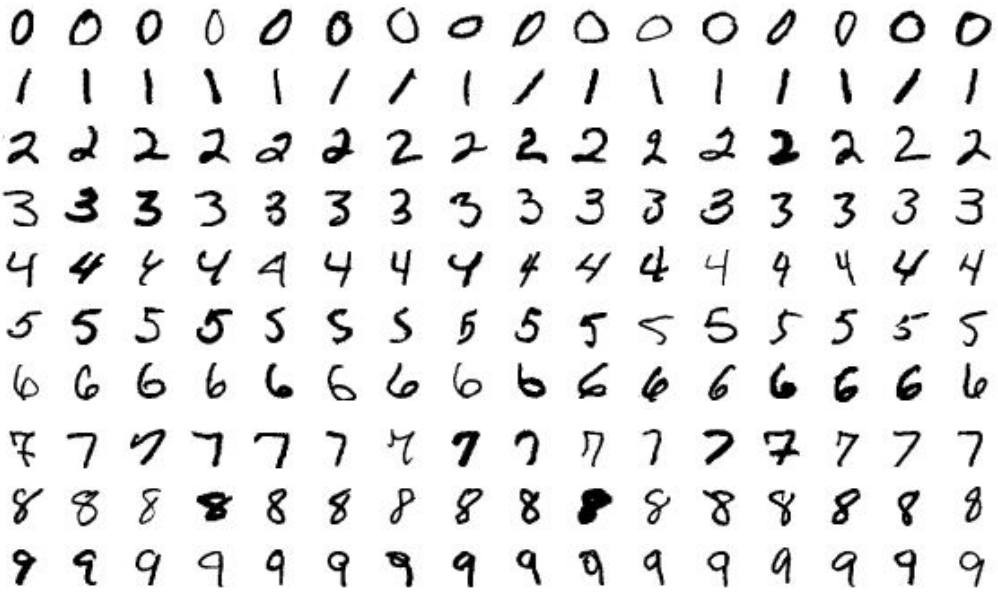
6.1 Conjunto de Dados para Validação

O conjunto de dados de dígitos manuscritos MNIST (Modified National Institute of Standards and Technology (NIST))(YANN; CORINNA; CHRISTOPHER, 1998) tem um conjunto de treinamento de 60.000 exemplos e um conjunto de teste de 10.000 exemplos, sendo introduzido por (LECUN et al., 1998) na área de aprendizagem profunda. Atualmente, é usado como o primeiro conjunto de dados a ser testado em validações e criação de novos métodos e técnicas de aprendizado profundo pela sua simplicidade e ampla utilização. Devido à sua simplicidade de treinamento e execução em bibliotecas de desenvolvimento de redes neurais com o Keras(KERAS, 2019), este conjunto foi usada na validação da arquitetura proposta. O MNIST foi criado pelo embaralhamento das amostras dos conjuntos de dados originais do NIST. Metade do conjunto de treinamento e metade do conjunto de testes foram retirados do conjunto de dados de treinamento do NIST, enquanto a outra metade do conjunto de treinamento e a outra metade do conjunto de testes foram retirados do conjunto de dados de testes do NIST. Como o conjunto de dados de treinamento do NIST foi retirado da escrita manuscrita dos funcionários do Departamento de Censo dos Estados Unidos, enquanto que conjunto de dados de teste foi retirado de estudantes do ensino médio americano, os criadores sentiram que tal distribuição poderia ser tendenciosa e não seria adequada para experimentos de AM. Além disso, as imagens em preto e branco do NIST foram normalizadas a partir do cálculo do centro de massa dos *pixels* de forma a caber em um tamanho de 28x28*pixels* com a realização de um *anti-aliasing*, possibilitando

[1] especificação pode ser encontrada em https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literat
iv-product-table.pdf

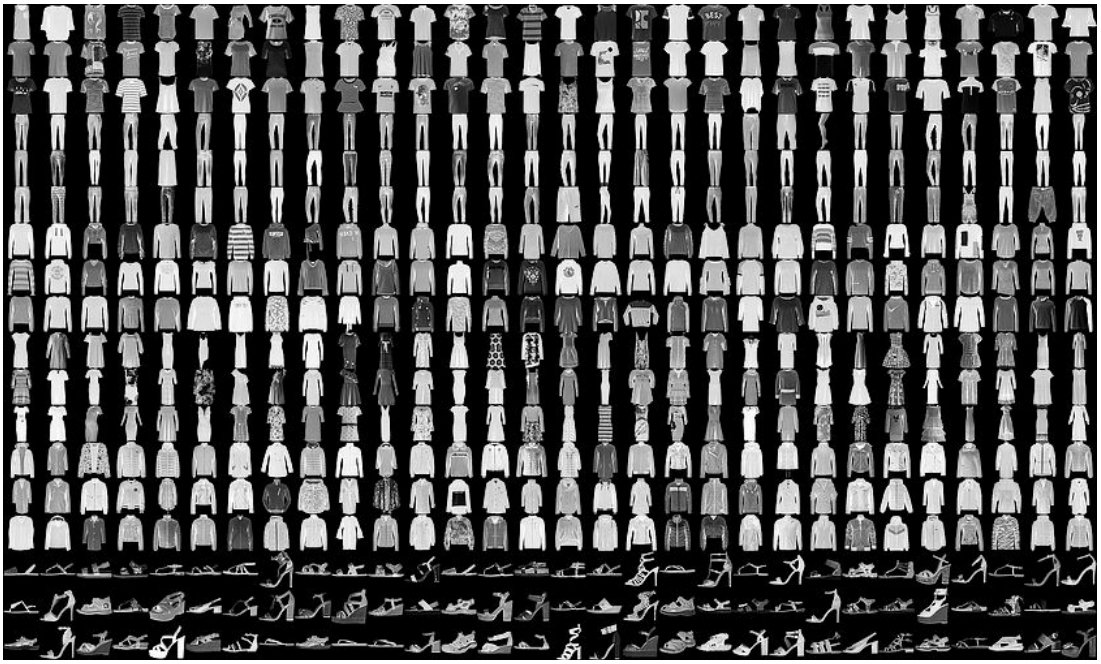
a introdução de níveis de escala de cinza (YANN; CORINNA; CHRISTOPHER, 1998). A Figura 37 apresenta uma amostra das imagens contidas no MNIST. Por fim, o conjunto de treinamento com 60.000 exemplos, é dividido em um subconjunto de treinamento com 48.000 exemplos e um subconjunto de testes com 12.000 exemplos.

Figura 37 – Imagens de amostra do conjunto de dados MNIST



O conjunto de dados Fashion-MNIST é baseado em produtos de moda do site da Zalando. Este conjunto possui imagens de diferentes produtos, por exemplo, calças, casacos e bolsas. Os autores do trabalho (XIAO; RASUL; VOLLGRAF, 2017) foram os responsáveis por selecionar as imagens e pré-processa-las para se tornar semelhante ao conjunto de dados do MNIST. O Fashion-MNIST tem imagens de 28x28 `pixels` em tons de cinza de produtos de moda totalizando 70.000 imagens em 10 categorias de produtos diferentes, com 7.000 imagens por categoria. Podemos observar que a quantidade de imagens do banco são semelhantes às do MNIST. Portanto, a distribuição é também semelhante, onde o conjunto de treinamento tem 60.000 imagens e o conjunto de testes tem 10.000 imagens. O conjunto de treinamento também foi dividido entre treinamento e validação com a proporção de 48.000 amostras de treinamento, e 12.000 amostras de validação, a mesma estratégia que foi realizada no MNIST. A Figura 38 apresenta uma amostra das imagens contidas na Fashion-MNIST.

Figura 38 – Imagens de amostra do conjunto de dados Fashion-MNIST



6.2 Resultados da Arquitetura

No capítulo 3 foram discutidos alguns trabalhos relacionados com este trabalho de dissertação. Fizemos um levantamento comparativo das características desejáveis para uma arquitetura de RNAs-MLP profunda. Nesta seção, vamos discutir se o projeto e desenvolvimento da arquitetura proposta neste trabalho nos capítulos 4 e 5, conseguiu atender às características importantes. A Tabela 4 apresenta o mesmo comparativo dos trabalhos relacionados apresentado no capítulo 3, mas agora considerando também o trabalho proposto.

Como vimos, a utilização da representação numérica em ponto flutuante de precisão simples de 32 *bits* melhora a precisão na capacidade de resolução dos problemas de AM. Portanto, a arquitetura foi projetada considerando todos os dados da RNA nesta representação numérica. Alguns trabalhos não fazem uso de tal representação, devido à quantidade massiva de dados que se faz necessário nas RNAs. A organização dos dados em pipeline proposta neste trabalho, onde o processamento de uma camada com n neurônios é realizada utilizando o mesmo caminho de dados e módulos, resulta no uso de menos dados alocados internamente no FPGA. Atrelado a isso, a implementação otimizada dos módulos possibilitou a utilização da representação ponto-flutuante sem impossibilitar o uso da arquitetura em aplicações de aprendizagem profunda. Na Tabela 4, podemos observar que os trabalhos que realizaram a validação utilizando um conjunto de dados de aprendizagem profunda, não adotaram a representação numérica em $32bits$, enquanto que os trabalho que a utilizaram, não fizeram uso de conjunto de dados de aprendizagem profunda para validação.

Tabela 4 – Comparativo dos trabalhos relacionados sob alguns critérios

Característica	Referência				**Trabalho Proposto**
	(Si; Harris, 2018)	(Huynh, 2017)	(KOYUNCU et al., 2017)	(AKLAH; ANDREWS, 2015)	
Representação Numérica	Representação linear em 8 *bits*	18 *bits*(onde, 2 bits de representação e 16 bits ponto flutuante meia precisão)	32 *bits*(ponto flutuante precisão simples)	32 *bits*(ponto flutuante ou ponto fixo)	32 *bits*(ponto flutuante precisão simples)
Implementação	SystemVerilog	Verilog	Não Informado	C/C++	SystemVerilog
Arquitetura Configurável	Não	Sim (Scrit em MATLAB)	Sim	Sim	Sim (Em Hardaware)
Quantidade de Entradas	Fixo	Parametrizável	Parametrizável	Parametrizável	Parametrizável
Quantidade de Neurônios	Fixo	Parametrizável	Parametrizável	Parametrizável	Parametrizável
Quantidade de Camadas	Fixo	Parametrizável (até 8)	Parametrizável	Parametrizável	Parametrizável
Função de Ativação Configurável	Fixa	Não informado	Sim (até 10 Funções)	Sim (até 3 Funções)	Sim (2 Funções Implementadas)
Validação utilizando Conjunto de Dados	Sim(MNIST)	Sim(MNIST)	Não (caso de uso)	Não (caso de uso)	Sim (MNIST e Fashion-MNIST)
Quantidade de Parâmetros da Rede	7.850	132.184(maior rede citada)	51	1.096 (maior rede citada)	184.214
Integração Hardware e Software	Não Informado	Não Informado	Não Informado	Não Informado	Sim

A implementação em linguagem de descrição de hardware SystemVerilog (SPEAR, 2008) permitiu explorar totalmente o paralelismo intrínseco das RNA-MLP, realizando uma implementação otimizada com recursos de geração parametrizável, de instanciação dos módulo em tempo de projeto e síntese da arquitetura. Estas características possibilitaram a construção do projeto da arquitetura configurável e flexível utilizando parâmetros de hardware, como discutido no capítulo 5. Assim sendo, foram criados parâmetros de hardware da topologia de uma RNA-MLP, como quantidade de entradas, quantidade de neurônios e de camadas da rede. Um dos trabalhos relacionados, (Huynh, 2017), realiza tal configuração da rede via software, através de um *script* de geração dos códigos sintetizáveis de hardware em MATLAB. Os trabalhos (KOYUNCU et al., 2017) e (AKLAH; ANDREWS, 2015) citam que são parametrizáveis à nível de hardware, mas infelizmente são foram validados no âmbito de aplicações de aprendizagem de máquina. As maiores redes citadas pelos autores são muito pequenas em quantidade de parâmetros como podemos observar na Tabela 4. Podemos concluir que suas arquiteturas não foram construídas orientadas a aplicações de aprendizagem profunda (centenas de milhares de parâmetros), portanto suas arquiteturas configuráveis tem uma escalabilidade reduzida.

Por fim, podemos observar na Tabela 4 que todos os trabalhos não mencionam nada sobre a prototipação de seus módulos de hardware com um software de aplicação da arquitetura. Esta é uma característica importante na avaliação de desempenho da arquitetura, onde dados de performance de comunicação entre o FPGA e uma CPU (por exemplo) são considerados no resultado final. Neste trabalho de dissertação a prototipação do hardware e sua integração com o software foram realizadas e os resultados serão discutidos na próxima seção.

6.3 Desempenho Obtido

Mencionamos anteriormente que para realizar a validação deste trabalho de dissertação, seria preciso fazer a prototipação do hardware e sua integração com o software de aplicação de RNA-MLPs profundas. Tais requisitos foram cumpridos e já discutidos neste trabalho. Portanto está seção irá apresentar os resultados de desempenho obtidos com a arquitetura proposta, além de fazer uma comparação concisa com os trabalhos relacionados que utilizaram o mesmo conjunto de dados de aprendizagem profunda que foram usados neste trabalho.

Como descrito na seção 5.1, foram projetadas as RNAs-MLP profundas, que serão usadas na aplicação com os conjuntos de dados MNIST e Fashin-MNIST. A Tabela 5 apresenta as informações de topologia da rede, quantidade de parâmetros e acurácia da rede aplicada ao conjunto de teste de cada conjunto de dados. Devido a similaridade entre os dois conjuntos de dados, para facilitar a etapa de projeto de IA nas etapas de desenvolvimento de software, que não foram o foco deste trabalho de dissertação, ambos os conjuntos de dados foram aplicados a mesma topologia de RNA-MLP. Portando,

ambas topologias apresentam o mesmo número de parâmetros (pesos e *bias*) da rede, vale lembrar que é uma quantidade considerável de parâmetros. Já a acurácia de teste obtida é diferente, pois são problemas similares em questão de padrão do conjunto de dados, mas são aplicações distintas. Para obter uma acurácia de teste melhor, seria preciso aumentar o esforço do projetista de IA na definição da RNA-MLP e treinamento da mesma.

Tabela 5 – Informações das RNAs-MLP Profundas utilizadas nos conjuntos de dados de validação da arquitetura proposta

Conjunto de Dados	Topologia da Rede	Quantidade de Parâmetros	Acurácia de Teste
MNIST	784-200-100-64-10	184.214	97.8%
Fashion-MNIST	784-200-100-64-10	184.214	88.1%

Com as configurações de rede e seus parâmetros (pesos e *bias*) na representação numérica correta, a arquitetura proposta pode ser configurada e sintetizada na ferramenta Intel Quartus Prime 18.1 (QUARTUS, 2019). O resultado de ocupação do Intel FPGA Stratix IV (EP4SGX530KH40C2) e frequência de operação obtida podem ser observados na Tabela 6. Como as topologias das RNAs-MLP utilizadas nos dois conjuntos de dados são iguais a síntese da arquitetura apresenta o mesmo resultado, como também mesmo desempenho, tanto em recursos utilizados quanto em tempo de execução.

Tabela 6 – Utilização de recursos e frequência de operação obtidos na plataforma Intel FPGA Stratix IV (EP4SGX530KH40C2)

	LUTs	Registradores	Memoria em Bits	Blocos de DSP	Frequência de Operação
MNIST e Fashion-MNIST	61.8k (15%)	104.6k (15%)	6748.7k (32%)	28(3%)	250Mhz

Podemos observar na Tabela 7 o tempo de execução de 10.000 imagens do conjunto de teste, em segundos, de ambas topologias de RNA-MLP Profunda para os dois conjuntos de dados. Os tempos de software para CPU e GPU foram medidos usando uma CPU Intel Core i7-7700HQ (8 núcleos a 2.80GHz) com 32GB de memória DDR4 (Ubuntu 64bits 16.04.3LTS) e uma GPU GeForce GTX 1050-Ti com 4GB de GDDR5. Ambas implementações foram feitas em Python v3.6.4 com a biblioteca Keras (versão 2.1.3) com suporte do Tensorflow (versão 1.4.0) durante as etapas de desenvolvimento de software deste trabalho. Podemos observar que a arquitetura proposta executa a mesma RNA-MLP profunda em hardware no FPGA 59.2 vezes mais rápido que na GPU e 366 vezes mais rápido que uma CPU, ambos com frequência de operação superior ao trabalho proposto em FPGA. Devemos destacar que a maior parte deste tempo de execução do FPGA é tempo de comunicação entre o software com o hardware via barramento PCI-Express 2.0.

Se fosse considerado apenas o tempo em execução no FPGA sem comunicação, esse tempo seria muito menor.

Tabela 7 – Performance obtida do trabalho proposto em FPGA em relação ao mesmo conjunto de dados de teste (10.000 imagens) executados em CPU e GPU

Conjunto de 10.000 imagens de Teste	Tempo em CPU	Tempo em GPU	**Tempo em FPGA**	Performance FPGA-GPU	Performance FPGA-CPU
MNIST e Fashion-MNIST	582s @2.8Ghz	94s @1.3Ghz	1.59s @250Mhz	59.2x	366x

A Tabela 8, apresenta uma comparação do tempo de execução obtido, com o tempo citado pelos trabalhos relacionados no capítulo 2, que não consideram o tempo de comunicação entre hardware e software.

Tabela 8 – Comparativo entre o tempo de execução citado pelos trabalhos relacionados e trabalho proposto em FPGA para o conjunto de teste (10.000 imagens)

Conjunto de 10.000 imagens de Teste	(SI; HARRIS, 2018)	(Huynh, 2017)	**Trabalho Proposto**
MNIST	3.8s	0.63	1.59s

Esses trabalhos fizeram sua validação com o conjunto de dados MNIST, o que possibilita essa comparação de tempo de execução, mas o uso dos recursos do FPGA não podem ser comparados, pois foram utilizadas plataformas diferentes em todos os trabalhos. Podemos observar que o tempo de execução da arquitetura proposta neste trabalho de dissertação é muito próxima aos trabalhos citados, alguns para mais (Huynh, 2017) e outros pra menos (SI; HARRIS, 2018).

7 CONCLUSÃO E TRABALHOS FUTUROS

O primeiro capítulo desta dissertação apresentou de maneira sucinta, os conceitos correlacionados de aprendizagem de máquina e sistemas ciber-físicos, motivando o uso de técnicas de aprendizagem profunda afim de contextualizar o uso das redes neurais artificiais na solução de problemas de aprendizagem profunda, e a necessidade de se melhorar o desempenho destas soluções através das plataformas FPGAs. No segundo capítulo foram discutidos os conceitos essenciais das RNAs necessários para o desenvolvimento da arquitetura proposta, seguido dos conceitos básicos a respeito dos FPGAs. No terceiro capítulo foram discutidos quatro trabalhos que estão relacionados de alguma forma com o projeto de uma arquitetura configurável para as RNAs. Ao final do capítulo foram levantadas e comparadas características importantes para a construção da arquitetura proposta neste trabalho.

O capítulo 4, apresenta a arquitetura configurável para redes neurais artificiais perceptron multicamadas. Onde a visão geral da arquitetura é formada pela quantidade configurável de camadas, conectadas entre si pelo modulo de interconexão que realiza a comunicação entre as camadas. O módulo de camada por sua vez, realiza a computação do impulso nervoso de todos os neurônios da camada em ponto flutuante $32 bits$ de precisão simples. Através de um caminho de dados em *pipeline* seguindo uma estrategia organizacional para a computação do estado de ativação utilizando apenas, 1 multiplicador de ponto flutuante e $\lfloor log_2(k) \rfloor + 1$ somados de ponto flutuante, onde k é igual a quantidade de entradas da camada. O resultado do estado de ativação é então utilizado para calcular a posição do valor de computação da função de ativação em um única tabela de consultas(com os valores possíveis da função de ativação em um determinado intervalo) para todos os neurônios da camada.

No quinto capítulo discutimos o fluxo de desenvolvimento adotado para realizar a implementação deste trabalho de dissertação. Foi apresentado uma sequência de desenvolvimento em etapas, divididas em desenvolvimento de software, hardware e por fim hardware e software. Nesse capítulo foram apresentados os detalhes da implementação de referencia da arquitetura em software, seguido dos detalhes da implementação dos módulos de hardware que compõe a arquitetura proposta. Por mim foram explicados os detalhes da prototipação hardware e software para a validação do trabalho proposto utilizando a plataforma educacional Terasic DE4 com um Intel FPGA Stratix IV (EP4SGX530KH40C2), interligado a um computador pessoal através de barramento PCI-Express 2.0 utilizando a biblioteca de estruturas de comunicação PCI-Express, RIFFA 2.2.2.

Por fim o sexto capítulo apresenta a conclusão deste trabalho de dissertação através dos resultados obtidos a partir da prototipação da arquitetura configurável proposta na execução de duas RNAs-MLP profundas para a aplicação envolvendo dois conjuntos de

dados (MNIST e Fashion-MNIST). O desempenho obtido pela execução da arquitetura prototipada no FPGA a 250Mhz foi 59.2 vezes mais rápido que uma GPU GeForce GTX 1050-Ti com 4GB de GDDR5 e 366 vezes mais rápido que uma CPU Intel Core i7-7700HQ (8 núcleos a 2.80GHz) com 32GB de memória DDR4.

Em comparação com os trabalhos relacionados o desempenho é similar, contudo os trabalhos relacionados não consideram o tempo de comunicação entre a plataforma de FPGA com o computador pessoal. Além disso, a arquitetura proposta nessa dissertação de mestrado atendeu a todas as características levantadas como sendo essenciais na construção de uma arquitetura configurável pra RNAs-MLP para aplicações de aprendizagem profunda.

A principal contribuição deste trabalho é o projeto de uma arquitetura configurável de alto desempenho para computação de redes neurais artificias perceptron multicamadas profundas. A arquitetura proposta possibilita assim uma flexibilidade da utilização das RNA-MLP Profundas em FPGA, com topologias complexas permitindo a parametrização livre da quantidade de camadas, neurônios por camada e quantidade de entradas por camada, totalizando centenas de milhares de parâmetros de uma RNA-MLP para aplicações de aprendizagem profunda. Além disso, a arquitetura permite uma flexibilização no uso de qualquer função de ativação que possa ser representada na forma de uma tabela de consulta. Tal feito é possivel graças a arquitetura proposta que buscou a utilização otimizada da organização dos dados em pipeline, reduzindo a quantidade de operações aritméticas de ponto flutuante por camada e armazenamento de tabelas de consultas em memória RAM. Esta estratégia permitiu uma redução significativa no consumo de recursos disponíveis do FPGA permitindo a configuração de RNA-MLP mais profundas.

7.1 Trabalho Futuros

Como trabalho futuro podemos sugerir primeiramente uma alteração na arquitetura para que o *bit* de válido da arquitetura seja correspondente a um dado de entrada e não a um conjunto de dados de entrada. Tal necessidade só foi notada, durante a prototipação da arquitetura, devido a possibilidade de ter "bolhas"na comunicação PCI-Express, o que interromperia o recebimento dos dados de entrada para conjuntos muito grandes.

Para melhorar a flexibilidade e uso da arquitetura seria interessante implementar-se em software uma ferramenta mais completa que a implementada neste trabalho, que faça a extração das caraterísticas de uma RNA-MLP definida em Keras. Esta ferramenta permite a extração das configurações de topologia da rede, configurando automaticamente o módulo principal da arquitetura, instanciando as camadas e suas interconexões. Além disso, a ferramenta poderia extrair parâmetros como os pesos e *bias*, convertendo-os para o formato de ponto flutuante. A construção dessa ferramenta mais robusta complementaria esse trabalho de dissertação permitindo uma maior facilidade na construção de aplicações RNA-MLP profundas em FPGA.

Uma outra melhoria atrelada a construção dessa ferramenta mais completa, seria a implementação de mais funções de ativação, como as citadas no capitulo 2, utilizando a abordagem com tabela de consulta proposta, enriquecendo assim a biblioteca de módulos pré-definidos da arquitetura.

REFERÊNCIAS

AKLAH, Z.; ANDREWS, D. A flexible multilayer perceptron co-processor for fpgas. In: SANO, K.; SOUDRIS, D.; HÜBNER, M.; DINIZ, P. C. (Ed.). *Applied Reconfigurable Computing*. Cham: Springer International Publishing, 2015. p. 427–434. ISBN 978-3-319-16214-0.

AMIN, H.; CURTIS, K. M.; HAYES-GILL, B. R. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings-Circuits, Devices and Systems*, IET, v. 144, n. 6, p. 313–317, 1997.

BAILEY, D. G. *Design for Embedded Image Processing on FPGAs*. 1st. ed. [S.l.]: Wiley Publishing, 2011. ISBN 0470828498, 9780470828496.

BARRETO, J. M. Introdução as redes neurais artificiais. *V Escola Regional de Informática da SBC Regional Sul*, Santa Maria, p. 41–71, 1997. Disponível em: <http://www.inf.ufsc.br/~j.barreto/tutoriais/Survey.pdf>.

BASTERRETXEA, K.; TARELA, J.; CAMPO, I. D. Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons. *IEE Proceedings-Circuits, Devices and Systems*, IET, v. 151, n. 1, p. 18–24, 2004.

BRAGA, A.; CARVALHO, A.; LUDERMIR, T. *Redes Neurais Artificiais: Teoria e Aplicações*. 2. ed. São Paulo: LTC Editora, 2007.

BROWN, S.; VRANESIC, Z. *Fundamentals of Digital Logic with VHDL Design*. [S.l.]: McGraw-Hill Higher Education, 2000.

COURTOY, M. Rapid system prototyping for real-time design validation. In: IEEE. *Rapid System Prototyping, 1998. Proceedings. 1998 Ninth International Workshop on*. [S.l.], 1998. p. 108–112.

DIAS, F. M.; ANTUNES, A.; MOTA, A. M. Artificial neural networks: a review of commercial hardware. *Engineering Applications of Artificial Intelligence*, v. 17, n. 8, p. 945 – 952, 2004. ISSN 0952-1976. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0952197604000934>.

DUTRA, B. H. T. C. Desenvolvimento de uma plataforma com uma arquitetura escalável para multiplicação de matrizes densas em sistemas reconfiguráveis de alto desempenho. In: *Dissertação de Mestrado*. [S.l.: s.n.], 2010.

FERREIRA, A. P. A. Desenvolvimento de uma arquitetura paralela para redes neurais artificiais mlp baseada em fpgas. In: *Dissertação de Mestrado*. [S.l.: s.n.], 2011.

FERREIRA, A. P. do A.; BARROS, E. N. da S. A high performance full pipelined arquitecture of mlp neural networks in fpga. In: *2010 17th IEEE International Conference on Electronics, Circuits and Systems*. [S.l.: s.n.], 2010. p. 742–745.

GRAPHICS, M. *ModelSim*. 2007.

HODGES, A. *Alan Turing: the enigma*. New York: Simon and Schuster, 1983. ISBN 978-0-671-49207-6 978-0-671-52809-6.

HOSCH, W. L. *Machine learning*. [S.l.]: Encyclopædia Britannica, 2016. Https://www.britannica.com/technology/machine-learning. Acessado em: 05-02-2019.

Huynh, T. V. Deep neural network accelerator based on fpga. In: *2017 4th NAFOSTED Conference on Information and Computer Science*. [S.l.: s.n.], 2017. p. 254–257.

JACOBSEN, M.; RICHMOND, D.; HOGAINS, M.; KASTNER, R. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, ACM, v. 8, n. 4, p. 22, 2015.

KERAS. 2019. Disponível em: <https://keras.io/>.

Khaitan, S. K.; McCalley, J. D. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, v. 9, n. 2, p. 350–365, June 2015. ISSN 1932-8184.

KOYUNCU, I.; SAHIN, I.; GLOSTER, C.; SARıTEKIN, N. K. A neuron library for rapid realization of artificial neural networks on fpga: A case study of rössler chaotic system. *Journal of Circuits, Systems and Computers*, v. 26, n. 01, 2017.

KUREMOTO, T.; KIMURA, S.; KOBAYASHI, K.; OBAYASHI, M. Time series forecasting using a deep belief network with restricted boltzmann machines. *Neurocomputing*, Elsevier, v. 137, p. 47–56, 2014.

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

LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, IEEE, v. 86, n. 11, p. 2278–2324, 1998.

LEE, J.; BAGHERI, B.; KAO, H.-A. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, v. 3, p. 18 – 23, 2015. ISSN 2213-8463. Disponível em: <http://www.sciencedirect.com/science/article/pii/S221384631400025X>.

LITJENS, G.; KOOI, T.; BEJNORDI, B. E.; SETIO, A. A. A.; CIOMPI, F.; GHAFOORIAN, M.; LAAK, J. A. van der; GINNEKEN, B. van; SÁNCHEZ, C. I. A survey on deep learning in medical image analysis. *Medical image analysis*, Elsevier, v. 42, p. 60–88, 2017.

MAGDALENO, E.; RODRÍGUEZ, M. Acceleration of computation speed for wavefront phase recovery using programmable logic. In: *Topics in Adaptive Optics*. [S.l.]: InTech, 2012.

MATHWORKS. *MATLAB*. 2019. Disponível em: <https://www.mathworks.com/products/matlab.html>.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

NIELSEN, M. A. *Neural Networks and Deep Learning*. [S.l.]: Determination Press, 2015.

QIU, X.; ZHANG, L.; REN, Y.; SUGANTHAN, P. N.; AMARATUNGA, G. Ensemble deep learning for regression and time series forecasting. In: IEEE. *Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on*. [S.l.], 2014. p. 1–6.

QUARTUS. 2019. Disponível em: <https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/overview.html>.

Rajkumar, R.; Lee, I.; Sha, L.; Stankovic, J. Cyber-physical systems: The next computing revolution. In: *Design Automation Conference*. [S.l.: s.n.], 2010. p. 731–736. ISSN 0738-100X.

ROCHA, R. C. F. Desenvolvimento de uma plataforma reconfigurável para modelagem 2d, em sísmica, utilizando fpga. In: *Dissertação de Mestrado*. [S.l.: s.n.], 2010.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, American Psychological Association, v. 65, n. 6, p. 386–408, 1958.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. *Neural Networks*, v. 61, p. 85 – 117, 2015. ISSN 0893-6080. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0893608014002135>.

Si, J.; Harris, S. L. Handwritten digit recognition system on an fpga. In: *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. [S.l.: s.n.], 2018. p. 402–407.

SI, J.; HARRIS, S. L. Handwritten digit recognition system on an fpga. In: *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. [S.l.: s.n.], 2018. p. 402–407.

SOUZA, V. L. S. Implementação de uma arquitetura para multiplicação de matrizes densas em sistemas reconfiguráveis de alto desempenho. In: *Dissertação de Mestrado*. [S.l.: s.n.], 2008.

SPEAR, C. *SystemVerilog for verification: a guide to learning the testbench language features*. [S.l.]: Springer Science & Business Media, 2008.

STANFORD, C. *CS231n: Convolutional Neural Networks for Visual Recognition, Module 1, Neural Networks Part 1: Setting up the Architecture*. 2019. Http://cs231n.github.io/neural-networks-1/. Acessado em: 05-02-2019.

TERASIC. *Altera DE4 Development and Education Board*. 2019. Disponível em: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=164&No=501>.

XIAO, H.; RASUL, K.; VOLLGRAF, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

XILINX. *Vivado Design Suite*. 2019. Disponível em: <https://www.xilinx.com/products/design-tools/vivado.html>.

XU, Y.; MO, T.; FENG, Q.; ZHONG, P.; LAI, M.; ERIC, I.; CHANG, C. Deep learning of feature representation with multiple instance learning for medical image analysis. In: IEEE. *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. [S.l.], 2014. p. 1626–1630.

YANN, L.; CORINNA, C.; CHRISTOPHER, J. B. *MNIST*. 1998. Disponível em: <http://yann.lecun.com/exdb/mnist/>.

YU, D.; DENG, L. Deep learning and its applications to signal and information processing [exploratory dsp]. *IEEE Signal Processing Magazine*, IEEE, v. 28, n. 1, p. 145–154, 2011.

YU, J.; WENG, K.; LIANG, G.; XIE, G. A vision-based robotic grasping system using deep learning for 3d object recognition and pose estimation. In: IEEE. *Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on*. [S.l.], 2013. p. 1175–1180.

ZHANG, M.; VASSILIADIS, S.; DELGADO-FRIAS, J. G. Sigmoid generators for neural computing using piecewise approximations. *IEEE transactions on Computers*, IEEE, v. 45, n. 9, p. 1045–1049, 1996.