

**Universidade Federal de Santa Catarina**  
**Campus Reitor João David Ferreira Lima**  
**Departamento de Engenharia Elétrica e Eletrônica**



**Luis Antonio Spader Simon**

**Exploração de Espaço de Projeto Automatizado para  
Arquiteturas Aceleradoras de Redes Neurais**

**Florianópolis**  
**2023**

**Luis Antonio Spader Simon**

# **Exploração de Espaço de Projeto Automatizado para Arquiteturas Aceleradoras de Redes Neurais**

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do título de Bacharel em Engenharia Eletrônica.

Orientador: Prof. Dr. Héctor Pettenghi Rodán

Coorientador: Prof. Dr. Mateus Grellert

Universidade Federal de Santa Catarina  
Campus Reitor João David Ferreira Lima  
Departamento de Engenharia Elétrica e Eletrônica

Florianópolis  
2023

Ficha de identificação da obra elaborada pelo autor,  
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Spader Simon, Luis Antonio  
Exploração de Espaço de Projeto Automatizado para  
Arquiteturas Aceleradoras de Redes Neurais / Luis Antonio  
Spader Simon ; orientador, Héctor Pettenghi Rodán,  
coorientador, Mateus Grellert da Silva, 2023.  
82 p.

Trabalho de Conclusão de Curso (graduação) -  
Universidade Federal de Santa Catarina, Centro Tecnológico,  
Graduação em Engenharia Eletrônica, Florianópolis, 2023.

Inclui referências.

1. Engenharia Eletrônica. 2. Redes Neurais Artificiais.  
3. Hardware. 4. framework. 5. ASIC. I. Pettenghi Rodán,  
Héctor. II. Grellert da Silva, Mateus. III. Universidade  
Federal de Santa Catarina. Graduação em Engenharia  
Eletrônica. IV. Título.

**Luis Antonio Spader Simon**

# **Exploração de Espaço de Projeto Automatizado para Arquiteturas Aceleradoras de Redes Neurais**

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de “Bacharel em Engenharia Eletrônica” e aprovado em sua forma final pelo Curso de Graduação em Engenharia Eletrônica.

Florianópolis, 18 de novembro de 2023

---

Prof. Fernando Rangel de Souza, Dr.  
Coordenador do Curso

**Banca Examinadora:**

---

Prof. Dr. Héctor Pettenghi Rodán  
Universidade Federal de Santa Catarina  
Orientador

---

Prof. Dr. Mateus Grellert da Silva  
Universidade Federal de Santa Catarina

---

Prof. Dr. Fabian L. Cabrera  
Universidade Federal de Santa Catarina

---

Prof. Dr. José Luiz Almada Güntzel  
Universidade Federal de Santa Catarina

---

Prof. Dra. Cristina Meinhardt  
Universidade Federal de Santa Catarina

Este trabalho é dedicado aos meus pais, irmã, namorada e amigos, os quais foram os pilares desta jornada. Sem vocês isso não seria possível.

# Agradecimentos

Gostaria de expressar meus sinceros agradecimentos a todas as pessoas que desempenharam papéis fundamentais na minha jornada acadêmica e na realização deste trabalho:

Aos meus pais, Nilssei e Donizete, que não apenas me apoiaram financeiramente, mas também me proporcionaram apoio emocional ao longo desses anos na UFSC.

À minha namorada, Milena, que esteve ao meu lado, oferecendo orientação, incentivo e persistência durante todo o processo de conclusão deste trabalho.

Aos meus professores orientadores, Mateus Grellert da Silva e Héctor Pettenghi Ródán, que investiram tempo e esforço significativos para me auxiliar na elaboração e desenvolvimento deste trabalho.

Aos professores do laboratório ECL, José Luiz Güntzel e Cristina Meinhardt, por fornecerem o suporte e a estrutura necessários para a realização deste projeto. Aos meus colegas de curso e os colegas do laboratório ECL, os quais me proporcionaram companhia e bons momentos que me ajudaram a suportar os desafios durante esta jornada de graduação.

# Resumo

Redes Neurais estão em tendência crescente de uso devido à sua comprovada eficiência para resolver problemas complexos como reconhecimento facial e geração de texto em linguagem natural. No entanto, a execução destes algoritmos em processadores de propósito geral ou mesmo em GPUs exige um consumo considerável de energia, gerando custos financeiros e também ambientais. Diversos estudos demonstram que o uso de arquiteturas dedicadas fornecem um notável aumento de eficiência energética e elevado poder de processamento. Contudo, o projeto de arquiteturas dedicadas requer tempo para otimizar os diversos parâmetros de projeto (como área, potência e atraso), pois cada solução precisa ser descrita em HDL, sintetizada e verificada. Estes dois processos de otimização combinados resultam em um espaço de projeto multidimensional, portanto soluções que buscam acelerar a exploração deste espaço são de extrema importância. Logo, a proposta deste trabalho se baseia na construção de um *framework* para geração automática de descrição de hardware em linguagem VHDL com objetivo de acelerar o tempo de desenvolvimento destas arquiteturas. O *framework* proposto suporta a tradução de diferentes arquiteturas de Redes Neurais através de configurações parametrizáveis para largura de bits dos dados, número de camadas, quantidade de neurônios por camada e função de ativação. Uma implementação em Python foi desenvolvida para servir como referência e permitir uma análise inicial do erro obtido através da quantização (redução da largura de bits) das entradas e pesos da rede. Como estudo de caso, a solução proposta analisou arquiteturas de Redes Neurais para reconhecimento de dígitos na tecnologia 65nm da ST Micron. Os resultados de síntese são comparados em termos de área, atraso e consumo de potência. Além disso, o efeito da quantização na acurácia dos modelos também é analisado e discutido. Após isto são apresentados os resultados de possíveis otimizações aritméticas à serem feitas nas arquiteturas geradas automaticamente pelo *framework*. O estudo revelou que a largura de bits é o fator que mais impacta na área e o consumo de energia. Modelos quantizados com 8 ou 10 bits se aproximam bastante da acurácia dos modelos originais, mas camadas extras apresentam um maior erro associado. Nas otimizações aritméticas, a substituição de multiplicadores por blocos de soma e deslocamentos otimiza a área, ao custo de maior atraso crítico e consumo de potência, sendo mais adequada em situações de restrição de área.

**Palavras-Chave:** 1. Redes Neurais Artificiais 2. Hardware 3. Autoencoder 4. ASIC 5. FPGA 6. framework

# Abstract

Neural Networks are increasingly used due to their proven efficiency in solving complex problems such as facial recognition and natural language text generation. However, running these algorithms on general-purpose processors or even GPUs requires considerable energy consumption, generating financial and environmental costs. Several studies demonstrate that the use of dedicated architectures provides a notable increase in energy efficiency and high processing power. However, the design of dedicated architectures requires time to optimize the various design parameters (such as area, power, and delay), as each solution needs to be described in HDL, synthesized, and verified. These two optimization processes combined result in a multidimensional design space, therefore solutions that seek to accelerate the exploration of this space are extremely important. Therefore, the proposal of this work is based on the construction of a framework for automatic generation of hardware description in VHDL language with the aim of accelerating the development time of these architectures. The proposed framework supports the translation of different Neural Network architectures through parameterizable settings for data bit width, number of layers, number of neurons per layer, and activation function. An implementation in Python was developed to serve as a reference and allow an initial analysis of the error obtained through quantization (reducing the bit width) of the network inputs and weights. As a case study, the proposed solution analyzed Neural Network architectures for digit recognition in ST Micron's 65nm technology. The synthesis results are compared in terms of area, delay, and power consumption. Furthermore, the effect of quantization on model accuracy is also analyzed and discussed. After this, the results of possible arithmetic optimizations to be made in the architectures automatically generated by framework are presented. The study revealed that bit width is the factor that most impacts area and energy consumption. Quantized models with 8 or 10 bits come very close to the accuracy of the original models, but extra layers have a greater associated error. In arithmetic optimizations, replacing multipliers with sum and shift blocks optimizes the area, at the cost of greater critical delay and power consumption, being more suitable in situations of area restrictions.

**Keywords:** 1. Artificial Neural Networks 2.Hardware 3.Autoencoder 4. ASIC 5. FPGA  
6. framework

# Listas de figuras

Figura 1 – Poder computacional usado no treinamento de sistemas de IA. . . . .	16
Figura 2 – Desempenho/Watt Relativo (TPU, CPU e GPU). . . . .	16
Figura 3 – Aumento de velocidade de Processamento. . . . .	17
Figura 4 – Aumento de eficiência energética (HAN et al., 2016). . . . .	17
Figura 5 – Comparação de desempenho entre arquitetura dedicada <i>Descartes</i> e CPU, GPU. . . . .	18
Figura 6 – Fluxo estimado de trabalho. . . . .	20
Figura 7 – Arquitetura de uma Rede Neural Artificial genérica (DNN $i \rightarrow h_1 \rightarrow h_2 \rightarrow h_n \rightarrow o$ ). . . . .	24
Figura 8 – Diagrama de blocos do Perceptron. . . . .	26
Figura 9 – Exemplo de Funções de Ativação. . . . .	26
Figura 10 – Arquitetura genérica de um <i>AutoEncoder</i> . . . . .	27
Figura 11 – Fluxo de trabalho do <i>framework</i> hls4ml. . . . .	30
Figura 12 – Fluxograma do Framework . . . . .	33
Figura 13 – Arquiteturas de multiplicação combinacional (pesos variáveis vs pesos fixos). . . . .	35
Figura 14 – Blocos de soma e deslocamentos (Spiral). . . . .	36
Figura 15 – Substituição das somas por um bloco de compressores. . . . .	37
Figura 16 – <i>Carry-Save Adders</i> . . . . .	38
Figura 17 – Aplicação web do <i>framework</i> . . . . .	39
Figura 18 – Estrutura de arquivos gerados pelo <i>framework</i> . . . . .	40
Figura 19 – Executando o <i>framework</i> através da IDE Microsoft VSCode. . . . .	41
Figura 20 – Executando o <i>framework</i> através do Linux Terminal. . . . .	41
Figura 21 – Arquivo de teste ( <i>testbench</i> ) da arquitetura de topo ( <i>top-level</i> ). . . . .	42
Figura 22 – Arquitetura genérica de uma Rede Neural Artificial de Perceptrons multi-camadas (MLPs) gerada pelo framework. . . . .	43
Figura 23 – Funções de ativação disponíveis . . . . .	44
Figura 24 – Exemplo de uma arquitetura em hardware de um MLP gerado pelo framework proposto. . . . .	45
Figura 25 – Bloco Shift-registers. . . . .	45
Figura 26 – Arquitetura genérica de um Neurônio (exemplo para arquitetura com 4 entradas). . . . .	46

Figura 27 – Resultados de síntese para a frequência alvo de 200 MHz. . . . .	48
Figura 28 – Desempenho na inferência de classificação considerando a variação do número total de camadas (N) e bits de precisão (B) (imagem 4x4). . .	51
Figura 29 – Desempenho na inferência de classificação considerando a variação do número total de camadas (N) e bits de precisão (B) (imagem 8x8). . .	51

# Listas de tabelas

Tabela 1 – Tabela Comparativa de <i>frameworks</i> . . . . .	31
Tabela 2 – Parâmetros de arquitetura testados (Valores padrão em negrito). . . . .	48
Tabela 3 – Multiplicadores vs Somas e deslocamentos: Área total. . . . .	53
Tabela 4 – Multiplicadores vs Somas e deslocamentos: Atraso crítico. . . . .	54
Tabela 5 – Multiplicadores vs Somas e deslocamentos: Potência total consumida.	54
Tabela 6 – Comparaçao de Área total entre arquitetura padrão (tipo 1) e com- pressores (tipo 2). . . . .	55
Tabela 7 – Comparaçao de Atraso máximo entre arquitetura padrão (tipo 1) e compressores (tipo 2). . . . .	56
Tabela 8 – Comparaçao de Potência total entre arquitetura padrão (tipo 1) e compressores (tipo 2). . . . .	56

# **Lista de Siglas e Abreviaturas**

<b>AE</b>	<i>Autoencoders</i>
<b>ASIC</b>	<i>Application-specific Integrated Circuit (Circuitos integrados de aplicação específica)</i>
<b>CPU</b>	<i>Central Processing Unit</i>
<b>CSA</b>	<i>Carry-Select Adder</i>
<b>CNN</b>	<i>Convolutional Neural Networks (Redes Neurais Convolucionais)</i>
<b>DNN</b>	<i>Deep-Neural Networks (Redes Profundas)</i>
<b>FPGA</b>	<i>Field Programmable Gate Array</i>
<b>GPU</b>	<i>Graphics Processing Unit (unidade de processamento gráfica)</i>
<b>HDL</b>	<i>Hardware Description Language (linguagem de descrição de hardware)</i>
<b>HLS</b>	<i>High-Level Synthesis (Síntese de alto-nível)</i>
<b>IA</b>	<i>Inteligência Artificial</i>
<b>IP</b>	<i>Intellectual Property (Propriedade Intelectual)</i>
<b>LUT</b>	<i>Look-Up Table</i>
<b>MAC</b>	<i>Multiply and Accumulate (Multiplica e acumula)</i>
<b>MLP</b>	<i>Multi-Layer Perceptron (Perceptron multicamadas)</i>
<b>RNA</b>	<i>Rede Neural Artificial</i>
<b>RTL</b>	<i>Register Transfer Level</i>
<b>VHDL</b>	<i>VHSIC Hardware Description Language</i>

# Sumário

1	INTRODUÇÃO . . . . .	14
1.1	Problema 1: Poder de Processamento e Consumo Energético	15
1.2	Problema 2: Projeto de arquiteturas dedicadas . . . . .	19
1.3	Metodologia . . . . .	19
1.4	Motivação e Objetivos . . . . .	20
1.4.1	Motivação . . . . .	20
1.4.2	Objetivos Gerais . . . . .	21
1.4.3	Objetivos Específicos . . . . .	21
2	REFERENCIAL TEÓRICO . . . . .	23
2.1	<i>Machine Learning</i> . . . . .	23
2.2	<i>Deep Learning</i> . . . . .	24
2.2.1	Perceptron . . . . .	25
2.2.2	Funções de Ativação . . . . .	26
2.2.3	Autoencoders . . . . .	27
3	TRABALHOS RELACIONADOS . . . . .	29
3.1	Ferramentas de código-aberto . . . . .	29
3.1.1	hls4ml . . . . .	30
3.1.2	NNGen . . . . .	31
3.2	Tabela Comparativa dos principais <i>Frameworks</i> . . . . .	31
4	PROPOSTA . . . . .	33
4.1	Framework . . . . .	33
4.2	Otimizações Aritméticas para Arquiteturas Geradas pelo Framework . . . . .	34
4.2.1	Substituição dos Multiplicadores por Blocos de Somas e Deslocamentos . . . . .	35
4.2.2	Troca dos Somadores por um Bloco de Compressão de Vetores	37
5	RESULTADOS . . . . .	39
5.1	Disponibilização e uso do Framework . . . . .	39
5.2	Framework . . . . .	42

5.2.1	Personalização da Largura de Representação de Bits . . . . .	43
5.2.2	Escolha de Funções de Ativação . . . . .	43
5.2.3	Inclusão de Barreiras de Registradores . . . . .	44
5.2.4	Modos de Atualização de Pesos e Inferência . . . . .	44
5.2.5	Modularização das Arquiteturas . . . . .	46
5.2.6	Resultados e Limitações . . . . .	47
5.3	Resultados de Síntese . . . . .	47
5.3.1	Resultados de Síntese . . . . .	47
5.3.2	Efeitos da Aproximação em Redes Neurais de Classificação .	49
5.4	Melhorias para arquiteturas geradas pelo <i>Framework</i> . . . .	52
5.4.1	Substituição dos Multiplicadores por Blocos de Somas e Deslocamentos . . . . .	53
5.4.2	Troca dos Somadores por um Bloco de Compressão de Vetores	55
6	CONCLUSÃO . . . . .	58
	REFERÊNCIAS BIBLIOGRÁFICAS . . . . .	60
	APÊNDICE A – CÓDIGOS VHDL DE UMA MLP EXEMPLO . . . . .	66

# 1 Introdução

Apesar de algoritmos de inteligência artificial terem começado seu estudo no ano 1956 no campus de Dartmouth College (USA), apenas recentemente, a partir dos anos 2000 os algoritmos de inteligência artificial (IA) vêm se popularizando. Isto se deu ao fato do aprimoramento dos algoritmos com o surgimento dos sistemas especialistas (*expert systems*) a partir dos anos 80 (NEWQUIST, 1994), aumento do poder de processamento dos computadores seguindo a Lei de Moore (KURZWEIL, 2006) com o uso das GPUs<sup>1</sup> (*Graphical Processing Units*) (SCHMIDHUBER, 2015) e atual demanda por diversas utilizações na indústria (PAN, 2016).

O mercado para este setor demonstra uma tendência crescente onde até o ano 2030, estima-se que o setor estará avaliado em 1.597,1 bilhões de dólares (PRECEDENCERESEARCH, 2022).

Devido ao aprimoramento dos algoritmos, atualmente encontram-se diversas aplicações para a utilização dos mesmos como por exemplo a detecção a fraude em bancos (DESROUSSEAUX; BERNARD; MARIAGE, 2021) em tempo real, através de algoritmos especializados na detecção de anomalias, *Video Coding & Decoding* (ZHOU; LV; YI, 2022) e (ZHANG et al., 2019) onde algoritmos de Inteligência Artificial (IA) podem ser utilizados para substituir ou melhorar parte do processo de codificação (LU et al., 2021) ou ainda toda a codificação e decodificação (PESSOA et al., 2020).

Se tratando dos modelos de algoritmo de IA, modelos de redes convolucionais (CNNs) e redes neurais artificiais recorrentes (RNNs) têm sido propostos para imagem, vídeo a processamento de fala (GUO et al., 2019). CNNs 1D foram propostas recentemente e imediatamente alcançaram níveis de desempenho de ponta em várias aplicações, como classificação de dados biomédicos personalizados e diagnóstico precoce, monitoramento de integridade estrutural, detecção e identificação de anomalias em eletrônica de potência e elétrica detecção de falha do motor (K et al., 2021).

Outra aplicação de IA é na orientação de veículos autônomos usando principalmente algoritmos de detecção de objetos (OYANG et al., 2020), planejadores de caminhos locais e sentido de rotas (ISELE et al., 2018) e (WANG et al., 2020), onde pretende-se

---

<sup>1</sup>**GPU ( (Graphical Processing Units):** Unidade de processamento gráfico, um processador especializado originalmente projetado para acelerar a renderização de gráficos. Projetada para processamento paralelo, a GPU é usada em uma ampla gama de aplicações, incluindo gráficos e renderização de vídeo. Embora sejam mais conhecidas por suas capacidades em jogos, as GPUs estão se tornando mais populares para uso em produção criativa e inteligência artificial (IA).

futuramente a associação sistemas de completos de direção autônoma, porém atualmente a principal dificuldade é a falta de algoritmos de segurança desenvolvidos de forma determinística e da dificuldade em se interpretar a segurança dos algoritmos diante das infinitas possibilidades de ocasiões (YURTSEVER et al., 2020).

## 1.1 Problema 1: Poder de Processamento e Consumo Energético

Conforme comentado anteriormente, o poder de processamento foi um dos fatores que permitiu a utilização de algoritmos de IA. O poder de processamento atual é em grande parte responsabilizado pelo uso das GPUs, que são utilizadas tanto para treinamento dos algoritmos, (que não construídos deterministicamente, mas sim passam por um ciclo iterativo até atingir o desempenho desejado para cada aplicação) quanto para inferência. O problema das GPUs é sua eficiência energética e o consumo de energia em algoritmos de IA está deixando marcas no planeta. Um trabalho de pesquisa da Universidade de Massachusetts Amherst (STRUBELL; GANESH; MCCALLUM, 2019) estimou que, durante uma vida útil média, as emissões de carbono associadas ao treinamento de um modelo de transformador, como BERT ou GPT-2, com pesquisa de arquitetura neural são equivalentes à pegada de carbono de um carro, incluindo combustível (MORGAN, 2021).

Para implementar uma Rede Neural Densa, conhecida como DNN (Deep Neural Networks)<sup>2</sup> de alta eficiência e baixo consumo energético, pode-se otimizar em 2 frentes: no design e algoritmo do modelo DNN ou na aceleração em hardware (HAO; CHEN, 2018).

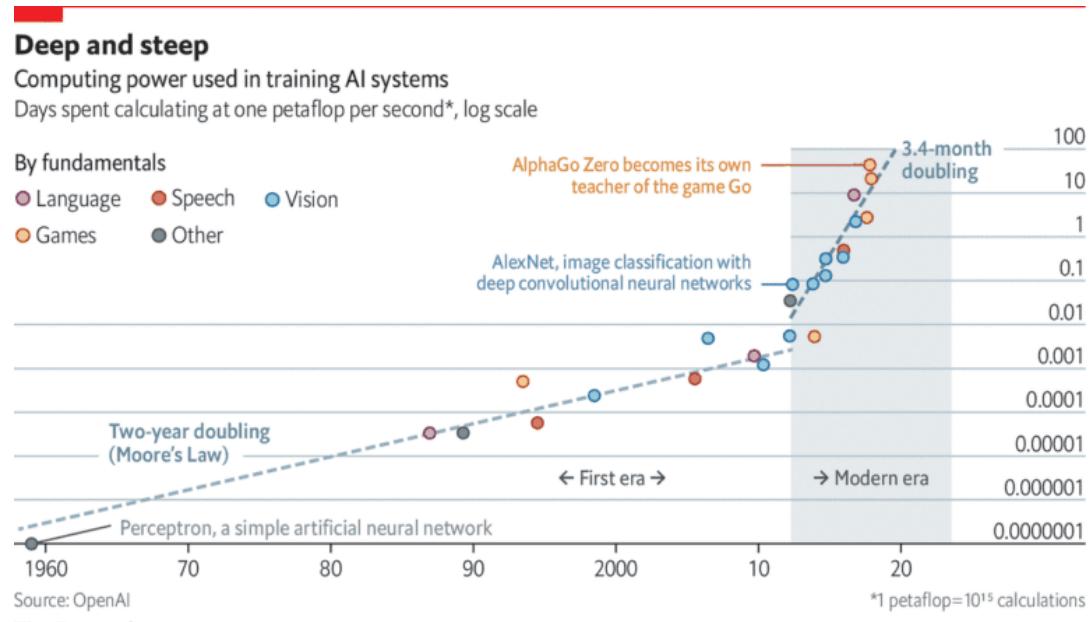
Apesar do alto poder de processamento das GPUs, como as mesmas não foram construídas especificamente para algoritmos de IA, arquiteturas dedicadas conseguem trazer uma eficiência e poder de processamento maiores (JOUSSI, 2017), (NURVITADHI et al., 2016) e (HAN et al., 2016).

Han et al. (2016) desenvolveram arquiteturas dedicadas que conseguem alcançar uma eficiência energética até 1052 vezes maior do que GPU (NVIDIA GeForce GTX Titan X) e 24.207 vezes maior que CPU (Intel Core i-7 5930k) e simultaneamente o processamento superior em até 4 vezes maior que a GPU citada anteriormente (HAN et al., 2016).

---

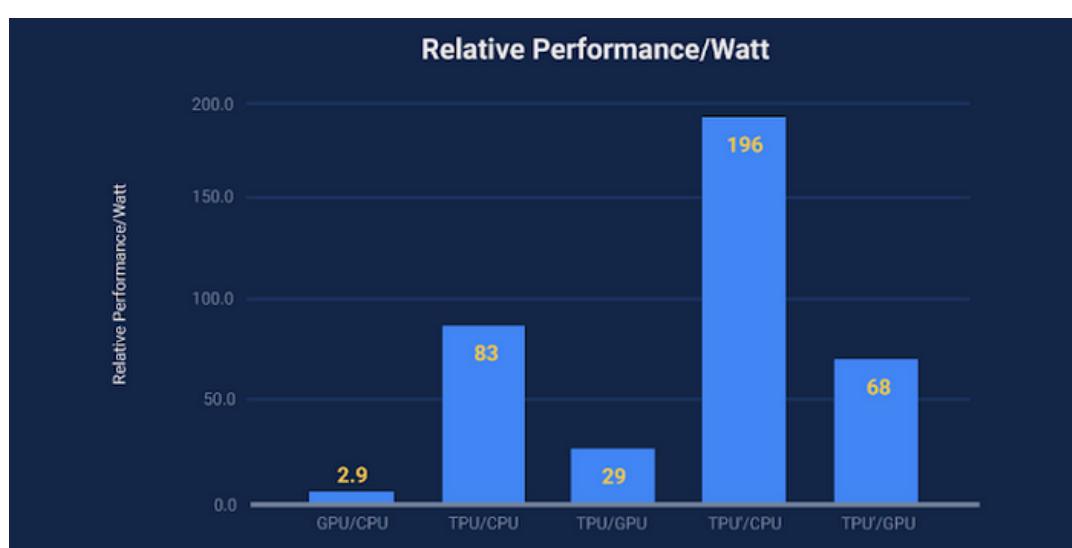
<sup>2</sup>DNN: *Deep Neural Networks* ou Redes Neurais Profundas são uma classe de algoritmos de aprendizado de máquina semelhantes a rede neural artificial e têm como objetivo imitar o processamento de informações do cérebro. A DNN possui mais de uma camada oculta situada entre as camadas de entrada e saída.

Figura 1 – Poder computacional usado no treinamento de sistemas de IA.



Fonte: (LAI SUBTAI AHMAD; MAVER, 2022).

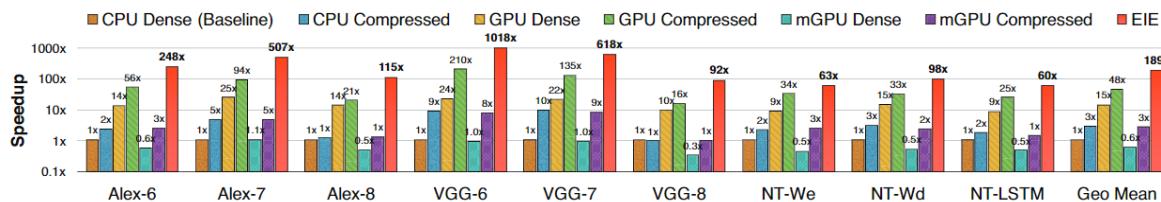
Figura 2 – Desempenho/Watt Relativo (TPU, CPU e GPU).



Fonte: (JOUUPPI, 2017).

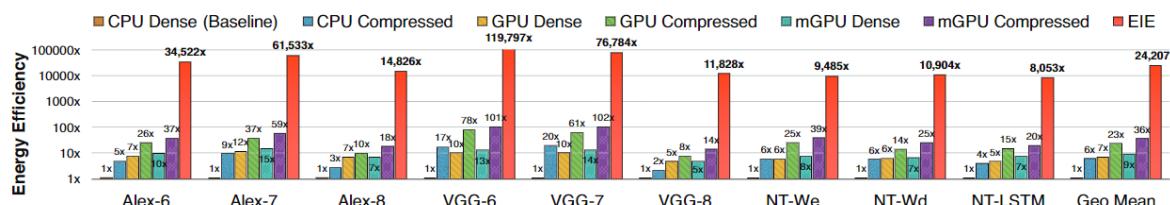
Recentemente a empresa Google optou por desenvolver arquiteturas próprias para uso em *Machine Learning* (ML), ao custo de dezenas de milhões de dólares. Apesar de todo este custo, (Jouppi et al., 2017) (THOMPSON; SPANUTH, 2018) afirma terem avançado em poder de processamento o equivalente a 7 anos da Lei de Moore (THOMPSON;

Figura 3 – Aumento de velocidade de Processamento.



Fonte: (HAN et al., 2016).

Figura 4 – Aumento de eficiência energética (HAN et al., 2016).



Fonte: (HAN et al., 2016).

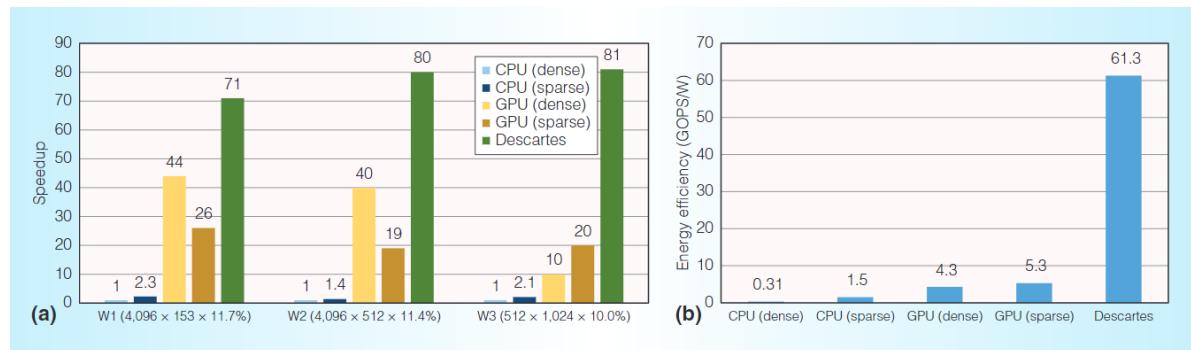
SPANUTH, 2018).

Segundo (THOMPSON; SPANUTH, 2018), em 2013 a empresa Google antecipou o aumento da demanda por pesquisa por voz e projetou que o aprendizado profundo exigiria uma duplicação da capacidade do seu data center. Em vez de adotar soluções convencionais, como processadores universais ou GPUs, o Google optou por desenvolver um processador altamente especializado chamado Tensor Processing Unit (TPU). Esses processadores personalizados são conhecidos como circuitos integrados específicos de aplicativos (ASICs). Embora a criação de um processador personalizado tenha sido um investimento considerável, estimado em dezenas de milhões de dólares, os benefícios foram significativos. O Google alega que o desempenho da TPU foi equivalente a sete anos de avanço previsto pela lei de Moore, sendo o projeto desenvolvido e implantado em 15 meses (SATO, 2017). Além disso, os custos evitados em infraestrutura compensaram o investimento inicial. Em 2017, o Google lançou a segunda geração da TPU, que era oito vezes mais rápida que as GPUs líderes de mercado, conforme medido pelo tempo necessário para treinar um modelo de tradução em larga escala.

Hardware de redes neurais que seja eficiente demanda métodos engenhosos para aproveitar os recursos disponíveis para alcançar alto poder de processamento ou baixo consumo (MISRA; SAHA, 2010). Pesquisas recentes com otimização (compressão do modelo) e projeto conjuntos de software e hardware mostram o desempenho superior

de uma arquitetura dedicada chamada de ‘*Descartes*’ implementada na placa FPGA<sup>3</sup> (*field-programmable gate array*) Xilinx XC7Z02 comparada a CPU Core i7-5930k CPU e a GPU mobile Pascal TitanX GPU) (GUO et al., 2017).

Figura 5 – Comparação de desempenho entre arquitetura dedicada *Descartes* e CPU, GPU.



Fonte: (GUO et al., 2017).

A Figura 5a mostra a aceleração em CPU/GPU implementações em redes densas/esparsas. Descartes até atinge cerca de 2 vezes a velocidade em relação à GPU Pascal TitanX. A Figura 5b mostra a comparação da eficiência energética. Descartes alcança eficiência energética 10 a 200 vezes melhor em relação às outras plataformas.

Apesar de implementações dedicadas em ASIC ou FPGA apresentarem eficiência energética superior a CPU ou GPU, dependendo das implementações o desempenho por inferência pode ser inferior em uma FPGA dependendo do tamanho do *Batch*<sup>4</sup> e do modelo da placa FPGA (NURVITADHI et al., 2016).

Os resultados acima citados, como por exemplo da arquitetura Descartes (Figura 5), demonstram o poder de otimização no processamento de algoritmos de Inteligência Artificial com a construção de arquiteturas dedicadas para tal. Os avançados atraentes destas arquiteturas, somadas com a saturação da Lei de Moore (DENG et al., 2020), empurram o desenvolvimento de hardware para arquiteturas cada vez mais dedicadas, ou seja, diminuindo e vertente de arquiteturas genéricas de propósito geral (THOMPSON; SPANUTH, 2018).

<sup>3</sup>**FPGA (field-programmable gate array):** Circuito integrado projeto para ser reprogramável, baseados em uma matriz de blocos reconfiguráveis (*configurable logic blocks - CLBs*).

<sup>4</sup>**Batch (lote):** O tamanho do lote é um hiper-parâmetro de gradiente descendente que controla o número de amostras de treinamento a serem trabalhadas antes que os parâmetros internos do modelo sejam atualizados.

## 1.2 Problema 2: Projeto de arquiteturas dedicadas

Embora a maioria das aplicações de Redes Neurais Artificiais (*Artificial Neural Networks* ANNs) existentes em uso comercial sejam frequentemente desenvolvidas apenas como software, existem aplicações específicas, como por exemplo detecção de fraude a bancos (conforme já citado na Seção 1.1) e compressão de *streaming* de vídeo, que exige processamento em tempo real de alto volume e aprendizado de grandes conjuntos de dados em tempo razoável exigindo arquiteturas energeticamente eficientes com grande capacidade de processamento paralelo (LIU et al., 2022).

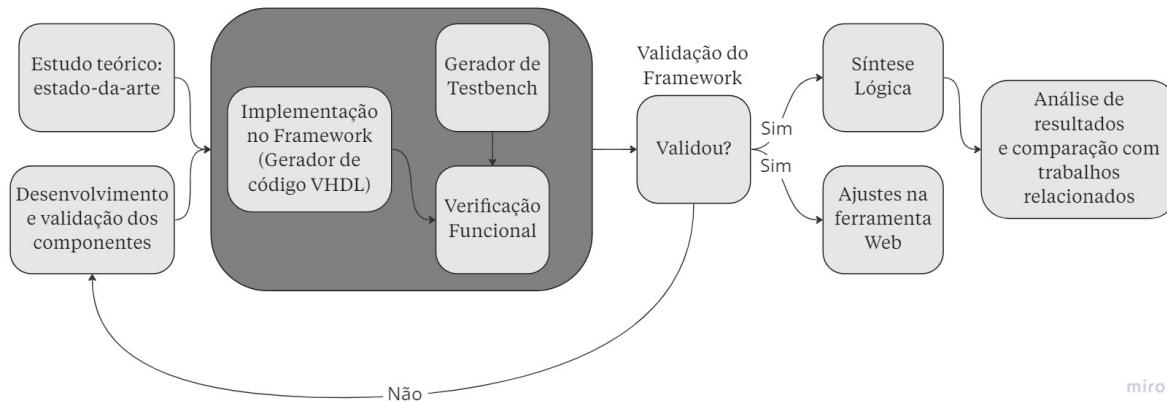
## 1.3 Metodologia

O desenvolvimento do Trabalho de Conclusão de Curso proposto seguiu as seguintes etapas:

- Estudo teórico: estado-da-arte;
- Desenvolvimento e Validação dos componentes;
- Finalização da implementação genérica do *framework* (sem otimizações aritméticas);
- Realização de simulações para verificação funcional de cada componente;
- Programação da Geração de *Testbench* automatizados pelo *framework* para Verificação Funcional;
- Implementação de síntese lógica e obtenção dos resultados estimados.
- Construção e estudo de otimizações aritméticas digitais.

Conforme mostrado na Figura 6 simultaneamente a parte inicial se baseou no estudo teórico do estado-da-arte em trabalhos de mesmo foco e no desenvolvimento e validação de componentes de hardware implementados. O estudo teórico teve foco em implementações de *Framework* para a síntese em alto-nível HLS de hardware dedicado a algoritmos de Redes Neurais Artificiais e/ou síntese de baixo nível de arquiteturas dedicadas de mesmo propósito.

Figura 6 – Fluxo estimado de trabalho.



Fonte: Elaborado pelo autor (2023).

Após estas etapas, o próximo passo foi o de implementar os componentes em linguagem de programação *Python*, o qual tem a função de gerar descrição de hardware em linguagem VHDL. Após isto foi desenvolvido a geração automática *dotestbench* do bloco de topo da DNN. Houve um processo iterativo, pois dependendo dos resultados de verificação funcional, na implementação de cada novo componente ou nova função, surgiu a necessidade ou não de novos ajustes. Quando ocorrido tudo bem, foi feita a síntese lógica.

A contribuição deste trabalho se deu: 1) Na construção do próprio framework; 2) Da disponibilização em repositório online (SIMON, 2023a); 3) Da disponibilização em formato Web acessível (SIMON, 2023b); 4) Do estudo da influência da escolha dos parâmetros do mesmo nos resultados de síntese, como área, atraso e potência consumidos; 5) Do estudo de possíveis otimizações a serem feitas nas arquiteturas que o framework gera atualmente.

Com os resultados de síntese, foi feito a análise dos resultados e comparação com trabalhos relacionados. Estes resultados foram documentados ao final do trabalho.

## 1.4 Motivação e Objetivos

### 1.4.1 Motivação

A motivação envolve os problemas citados:

- GPUs são caras e não otimizadas para os algoritmos de IA, havendo espaço para

otimização em eficiência energética e poder de processamento

- Para projetar arquiteturas dedicadas e otimizadas para cada tipo de arquitetura, é necessário conhecimento sobre os algoritmos de IA, conhecimentos em descrição de hardware (HDL) e tempo de projeto.

Conforme (COLUCCI et al., 2020), tendências da indústria e pesquisa avançaram na direção de IPs<sup>5</sup> (*Intellectual Property*) e ASICs para aceleradores dedicados, tanto para inferência quanto para treinamento dos modelos de IA. O problema dos projetos baseados em ASIC é sua demora para chegar ao mercado, devido ao longo tempo de projeto e fabricação. Outro fator é que os algoritmos de IA estão em rápida evolução, trazendo a necessidade de novas extensões de hardware, o que é inviável devido ao curto período de tempo.

Desta forma um *framework* para geração de código VHDL otimizado para o modelo treinado, traz uma arquitetura otimizada sem a necessidade de conhecimentos em descrição de hardware, em muito menor tempo de projeto, ainda trazendo opções de customizações (para os mais chegados em hardware digital). Resumidamente, a proposta surge para diminuir a complexidade de projeto de uma arquitetura dedicada, com a implantação e otimização automática de aceleradores de DNNs.

### 1.4.2 Objetivos Gerais

O objetivo deste trabalho é desenvolver um *framework* para geração automatizada de código VHDL sintetizável para algoritmos de Redes Neurais Artificiais já treinadas e estudar o efeito de diferentes configurações de arquiteturas no desempenho do algoritmo, assim como seu consumo de área e potência, desta forma, trazendo um ‘guia’ para o usuário na hora de ajuste dos parâmetros no uso do *framework*, dependendo de suas preferências e modelo.

### 1.4.3 Objetivos Específicos

Se tratando dos objetivos específicos, um deles é permitir ao usuário a implementação de seus próprios multiplicadores e somadores aritméticos, criados por ele mesmo.

---

<sup>5</sup>**IP:** Núcleo de propriedade intelectual (núcleo IP) é um bloco funcional de lógica ou dados usados para criar um FPGA ou um circuito integrado de aplicação específica para um produto (ASIC). Comumente usado em semicondutores, um núcleo IP é uma unidade reutilizável de lógica ou projeto de layout de circuito integrado (IC) (AWATI, 2022).

Além da construção das ferramentas no framework, pretende-se: 1) realizar um estudo das diferentes arquiteturas geradas; 2) Realizar uma análise comparativa dessas arquiteturas, levando em consideração o consumo energético, a ocupação de área e a latência; 3) encontrar relações que possam orientar os usuários do framework na escolha da arquitetura mais adequada de acordo com a aplicação desejada. Por último pretende-se trazer um estudo de possíveis otimizações aritméticas possíveis nas arquiteturas geradas pelo framework.

## 2 Referencial Teórico

Este capítulo apresenta uma visão geral sobre os conceitos de Machine Learning (ML) e Deep Learning, destacando suas aplicações e características principais. Serão mencionadas as técnicas de treinamento em ML, como o aprendizado supervisionado, não-supervisionado e por reforço.

Em seguida, é destacada a estrutura das DNNs, com suas camadas de neurônios interconectados, e a importância das funções de ativação na obtenção de respostas não-lineares.

Este capítulo fornecerá uma base teórica necessária para compreender os temas abordados nos próximos capítulos, que se concentra em ferramentas e arquiteturas relacionadas ao desenvolvimento de modelos de Deep Learning.

### 2.1 *Machine Learning*

*Machine Learning* (ML) é uma disciplina da área da Inteligência Artificial que, por meio de algoritmos, dá aos computadores a capacidade de identificar padrões em dados e fazer previsões (análise preditiva). Essa aprendizagem permite que os computadores efetuam tarefas específicas de forma autônoma, ou seja, sem necessidade de serem programados, apenas treinados, com ou sem supervisão (IBERDROLA.COM, 2022).

Modelos de aprendizado de máquina (*Machine Learning*) se diferenciam por suas estruturas e aplicações.

ML tradicionalmente possui 3 técnicas de treinamento:

1. Aprendizado supervisionado
2. Aprendizado não-supervisionado
3. Aprendizado por reforço

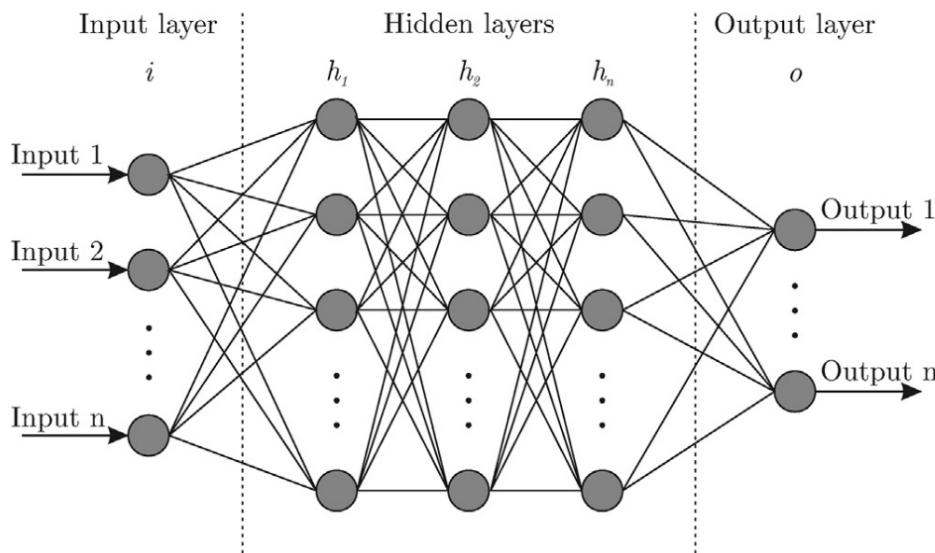
O aprendizado supervisionado envolve a utilização de dados rotulados para treinar algoritmos, ou seja, o modelo aprende a partir de exemplos com respostas conhecidas. Já o aprendizado não-supervisionado não utiliza rótulos; o modelo procura padrões ou estruturas nos dados por conta própria, sem orientação explícita, como em agrupamentos ou redução de dimensionalidade.

## 2.2 Deep Learning

Deep Learning é uma categoria da disciplina de ML que faz o uso de Redes Neurais Artificiais (Artificial Neural Networks DNNs).

DNNs são nada mais do que uma rede de nodos interligados chamados de neurônios artificiais. Em cada camada da DNN os neurônios recebem múltiplas entradas diferentes vindas da camada anterior da DNN e saídas iguais para os neurônios da próxima camada.

Figura 7 – Arquitetura de uma Rede Neural Artificial genérica (DNN  $i \rightarrow h_1 \rightarrow h_2 \rightarrow h_n \rightarrow o$  ).



Fonte: (BRE; GIMENEZ; FACHINOTTI, 2018).

Na Figura 7 podemos ver a estrutura genérica de uma DNN, constituída pela camada de entrada (*Input Layer*), camadas intermediárias ou ocultas (*Hidden layers*) e a camada de saída (*Output layer*).

As redes MLP (Multi-Layer Perceptron) são caracterizadas pela composição de múltiplas funções em cascata. Uma rede com três camadas, por exemplo, pode ser representada como  $f(x) = f(3)(f(2)(f(1)(x)))$ . Cada uma dessas camadas é composta por unidades que realizam uma transformação afim, que equivale a uma soma linear das entradas ponderadas. Matematicamente, cada camada pode ser representada como  $Y = f(Wx^T + \gamma)$ , onde  $f$  é a função de ativação (a qual será discutida posteriormente),  $W$  representa o conjunto de parâmetros ou pesos específicos daquela camada,  $x$  é o vetor de entrada, que também pode ser a saída da camada anterior, e  $\gamma$  é o vetor de polarização.

Para representar melhor o conjunto  $Y$  é mostrado abaixo uma matriz de neurônios, onde os índices  $i$  e  $j$  representam as linhas (neurônios) e as colunas (camadas), ou seja,  $y_{23}$  é o segundo neurônio da terceira camada.

$$\begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{bmatrix} \quad (2.1)$$

Na equação abaixo, é possível ver que o primeiro neurônio da segunda camada ( $y_{12}$ ) terá seu resultado sendo a multiplicação de cada um dos resultados dos neurônios da camada anterior ( $y_{11}$ ,  $y_{21}$  e  $y_{31}$ ) multiplicados pelos seus pesos respectivos ( $w_{1_{12}}$ ,  $w_{2_{12}}$  e  $w_{3_{12}}$ ) sendo ao final somados ao viés  $\gamma_{12}$ .

$$y_{12} = f \left( \sum_{i=1}^{I_1} y_{i1} \cdot w_{i_{12}} + \gamma_{12} \right) \quad (2.2)$$

É relevante observar que as camadas de um MLP são tipicamente construídas como camadas totalmente conectadas, o que significa que cada unidade em uma camada está conectada a todas as unidades da camada anterior. Dentro de uma camada totalmente conectada, os parâmetros de cada unidade são independentes das demais unidades da mesma camada, resultando em um conjunto único de pesos para cada unidade.

O que diferencia uma DNN de outra são diversos fatores, dentre os mais importantes: o número de camadas, número de neurônios em cada camada, valores dos pesos e viés de cada neurônio, funções de ativação de cada camada.

DNNs com um número muito grande de camadas ocultas são conhecidas como DNNs (*Dense Neural Networks*) ou em outras vezes MLPs (*Multi-Layer Perceptrons*).

## 2.2.1 Perceptron

Os neurônios possuem uma estrutura interna que se diferencia do seu modelo. Neurônios artificiais conhecidos como Perceptron, multiplicam cada uma das diferentes entradas por diferentes pesos e somam os resultados de todas as multiplicações com um valor de deslocamento conhecido como *bias*. Este processo de multiplicações e somas é conhecido como MAC (*Multiply And Accumulate* ).

$$y = f \left( \sum_{j=1}^n x_j \cdot w_j + \gamma \right) \quad (2.3)$$

O modelo matemático do funcionamento de um Perceptron é apresentado na equação 2.3. O diagrama de blocos de um Perceptron é representado na Figura 8, onde os pesos

são indicados como  $w_1, w_2, \dots, w_j$ , os dados de entrada são referidos como  $x_1, x_2, \dots, x_j$ , o viés é representado por  $\gamma$  e a função de ativação é  $f$ .

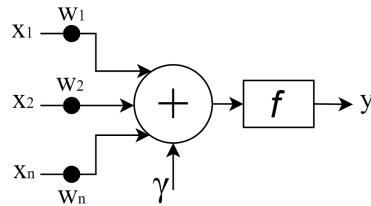
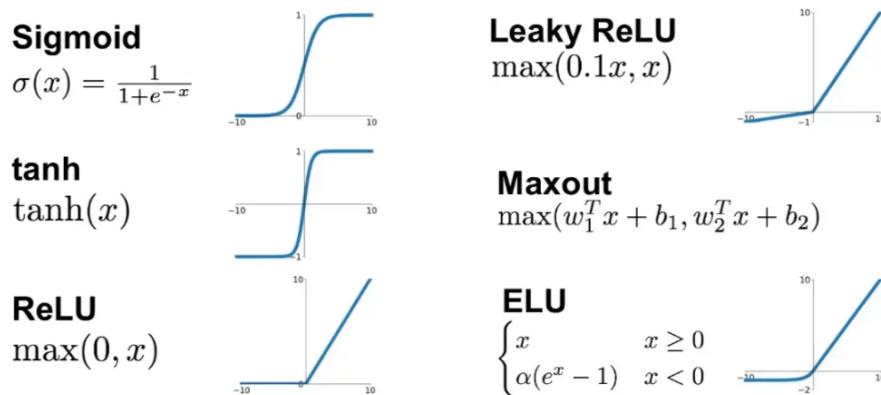


Figura 8 – Diagrama de blocos do Perceptron.

## 2.2.2 Funções de Ativação

Após todo este processo, o valor resultante do MAC passa por uma função de ativação, que permite saídas não-lineares às entradas. O poder de aprendizado e generalização dos dados de treinamento do algoritmo de *Deep Learning* é graças à resposta não-linear das funções de ativação de cada neurônio. Esta diferença dos demais algoritmos é uma das principais características que diferencia as Redes Neurais Artificiais de outros algoritmos de ML em seu potencial e aplicações.

Figura 9 – Exemplo de Funções de Ativação.



Fonte: (JADON, 2018).

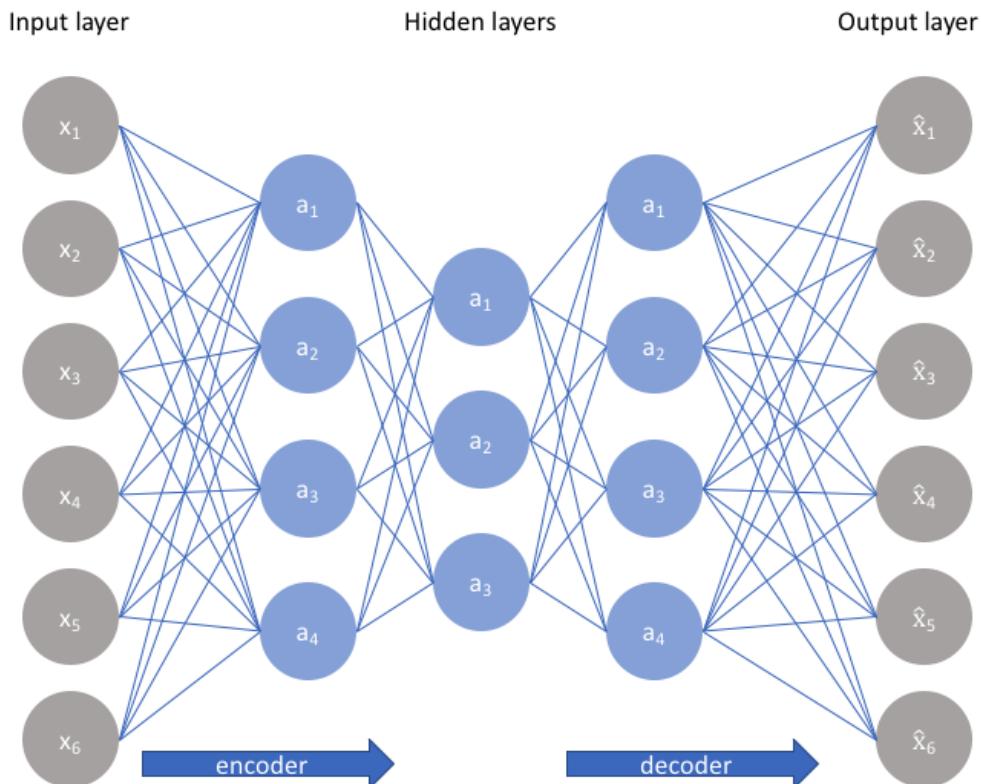
Funções de ativação definem a saída de um nó dada uma entrada ou conjunto de entradas. Funções de ativação lineares nos fornecem Perceptrons lineares. Funções de ativação não lineares, tornam os Perceptrons não-lineares e permitem que essas redes calculem problemas não triviais usando apenas um pequeno número de nós (HINKELMANN, 2022).

Dentre as funções de ativação possíveis, três das mais conhecidas e comumente usadas são a ReLU (*Rectified and Logic Unit*), Sigmoide ou Logistica e Leaky ReLU, porém muitos outros tipos existem e alguns deles podem ser vistos na Figura 9.

### 2.2.3 Autoencoders

*AutoEncoders* feitos com MLP (*Multi-layer Perceptrons*) são redes neurais artificiais usadas para aprendizado não supervisionado e redução de dimensionalidade. Eles consistem em duas partes: um codificador, que mapeia dados de alta dimensão para uma representação de menor dimensão, e um decodificador, que reconstrói os dados originais a partir da representação reduzida. Essas redes são úteis para tarefas como compressão de dados, redução de dimensionalidade (AMORIM, 2022), codificadores de vídeo (HABIBIAN et al., 2019), remoção de ruídos (DSA, 2019) em imagens, e extração de características relevantes, como em reconhecimento de padrões, detecção de anomalias (NASCIMENTO et al., 2021) e geração de conteúdo.

Figura 10 – Arquitetura genérica de um *AutoEncoder*.



Fonte: (JORDAN, 2018).

Modelos conhecidos como *AutoEncoders*, são utilizados principalmente para aprendizado não-supervisionado, ou seja, quando não há um ‘gabarito’ de respostas corretas.

Os *Autoencoders* (AE) são redes neurais que visam copiar suas entradas para suas saídas. Eles trabalham compactando a entrada em uma representação de espaço latente e, em seguida, reconstruindo a saída dessa representação (DSA, 2019).

Conforme já falado, esse tipo de rede é composto de duas partes (como mostra a Figura 10):

Codificador (*Encoder*): é a parte da rede que compacta a entrada em uma representação de espaço latente (codificando a entrada). Pode ser representado por uma função de codificação  $h = f(x)$ .

Decodificador (*Decoder*): Esta parte tem como objetivo reconstruir a entrada da representação do espaço latente. Pode ser representado por uma função de decodificação  $r = g(h)$ , Academy (2022).

Em resumo, os Autoencoders baseados em MLP (Multi-layer Perceptrons) são poderosas ferramentas de aprendizado não supervisionado, destacadas por seu papel na redução de dimensionalidade e na reconstrução eficiente de dados. Compostos por um codificador e um decodificador, essas redes têm aplicações amplas, incluindo compressão de dados, remoção de ruídos em imagens, detecção de anomalias e geração de conteúdo. Sua capacidade de aprender padrões sem depender de rótulos de resposta correta os torna valiosos para explorar correlações intrínsecas nos dados. Este capítulo apresentou uma visão abrangente dos Autoencoders, ressaltando seu papel em diversas aplicações de processamento de informações e reconhecimento de padrões.

### 3 Trabalhos relacionados

Conforme Lebedev e Belecky (2021), atualmente existem diversas ferramentas e frameworks para algoritmos de inteligência artificial as quais podem ser divididas em 3 grupos:

Ferramentas que otimizam e preparam um modelo de Rede Neural Artificial:

1. Para ser sintetizável em um modelo RTL<sup>1</sup> que pode tanto ser executado em uma FPGA quanto implementado em ASIC<sup>2</sup> (sintetizadores em RTL);
2. Em um executável binário para um processador de arquitetura específica (compiladores);
3. Para placas de propósito específico como FPGAs (sintetizadores para placas específicas);

Assim como a proposta deste presente trabalho, a revisão do estado da arte engloba apenas trabalhos do primeiro grupo, ou seja, modelos sintetizáveis em RTL.

#### 3.1 Ferramentas de código-aberto

A seção apresenta dois notáveis frameworks de código aberto para a geração de arquiteturas de aceleradores em redes neurais artificiais. O hls4ml, abrangente e versátil, aceita modelos em diversos formatos e realiza otimizações como o treinamento consciente da quantização. Já o NNGen, ao aceitar modelos em formato ONNX ou usando descrições Python, gera hardware completo, incluindo processamento, memória e controladores, sem requerer controle externo pós-processamento. A Tabela comparativa destaca suas características distintivas. O hls4ml destaca-se como a única ferramenta que realiza síntese em ASIC automaticamente, embora seja fundamental considerar a relação entre funcionalidades e qualidade das arquiteturas geradas.

---

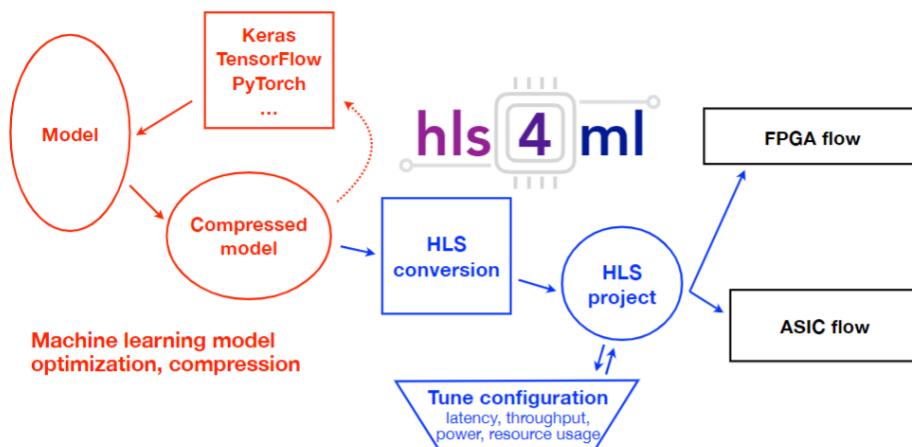
<sup>1</sup>**Register Transfer Level (RTL):** RTL é uma abstração dos componentes digitais de um circuito de hardware. É a principal abstração usada para definir sistemas eletrônicos atualmente e geralmente serve como modelo de ouro no fluxo de design e verificação (SEMIENGINEERING.COM, 2021).

<sup>2</sup>**ASIC:** Circuitos integrados de aplicação específica (*application-specific integrated circuit* ASIC) é um chip de circuito integrado projetado para uma finalidade específica.

### 3.1.1 hls4ml

hls4ml é um *framework* de código aberto (*open-source*) sendo um dos mais completos. Seu sistema aceita modelos de Rede Neural Artificial em diversos formatos de entrada, como ONNX, Tensorflow, Keras e Pytorch, sendo as bibliotecas mais conhecidas e utilizadas atualmente, otimiza estes modelos com técnicas como *quantization-aware training*<sup>3</sup> e *Pruning*<sup>4</sup>, implementa a conversão em síntese de alto-nível (HLS<sup>5</sup>) em linguagem C/C++, implementa ajuste finos dependendo do foco de arquitetura e de projeto, analisando latência, desempenho e consumo de potência, consumo de recursos (componentes disponíveis na placa desejada ou área ocupada em uma síntese ASIC) trazendo ainda a possibilidade de implementações com foco em baixo consumo energético. O fluxo de trabalho do *framework* hls4ml pode ser visto na Figura 11.

Figura 11 – Fluxo de trabalho do *framework* hls4ml.



Fonte: (FAHIM et al., 2021).

<sup>3</sup>**Pruning (Poda):** Esta técnica consiste em retirar partes do modelo que menos colaboram para o resultado de saída do modelo, com foco em reduzir o número de área consumida e operações realizadas como por exemplo: neurônios "mortos" em que sua saída está sempre tendendo a valores muito baixos (ocorre muito com a função de ativação ReLU) que resultariam em pouco impacto na camada posterior. A técnica de Poda também pode ser realizada para retirar partes do modelo com valores muito altos, buscando assim a diminuição do consumo energético (SZE et al., 2017).

<sup>4</sup>**Quantization-aware training:** Esta é uma treinamento do modelo já consciente da quantização dos pesos da rede. Estudos mostram que quantizar os pesos durante o treinamento trazem uma acurácia melhor comparado a treinar o modelo normalmente e só ao final quantizar os pesos do modelo (FAHIM et al., 2021).

<sup>5</sup>**HLS (High Level Synthesis) :** A síntese de alto nível (HLS), às vezes chamada de síntese algorítmica ou síntese comportamental, é um processo de design automatizado que pega uma especificação comportamental abstrata de um sistema digital e encontra um registro -estrutura de nível de transferência (RTL) que realiza o comportamento dado Wikipedia (2022).

### 3.1.2 NNGen

NNGen é um *framework* de código aberto que aceita modelos descritos em formato ONNX ou utilizando uma descrição própria em linguagem de programação Python. O hardware gerado é completo, o que inclui mecanismo de processamento, memória on-chip, rede on-chip, controlador DMA<sup>6</sup> e circuitos de controle. Portanto, o hardware gerado não requer nenhum controle adicional de um circuito externo ou da CPU após o início do processamento (YAMAZAKI, 2022).

## 3.2 Tabela Comparativa dos principais *Frameworks*

Uma análise qualitativa de trabalhos similares foi feita e pode ser vista na Tabela 1. A coluna 'selecionados', indica se o trabalho foi ou não comentado no capítulo 3. Importante denotar que o único trabalho que realiza a síntese em ASIC automaticamente é o *framework* **hls4ml**, o qual possui sua última atualização no ano de 2021 porém demonstra ser a ferramenta com mais funcionalidades, o que não necessariamente traz as melhores arquiteturas.

Tabela 1 – Tabela Comparativa de *frameworks*

Nome	Ano Public.	Ano Atualiz.	Selec.	Comprime o modelo	Treina o modelo	Ajuste fino parâmetros	Otimização para Zynq-7000	Configurações de arquitetura personalizáveis	Código aberto	Faz verificação	Sintetiza em ASIC	Faz validação da síntese
hls4ml	2021	2021	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LeFlow	2018	2019	X	X	X	X	X	X	✓	X	X	X
ONNC	2019	2020	X	X	X	X	X	X	✓	X	X	X
NNGen	-	2022	✓	X	X	X	X	✓	✓	✓	X	X
DnnWeaver	2016	2019	X	X	X	✓	X	X	✓	X	X	X
PandA-bambu	2021	2022	X	✓	X	✓	✓	✓	✓	✓	X	X
FP-DNN	2017	-	X	✓	X	X	X	X	X	X	X	X
NVDLA	2018	2018	X	✓	✓	X	X	✓	✓	✓	X	X
yolowell	2019	2020	X	✓	X	X	X	X	✓	X	X	X
Vitis AI (Xilinx)	2020	2022	X	✓	X	✓	✓	✓	X	X	X	✓
OpenVINO (Intel Arria FPGA)	2022	2022	X	✓	✓	✓	X	X	X	✓	X	X

Fonte: Elaborado pelo autor (2023).

A coluna (Ano Public.) indica o último momento de atualização do framework e dá uma ideia do quanto recentemente ele foi lançado. A coluna (Ano Atualiz.) indica o quanto recente é a última atualização do *framework*. A quinta coluna indica se há algum tipo de compressão do modelo treinado para melhor se adaptar à realidade em hardware. A sexta coluna indica se o *framework* já faz algum tipo de treinamento de modelos em software. A sétima coluna indica se há otimizações de arquitetura específicas para a placa FPGA Zynq-7000 da Xilinx. A coluna (Configurações de arquitetura personalizáveis) indica se

<sup>6</sup>**DMA:** O DMA ou *Direct Memory Access* é o método que permite que um dispositivo de entrada e saída envie ou receba dados diretamente da memória principal, ignorando a CPU, acelerando as operações que envolvem a memória (TECNOBLOG, 2022).

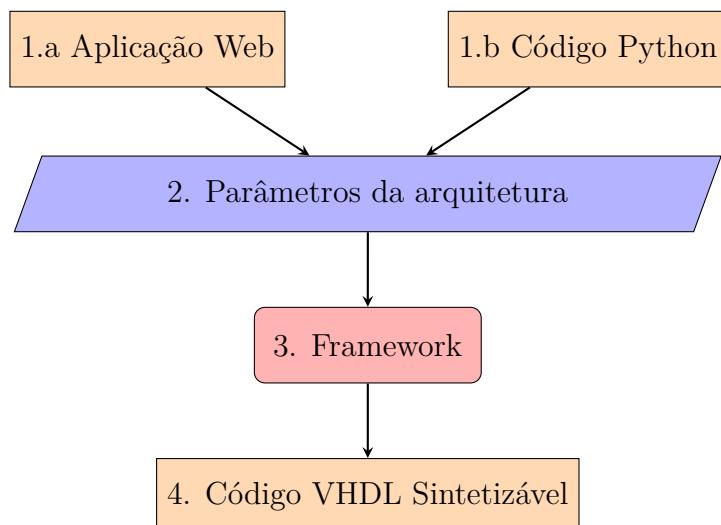
o usuário possui alguma liberdade de configuração do tipo e modo de arquitetura gerada em hardware. A coluna (Faz verificação) indica se o framework fornece arquivos de teste ou faz algum tipo de verificação da arquitetura gerada em hardware. As duas últimas colunas dizem respeito à síntese física, indicando se o *framework* faz a síntese do RTL gerado e se faz validação da síntese, estimando área, atraso crítico e potência limitados pelo projetista.

# 4 Proposta

## 4.1 Framework

A proposta se baseia na construção de um *framework* HLS<sup>5</sup> para a rápida síntese e prototipação de arquiteturas de algoritmos de Redes Neurais Artificiais de estrutura MLP<sup>1</sup>, utilizando linguagem de programação Python e descrição de hardware VHDL. O mesmo permite a exploração de diferentes configurações de arquiteturas através de configurações parametrizáveis e modelos de entrada e a possibilidade ao usuário da implementação de novas estruturas (código-aberto).

Figura 12 – Fluxograma do Framework



Fonte: Elaborado pelo autor (2023).

Conforme Figura 12, o usuário poderá optar por 2 alternativas como entrada do *framework*

- **Aplicação Web:** versão mais acessível a usuários sem conhecimento de linguagens HDL e Python. Permite a geração de uma arquitetura em poucos minutos (SIMON, 2023b).

---

<sup>1</sup>**MLP - Multi-Layer Perceptron :** Um multilayer perceptron (MLP) é uma classe totalmente conectada de rede neural artificial (DNN) *feedforward* (Multilayer perceptron, 2022), ou seja, suas conexões não formam um ciclo.

- **Código-aberto:** versão que disponibiliza novas implementações e modificações a usuários mais imersos em descrição de hardware e linguagem Python (SIMON, 2023a). Possibilita uma maior costumização por parte do usuário.

## 4.2 Otimizações Aritméticas para Arquiteturas Geradas pelo Framework

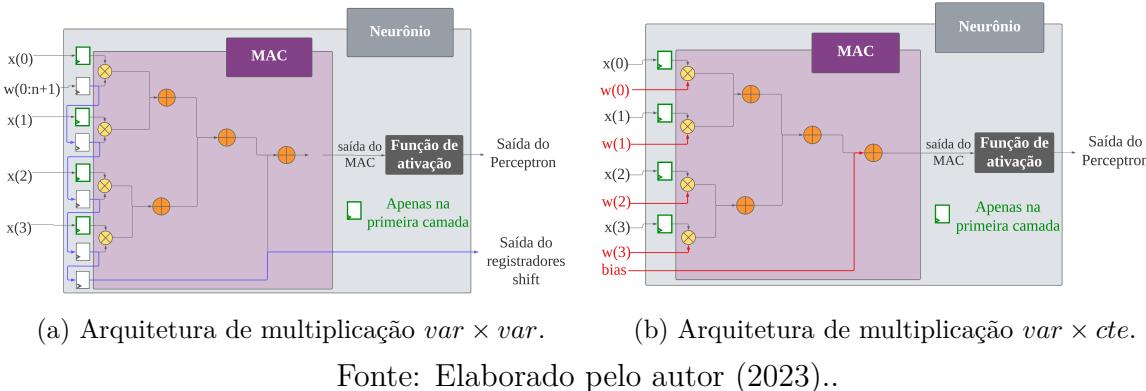
Este capítulo aborda a investigação de possíveis melhorias para as arquiteturas geradas pelo framework desenvolvido neste trabalho. Devido à necessidade de implementar um conversor de modelos treinados (Pytorch, Tensorflow, etc.) para gerar um arquivo texto de entrada, essencial para a etapa de atualização da arquitetura (atualização dos registradores de pesos e viéses), os testes abrangentes nas arquiteturas geradas pelo framework não puderam ser conduzidos no momento. Inicialmente, utilizou-se um algoritmo Python desenvolvido em colaboração com o aluno Lucas Soares, sob a orientação do professor Mateus Grellert, que simula as operações feitas em RTL. O mesmo pode ser encontrado em pode ser encontrado em (SIMON, 2023a) na pasta "truncatedNN". No entanto, a realização de testes mais rigorosos diretamente na arquitetura (por meio de *testbenches*) poderia ser considerada no futuro para garantir resultados mais confiáveis. Essa limitação oferece uma oportunidade para áreas de aprimoramento e sugestões para trabalhos futuros, além de fornecer contribuições para outros pesquisadores interessados na criação de arquiteturas de Redes Neurais Artificiais (RNAs) que fazem uso de unidades aritméticas de multiplicação e soma, conhecidas como MACs (Multiply and Accumulate) para redes multi-camadas de perceptrons (MLPs).

Como as arquiteturas geradas pelo framework, possuem sua unidades aritméticas de multiplicações e somas (MACs) combinacionais, esta abordagem traz um grande consumo de área, pois traz um multiplicador individual para cada operação de multiplicação e um somador para cada operação de adição.

Logo baseado neste grande consumo de área, a investigação para aprimorar as arquiteturas geradas pelo framework concentrou-se em duas abordagens principais: a substituição dos multiplicadores dos MACs por blocos de somas e deslocamentos (*shifts*) e a troca de todos os somadores por um bloco de compressão dos vetores. Essas melhorias têm como objetivo otimizar o desempenho e a eficiência das arquiteturas geradas, levando em consideração as restrições de recursos e as necessidades específicas dos sistemas embarcados que utilizam essas RNAs.

Entretanto, é importante destacar que a comparação não pode ser considerada com-

Figura 13 – Arquiteturas de multiplicação combinacional (pesos variáveis vs pesos fixos).



Fonte: Elaborado pelo autor (2023)..

pletamente justa. Na primeira versão da rede neural, que é a única gerada pelo framework, os valores de entrada ( $x_i$ ) e pesos ( $w_i$ ) são variáveis na arquitetura, devido os pesos e viés poderem ser alterados a qualquer momento, pois estão salvos nos registradores, que podem ser atualizados sempre que necessário, conforme Figura 13a. Já na segunda versão, apenas as entradas eram variáveis ( $x_i$ ), com os pesos sendo agora fixos pois estão contidos nas arquiteturas fixas de somas e deslocamentos (destacados em vermelho). Logo para uma comparação justa (usando pesos sempre como constantes) e devido à possibilidade de se fazer uma série de otimizações para este tipo de arquitetura (quando já sabemos quais serão os valores dos pesos), as análises posteriores fazem a comparação com a arquitetura de multiplicação de variáveis (entradas) por valores fixos (pesos e vieses), entradas e pesos respectivamente, conforme Figura 13b.

#### 4.2.1 Substituição dos Multiplicadores por Blocos de Somas e Deslocamentos

Uma das estratégias investigadas consiste na substituição dos multiplicadores dos MACs por blocos de somas e deslocamentos. Essa abordagem se baseia na observação de que, em algumas aplicações de RNAs, certos padrões de multiplicação podem ser representados de forma mais eficiente por operações de soma e deslocamento. A substituição dos multiplicadores por blocos de somas e deslocamentos pode reduzir a complexidade dos circuitos gerados, diminuindo a área ocupada e o consumo de energia. As arquiteturas de Somas e Deslocamentos foram geradas com o auxílio da ferramenta Spiral, de autoria de Yevgen Voronenko para o projeto Spiral, pela universidade de Carnegie Mellon (PÜSCHEL, 2007). Uma das abordagens eficientes para a multiplicação

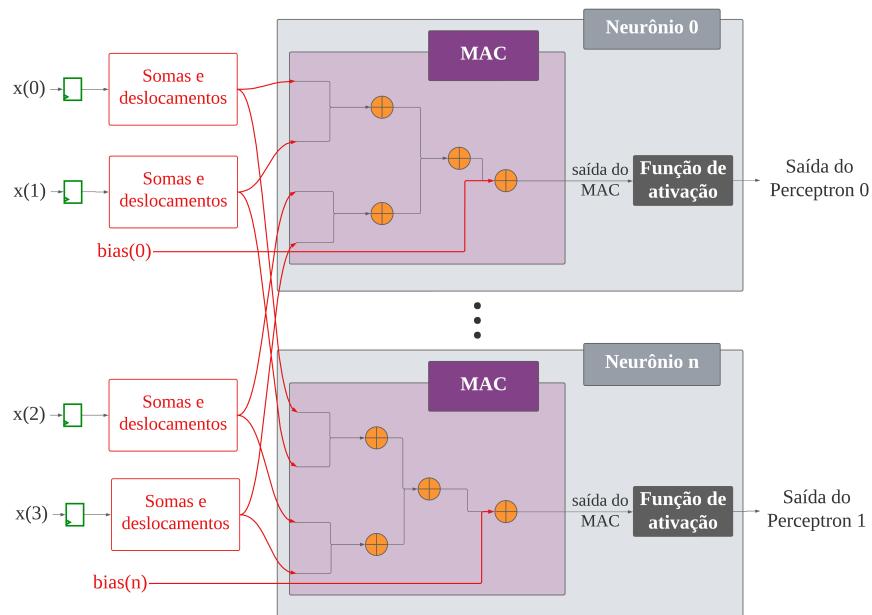
de uma variável por um conjunto de constantes em ponto fixo é por meio de um bloco multiplicador que consiste exclusivamente de adições, subtrações e deslocamentos. A geração desse bloco multiplicador a partir do conjunto de constantes é conhecida como o problema da multiplicação por múltiplas constantes (*Multiple Constant Multiplication - MCM*). Encontrar a solução ótima, ou seja, aquela que requer o menor número de adições e subtrações, é conhecido por ser um problema NP-completo.

Um exemplo de equação que representa a equivalência de uma multiplicação por somas e deslocamento é:

$$9a = 8a + a \quad (4.1)$$

Nesta equação,  $a$  é a entrada variável que pode assumir qualquer valor e 9 é a constante de multiplicação. Nas arquiteturas de RNAs estas constantes são os pesos e viés (bias) dos neurônios.

Figura 14 – Blocos de soma e deslocamentos (Spiral).



Fonte: Elaborado pelo autor (2023)..

A Figura 14 apresentada ilustra a adaptação da arquitetura de múltiplos neurônios em uma rede neural, onde é aproveitada a vantagem de compartilhar as mesmas entradas da camada anterior para cada neurônio, enquanto cada neurônio possui seus próprios pesos e viés individuais. Para implementar todas as multiplicações correspondentes às

mesmas entradas, provenientes de neurônios diferentes, a arquitetura utiliza blocos de somas e deslocamentos em vez de multiplicadores.

A Figura 14 mostra por exemplo, a utilização desses blocos de somas e deslocamentos para a entrada  $x(0)$ , onde um único bloco é responsável por realizar todas as operações de multiplicação necessárias para cada neurônio da camada. Cada saída do bloco de somas e deslocamentos é conectada a um neurônio específico na camada, permitindo que os pesos individuais de cada neurônio sejam aplicados às entradas correspondentes.

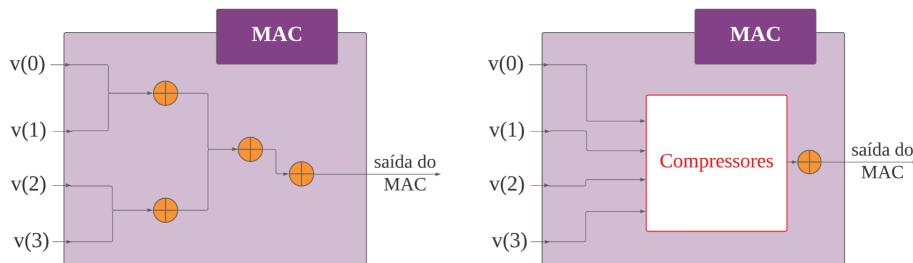
Essa abordagem proporciona eficiência na implementação, pois reduz a necessidade de multiplicadores individuais para cada neurônio e aproveita o compartilhamento de entradas. Ao agrupar as operações de multiplicação em um único bloco de somas e deslocamentos relacionado a cada entrada, a arquitetura simplifica o circuito e melhora a eficiência computacional.

Essa figura representa uma solução alternativa e eficaz para implementar a multiplicação em arquiteturas de redes neurais, permitindo o processamento paralelo e otimizado de múltiplos neurônios em uma única camada e o reaproveitamento de hardware.

#### 4.2.2 Troca dos Somadores por um Bloco de Compressão de Vetores

Esta abordagem visou a substituição de todos os somadores presentes nas arquiteturas geradas por um bloco de compressão de vetores, conforme exemplificado na Figura 15. Na arquitetura gerada pelo framework valores multiplicados são acumulados com uma árvore somadora, que possui níveis de somadores  $\log_2(N)$ . Em seguida, é realizada uma adição final com o viés, após disso a saída MAC está pronta.

Figura 15 – Substituição das somas por um bloco de compressores.



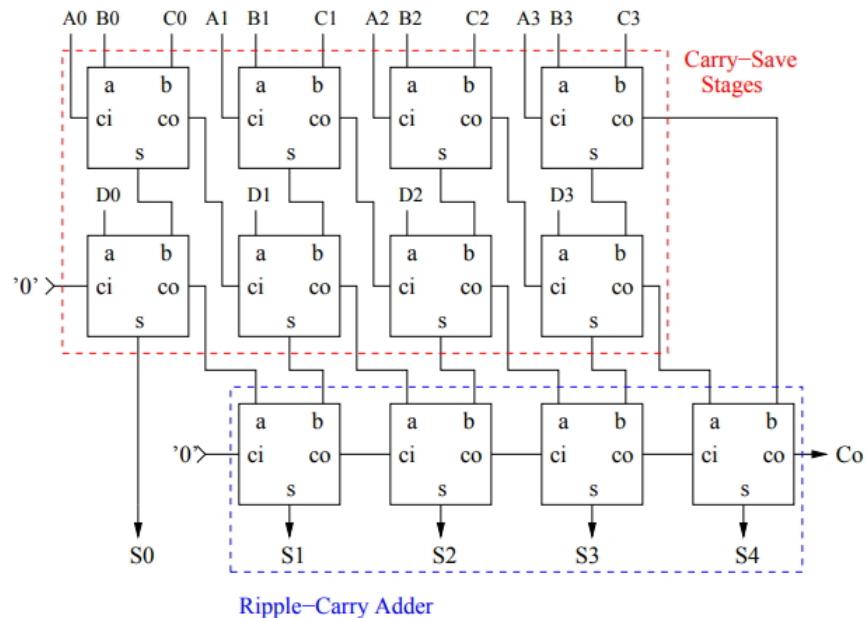
Fonte: Elaborado pelo autor (2023)..

Essa estratégia se baseia no conceito de que, em muitas aplicações de RNAs, os somadores convencionais podem ser substituídos por técnicas mais eficientes de compressão

de dados. O objetivo dessa substituição é reduzir a área ocupada pelos somadores a troca de um provável maior consumo de energia e tempo de processamento.

A investigação envolveu o uso de compressores conhecidos como CSAs (*Carry-Save Adders*), arquitetura que possui uma série de compressores em cascata dos múltiplos vetores de soma da operação do MAC, para ao final, restando apenas 2 vetores, os mesmos são somados com um somador *Ripple-Carry*, conforme Figura 16.

Figura 16 – *Carry-Save Adders*.



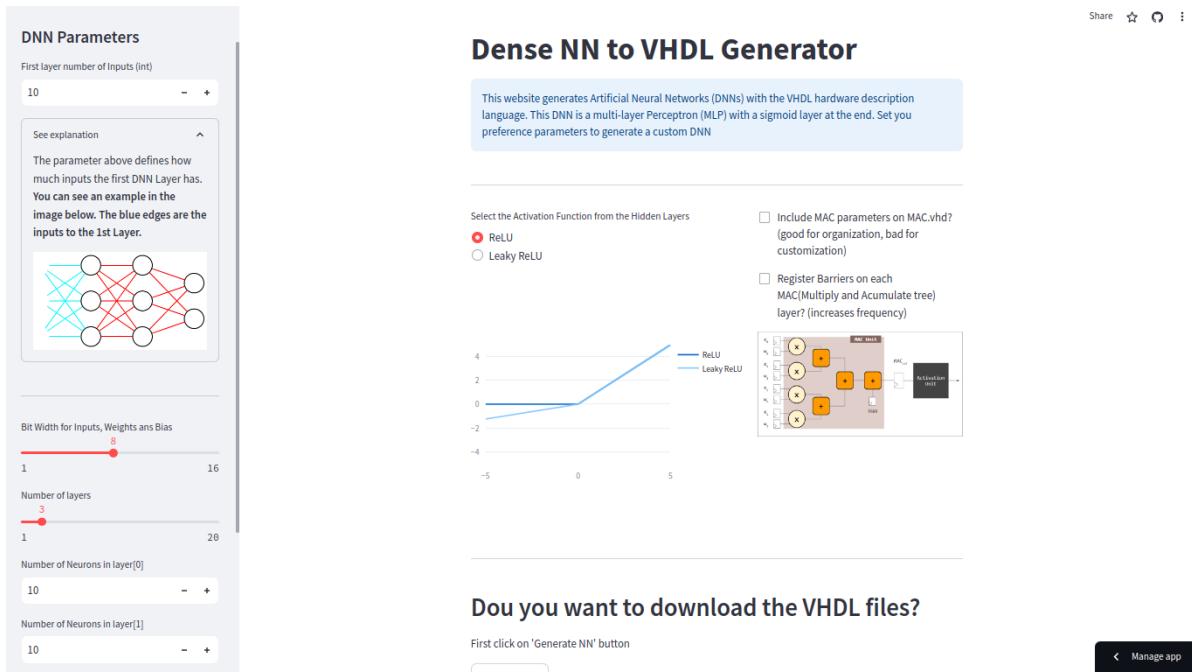
Fonte: (GYAN, 2022).

# 5 Resultados

## 5.1 Disponibilização e uso do Framework

O *framework* foi disponibilizado tanto em repositório online (SIMON, 2023a), quanto em aplicação web (SIMON, 2023b). A captura de tela da aplicação web do *framework* na Figura 17 revela uma interface intuitiva que oferece um ajuste fácil de parâmetros cruciais para a criação de redes neurais. Notavelmente, o usuário pode personalizar o número de entradas da rede neural, a largura de bits de representação, o número de camadas, e a quantidade de neurônios por camada, entre outros. Essa flexibilidade é destacada pela simplicidade de alguns cliques, permitindo ao usuário realizar ajustes precisos. Um elemento proeminente na interface (que não pôde ser visto na imagem) é o botão "Generate NN", oferecendo uma solução conveniente para gerar e baixar uma pasta compactada contendo todos os arquivos VHDL da arquitetura personalizada escolhida. Essa abordagem simplificada destaca a usabilidade e acessibilidade do framework, proporcionando uma experiência eficiente ao usuário.

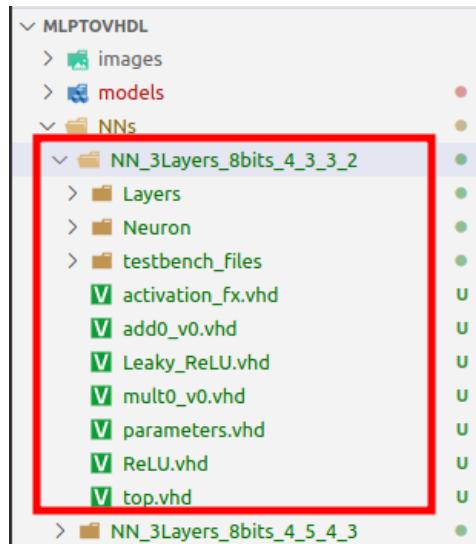
Figura 17 – Aplicação web do *framework*.



Fonte: Elaborado pelo autor (SIMON, 2023b).

Na Figura 18 é possível ver a estrutura de arquivos gerados para um tipo específico de arquitetura. Nela é possível ver alguns arquivos VHDL e algumas pastas, como por exemplo a pasta de *testbenches*, a qual possui um testbench de varredura de valores possíveis de entrada no arquivo de topo (arquivo RTL de hierarquia mais alta da arquitetura).

Figura 18 – Estrutura de arquivos gerados pelo *framework*.



Fonte: Elaborado pelo autor (2023).

Um exemplo de como executar a geração de arquiteturas pelo *framework* através da IDE Microsoft VSCode, pode ser visto na Figura 19. Os destaque em verde, indicam qual arquivo já pré-preenchido (`RUN_GEN_TOP_LEVEL_HDL.py`) pode ser usado para executar uma arquitetura. Neste arquivo, pode-se ver a definição dos parâmetros mais básicos, como por exemplo o parâmetro da linha 6 (`LAYER_NEURONS_NUMBER_LIST`). O mesmo definido como igual a `[3, 3, 2]` indica um MLP com 3 camadas, sendo 2 camadas intermediárias, cada uma com 3 neurônios e uma camada de saída com 2 neurônios, sendo os seus arquivos VHDL mostrados na Figura 18 comentada anteriormente. Alguns códigos VHDL desta arquitetura são mostrados no Apêndice A. Parâmetros de customização mais avançados podem ser definidos nos dicionários base para as camadas ocultas (`layer_dict_hidden`) e camada de saída (`layer_dict_softmax`), ambos se encontram no arquivo abaixo.

`Python_script/utils/standard_dicts.py`

O destaque em vermelho sinaliza como executar o arquivo através da IDE.

Já a Figura 20 mostra um exemplo em como executar o *framework* através do prompt

Figura 19 – Executando o *framework* através da IDE Microsoft VSCode.

RUN\_GEN\_TOP\_LEVEL\_HDL.py - MLptoVHDL - Visual Studio Code

Edit Selection View Go Run Terminal Help

EXPLORER

OPEN EDITORS

RUN\_GEN\_TOP\_LEVEL\_HDL.py u

MLPTOHDL

images

models

NNS

Python script

\_mpyache\_

ignore

model\_conversion

planning

utils

dict\_to\_dense.py

gen\_Autoencoder\_models.py

GEN\_TOP\_LEVEL\_HDL teste .cp.py

GEN\_TOP\_LEVEL\_HDL.py

LICENSE.txt

README.md

RUN GEN TOP LEVEL HDL.py

setup.py

truncatedNN

web

.gitignore

framework\_workflow\_2.png

framework\_workflow.png

LICENSE

README.md

requirements.txt

web.py

OUTLINE

TIMELINE

SVN

Python: RUN\_GEN\_TOP\_LEVEL\_HDL.py

```
1 from GEN_TOP_LEVEL_HDL import *
2
3 INPUTS_NUMBER = 4
4
5 BIT_WIDTH = 8
6 LAYER_NEURONS_NUMBER_LIST = [3, 3, 2]
7 IO_TYPE_STR = 'signed'
8
9 BASE_DICT_HIDDEN = layer_dict_hidden
10 BASE_DICT_SOFTMAX = layer_dict_softmax
11 OUTPUT_BASE_DIR_PATH = './NNS'
12
13 DOWNLOAD_VHD = True # True= para baixar || False = não baixar
14 DEAD_NEURONS = False # gera neurônios mortos
15
16 INCLUDE_PARAMETERS_ON_FOLDERNAME = True
17 # INCLUDE_MAC_TYPE: Include MAC parameters on MAC.vhd? (good for organization, bad for customization)
18 INCLUDE_MAC_TYPE = True
19 BASE_DICT_HIDDEN['Neuron_arch'][INCLUDE_MAC_type'] = INCLUDE_MAC_TYPE
20 BASE_DICT_SOFTMAX['Neuron_arch'][INCLUDE_MAC_type'] = INCLUDE_MAC_TYPE
21 BARRIERS = False
22
23 BASE_DICT_HIDDEN['Neuron_arch'][BARRIERS] = BARRIERS
24 BASE_DICT_SOFTMAX['Neuron_arch'][BARRIERS] = BARRIERS
25
26 GEN_TOP_LEVEL_HDL(INPUTS_NUMBER=INPUTS_NUMBER,
27                   BIT_WIDTH=BIT_WIDTH,
28                   IO_TYPE_STR=IO_TYPE_STR,
29                   LAYER_NEURONS_NUMBER_LIST=LAYER_NEURONS_NUMBER_LIST,
30                   BASE_DICT_HIDDEN=BASE_DICT_HIDDEN,
31                   BASE_DICT_SOFTMAX=BASE_DICT_SOFTMAX,
32                   OUTPUT_BASE_DIR_PATH=OUTPUT_BASE_DIR_PATH,
33                   INCLUDE_PARAMETERS_ON_FOLDERNAME=INCLUDE_PARAMETERS_ON_FOLDERNAME,
34                   DOWNLOAD_VHD=DOWNLOAD_VHD,
35                   DEAD_NEURONS=DEAD_NEURONS,
36                   DEBUG=False
37
38
39 )
```

Fonte: Elaborado pelo autor (2023).

de comando do Linux, conhecido como Terminal. O usuário antes de executar tal comando, só precisa lembrar de configurar o arquivo `RUN_GEN_TOP_LEVEL_HDL.py` conforme sua arquitetura desejada.

Figura 20 – Executando o *framework* através do Linux Terminal.

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS COMMENTS Python + ⌂ ⌂ ... ⌘ X

Fonte: Elaborado pelo autor (2023).

Na Figura 21 é possível ver parcialmente o arquivo de teste gerado pela arquitetura exemplo. Nas caixas destacadas em vermelho estão sendo utilizados arquivos de texto que devem ser fornecidos pelo usuário na localização destacada pela caixa azul. Estes arquivos (`weights_bin.txt` e `tb_inputs.txt`) são os arquivos com os valores em binário dos pesos da arquitetura e dos valores de entrada para teste da arquitetura. Estes arquivos não são gerados automaticamente pois requer uma conversão de um modelo em software para poder gerar o arquivo de pesos e as entradas de teste, conforme comentado

na posterior Seção 5.2.6 e ficam como sugestão para trabalhos futuros. A caixa verde destaca o arquivo que será gerado após execução da fase de testes, registrando os valores de saída da arquitetura.

Figura 21 – Arquivo de teste (*testbench*) da arquitetura de topo (*top-level*).

```

top_tbvhdl - MLPToVHDL - Visual Studio Code

Edit Selection View Go Run Terminal Help

OPEN EDITORS
RUN_GEN_TOP_LEVEL_HDL.py u top.vhd NNs/NN_3Layers_8bits_4_3_3_2/testbench_files > top_tb.vhd s,u File io
NNs > NN_3Layers_8bits_4_3_3_2/testbench_files > top_tb.vhd > tb (top_tb) > File io

62 DOWNTO 0) := (OTHERS => '0'); --signal
63 VARIABLE val_IO_in: STD_LOGIC_VECTOR(TOTAL_BITS - 1 DOWNTO 0) := (OTHERS => '0'); --signal
VARIABLE val_SPACE:
CHARACTER;
-- espacos da leitura de cada linha de entrada

64 BEGIN
65   ----- ATUALIZACAO DOS PESOS DA NN -----
66   file_open(NN_weights_buf, "./NNs/NN_3Layers_8bits_4_3_3_2/testbench_files/weights bin.txt", read_mode);
67   rst <= '1', '0' AFTER clk_period;
68   WAIT UNTIL rst = '0';
69   WHILE NOT endfile(NN_weights_buf) LOOP --enquanto arquivo nao terminar de ler-
70     file_close(NN_weights_buf); --fecha leitura arquivo dos pesos da NN
71     update weights <= '0';
72   ----- LEITURA ENTRADA E ESCRITA NO ARQUIVO DE SAIDA -----
73   WAIT FOR (sigmid_read_time);
74   -- arquivo de entrada do tb;
75   file_open(input_buf, "./NNs/NN_3Layers_8bits_4_3_3_2/testbench_files/tb_inputs.txt", read_mode);
76   -- arquivo de saida do tb;
77   file_open(output_buf, "./NNs/NN_3Layers_8bits_4_3_3_2/testbench_files/tb_outouts.txt", write_mode);
78   WHILE NOT endfile(input_buf) LOOP
79     readline(input_buf, read_col_from_input_buf); --le linha buffer primeira linha -> escreve na variavel
80     read(read_col_from_input_buf, val_IO_in);
81     IO_in <= signed(val_IO_in);
82     buff_out <= STD_LOGIC_VECTOR(c2_n0_IO_out & c2_n1_IO_out);
83     WAIT FOR (5 * clk_period);
84     write(write_col_to_output_buf, buff_out); --Pega valor da saida e associa ao sinal
85     writeln(output_buf, write_col_to_output_buf); --Escreve valor da saida (do sinal) no arquivo de texto
86   END LOOP;
87   write(write_col_to_output_buf, STRING("END!")); --para confirmar que saiu do loop e estah tudo ok
88   writeln(output_buf, write_col_to_output_buf);
89   file_close(input_buf); --fecha leitura arquivo INPUTS
90   file_close(output_buf); --fecha arquivo OUTPUTS
91   WAIT;
92   --sem ele m funciona; -->Pq?
93   END PROCESS;
94 END tb;
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113

```

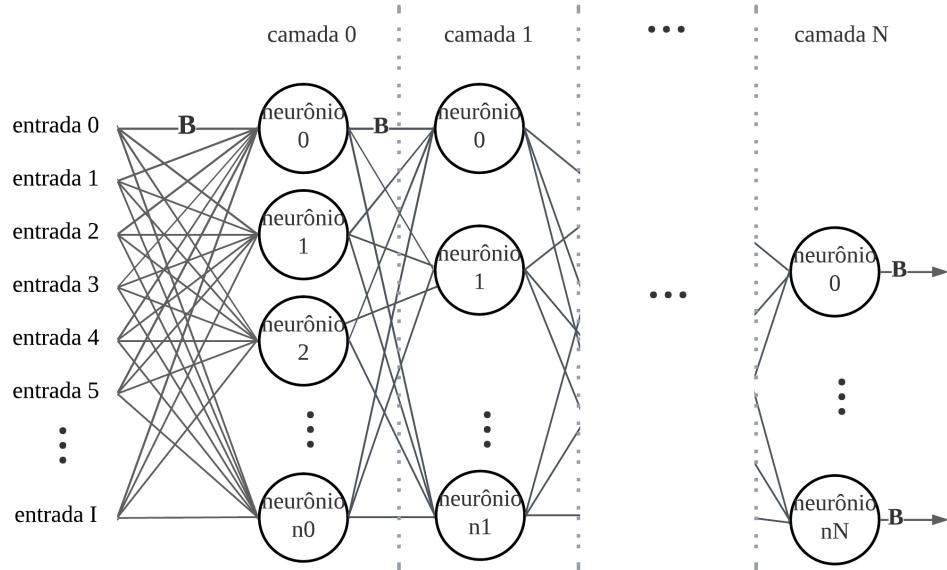
Fonte: Elaborado pelo autor (2023).

## 5.2 Framework

Os resultados alcançados durante este projeto, trazem um framework que permite a geração de código VHDL para redes MLP (Multi-Layer Perceptron), ou seja, se limita à este tipo de arquitetura, não gerando Redes Neurais Convolucionais por exemplo.

Conforme ilustrado na Figura 22, o *framework* permite a geração de MLPs com qualquer número arbitrário de  $B$  bits de representação,  $N$  de camadas,  $I$  entradas e  $n$  neurônios para cada camada (podendo cada camada possuir um número individual de neurônios). Essa flexibilidade oferece uma vantagem significativa aos projetistas, permitindo a criação de redes neurais com arquiteturas personalizadas para atender às necessidades específicas de cada projeto. Além disso, essa capacidade de geração automatizada de código VHDL reduz consideravelmente o tempo necessário para o desenvolvimento de redes neurais extensas.

Figura 22 – Arquitetura genérica de uma Rede Neural Artificial de Perceptrons multicamadas (MLPs) gerada pelo framework.



Fonte: Elaborado pelo autor (2023).

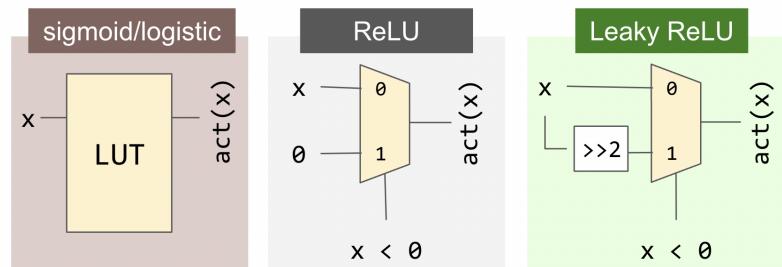
### 5.2.1 Personalização da Largura de Representação de Bits

É importante ressaltar a funcionalidade do framework de personalização da largura de representação de bits  $B$  em ponto-fixo nas unidades aritméticas e nos pesos e viéses da arquitetura. Isso permite que o projetista ajuste a precisão numérica de acordo com os requisitos do projeto, encontrando um equilíbrio entre desempenho e consumo de recursos.

### 5.2.2 Escolha de Funções de Ativação

O framework oferece três opções de funções de ativação: ReLU e Leaky ReLU implementados com multiplexadores e Sigmoid como LUT (Look Up Table), conforme Figura 9. Essa escolha permite que o projetista selecione a função de ativação mais adequada para cada camada da rede neural, considerando as características do problema em questão. Essa flexibilidade adiciona uma camada de personalização às redes neurais geradas, ampliando suas capacidades de representação.

Figura 23 – Funções de ativação disponíveis



Fonte: Elaborado pelo autor (2023)..

### 5.2.3 Inclusão de Barreiras de Registradores

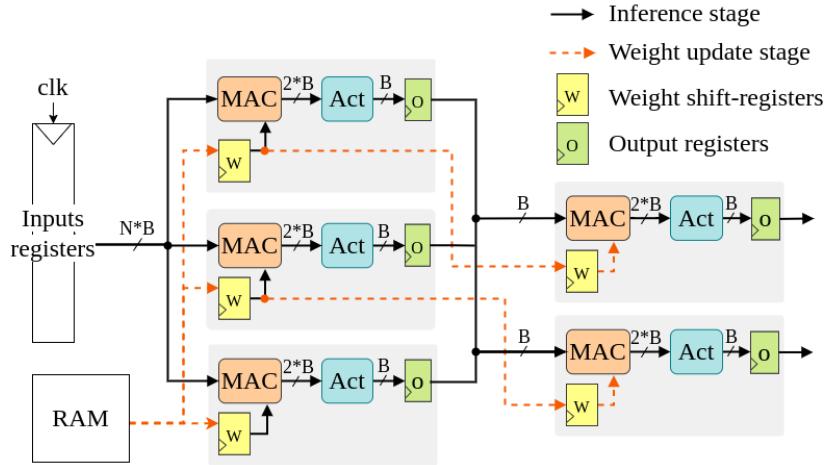
O framework também permite a inclusão de barreiras de registradores nas unidades aritméticas dos neurônios, que podem ser vistos como registradores em azul, na Figura 26. Essa funcionalidade visa otimizar o atraso crítico da rede neural, reduzindo o tempo necessário para propagar os sinais através das unidades aritméticas. No entanto, é importante ressaltar que essa otimização resulta em um aumento no consumo de área e potência da rede neural.

### 5.2.4 Modos de Atualização de Pesos e Inferência

As arquiteturas de redes neurais implementadas pelo framework proposto possuem dois principais modos de operação: (i) atualização de pesos, onde os pesos e vieses armazenados em cada camada são atualizados com novos valores, e (ii) inferência, quando a rede realiza a propagação dos dados de entrada através da rede, produzindo as saídas previstas. A motivação para isso decorre do fato de que os modelos de redes neurais estão constantemente sendo atualizados, sendo importante fornecer mecanismos que permitam aos usuários apesar de treinar um novo modelo mais atualizado, passar estes novos valores de pesos para a arquitetura sempre que necessário (importante para arquiteturas em ASIC).

Um exemplo de arquitetura de hardware geral com uma camada oculta com 3 neurônios ( $n = 3$ ) e uma camada de saída com 2 neurônios ( $O = 2$ ) é mostrado na Figura 24. O retângulo azul representa a unidade de ativação. Para reduzir a congestão de roteamento, a arquitetura desloca os dados pelos registros de peso durante a fase de atualização de pesos. A estrutura dos registradores de deslocamento (*shift-registers*) é ilustrada na Figura 25, onde a seção amarela, identifica o bloco registrador 'w' da Figura 24, ou seja, o bloco 'w' na verdade é um composto de registradores individuais (um para

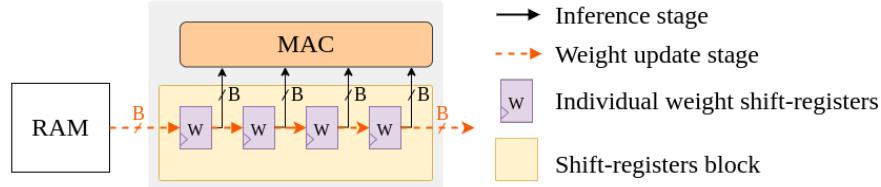
Figura 24 – Exemplo de uma arquitetura em hardware de um MLP gerado pelo framework proposto.



Fonte: Elaborado pelo autor (2023).

cada peso). As conexões laranjas, são utilizadas apenas na etapa de atualização dos pesos, que possuem conexão entre si, dessa forma cada neurônio só irá ter uma entrada de  $B$  bits de largura. Esse deslocamento de pesos entre os registradores, seguirá até a última camada, conforme ilustrado na Figura 24.

Figura 25 – Bloco Shift-registers.



Fonte: Elaborado pelo autor (2023).

Portanto, para  $L$  camadas ocultas, a atualização de pesos irá requerer  $(N + 1) + \sum_{i=2}^{L+1} (n_{i-1} + 1)$  ciclos para ser concluída, onde  $(n_{i-1} + 1)$  é o número de neurônios da camada anterior, que define o número de entradas da camada atual, mais o viés do neurônio. Os  $N + 1$  representam um peso para cada uma das  $N$  entradas, mais o viés do neurônio. Os  $L + 1$  ciclos extras representam a camada de saída.

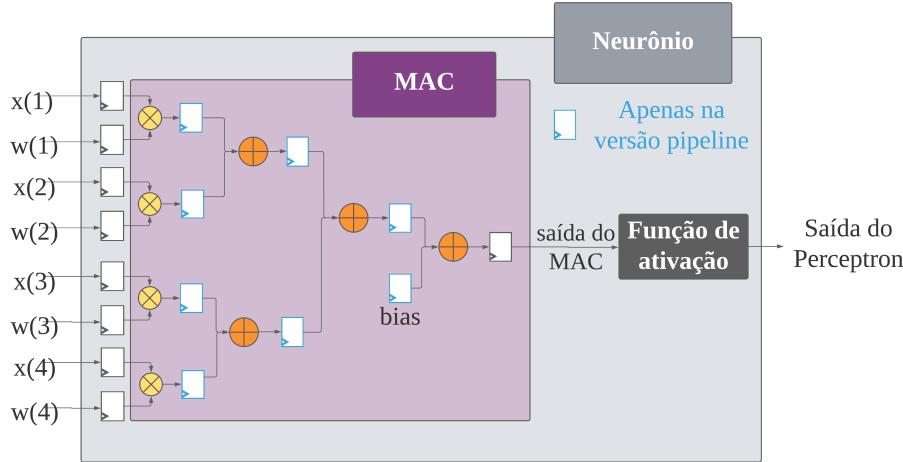
Uma vez que os pesos são carregados, o modo de inferência pode ser realizado. A quantidade de ciclos depende da implementação da multiplicação-adição acumulada (MAC, na sigla em inglês) utilizada (MAC de ciclo único ou pipeline). Para MACs de ciclo único, a etapa de inferência requer  $(L + 1)$  ciclos.

Para MACs de pipeline, o número de nós ( $n$ ) em cada camada também faz parte da equação, e o número de ciclos é obtido por  $(N + 1) + \sum_{i=2}^{L+1}(\log_2(n_{i-1}) + 1)$  pois existem registradores entre cada nível da árvore de somas (para todos os neurônios).

### 5.2.5 Modularização das Arquiteturas

Uma das principais vantagens do framework é a modularização das arquiteturas geradas. Cada camada e unidade aritmética possui um nome de identificação, o que permite ao usuário alterar facilmente as arquiteturas conforme necessário. Essa modularidade simplifica a realização de testes rápidos com a criação de arquiteturas próprias, substituindo os módulos existentes, pois o usuário pode focar em uma determinada unidade ou camada, sem afetar o restante da rede neural. Essa flexibilidade facilita a exploração de diferentes configurações arquiteturais e promove um processo de projeto iterativo muito mais rápido e eficiente.

Figura 26 – Arquitetura genérica de um Neurônio (exemplo para arquitetura com 4 entradas).



Fonte: Elaborado pelo autor (2023)..

Diferentemente dos frameworks existentes abordados no Capítulo 4, que requerem a instalação de várias ferramentas e softwares licenciados, o framework desenvolvido neste trabalho oferece uma interface web intuitiva que pode ser acessada em (SIMON, 2023b). Essa interface permite que o usuário ajuste facilmente as configurações desejadas para a geração do código VHDL, tornando todo o processo rápido e acessível. Com apenas alguns minutos de configuração, o usuário pode obter o código VHDL completo para a rede neural desejada.

### 5.2.6 Resultados e Limitações

Embora o framework tenha atingido seus objetivos principais e demonstrado várias capacidades e facilidades para o projeto de redes neurais artificiais em VHDL, é importante reconhecer suas limitações. O framework gera arquiteturas de MLP, ou seja, Perceptrons multi-camada, não possuindo a possibilidade de gerar camadas convolucionais (muito utilizadas para imagem, áudio e vídeo). Devido à restrição de tempo para a entrega deste trabalho, os testes para verificar a funcionalidade completa das arquiteturas geradas não puderam ser realizados. Portanto, é necessário realizar testes e validações adicionais para garantir o correto funcionamento das redes geradas pelo framework em diferentes cenários. Outro ponto é que o framework não possui um conversor direto dos modelos mais conhecidos como Tensorflow, Pytorch ou ONNX por exemplo, necessitando ao usuário atualmente, obter os pesos e viéses da arquitetura, para salvar em arquivo para leitura e atualização dos valores na arquitetura em hardware gerada.

## 5.3 Resultados de Síntese

Esta seção apresenta dois conjuntos de experimentos: o primeiro revela a relação entre os parâmetros do framework e sua influência nos resultados da síntese de hardware. Ao variar esses parâmetros, pretendemos elucidar os efeitos consequentes em métricas críticas, como área, atraso e consumo de energia. A segunda análise avalia como esses parâmetros, combinados com o uso da aritmética de ponto fixo, afetam o desempenho do modelo.

### 5.3.1 Resultados de Síntese

Todas as arquiteturas foram sintetizadas considerando um fluxo de projeto ASIC para uma biblioteca de células padrão de 65nm da ST Micron, usando a ferramenta Cadence Genus. A frequência alvo foi definida como 200 MHz para todos os projetos devido ao grande tempo consumido em se sintetizar um grande número de circuitos buscando a frequência mínima de cada um. Os resultados de consumo de energia foram estimados com os valores padrão de atividade de comutação da ferramenta. Além disso, o ModelSim foi utilizado para realizar a verificação funcional dos circuitos.

Para mostrar um exemplo prático de nosso framework, várias arquiteturas de redes neurais (geradas automaticamente) são comparadas. As arquiteturas foram geradas com diferentes valores de precisão ( $B$  bits), paralelismo de entrada ( $N$  entradas e pesos), nú-

mero de neurônios por camada ( $n$ ) e número de camadas ( $L$ ). Alguns projetos requeriam vários minutos para serem sintetizados devido ao tamanho do circuito, então mantivemos alguns parâmetros constantes, a saber, tamanho da camada de entrada/saída (16 e 10), implementação do MAC (ciclo único) e função de ativação (ReLU).

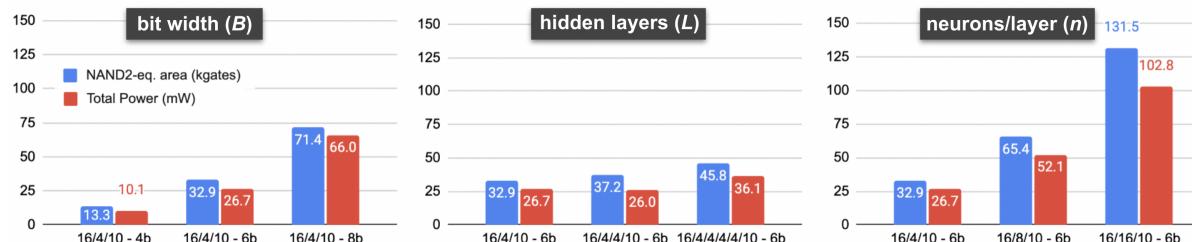
Tabela 2 – Parâmetros de arquitetura testados (Valores padrão em negrito).

Parâmetro	Valores
Unidade MAC	<b>Um ciclo</b>
Função de Ativação	<b>ReLU</b>
Número de entradas (N)	<b>16</b>
Número de saídas	<b>10</b>
Largura de bits de representação (B)	4, <b>6</b> , 8
Nº. de neurônios por camada (n)	4, 8, <b>16</b>
Camadas ocultas (L)	<b>1, 2, 4</b>

Fonte: Elaborado pelo autor (2023).

As configurações testadas (um total de 9) são apresentadas na Tabela 2. Quando mais de um valor foi testado para um determinado parâmetro, os demais foram mantidos com os valores indicados em negrito.

Figura 27 – Resultados de síntese para a frequência alvo de 200 MHz.



Fonte: Elaborado pelo autor (2023).

A Fig. 27 mostra os resultados de síntese para os parâmetros testados em termos de área (equivalente a NAND-2) e dissipação total de energia. Para maior esclarecimento das legendas de cada arquitetura, a arquitetura 16/4/4/10 - 6b possui 16 entradas, possui 2 camadas intermediárias, cada uma com 4 neurônios e 10 neurônios na última camada (função de ativação sigmóide). Os resultados de estudo para cada parâmetro (B, L ou n) indicados na Fig. 27 são discutidos nos parágrafos seguintes. **Largura de bits:** Aumentar a largura de bits em um único incremento resulta em um aumento substancial na área de 14.525 kgates e um aumento significativo na potência de 13.975 mW. Ao dobrar a largura de bits, a área se multiplica consideravelmente por um fator

de 5.368x, e a potência aumenta substancialmente em 6.534x. Isso enfatiza o *trade-off* entre precisão (largura de bits) e utilização de recursos de hardware, onde larguras de bits mais altas demandam mais área e potência.

**Camadas ocultas:** A adição de cada camada oculta contribui com 4.3 kgates para a área total e 5.05 mW para o consumo de potência. Ao dobrar o número de camadas ocultas, a área experimenta um aumento modesto de 1,23x no máximo, enquanto o consumo de potência aumenta em um máximo de 1,38x. Esses resultados sugerem que o impacto das camadas ocultas na área e na potência permanece relativamente limitado, mesmo com incrementos significativos de camadas, mas é importante observar que esses resultados são intrinsecamente influenciados pelo número de neurônios por camada que escolhemos manter.

**Neurônios por camada:** Cada incremento no número de neurônios por camada resulta em um acréscimo de 8.262,5 kgates na área e 6.337 mW na potência. Dobrar o número de neurônios por camada leva a um aumento máximo na área de 2,01x e um aumento na potência de 1,97x. Essas descobertas destacam que o número de neurônios por camada tem um crescimento quase linear com a área e a potência.

### 5.3.2 Efeitos da Aproximação em Redes Neurais de Classificação

Esta seção avalia como a relação entre os parâmetros do framework, combinados com o uso da aritmética de ponto fixo, afetam o desempenho do modelo. Os resultados aqui mostrados, avaliam algumas arquiteturas em comum com as da Figura 27, o que torna possível a correlação entre acurácia e parâmetros de hardware como área, atraso e potência.

Para evitar unidades aritméticas de ponto flutuante, os multiplicadores são implementados usando aritmética de ponto fixo. Como isso introduz um erro em nosso cálculo MAC, foram testados diferentes níveis de precisão. No entanto, existe um preço a ser pago em termos de desempenho do modelo quando multiplicadores de ponto fixo são utilizados. No entanto, é difícil avaliar essa perda porque as bibliotecas de software para treinamento de redes neurais trabalham com precisão de ponto flutuante e não oferecem alternativas de ponto fixo.

Para avaliar o impacto dessas adaptações na acurácia dos modelos, desenvolveu-se um algoritmo em Python, liderado pelo aluno Lucas Soares, com o apoio do professor coorientador Mateus Grellert. Este algoritmo tem a capacidade de transformar um modelo previamente treinado do tipo MLP, ou seja, formado puramente de neurônios,

armazenado em formato '.h5', em um modelo adaptado. Ele realiza essa adaptação ajustando os pesos para a representação em ponto-fixo, com a possibilidade de definir a largura de bits desejada e, ao mesmo tempo, imita a arquitetura original, alterando a aritmética para simular o comportamento da arquitetura gerada em hardware.

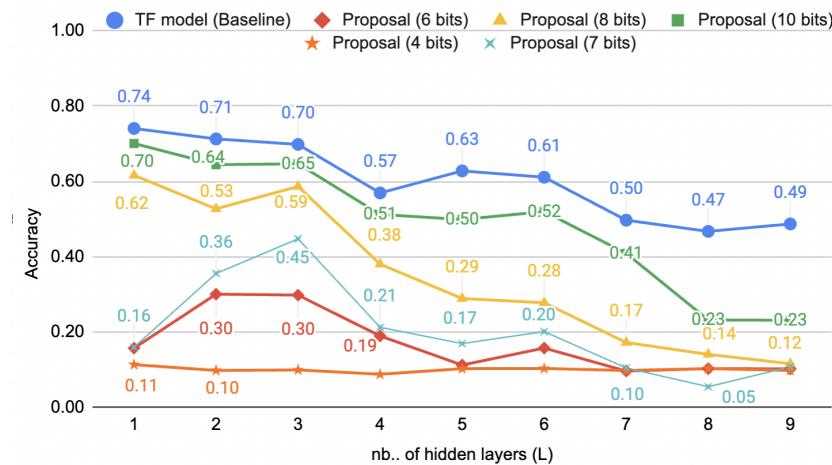
Através deste processo, o algoritmo é capaz de realizar operações de inferência utilizando a aritmética adaptada e registra os resultados de saída do modelo adaptado. Essa abordagem possibilita o cálculo da acurácia do modelo adaptado, fornecendo revelações valiosas sobre como as adaptações impactam o desempenho do modelo, dependendo de cada configuração de arquitetura.

O estudo concentrou-se em um modelo de classificação que utiliza o conjunto de dados MNIST (DENG, 2012) com 10 classes. Os conjuntos de treinamento e teste contêm, respectivamente, 60.000 e 10.000 amostras. Devido às limitações de tamanho do netlist e à demora de síntese para arquiteturas muito maiores, tivemos que reduzir o tamanho da imagem de entrada de 28x28 para 4x4 através de bibliotecas de conversão, o que prejudicou significativamente o desempenho. No entanto, isso não afeta o objetivo principal desta análise, que se concentra na comparação entre modelos de rede precisos e aproximados. Nossa configuração utilizou Python 3.9.5, com 30 épocas, uma taxa de aprendizado de 0,001, otimizador Adam e um tamanho de lote de 32. A comparação foi feita variando o número de camadas, mantendo o número de neurônios em cada camada ( $n$ ) em 4. Também é importante destacar que as redes neurais de ponto flutuante (referência) foram convertidas para ponto fixo após o treinamento, de modo que a aproximação está presente apenas na fase de inferência. Os resultados são apresentados na Figura 28.

Os resultados da Figura 28 mostram que os modelos quantizados apresentam desempenho variado, especialmente quando menos bits são usados para representar os dados. Quando são utilizados 8 ou 10 bits, o desempenho do modelo aproximado está relativamente próximo da melhor versão precisa (que apresentou a melhor precisão de 74%), atingindo picos de desempenho de 70% e 62% para os modelos de 10 bits e 8 bits, com uma única camada oculta. No entanto, a diferença aumenta a medida que mais camadas ocultas são usadas. Isso pode parecer contra-intuitivo a primeira vista, mas pode ser resultado de overfitting ou porque mais épocas de treinamento são necessárias (já que mais camadas se traduzem em mais parâmetros a serem treinados).

Outro resultado interessante evidencia a necessidade de equilíbrio entre tamanho de arquitetura e desempenho do modelo. De forma a se economizar em área e consumo de potência, é natural a busca por modelos mais enxutos (imagens de treinamento menores, menos entradas, menos neurônios e menor representação de bits). Porém quando levado

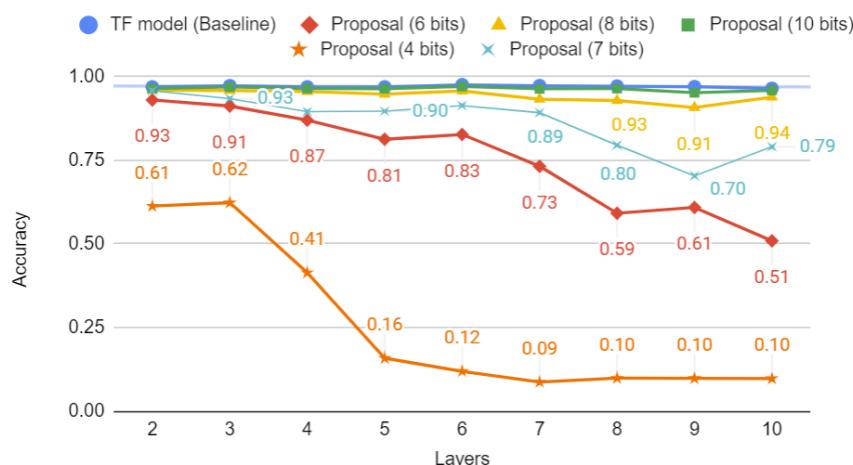
Figura 28 – Desempenho na inferência de classificação considerando a variação do número total de camadas (N) e bits de precisão (B) (imagem 4x4).



Fonte: Elaborado pelo autor (2023).

isso ao extremo, compromete a acurácia do modelo. Agora reduzindo o tamanho da imagem de entrada de 28x28 para 8x8 (ao invés de 4x4), e alterando o número de neurônios por camada oculta em 64 (ao invés de 4) o desempenho dos modelos foi significativamente melhor. Conforme a Figura 29, mesmo um modelo com representação de 6 bits apresenta um desempenho relativo satisfatório comparado ao modelo original (para um número de até 6 camadas ocultas), ou então um modelo de 8 bits para todos os números de camadas ocultas testados.

Figura 29 – Desempenho na inferência de classificação considerando a variação do número total de camadas (N) e bits de precisão (B) (imagem 8x8).



Fonte: Elaborado pelo autor (2023).

Em conclusão, nossas análises de regressão forneceram insights valiosos sobre a influência dos principais parâmetros de arquitetura de redes neurais nas métricas de síntese de hardware e sua correlação com o desempenho da precisão do modelo.

O parâmetro de largura de bits emerge como o fator mais influente, com implicações substanciais em termos de área e consumo de energia. Maior precisão exige significativamente mais recursos, destacando o trade-off entre largura de bits e utilização de hardware.

Quanto à precisão do modelo, os modelos quantizados têm desempenho próximo a versão de maior precisão com 8 ou 10 bits. No entanto, a introdução de mais camadas ocultas amplia a lacuna de desempenho (resultado esperado devido ao erro propagado por mais camadas), sugerindo também desafios potenciais relacionados ao overfitting ou a duração do treinamento. Outro ponto a se ressaltar é que o aumento do número de camadas também irá aumentar o consumo de área e potência, conforme Figura 27.

## 5.4 Melhorias para arquiteturas geradas pelo *Framework*

Os resultados do estudo descrito nas próximas 2 seções se conceve em um número de 4 a 20 vetores de entrada para ambas arquiteturas, pois o algoritmo Spiral com interface web possui a limitação de no máximo 20 constantes de multiplicação. Porém este número está longe de um número de vetores em uma aplicação real. Arquiteturas com aplicações reais costumam passar em muito deste número, como por exemplo a arquitetura MLP-Mixer, que possui blocos de camadas MLPs completamente conectadas 196 vetores de entrada, (TOLSTIKHIN et al., 2021).

É importante salientar que devido aos vetores de entrada de cada neurônio, que representam suas entradas e pesos respectivos possuírem uma largura  $B$  de representação, **os vetores resultantes da etapa de multiplicação (mostrados na seção 5.4.1) possuem uma largura  $2 * B$ , ou seja, para uma MLP com 8 bits de representação para suas entradas e saídas, possuirá valores destes vetores intermediários com 16 bits de largura.** Estes 16 bits entram na função de ativação, onde são truncados novamente para 8 bits, conforme Figura 24.

Tabela 3 – Multiplicadores vs Somas e deslocamentos: Área total.

Número de vetores	Área total	Área total	Variação
	arquitetura A	arquitetura B	
4	7800	8661	11.04%
8	13172	11802	-10.40%
12	23720	20754	-12.50%
16	33060	27153	-17.87%
20	40117	34865	-13.09%

Fonte: Elaborado pelo autor (2023).

#### 5.4.1 Substituição dos Multiplicadores por Blocos de Somas e Deslocamentos

Na seção 4.2.1 deste estudo, foi detalhada a comparação entre duas arquiteturas diferentes, a **arquitetura A, que usa multiplicadores padrão, e a arquitetura B, que substitui esses multiplicadores por blocos de somas e deslocamentos usando a ferramenta Spiral (Figura 14)**. Foram analisadas três métricas principais: área total, atraso crítico e potência total e avaliada a variação entre as duas arquiteturas.

Com relação à área total, observamos que a arquitetura B, com os blocos de somas e deslocamentos, teve um acréscimo de 11,04% em relação à arquitetura A para 4 vetores. No entanto, a medida que o número de vetores aumenta, a arquitetura B apresentou uma redução significativa na área necessária. Para 20 vetores, a arquitetura B apresentou uma redução de 13,09% em relação à arquitetura A. Esses resultados indicam que a substituição dos multiplicadores por somas e deslocamentos pode resultar em uma diminuição geral da área ocupada pela arquitetura, principalmente para maior quantidade de vetores, ou seja, para MLPs com maior número de neurônios por camada.

Em relação ao atraso crítico, olhamos para o tempo necessário para um sinal percorrer o caminho mais longo na arquitetura. Descobrimos que a Arquitetura B experimentou aumentos significativos no atraso crítico em comparação com a arquitetura A. Para 4 vetores, observamos um aumento de 42,40% em um atraso crítico na arquitetura B. O ganho proporcional se manteve próximo deste valor mesmo quando o número de vetores aumentou, ou seja, a arquitetura B mostrou atrasos maiores em comparação com a arquitetura A em todos os casos. Esses resultados sugerem que a substituição de multiplicadores por somas impactará o desempenho da arquitetura em termos de atraso crítico, o que sugere uma troca entre atraso crítico e área total.

Os resultados gerados demonstram que a arquitetura B, com os blocos de adição e

Tabela 4 – Multiplicadores vs Somas e deslocamentos: Atraso crítico.

Número de vetores	Atraso crítico	Atraso crítico	Variação
	arquitetura A	arquitetura B	
4	3064	4363	42.40%
8	3064	4232	38.12%
12	3528	4814	36.45%
16	3756	5246	39.67%
20	3714	5220	40.55%

Fonte: Elaborado pelo autor (2023).

Tabela 5 – Multiplicadores vs Somas e deslocamentos: Potência total consumida.

Número de vetores	Potência total	Potência total	Variação
	arquitetura A	arquitetura B	
4	411663.426	596617.265	44.93%
8	708515.042	731622.504	3.26%
12	1321226.936	1537422.441	16.36%
16	1883624.351	2105948.442	11.80%
20	2281310.749	2622236.593	14.94%

Fonte: Elaborado pelo autor (2023).

deslocamento, apresentou aumentos na potência total em comparação com a arquitetura A. Para 4 vetores, a arquitetura B apresentou um aumento de 44,93% na potência total. No entanto, a medida que o número de vetores aumentou, a diferença na potência total entre as duas arquiteturas diminui, reforçando o maior benefício do uso desta técnica para arquiteturas MLPs de maior número de neurônios. Em todos os casos, a arquitetura B apresentou maior potência total comparada à arquitetura A. Esses resultados indicam que a substituição dos multiplicadores por somas e deslocamentos resulta em um aumento geral no consumo de energia da arquitetura.

Em resumo, os resultados constatam que a substituição de multiplicadores por blocos de somas e deslocamentos, utilizando a ferramenta Spiral, traz impactos positivos na ocupação de área total porém ao custo de impactos negativos no atraso crítico e consumo de potência. No geral, utilizar técnicas de somas e deslocamentos se mostra válida para arquiteturas em que a área ocupada seja um fator crítico. Esses resultados destacam a importância de considerar cuidadosamente as compensações entre essas análises ao escolher substituir multiplicadores por somas e mudanças em uma arquitetura de rede neural, dependendo do objetivo do projeto.

### 5.4.2 Troca dos Somadores por um Bloco de Compressão de Vetores

A pesquisa teve como objetivo comparar dois tipos de árvore de soma em hardware considerando diferentes valores de largura de bits  $B$  e número de vetores  $N$ . Os dois tipos de árvore de soma foram denominados **tipo 1** e **tipo 2**. O **tipo 1** (exemplificado na Figura 13b), realizou a soma direta dos vetores utilizando os operadores de soma '+' no código de descrição de hardware (*HDL*), enquanto o **tipo 2** (exemplificado na Figura 15) utilizou a técnica de compressão dos vetores através de *Carry-Save Adders (CSAs)*, conforme seção 4.2.2.

Tabela 6 – Comparação de Área total entre arquitetura padrão (tipo 1) e compressores (tipo 2).

N	12 bits		16 bits		20 bits	
	Tipo 1	Tipo 2	Tipo 1	Tipo 2	Tipo 1	Tipo 2
4	4375	4343	5853	5816	7333	7296
8	10101	10052	13501	13451	16900	16850
12	15829	15581	21148	20900	26467	26219
16	21588	21536	28826	28774	36065	36013
20	27321	27333	36480	36371	45638	45410

Fonte: Elaborado pelo autor (2023).

Para diferentes valores de  $B$  (6, 8 e 10), conforme descrito na seção 5.4 trazem resultados intermediários com o dobro deste número de bits (12, 16 e 20).

Em comparação com o **tipo 1**, o uso da arquitetura do **tipo 2** apresentou uma pequena redução na área total de até 1,59%, 1,19% e 0,95% para para  $B = 6, 8$  e 10 bits. Isso indica que a técnica de compressão dos vetores utilizando *Carry-Save Adders (CSAs)* traz uma economia muito sutil na área total do circuito, principalmente para arquiteturas com menor número de bits, o que não justifica o uso da mesma.

A arquitetura do tipo 2 apresentou aumento de até 27,26% no atraso crítico para  $B = 6, N = 16$  e até 20,19% para  $B = 10, N = 16$ , o que sugere que o aumento da largura de representação  $B$  torna a utilização da arquitetura do tipo 2 mais atrativa, pois a diferença tende a diminuir a medida que  $B$  aumenta. O aumento no número de vetores de entrada  $N$  parece trazer resultados não conclusivos, uma vez que os mesmos variam bastante, possuindo o tipo 2 valores de atraso crítico menores apenas para  $N = 4$ .

Os resultados de potência total mostraram os melhores resultados para o tipo 2 com o número de vetores  $N = 12$ , com valores 30,44% menores do que a arquitetura do tipo 1. Porém estes resultados promissores não se repetem para outros números de vetores

Tabela 7 – Comparação de Atraso máximo entre arquitetura padrão (tipo 1) e compressores (tipo 2).

N	12 bits		16 bits		20 bits	
	Tipo 1	Tipo 2	Tipo 1	Tipo 2	Tipo 1	Tipo 2
4	3446	3748	4434	4393	5421	5381
8	3967	4542	4955	5529	5942	6517
12	4437	4610	5425	5598	6412	6586
16	4637	5249	5625	6237	6640	7225
20	5096	5261	6084	6248	7071	7269

Fonte: Elaborado pelo autor (2023).

Tabela 8 – Comparação de Potência total entre arquitetura padrão (tipo 1) e compressores (tipo 2).

N	12 bits		16 bits		20 bits	
	Tipo 1	Tipo 2	Tipo 1	Tipo 2	Tipo 1	Tipo 2
4	401702.457	401728.741	553990.823	545848.085	661804.691	687322.391
8	1297960.433	1397620.704	1793935.518	1934067.116	2243942.746	2361437.367
12	2425587.731	1943495.959	3389768.361	2688477.451	4283689.599	3283983.452
16	3860425.249	4041805.554	5317028.93	5515710.797	6704144.651	6959152.4
20	5172204.588	5621586.712	7012814.39	7458084.027	8742118.715	9448382.084

Fonte: Elaborado pelo autor (2023).

de entrada, o que traz uma indicação de que existem números de vetores ótimos que tornam a arquitetura tipo 2 mais atrativa. Os dois melhores números de  $N$  vetores se revelam sendo para  $N = 4$  e  $N = 12$ , dentro do alcance estudado neste trabalho.

A pesquisa comparativa entre os dois tipos de árvore de soma em hardware, **tipo 1** e **tipo 2**, revelou que a utilização de Carry-Save Adders para compressão dos vetores não apresenta vantagens significativas em termos de área total do circuito ou atraso crítico. Independentemente da largura de bits utilizada, o **tipo 2** resultou em uma área total sutilmente menor em comparação com o **tipo 1**, ao passo de resultados consideravelmente piores no atraso crítico e consumo de potência. Em teoria compressores de soma deveriam ser muito mais eficientes em termos de atraso, porém a ferramenta de síntese utilizada demonstra já fazer uma série de otimizações, não compensando o uso de compressores de soma ao invés dos operadores '+'. Uma investigação mais profunda do por quê deste resultado revelou que a ferramenta acaba por não implementar os compressores da forma esperada, não sendo considerado de fato um compressor. O mesmo ocorreu também na tentativa de se adicionar um *Carry-Select Adder* ao final da soma dos compressores, onde a ferramenta acaba por implementar um *Ripple-Carry Adder*.

Uma sugestão de trabalhos futuros seria a investigação de modos para a ferramenta de síntese implementar as arquiteturas desejadas (compressores e *Carry-Select Adders*).

## 6 Conclusão

No âmbito deste Trabalho de Conclusão de Curso (TCC) em Engenharia Eletrônica, desenvolveu-se um framework em Python voltado para a geração automatizada de código VHDL sintetizável destinado a aceleradores de Perceptrons de Múltiplas Camadas, do inglês "Multi-Layer Perceptrons"(MLPs). O trabalho abrangeu três principais áreas de pesquisa que contribuíram significativamente para o avanço na otimização dessas arquiteturas de hardware.

No que diz respeito à criação do framework, observou-se que a largura de bits dos dados tem um impacto substancial na área e no consumo de energia. Aumentar a largura de bits resulta em um aumento considerável tanto na área quanto na potência, enfatizando um claro trade-off entre precisão (largura de bits) e a utilização de recursos de hardware. A adição de camadas ocultas teve um impacto negativo em termos de área e potência. No entanto, é importante considerar que esses resultados são influenciados pelo número de neurônios por camada escolhido, sendo o correto equacionamento deste impacto, o número total de multiplicadores e somadores na arquitetura.

No estudo sobre os efeitos da aproximação em redes neurais de classificação, ficou evidente mais uma vez que a largura de bits é o fator mais influente, com implicações substanciais em termos de área e consumo de energia. Modelos quantizados apresentaram desempenho próximo ao da versão de maior precisão com 8 ou 10 bits. No entanto, a introdução de mais camadas ocultas ampliou a lacuna de desempenho, sugerindo desafios relacionados ao erro propagado, overfitting e/ou à duração do treinamento.

O último estudo focou-se em possíveis otimizações na arquitetura do acelerador de MLPs. Duas abordagens foram comparadas: a substituição de multiplicadores por blocos de somas e deslocamentos, bem como a troca de somadores por um bloco de compressão de vetores. Constatou-se que a primeira abordagem, embora otimize a área ocupada, implica em impactos negativos no atraso crítico e no consumo de potência. Portanto, essa técnica se mostra adequada para situações em que a área é o fator crítico. Quanto a comparação entre os dois tipos de árvore de soma em hardware, a utilização de Carry-Save Adders para compressão dos vetores não apresentou vantagens significativas em termos de área total do circuito, resultando em resultados consideravelmente piores em atraso crítico e consumo de potência.

Em resumo, este trabalho proporcionou um estudo significativo na otimização de aceleradores de MLPs em hardware por meio da criação de um framework de geração de

código VHDL, do estudo dos efeitos da aproximação na acurácia do modelo e da análise de possíveis otimizações na arquitetura do acelerador. Essas descobertas contribuem para a compreensão mais profunda dos trade-offs envolvidos na adaptação de modelos de redes neurais para implementações em hardware e fornecem intuições valiosas para futuros desenvolvimentos na área.

Para pesquisas futuras, sugerem-se as seguintes direções:

Desenvolvimento de uma ferramenta de conversão automática capaz de transformar modelos em formatos comuns, como ONNX, PyTorch e TensorFlow, em descrições em VHDL. Isso ampliaria significativamente a utilidade do framework desenvolvido, facilitando a adaptação de uma variedade de modelos de redes neurais para implementações em hardware.

Exploração da substituição dos MACs (Multiply-Accumulate Units) combinacionais por blocos sequenciais. Essa mudança poderia impactar o desempenho, consumo de área, potência e atraso das arquiteturas de hardware. Investigar os trade-offs envolvidos nessa transição seria um campo promissor de pesquisa.

Considerando o amplo uso das camadas convolucionais em Redes Neurais Convolucionais (CNNs) para aplicações de visão e áudio, uma área de pesquisa interessante seria a geração automatizada de camadas convolucionais. Isso contribuiria para a expansão do framework para suportar uma variedade mais ampla de arquiteturas de redes neurais, abrangendo domínios além de MLPs.

# Referências Bibliográficas

ACADEMY, D. S. *Capítulo 58 - Introdução aos Autoencoders - Deep Learning Book*. 2022. <<https://www.deeplearningbook.com.br/introducao-aos-autoencoders/>>. (Accessed on 12/09/2022). 28

AMORIM, P. W. F. *Autoencoder*. 2022. Disponível em: <<https://lamfo-unb.github.io/2020/11/21/Autoencoder/>>. 27

AWATI, R. *What is an intellectual property core (IP core)? – techtarget definition*. TechTarget, 2022. Disponível em: <<https://www.techtarget.com/whatis/definition/IP-core-intellectual-property-core>>. 21

BRE, F.; GIMENEZ, J. M.; FACHINOTTI, V. D. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, v. 158, p. 1429–1441, 2018. ISSN 0378-7788. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S0378778817325501>>. 24

COLUCCI, A. et al. A fast design space exploration framework for the deep learning accelerators: Work-in-progress. In: *2020 International Conference on Hardware/Software Codesign and System Synthesis (CODESISSS)*. IEEE, 2020. Disponível em: <<https://doi.org/10.1109/codesisss51650.2020.9244038>>. 21

DENG, B. L. et al. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, Institute of Electrical and Electronics Engineers (IEEE), v. 108, n. 4, p. 485–532, abr. 2020. Disponível em: <<https://doi.org/10.1109/jproc.2020.2976475>>. 18

DENG, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, IEEE, v. 29, n. 6, p. 141–142, 2012. 50

DESROUSSEAUX, R.; BERNARD, G.; MARIAGE, J.-J. Profiling money laundering with neural networks: a case study on environmental crime detection. In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. [S.l.: s.n.], 2021. p. 364–369. 14

DSA, E. *Capítulo 58 - Introdução AOS autoencoders*. 2019. Disponível em: <<https://www.deeplearningbook.com.br/introducao-aos-autoencoders/>>. 27, 28

FAHIM, F. et al. *hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices*. arXiv, 2021. Disponível em: <<https://arxiv.org/abs/2103.05579>>. 30

- GUO, K. et al. Software-hardware codesign for efficient neural network acceleration. *IEEE Micro*, Institute of Electrical and Electronics Engineers (IEEE), v. 37, n. 2, p. 18–25, mar. 2017. Disponível em: <<https://doi.org/10.1109/mm.2017.39>>. 18
- GUO, K. et al. [DL] a survey of FPGA-based neural network inference accelerators. *ACM Transactions on Reconfigurable Technology and Systems*, Association for Computing Machinery (ACM), v. 12, n. 1, p. 1–26, mar. 2019. Disponível em: <<https://doi.org/10.1145/3289185>>. 14
- GYAN, V. *Carry save adder verilog code: Verilog implementation of carry save adder*. Admin, 2022. Disponível em: <<http://vlsigyan.com/carry-save-adder-verilog-code/>>. 38
- HABIBIAN, A. et al. Video compression with rate-distortion autoencoders. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. [S.l.: s.n.], 2019. p. 7032–7041. 27
- HAN, S. et al. EIE: Efficient inference engine on compressed deep neural network. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016. Disponível em: <<https://doi.org/10.1109/isca.2016.30>>. 8, 15, 17
- HAO, C.; CHEN, D. Deep neural network model and FPGA accelerator co-design: Opportunities and challenges. In: *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. IEEE, 2018. Disponível em: <<https://doi.org/10.1109/icsict.2018.8564956>>. 15
- HINKELMANN, P. D. K. *Wayback Machine*. 2022. <[https://web.archive.org/web/20181006235506/http://didattica.cs.unicam.it/lib/exe/fetch.php?media=didattica:magistrale:kebi:ay\\_1718:ke-11\\_neural\\_networks.pdf](https://web.archive.org/web/20181006235506/http://didattica.cs.unicam.it/lib/exe/fetch.php?media=didattica:magistrale:kebi:ay_1718:ke-11_neural_networks.pdf)>. (Accessed on 12/09/2022). 26
- IBERDROLA.COM. *Conheça os principais benefícios do 'Machine Learning'*. 2022. Disponível em: <<https://www.iberdrola.com/inovacao/o-que-e-machine-learning>>. 23
- ISELE, D. et al. Navigating occluded intersections with autonomous vehicles using deep reinforcement learning. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018. Disponível em: <<https://doi.org/10.1109/icra.2018.8461233>>. 14
- JADON, S. *Introduction to Different Activation Functions for Deep Learning / by Shruti Jadon / Medium*. 2018. <<https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092>>. (Accessed on 12/09/2022). 26
- JORDAN, J. *Introduction to autoencoders*. Jeremy Jordan, 2018. Disponível em: <<https://www.jeremyjordan.me/autoencoders>>. 27
- JOUPPI, N. *Quantifying the performance of the TPU, our first machine learning chip / Google Cloud Blog*. 2017. <<https://cloud.google.com/blog/products/gcp/quantifying-the-performance-of-the-tpu-our-first-machine-learning-chip>>. (Accessed on 12/09/2022). 15, 16

K, M. et al. 1d convolution approach to human activity recognition using sensor data and comparison with machine learning algorithms. *International Journal of Cognitive Computing in Engineering*, Elsevier BV, v. 2, p. 130–143, jun. 2021. Disponível em: <<https://doi.org/10.1016/j.ijcce.2021.09.001>>. 14

KURZWEIL, R. *The singularity is near: When humans transcend biology / worldcat.org*. Penguin Books, New York, 2006, 2006. Disponível em: <<https://www.worldcat.org/title/The-singularity-is-near--when-humans-transcend-biology/oclc/71826177>>. 14

LAI SUBUTAI AHMAD, D. D. C.; MAVER, C. *AI is harming our planet: addressing AI's staggering energy cost*. 2022. <<https://www.numenta.com/blog/2022/05/24/ai-is-harming-our-planet/>>. (Accessed on 12/09/2022). 16

LEBEDEV, M.; BELECKY, P. A survey of open-source tools for FPGA-based inference of artificial neural networks. In: *2021 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2021. Disponível em: <<https://doi.org/10.1109/ivmem53963.2021.00015>>. 29

LIU, X. et al. Compressed ultrahigh-speed photography enabled by a snapshot-to-video autoencoder. In: *2022 Photonics North (PN)*. IEEE, 2022. Disponível em: <<https://doi.org/10.1109/pn56061.2022.9908322>>. 19

LU, W. et al. Secure robust jpeg steganography based on autoencoder with adaptive bch encoding. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 31, n. 7, p. 2909–2922, 2021. 14

MISRA, J.; SAHA, I. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, Elsevier BV, v. 74, n. 1-3, p. 239–255, dez. 2010. Disponível em: <<https://doi.org/10.1016/j.neucom.2010.03.021>>. 17

MORGAN, L. *AI carbon footprint: Helping and hurting the environment / TechTarget*. 2021. <<https://www.techtarget.com/searchenterpriseai/feature/AI-carbon-footprint-Helping-and-hurting-the-environment>>. (Accessed on 12/09/2022). 15

Multilayer perceptron. *Multilayer perceptron — Wikipedia, The Free Encyclopedia*. 2022. [Online; accessed 28-December-2022]. Disponível em: <[https://en.wikipedia.org/wiki/Multilayer\\_perceptron](https://en.wikipedia.org/wiki/Multilayer_perceptron)>. 33

NASCIMENTO, R. S. F. do et al. Detecção de anomalias em poços de petróleo surgentes com stacked autoencoders. In: *Proceedings do XV Simpósio Brasileiro de Automação Inteligente*. SBA Sociedade Brasileira de Automática, 2021. Disponível em: <<https://doi.org/10.20906/sbai.v1i1.2856>>. 27

NEWQUIST, H. P. *The Brain Makers: [genius, ego, and greed in the quest ... - worldcat.org*. Sams Publ., Indianapolis, Ind., 1994, 1994. Disponível em: <<https://www.worldcat.org/title/313139906>>. 14

NURVITADHI, E. et al. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In: *2016 26th International Conference*

- on Field Programmable Logic and Applications (FPL)*. IEEE, 2016. Disponível em: <<https://doi.org/10.1109/fpl.2016.7577314>>. 15, 18
- OUYANG, Z. et al. Deep CNN-based real-time traffic light detector for self-driving vehicles. *IEEE Transactions on Mobile Computing*, Institute of Electrical and Electronics Engineers (IEEE), v. 19, n. 2, p. 300–313, fev. 2020. Disponível em: <<https://doi.org/10.1109/tmc.2019.2892451>>. 14
- PAN, Y. Heading toward artificial intelligence 2.0. *Engineering*, v. 2, n. 4, p. 409–413, 2016. ISSN 2095-8099. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S2095809917300772>>. 14
- PESSOA, J. et al. End-to-end learning of video compression using spatio-temporal autoencoders. In: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. [S.l.: s.n.], 2020. p. 1–6. 14
- PRECEDENCERESSEARCH. *Artificial Intelligence Market*. 2022. Disponível em: <<https://www.precedenceresearch.com/artificial-intelligence-market>>. 14
- PÜSCHEL, Y. V. . M. *SPIRAL Multiplier Block Generator*. 2007. Disponível em: <<https://spiral.ece.cmu.edu/mcm/gen.html>>. 35
- SATO, K. *An in-depth look at Google's first tensor processing unit (TPU) / google cloud blog*. Google, 2017. Disponível em: <<https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu>>. 17
- SCHMIDHUBER, J. Deep learning in neural networks: An overview. *Neural Networks*, v. 61, p. 85–117, 2015. ISSN 0893-6080. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S0893608014002135>>. 14
- SEMIENGINEERING.COM. *RTL (Register Transfer Level)*. semiengineering.com, 2021. Disponível em: <[https://semiengineering.com/knowledge\\_centers/eda-design/definitions/register-transfer-level/](https://semiengineering.com/knowledge_centers/eda-design/definitions/register-transfer-level/)>. 29
- SIMON, L. A. S. *MLP to VHDL Github repository*. 2023. Disponível em: <<https://github.com/LuisSpader/MLPtoVHDL>>. 20, 34, 39
- SIMON, L. A. S. *MLP to VHDL Web App Framework*. Streamlit, 2023. Disponível em: <<https://mlptovhdl.streamlit.app/>>. 20, 33, 39, 46
- STRUBELL, E.; GANESH, A.; MCCALLUM, A. *Energy and Policy Considerations for Deep Learning in NLP*. arXiv, 2019. Disponível em: <<https://arxiv.org/abs/1906.02243>>. 15
- SZE, V. et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. arXiv, 2017. Disponível em: <<https://arxiv.org/abs/1703.09039>>. 30

TECNOBLOG, L. K. *O que É dma? [Acesso Direto à memória]*. tecnoblog.net, 2022. Disponível em: <<https://tecnoblog.net/responde/o-que-e-dma-acesso-direto-a-memoria/>>. 31

THOMPSON, N.; SPANUTH, S. The decline of computers as a general purpose technology: Why deep learning and the end of moore's law are fragmenting computing. *SSRN Electronic Journal*, Elsevier BV, 2018. Disponível em: <<https://doi.org/10.2139/ssrn.3287769>>. 16, 17, 18

TOLSTIKHIN, I. O. et al. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems*, v. 34, p. 24261–24272, 2021. 52

WANG, J. et al. Neural RRT: Learning-based optimal path planning. *IEEE Transactions on Automation Science and Engineering*, Institute of Electrical and Electronics Engineers (IEEE), v. 17, n. 4, p. 1748–1758, out. 2020. Disponível em: <<https://doi.org/10.1109/tase.2020.2976560>>. 14

WIKIPEDIA. *High-level synthesis - Wikipedia*. 2022. <[https://en.wikipedia.org/wiki/High-level\\_synthesis](https://en.wikipedia.org/wiki/High-level_synthesis)>. (Accessed on 12/09/2022). 30

YAMAZAKI. *NNgen/nngen: NNgen: A fully-customizable hardware synthesis compiler for deep neural network*. Shinya Takamaeda-Yamazaki and Contributors, 2022. Disponível em: <<https://github.com/NNgen/nngen>>. 31

YURTSEVER, E. et al. A survey of autonomous driving: icommon practices and emerging technologies/i. *IEEE Access*, Institute of Electrical and Electronics Engineers (IEEE), v. 8, p. 58443–58469, 2020. Disponível em: <<https://doi.org/10.1109/access.2020.2983149>>. 15

ZHANG, G. et al. Cnn-based sample adaptive offset optimization in hevc for streaming video. In: *2019 IEEE International Conference on Real-time Computing and Robotics (RCAR)*. [S.l.: s.n.], 2019. p. 263–266. 14

ZHOU, J.; LV, T.; YI, X. End-to-end distributed video coding. In: *2022 Data Compression Conference (DCC)*. [S.l.: s.n.], 2022. p. 496–496. 14

# Apêndices

# APÊNDICE A – Códigos VHDL de uma MLP exemplo

Os códigos abaixo mostram a arquitetura modular gerada pelo *framework*, evidenciando ao usuário a possibilidade de testar implementações próprias para cada bloco ao passo do aproveitamento da geração automática do restante da arquitetura. Os exemplos de código mostrados representam a estrutura comentada na Seção 5.1, ou seja, uma MLP com 4 entradas, 2 camadas ocultas com 3 neurônios cada, utilizando a função de ativação ReLU e uma camada de saída com 2 neurônios utilizando a função de ativação Sigmóide como LUT. A largura de representação utilizada é de 8 bits.

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE std.textio.ALL;
5 USE ieee.std_logic_textio.ALL; -- para tratamento de arquivos e texto-> file_open...
6 USE work.parameters.ALL;
7 ENTITY top_tb IS
8
9 GENERIC (
10     BITS: NATURAL := 8;
11     NUM_INPUTS: NATURAL := 4;
12     TOTAL_BITS: NATURAL := 32
13 );
14
15 END top_tb;
16 ARCHITECTURE tb OF top_tb IS
17     CONSTANT clk_hz : INTEGER
18             := 100e6;
19     CONSTANT clk_period : TIME
20             := 1 sec / clk_hz;
21     SIGNAL buff_out : STD_LOGIC_VECTOR(((4) * BITS) - 1 DOWNTO 0) := (OTHERS => '0');
22     CONSTANT sigmoid_read_time : TIME
23             := 16 * clk_period;
24     -- SIGNAL clk, rst, update_weights : STD_LOGIC
25             := '0';
26     -- SIGNAL IO_in : signed(TOTAL_BITS * NUM_INPUTS - 1 DOWNTO 0);
27     -- SIGNAL buff_in : STD_LOGIC_VECTOR(TOTAL_BITS * NUM_INPUTS - 1 DOWNTO 0);
28     SIGNAL clk, rst, update_weights: STD_LOGIC;
29     SIGNAL IO_in: signed(TOTAL_BITS - 1 DOWNTO 0);
30     SIGNAL c0_n0_W_in, c0_n1_W_in, c0_n2_W_in: signed(BITS - 1 DOWNTO 0);
31     SIGNAL c2_n0_IO_out, c2_n1_IO_out: signed(BITS - 1 DOWNTO 0);

```

```

28 BEGIN
29   -- port map do componente 'top.vhd'
30   UUT : ENTITY work.top PORT MAP(
31     clk => clk,
32     rst => rst,
33     update_weights => update_weights,
34     IO_in => IO_in,
35     c0_n0_W_in => c0_n0_W_in,
36     c0_n1_W_in => c0_n1_W_in,
37     c0_n2_W_in => c0_n2_W_in,
38     c2_n0_IO_out => c2_n0_IO_out,
39     c2_n1_IO_out => c2_n1_IO_out
40   );
41   -- processo gerador de clock
42   clk_gen : PROCESS
43     --constant period: time := 20 ns;
44 BEGIN
45   clk <= '0';
46   WAIT FOR clk_period/2;
47   clk <= '1';
48   WAIT FOR clk_period/2;
49 END PROCESS;
50   -- processo para leitura das entradas e escrita das saídas
51   file_io : PROCESS
52     --SIGNALS AND VARIABLES
53     VARIABLE read_col_from_input_buf : line
54     ; -- buffers de entrada e saída
55     FILE input_buf : text
56     ; --text is keyword ->??
57     VARIABLE read_col_from_sigmoid_buf : line
58     ;
59     FILE NN_weights_buf : text
60     ; --text is keyword -->??
61     VARIABLE write_col_to_output_buf : line
62     ;
63     FILE output_buf : text
64     ; --text is keyword -->??
65     VARIABLE val_address : STD_LOGIC_VECTOR(bits - 1 DOWNTO 0) := (OTHERS => '0');
66     VARIABLE val_c0_n0_W_in, val_c0_n1_W_in, val_c0_n2_W_in: STD_LOGIC_VECTOR(BITS
67     - 1 DOWNTO 0) := (OTHERS => '0'); --signal
68     -- VARIABLE val_n0_IO_in, val_n1_IO_in, val_n2_IO_in, val_n3_IO_in,
69     val_n4_IO_in : STD_LOGIC_VECTOR(TOTAL_BITS - 1 DOWNTO 0) := (OTHERS => '0'); -- signal
70     VARIABLE val_IO_in: STD_LOGIC_VECTOR(TOTAL_BITS - 1 DOWNTO 0) := (OTHERS =>
71     '0'); --signal
72     VARIABLE val_SPACE : CHARACTER;
73     de cada linha de entrada
74 BEGIN
75   ----- ATUALIZACAO DOS PESOS DA NN -----
76   file_open(NN_weights_buf, "./NNs/NN_3Layers_8bits_4_3_3_2/testbench_files/
77   weights_bin.txt", read_mode);
78   rst <= '1', '0' AFTER clk_period;

```



```

112     END PROCESS;
113 END tb;
```

Código A.1 – Arquivo de teste da arquitetura topo de uma MLP (*top-level MLP testbench*)

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE work.parameters.ALL;
5
6 ENTITY top IS
7 GENERIC (
8   BITS : NATURAL := BITS;
9   NUM_INPUTS : NATURAL := 4;
10  TOTAL_BITS : NATURAL := 32
11 );
12 PORT (
13   clk, rst, update_weights: IN STD_LOGIC;
14   IO_in: IN signed(TOTAL_BITS - 1 DOWNTO 0);
15   c0_n0_W_in, c0_n1_W_in, c0_n2_W_in: IN signed(BITS - 1 DOWNTO 0);
16   -----
17   c2_n0_IO_out, c2_n1_IO_out: OUT signed(BITS -1 DOWNTO 0)
18 );
19 end ENTITY;
20
21 ARCHITECTURE arch OF top IS
22 -- SIGNALS
23   SIGNAL c0_n0_W_out, c0_n1_W_out, c0_n2_W_out, c1_n0_W_out, c1_n1_W_out: signed(BITS -
24   1 DOWNTO 0);
25   SIGNAL c1_IO_in: signed((BITS*3) - 1 DOWNTO 0);
26   SIGNAL c2_IO_in: signed((BITS*3) - 1 DOWNTO 0);
27   SIGNAL c0_n0_IO_out, c0_n1_IO_out, c0_n2_IO_out: SIGNED(BITS -1 DOWNTO 0);
28   SIGNAL c1_n0_IO_out, c1_n1_IO_out, c1_n2_IO_out: SIGNED(BITS -1 DOWNTO 0);
29   SIGNAL reg_IO_in: signed(TOTAL_BITS - 1 DOWNTO 0);
30   SIGNAL en_registers: STD_LOGIC;
31 BEGIN
32
33   en_registers <= update_weights AND clk;
34   c1_IO_in <= c0_n0_IO_out & c0_n1_IO_out & c0_n2_IO_out;
35   c2_IO_in <= c1_n0_IO_out & c1_n1_IO_out & c1_n2_IO_out;
36
37   PROCESS (clk, rst)
38   BEGIN
39     IF rst = '1' THEN
40       reg_IO_in <= (OTHERS => '0');
41     ELSIF clk'event AND clk = '1' THEN
42       reg_IO_in <= IO_in;
43     END IF;
44   END PROCESS;
45
46 camada0_inst_0: ENTITY work.camada0_ReLU_3neuron_8bits_4n_signed
47   PORT MAP (
```

```

48      ----- Entradas -----
49      -- ['IN'][]['STD_LOGIC']
50      clk=> clk,
51      rst=> rst,
52      update_weights=> en_registers,
53      -- ['IN'][]['manual']
54      IO_in=> reg_IO_in,
55      c0_n0_W_in=> c0_n0_W_in,
56      c0_n1_W_in=> c0_n1_W_in,
57      c0_n2_W_in=> c0_n2_W_in,
58      ----- Saidas -----
59      -- ['OUT'][]['SIGNED']
60      c0_n0_IO_out=> c0_n0_IO_out,
61      c0_n1_IO_out=> c0_n1_IO_out,
62      c0_n2_IO_out=> c0_n2_IO_out,
63      -- ['OUT'][]['manual']
64      c0_n0_W_out=> c0_n0_W_out,
65      c0_n1_W_out=> c0_n1_W_out,
66      c0_n2_W_out=> c0_n2_W_out
67  );
68
69 camada1_inst_1: ENTITY work.camada1_ReLU_3neuron_8bits_3n_signed
70 PORT MAP (
71      ----- Entradas -----
72      -- ['IN'][]['STD_LOGIC']
73      clk=> clk,
74      rst=> rst,
75      update_weights=> en_registers,
76      -- ['IN'][]['manual']
77      IO_in=> c1_IO_in,
78      c1_n0_W_in=> c0_n0_W_out,
79      c1_n1_W_in=> c0_n1_W_out,
80      c1_n2_W_in=> c0_n2_W_out,
81      ----- Saidas -----
82      -- ['OUT'][]['SIGNED']
83      c1_n0_IO_out=> c1_n0_IO_out,
84      c1_n1_IO_out=> c1_n1_IO_out,
85      c1_n2_IO_out=> c1_n2_IO_out,
86      -- ['OUT'][]['manual']
87      c1_n0_W_out=> c1_n0_W_out,
88      c1_n1_W_out=> c1_n1_W_out
89  );
90
91 camada2_inst_2: ENTITY work.camada2_Sigmoid_2neuron_8bits_3n_signed
92 PORT MAP (
93      ----- Entradas -----
94      -- ['IN'][]['STD_LOGIC']
95      clk=> clk,
96      rst=> rst,
97      update_weights=> en_registers,
98      -- ['IN'][]['manual']
99      IO_in=> c2_IO_in,
100     c2_n0_W_in=> c1_n0_W_out,
101     c2_n1_W_in=> c1_n1_W_out,

```

```

102      ----- Saidas -----
103      -- ['OUT'][‘SIGNED’]
104      c2_n0_I0_out=> c2_n0_I0_out,
105      c2_n1_I0_out=> c2_n1_I0_out
106  );
107
108 END ARCHITECTURE;

```

Código A.2 – Arquitetura topo de uma MLP

Código A.3 – Primeira camada oculta com 8 bits

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.std_logic_unsigned.ALL;
4 USE ieee.numeric_std.ALL;
5 USE ieee.math_real.ALL;
6 USE work.parameters.ALL;
7
8 ENTITY neuron_comb_layer1_3n_8bit_signed_mult0_v0_add0_v0_out IS
9   GENERIC (
10     BITS : NATURAL := BITS;
11     NUM_INPUTS : NATURAL := 3;
12     TOTAL_BITS : NATURAL := 24
13   );
14   PORT (
15     clk, rst, update_weights: IN STD_LOGIC;
16     IO_in : IN signed(TOTAL_BITS - 1 DOWNTO 0);
17     W_in : IN signed(BITS - 1 DOWNTO 0);
18     -----
19     IO_out: OUT signed(BITS -1 DOWNTO 0);
20     W_out : OUT signed(BITS - 1 DOWNTO 0)
21   );
22 end ENTITY;
23
24 ARCHITECTURE behavior of neuron_comb_layer1_3n_8bit_signed_mult0_v0_add0_v0_out is
25   -----
26   COMPONENT MAC_comb_3n_8bit_signed_mult0_v0_add0_v0 IS
27     GENERIC (
28       BITS : NATURAL := BITS;
29       NUM_INPUTS : NATURAL := 3;
30       TOTAL_BITS : NATURAL := 24
31     );
32     PORT (
33       clk, rst: IN STD_LOGIC;
34       IO_in : IN signed(TOTAL_BITS - 1 DOWNTO 0);
35       W_in : IN signed((BITS * (NUM_INPUTS + 1)) - 1 DOWNTO 0);
36       -----
37       IO_out: OUT signed((MAC_OUT_BITS_rescale*BITS) -1 DOWNTO 0)
38     );
39   end COMPONENT;
40
41   COMPONENT shift_reg_3n IS

```

```

42      GENERIC (
43          BITS : NATURAL := BITS;
44          NUM_INPUTS : NATURAL := NUM_INPUTS
45      );
46      PORT (
47          clk, rst : IN STD_LOGIC;
48          W_in : IN signed(BITS - 1 DOWNTO 0);
49          -- Win : IN signed(BITS - 1 DOWNTO 0);
50          W_out : OUT signed((BITS * (NUM_INPUTS + 1)) - 1 DOWNTO 0)
51      );
52  END COMPONENT;
53
54
55 COMPONENT activation_fx IS
56     GENERIC (
57         BITS_FX_IN : NATURAL := BITS_FX_IN;
58         BITS_FX_OUT : NATURAL := BITS_FX_OUT;
59         ACTIVATION_TYPE : NATURAL := 0; -- 0: ReLU, 1: Leaky ReLU, 2: Sigmoid, 3:
60         linear
61         Leaky_attenuation : NATURAL := Leaky_attenuation;
62         Leaky_ReLU_ones : signed := Leaky_ReLU_ones
63     );
64     PORT (
65         clk, rst : IN STD_LOGIC;
66         fx_in : IN signed(BITS_FX_IN - 1 DOWNTO 0);
67         fx_out : OUT signed (BITS_FX_OUT - 1 DOWNTO 0)
68     );
69 END COMPONENT;
70 ----- SIGNALS -----
71 SIGNAL out_reg_MAC : signed ((2*BITS)-1 DOWNTO 0); --reg da saida do MAC
72 SIGNAL s_Wout : signed((BITS * (NUM_INPUTS + 1)) - 1 DOWNTO 0);
73 BEGIN
74
75     -- MAC ja registra a saida
76     U_MAC : MAC_comb_3n_8bit_signed_mult0_v0_add0_v0 PORT MAP(
77         clk, rst,
78         IO_in,
79         s_Wout,
80         out_reg_MAC );
81
82     inst_shift_reg : shift_reg_3n PORT MAP(update_weights, rst, W_in , s_Wout );
83     W_out <= s_Wout((BITS * (NUM_INPUTS + 1)) - 1 DOWNTO (BITS * (NUM_INPUTS + 0)))
84 ;
85
86     fx_activation_inst : activation_fx PORT MAP(
87         clk, rst,
88         out_reg_MAC ,
89         IO_out
90     );
91 END behavior;

```

Código A.4 – Neurônio de 8 bits da segunda camada oculta (layer 1) com 3 entradas e saída para deslocamento dos registradores de peso e viés

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE ieee.math_real.ALL;
5 USE work.parameters.ALL;
6
7 ENTITY MAC_comb_3n_8bit_signed_mult0_v0_add0_v0 IS
8   GENERIC (
9     BITS : NATURAL := BITS;
10    NUM_INPUTS : NATURAL := 3;
11    TOTAL_BITS : NATURAL := 24
12  );
13  PORT (
14    clk, rst: IN STD_LOGIC;
15    IO_in : IN signed(TOTAL_BITS - 1 DOWNTO 0);
16    W_in : IN signed((BITS * (NUM_INPUTS + 1)) - 1 DOWNTO 0);
17    -----
18    IO_out: OUT signed((MAC_OUT_BITS_rescale*BITS) -1 DOWNTO 0)
19  );
20 end ENTITY;
21
22 ARCHITECTURE arch OF MAC_comb_3n_8bit_signed_mult0_v0_add0_v0 IS
23
24  -----
25  SIGNAL sum_all : signed((2*BITS) - 1 DOWNTO 0);
26  SIGNAL s_Xi : signed((BITS * NUM_INPUTS) - 1 DOWNTO 0);
27  SIGNAL s_Win : signed((BITS * (NUM_INPUTS + 1)) - 1 DOWNTO 0);
28  SIGNAL s_mult : signed(((2 * BITS) * (NUM_INPUTS)) - 1 DOWNTO 0);
29
30 COMPONENT mult0_v0 IS
31   GENERIC (
32     BITS : NATURAL := BITS
33   );
34   PORT (
35     X : IN signed((BITS) - 1 DOWNTO 0);
36     W : IN signed((BITS) - 1 DOWNTO 0);
37     Y : OUT signed((2 * BITS) - 1 DOWNTO 0)
38   );
39 END COMPONENT;
40
41 BEGIN
42   s_Xi <= IO_in;
43   s_Win <= W_in;
44
45   sum_all <= (s_mult(((2 * BITS) * (0 + 1)) - 1 DOWNTO ((2 * BITS) * (0))) +
46   s_mult(((2 * BITS) * (1 + 1)) - 1 DOWNTO ((2 * BITS) * (1))) +
47   s_mult(((2 * BITS) * (2 + 1)) - 1 DOWNTO ((2 * BITS) * (2))) +
48   s_Win((BITS * (3 + 1)) - 1 DOWNTO (BITS * (3))));
49
50 loop_Mult_port_map : FOR i IN 0 TO (NUM_INPUTS - 1) GENERATE
51   mult0_v0_inst_loop : mult0_v0
52   PORT MAP(
53     X => s_Xi((BITS * (i + 1)) - 1 DOWNTO (BITS * (i))),
54     W => s_Win((BITS * (i + 1)) - 1 DOWNTO (BITS * (i))),
```

```

55      Y => s_mult(((2 * BITS) * (i + 1)) - 1 DOWNTO ((2 * BITS) * (i)))
56  );
57 END GENERATE;
58 IO_out <= signed(sum_all);
59 END arch;
```

Código A.5 – MAC de 3 entradas 8 bits e utilizando multiplicadores e somadores do tipo 0 versão 0

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE work.parameters.ALL;
5
6
7 ENTITY mult0_v0 IS
8   GENERIC (
9     BITS : NATURAL := 8
10  );
11  PORT (
12    X : IN signed((1* BITS) - 1 DOWNTO 0);
13    W : IN signed((1* BITS) - 1 DOWNTO 0);
14    Y : OUT signed((1*2 * BITS) - 1 DOWNTO 0)
15  );
16 END ENTITY;
17
18 ARCHITECTURE rtl OF mult0_v0 IS
19
20 BEGIN
21   Y <= X * W;
22 END ARCHITECTURE;
```

Código A.6 – Multiplicador utilizando operator '\*'

```

1
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.ALL;
4 USE ieee.numeric_std.ALL;
5 USE work.parameters.ALL;
6
7 ENTITY activation_fx IS
8   GENERIC (
9     BITS_FX_IN       : NATURAL := BITS_FX_IN;
10    BITS_FX_OUT      : NATURAL := BITS_FX_OUT;
11    ACTIVATION_TYPE  : NATURAL := 2; -- 0: ReLU, 1: Leaky ReLU, 2: Sigmoid, 3:
12    linear
13    Leaky_attenuation : NATURAL := Leaky_attenuation;
14    Leaky_ReLU_ones   : signed  := Leaky_ReLU_ones
15  );
16  PORT (
17    clk, rst : IN STD_LOGIC;
18    fx_in    : IN signed(BITS_FX_IN - 1 DOWNTO 0);
19    fx_out   : OUT signed (BITS_FX_OUT - 1 DOWNTO 0)
```

```

19    );
20 END ENTITY;
21 ARCHITECTURE arch OF activation_fx IS
22     ----- COMPONENTS -----
23
24 COMPONENT ReLU IS
25     PORT (
26         fx_in : IN signed(BITS_FX_IN - 1 DOWNTO 0);
27         fx_out : OUT signed (BITS_FX_OUT - 1 DOWNTO 0)
28     );
29 END COMPONENT;
30
31 COMPONENT Leaky_ReLU IS
32     PORT (
33         fx_in : IN signed(BITS_FX_IN - 1 DOWNTO 0);
34         fx_out : OUT signed (BITS_FX_OUT - 1 DOWNTO 0)
35     );
36 END COMPONENT;
37
38 -- ROM
39 COMPONENT ROM_fx_8bitaddr_8width IS
40     PORT (
41         address : IN STD_LOGIC_VECTOR (BITS - 1 DOWNTO 0);
42         -----
43         data_out : OUT STD_LOGIC_VECTOR (BITS - 1 DOWNTO 0)
44     );
45     -- input: address (8 bits)
46     -- output: data_out (8 bits)
47 END COMPONENT;
48 ----- SIGNALS -----
49 SIGNAL s_fx_out      : signed(BITS_FX_OUT - 1 DOWNTO 0);
50 SIGNAL s_fx_out_std : STD_LOGIC_VECTOR(BITS_FX_OUT - 1 DOWNTO 0);
51 SIGNAL fx_in_ROM     : signed(BITS - 1 DOWNTO 0);
52
53 BEGIN
54
55     ReLU_inst : IF ACTIVATION_TYPE = 0 GENERATE
56         ReLU_inst : ReLU PORT MAP(fx_in, s_fx_out);
57     END GENERATE;
58
59     Leaky_ReLU_inst : IF ACTIVATION_TYPE = 1 GENERATE
60         Leaky_ReLU_inst : Leaky_ReLU PORT MAP(fx_in, s_fx_out);
61     END GENERATE;
62
63     Sigmoid_ROM_inst : IF ACTIVATION_TYPE = 2 GENERATE -- it's even
64         -- BEGIN
65         -- fx_in_ROM <= to_signed(to_integer(fx_in), fx_in_ROM'length); -- Numeric_std
66         fx_in_ROM <= fx_in((2 * BITS) - 1 DOWNTO BITS);
67         U_ROM : ROM_fx_8bitaddr_8width PORT MAP(
68             STD_LOGIC_VECTOR(fx_in_ROM),
69             s_fx_out_std
70         ); -- input: address (8), output: data_out (8)
71         -- END PROCESS fx_activation_inst;
72         s_fx_out <= signed(s_fx_out_std);

```

```

73  END GENERATE;
74
75  linear_inst : IF ACTIVATION_TYPE = 3 GENERATE
76      fx_in_ROM <= fx_in((2 * BITS) - 1 DOWNTO BITS);
77      s_fx_out  <= fx_in_ROM;
78  END GENERATE;
79
80  PROCESS (clk, rst)
81  BEGIN
82      IF (rst = '1') THEN
83          fx_out <= (OTHERS => '0');
84      ELSE
85          IF clk'event AND clk = '1' THEN
86              fx_out <= s_fx_out;
87          END IF;
88      END IF;
89  END PROCESS;
90
91 END ARCHITECTURE;

```

Código A.7 – Bloco instanciador dos módulos das funções de ativação

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE work.parameters.ALL;
5
6 ENTITY ReLU IS
7     PORT (
8         fx_in : IN signed(BITS_FX_IN - 1 DOWNTO 0);
9         fx_out : OUT signed (BITS_FX_OUT - 1 DOWNTO 0)
10    );
11 END ENTITY;
12
13 ARCHITECTURE rtl OF ReLU IS
14
15 BEGIN
16
17     PROCESS (fx_in)
18     BEGIN
19         IF fx_in > 0 THEN -- X > 0
20             -- s_fx_out <= fx_in;
21             IF fx_in > signed_max_2xbit THEN
22                 fx_out <= to_signed(to_integer(signed_max), fx_out'length);
23             ELSE
24                 fx_out <= to_signed(to_integer(fx_in), fx_out'length); -- Numeric_std
25             END IF;
26
27         ELSE -- X < 0
28             fx_out <= (OTHERS => '0');
29         END IF;
30     END PROCESS;
31
32

```

33 END ARCHITECTURE;

### Código A.8 – Função de ativação ReLU

```

1
2 --https://stackoverflow.com/questions/17579716/implementing-rom-in-xilinx-vhdl
3 LIBRARY ieee ;
4 USE ieee.std_logic_1164.all ;
5 USE ieee.numeric_std.all;
6 -----
7 ENTITY ROM_fx_8bitaddr_8width IS
8 generic(addr_height : integer := 256; -- store 256 elements
9     addr_bits : integer := 8; -- required bits to store 256 elements
10    data_width : integer := 8 -- each element has 8-bits
11 );
12 PORT (
13     address : IN STD_LOGIC_VECTOR(addr_bits - 1 DOWNTO 0);
14     data_out : OUT STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0)
15 );
16 END ENTITY;
17 -----
18 architecture arch of ROM_fx_8bitaddr_8width is
19 type memory is array ( 0 to addr_height-1 ) of std_logic_vector(data_width-1 downto 0
20 );
21 constant myrom : memory := (
22 --"int_f_x",--(address) =integer_MAC|| f(x)          = int_f_x
23 "10000000",-- (00000000) = 0.0      || 128.0          = 128.0
24 "10000010",-- (00000001) = 1.0      || 130.79955347088494 = 130.0
25 "10000101",-- (00000010) = 2.0      || 133.59642981673   = 133.0
26 "10001000",-- (00000011) = 3.0      || 136.38796214291455 = 136.0
27 "10001011",-- (00000100) = 4.0      || 139.1715039326922   = 139.0
28 "10001101",-- (00000101) = 5.0      || 141.9444390298694   = 141.0
29 "10010000",-- (00000110) = 6.0      || 144.70419137717758 = 144.0
30 "10010011",-- (00000111) = 7.0      || 147.44823443416874 = 147.0
31 "10010110",-- (00001000) = 8.0      || 150.1741002028365   = 150.0
32 "10011000",-- (00001001) = 9.0      || 152.8793877944536   = 152.0
33 "10011011",-- (00001010) = 10.0     || 155.56177147722028 = 155.0
34 "10011110",-- (00001011) = 11.0     || 158.21900815110592 = 158.0
35 "10100000",-- (00001100) = 12.0     || 160.84894420361377 = 160.0
36 "10100011",-- (00001101) = 13.0     || 163.4495217079567 = 163.0
37 "10100110",-- (00001110) = 14.0     || 166.01878393316375 = 166.0
38 "10101000",-- (00010000) = 15.0     || 168.55488014379085 = 168.0
39 "10101011",-- (00010001) = 16.0     || 171.05606967505054 = 171.0
40 "10101111",-- (00010010) = 17.0     || 173.52072527716751 = 173.0
41 "10110010",-- (00010011) = 18.0     || 175.94733573048427 = 175.0
42 "10110100",-- (00010100) = 19.0     || 178.3345077401747 = 178.0
43 "10110110",-- (00010101) = 20.0     || 180.68096712627488 = 180.0
44 "10111001",-- (00010110) = 21.0     || 182.98555933102264 = 182.0
45 "10111011",-- (00010111) = 22.0     || 185.24724927115267 = 185.0
46 "10111101",-- (00011000) = 23.0     || 187.46512056776362 = 187.0
47 "10111111",-- (00011001) = 24.0     || 189.6383741906314 = 189.0
48 "11000001",-- (00011010) = 25.0     || 191.76632655737023 = 191.0
49 "11000011",-- (00011011) = 26.0     || 193.84840713063423 = 193.0
50 "11000111",-- (00011100) = 27.0     || 195.88415555862744 = 195.0

```

```

50 "11000101",-- (00011100) = 28.0    || 197.87321840556507 = 197.0
51 "11000111",-- (00011101) = 29.0    || 199.81534551944442 = 199.0
52 "11001001",-- (00011110) = 30.0    || 201.71038608458105 = 201.0
53 "11001011",-- (00011111) = 31.0    || 203.55828440590156 = 203.0
54 "11001101",-- (00100000) = 32.0    || 205.3590754709969 = 205.0
55 "11001111",-- (00100001) = 33.0    || 207.1128803345236 = 207.0
56 "11010000",-- (00100010) = 34.0    || 208.819901367714 = 208.0
57 "11010010",-- (00100011) = 35.0    || 210.48041741361928 = 210.0
58 "11010100",-- (00100100) = 36.0    || 212.09477888629738 = 212.0
59 "11010101",-- (00100101) = 37.0    || 213.66340284954757 = 213.0
60 "11010111",-- (00100110) = 38.0    || 215.1867681080277 = 215.0
61 "11011000",-- (00100111) = 39.0    || 216.66541034073072 = 216.0
62 "11011010",-- (00101000) = 40.0    || 218.0999173038875 = 218.0
63 "11011011",-- (00101001) = 41.0    || 219.49092412744895 = 219.0
64 "11011100",-- (00101010) = 42.0    || 220.8391087264176 = 220.0
65 "11011110",-- (00101011) = 43.0    || 222.1451873454842 = 222.0
66 "11011111",-- (00101100) = 44.0    || 223.40991025270483 = 223.0
67 "11100000",-- (00101101) = 45.0    || 224.6340575953512 = 224.0
68 "11100001",-- (00101110) = 46.0    || 225.8184354286031 = 225.0
69 "11100010",-- (00101111) = 47.0    || 226.96387192543793 = 226.0
70 "11100100",-- (00110000) = 48.0    || 228.0712137739231 = 228.0
71 "11100101",-- (00110001) = 49.0    || 229.14132276613313 = 229.0
72 "11100110",-- (00110010) = 50.0    || 230.17507258110555 = 230.0
73 "11100111",-- (00110011) = 51.0    || 231.17334576261152 = 231.0
74 "11101000",-- (00110100) = 52.0    || 232.13703089105314 = 232.0
75 "11101001",-- (00110101) = 53.0    || 233.0670199474983 = 233.0
76 "11101001",-- (00110110) = 54.0    || 233.9642058667285 = 233.0
77 "11101010",-- (00110111) = 55.0    || 234.82948027519006 = 234.0
78 "11101011",-- (00111000) = 56.0    || 235.66373140890153 = 235.0
79 "11101100",-- (00111001) = 57.0    || 236.4678422056685 = 236.0
80 "11101101",-- (00111010) = 58.0    || 237.24268856538296 = 237.0
81 "11101101",-- (00111011) = 59.0    || 237.9891377717278 = 237.0
82 "11101110",-- (00111100) = 60.0    || 238.70804706825496 = 238.0
83 "11101111",-- (00111101) = 61.0    || 239.40026238155605 = 239.0
84 "11110000",-- (00111110) = 62.0    || 240.06661718407545 = 240.0
85 "11110000",-- (00111111) = 63.0    || 240.7079314890315 = 240.0
86 "11110001",-- (01000000) = 64.0    || 241.3250109698896 = 241.0
87 "11110001",-- (01000001) = 65.0    || 241.91864619687576 = 241.0
88 "11110010",-- (01000010) = 66.0    || 242.48961198310965 = 242.0
89 "11110011",-- (01000011) = 67.0    || 243.03866683307706 = 243.0
90 "11110011",-- (01000100) = 68.0    || 243.56655248633442 = 243.0
91 "11110100",-- (01000101) = 69.0    || 244.07399354954848 = 244.0
92 "11110100",-- (01000110) = 70.0    || 244.56169721020154 = 244.0
93 "11110101",-- (01000111) = 71.0    || 245.03035302554926 = 245.0
94 "11110101",-- (01001000) = 72.0    || 245.48063278068096 = 245.0
95 "11110101",-- (01001001) = 73.0    || 245.9131904098127 = 245.0
96 "11110110",-- (01001010) = 74.0    || 246.3286619752277 = 246.0
97 "11110110",-- (01001011) = 75.0    || 246.72766569856648 = 246.0
98 "11110111",-- (01001100) = 76.0    || 247.11080203946045 = 247.0
99 "11110111",-- (01001101) = 77.0    || 247.47865381678946 = 247.0
100 "11110111",-- (01001110) = 78.0    || 247.8317863681286 = 247.0
101 "11111000",-- (01001111) = 79.0    || 248.17074774322967 = 248.0
102 "11111000",-- (01010000) = 80.0    || 248.49606892765277 = 248.0
103 "11111000",-- (01010001) = 81.0    || 248.80826409293064 = 248.0

```

```

104 "11111001",-- (01010010) = 82.0    || 249.10783086989977 = 249.0
105 "11111001",-- (01010011) = 83.0    || 249.3952506420807 = 249.0
106 "11111001",-- (01010100) = 84.0    || 249.67098885622383 = 249.0
107 "11111001",-- (01010101) = 85.0    || 249.93549534736016 = 249.0
108 "11111010",-- (01010110) = 86.0    || 250.18920467591147 = 250.0
109 "11111010",-- (01010111) = 87.0    || 250.4325364746162 = 250.0
110 "11111010",-- (01011000) = 88.0    || 250.6658958032178 = 250.0
111 "11111010",-- (01011001) = 89.0    || 250.88967350904375 = 250.0
112 "11111011",-- (01011010) = 90.0    || 251.10424659177247 = 251.0
113 "11111011",-- (01011011) = 91.0    || 251.30997857084472 = 251.0
114 "11111011",-- (01011100) = 92.0    || 251.50721985412497 = 251.0
115 "11111011",-- (01011101) = 93.0    || 251.6963081065567 = 251.0
116 "11111011",-- (01011110) = 94.0    || 251.87756861768625 = 251.0
117 "11111100",-- (01011111) = 95.0    || 252.05131466704864 = 252.0
118 "11111100",-- (01100000) = 96.0    || 252.2178478865221 = 252.0
119 "11111100",-- (01100001) = 97.0    || 252.3774586188595 = 252.0
120 "11111100",-- (01100010) = 98.0    || 252.53042627170205 = 252.0
121 "11111100",-- (01100011) = 99.0    || 252.6770196664673 = 252.0
122 "11111100",-- (01100100) = 100.0   || 252.81749738158442 = 252.0
123 "11111100",-- (01100101) = 101.0   || 252.95210808962463 = 252.0
124 "11111101",-- (01100110) = 102.0   || 253.08109088794197 = 253.0
125 "11111101",-- (01100111) = 103.0   || 253.20467562250147 = 253.0
126 "11111101",-- (01101000) = 104.0   || 253.32308320462894 = 253.0
127 "11111101",-- (01101001) = 105.0   || 253.43652592046766 = 253.0
128 "11111101",-- (01101010) = 106.0   || 253.54520773297327 = 253.0
129 "11111101",-- (01101011) = 107.0   || 253.64932457632193 = 253.0
130 "11111101",-- (01101100) = 108.0   || 253.74906464264322 = 253.0
131 "11111101",-- (01101101) = 109.0   || 253.84460866102432 = 253.0
132 "11111101",-- (01101110) = 110.0   || 253.93613016876247 = 253.0
133 "11111110",-- (01101111) = 111.0   || 254.0237957748705 = 254.0
134 "11111110",-- (01110000) = 112.0   || 254.10776541586384 = 254.0
135 "11111110",-- (01110001) = 113.0   || 254.18819260387994 = 254.0
136 "11111110",-- (01110010) = 114.0   || 254.26522466719953 = 254.0
137 "11111110",-- (01110011) = 115.0   || 254.3390029832559 = 254.0
138 "11111110",-- (01110100) = 116.0   || 254.40966320423354 = 254.0
139 "11111110",-- (01110101) = 117.0   || 254.47733547536947 = 254.0
140 "11111110",-- (01110110) = 118.0   || 254.54214464608148 = 254.0
141 "11111110",-- (01110111) = 119.0   || 254.60421047405794 = 254.0
142 "11111110",-- (01111000) = 120.0   || 254.66364782244906 = 254.0
143 "11111110",-- (01111001) = 121.0   || 254.72056685030938 = 254.0
144 "11111110",-- (01111010) = 122.0   || 254.775073196443 = 254.0
145 "11111110",-- (01111011) = 123.0   || 254.82726815680954 = 254.0
146 "11111110",-- (01111100) = 124.0   || 254.87724885565115 = 254.0
147 "11111110",-- (01111101) = 125.0   || 254.92510841050319 = 254.0
148 "11111110",-- (01111110) = 126.0   || 254.97093609125218 = 254.0
149 "11111111",-- (01111111) = 127.0   || 255.01481747340657 = 255.0
150 "00000000",-- (10000000) = -128.0  || 0.9431654142556132 = 0.0
151 "00000000",-- (10000001) = -127.0  || 0.9851825265934124 = 0.0
152 "00000001",-- (10000010) = -126.0  || 1.0290639087477775 = 1.0
153 "00000001",-- (10000011) = -125.0  || 1.0748915894967983 = 1.0
154 "00000001",-- (10000100) = -124.0  || 1.1227511443488492 = 1.0
155 "00000001",-- (10000101) = -123.0  || 1.1727318431904823 = 1.0
156 "00000001",-- (10000110) = -122.0  || 1.2249268035570005 = 1.0
157 "00000001",-- (10000111) = -121.0  || 1.2794331496906086 = 1.0

```

```

158 "00000001",-- (10001000) = -120.0 || 1.3363521775509497 = 1.0
159 "00000001",-- (10001001) = -119.0 || 1.3957895259420845 = 1.0
160 "00000001",-- (10001010) = -118.0 || 1.457855353918504 = 1.0
161 "00000001",-- (10001011) = -117.0 || 1.5226645246305492 = 1.0
162 "00000001",-- (10001100) = -116.0 || 1.590336795766455 = 1.0
163 "00000001",-- (10001101) = -115.0 || 1.6609970167441164 = 1.0
164 "00000001",-- (10001110) = -114.0 || 1.73477533280049 = 1.0
165 "00000001",-- (10001111) = -113.0 || 1.811807396120073 = 1.0
166 "00000001",-- (10010000) = -112.0 || 1.892234584136186 = 1.0
167 "00000001",-- (10010001) = -111.0 || 1.9762042251295215 = 1.0
168 "00000010",-- (10010010) = -110.0 || 2.06386983123754 = 2.0
169 "00000010",-- (10010011) = -109.0 || 2.155391338975693 = 2.0
170 "00000010",-- (10010100) = -108.0 || 2.2509353573567803 = 2.0
171 "00000010",-- (10010101) = -107.0 || 2.3506754236780787 = 2.0
172 "00000010",-- (10010110) = -106.0 || 2.454792267026737 = 2.0
173 "00000010",-- (10010111) = -105.0 || 2.563474079532323 = 2.0
174 "00000010",-- (10011000) = -104.0 || 2.6769167953710262 = 2.0
175 "00000010",-- (10011001) = -103.0 || 2.7953243774985372 = 2.0
176 "00000010",-- (10011010) = -102.0 || 2.9189091120580306 = 2.0
177 "00000011",-- (10011011) = -101.0 || 3.0478919103753572 = 3.0
178 "00000011",-- (10011100) = -100.0 || 3.18250261841557 = 3.0
179 "00000011",-- (10011101) = -99.0 || 3.3229803335326857 = 3.0
180 "00000011",-- (10011110) = -98.0 || 3.4695737282979517 = 3.0
181 "00000011",-- (10011111) = -97.0 || 3.6225413811405303 = 3.0
182 "00000011",-- (10100000) = -96.0 || 3.7821521134779053 = 3.0
183 "00000011",-- (10100001) = -95.0 || 3.948685332951331 = 3.0
184 "00000100",-- (10100010) = -94.0 || 4.122431382313737 = 4.0
185 "00000100",-- (10100011) = -93.0 || 4.303691893443283 = 4.0
186 "00000100",-- (10100100) = -92.0 || 4.492780145875043 = 4.0
187 "00000100",-- (10100101) = -91.0 || 4.690021429155295 = 4.0
188 "00000100",-- (10100110) = -90.0 || 4.895753408227562 = 4.0
189 "00000101",-- (10100111) = -89.0 || 5.110326490956257 = 5.0
190 "00000101",-- (10101000) = -88.0 || 5.3341041967821905 = 5.0
191 "00000101",-- (10101001) = -87.0 || 5.567463525383797 = 5.0
192 "00000101",-- (10101010) = -86.0 || 5.81079532408854 = 5.0
193 "00000110",-- (10101011) = -85.0 || 6.064504652639856 = 6.0
194 "00000110",-- (10101100) = -84.0 || 6.3290111437761585 = 6.0
195 "00000110",-- (10101101) = -83.0 || 6.604749357919308 = 6.0
196 "00000110",-- (10101110) = -82.0 || 6.892169130100257 = 6.0
197 "00000111",-- (10101111) = -81.0 || 7.191735907069339 = 7.0
198 "00000111",-- (10110000) = -80.0 || 7.503931072347218 = 7.0
199 "00000111",-- (10110001) = -79.0 || 7.829252256770357 = 7.0
200 "00001000",-- (10110010) = -78.0 || 8.168213631871376 = 8.0
201 "00001000",-- (10110011) = -77.0 || 8.521346183210536 = 8.0
202 "00001000",-- (10110100) = -76.0 || 8.889197960539537 = 8.0
203 "00001001",-- (10110101) = -75.0 || 9.272334301433533 = 9.0
204 "00001001",-- (10110110) = -74.0 || 9.67133802477231 = 9.0
205 "00001010",-- (10110111) = -73.0 || 10.086809590187293 = 10.0
206 "00001010",-- (10111000) = -72.0 || 10.519367219319044 = 10.0
207 "00001010",-- (10111001) = -71.0 || 10.969646974450724 = 10.0
208 "00001011",-- (10111010) = -70.0 || 11.438302789798453 = 11.0
209 "00001011",-- (10111011) = -69.0 || 11.926006450451535 = 11.0
210 "00001100",-- (10111100) = -68.0 || 12.43344751366557 = 12.0
211 "00001100",-- (10111101) = -67.0 || 12.961333166922968 = 12.0

```

```

212 "00001101",-- (10111110) = -66.0    || 13.510388016890333 = 13.0
213 "00001110",-- (10111111) = -65.0    || 14.08135380312426 = 14.0
214 "00001110",-- (11000000) = -64.0    || 14.674989030110401 = 14.0
215 "00001111",-- (11000001) = -63.0    || 15.292068510968507 = 15.0
216 "00001111",-- (11000010) = -62.0    || 15.933382815924531 = 15.0
217 "00010000",-- (11000011) = -61.0    || 16.59973761844398 = 16.0
218 "00010001",-- (11000100) = -60.0    || 17.29195293174505 = 17.0
219 "00010010",-- (11000101) = -59.0    || 18.01086222827223 = 18.0
220 "00010010",-- (11000110) = -58.0    || 18.757311434617044 = 18.0
221 "00010011",-- (11000111) = -57.0    || 19.532157794331532 = 19.0
222 "00010100",-- (11001000) = -56.0    || 20.33626859109847 = 20.0
223 "00010101",-- (11001001) = -55.0    || 21.170519724809946 = 21.0
224 "00010110",-- (11001010) = -54.0    || 22.035794133271526 = 22.0
225 "00010110",-- (11001011) = -53.0    || 22.93298005250171 = 22.0
226 "00010111",-- (11001100) = -52.0    || 23.862969108946842 = 23.0
227 "00011000",-- (11001101) = -51.0    || 24.826654237388475 = 24.0
228 "00011001",-- (11001110) = -50.0    || 25.824927418894468 = 25.0
229 "00011010",-- (11001111) = -49.0    || 26.858677233866867 = 26.0
230 "00011011",-- (11010000) = -48.0    || 27.928786226076923 = 27.0
231 "00011101",-- (11010001) = -47.0    || 29.036128074562075 = 29.0
232 "00011110",-- (11010010) = -46.0    || 30.181564571396905 = 30.0
233 "00011111",-- (11010011) = -45.0    || 31.36594240464876 = 31.0
234 "00100000",-- (11010100) = -44.0    || 32.59008974729516 = 32.0
235 "00100001",-- (11010101) = -43.0    || 33.8548126545158 = 33.0
236 "00100011",-- (11010110) = -42.0    || 35.1608912735824 = 35.0
237 "00100100",-- (11010111) = -41.0    || 36.509075872551044 = 36.0
238 "00100101",-- (11011000) = -40.0    || 37.900082696112506 = 37.0
239 "00100111",-- (11011001) = -39.0    || 39.33458965926929 = 39.0
240 "00101000",-- (11011010) = -38.0    || 40.813231891972286 = 40.0
241 "00101010",-- (11011011) = -37.0    || 42.336597150452434 = 42.0
242 "00101011",-- (11011100) = -36.0    || 43.90522111370262 = 43.0
243 "00101101",-- (11011101) = -35.0    || 45.51958258638069 = 45.0
244 "00101111",-- (11011110) = -34.0    || 47.18009863228597 = 47.0
245 "00110000",-- (11011111) = -33.0    || 48.88711966547641 = 48.0
246 "00110010",-- (11100000) = -32.0    || 50.64092452900307 = 50.0
247 "00110100",-- (11100001) = -31.0    || 52.441715594098426 = 52.0
248 "00110110",-- (11100010) = -30.0    || 54.28961391541896 = 54.0
249 "00111000",-- (11100011) = -29.0    || 56.184654480555785 = 56.0
250 "00111010",-- (11100100) = -28.0    || 58.126781594434945 = 58.0
251 "00111100",-- (11100101) = -27.0    || 60.11584444137255 = 60.0
252 "00111110",-- (11100110) = -26.0    || 62.15159286936576 = 62.0
253 "01000000",-- (11100111) = -25.0    || 64.23367344262975 = 64.0
254 "01000010",-- (11101000) = -24.0    || 66.36162580936859 = 66.0
255 "01000100",-- (11101001) = -23.0    || 68.5348794322364 = 68.0
256 "01000110",-- (11101010) = -22.0    || 70.75275072884732 = 70.0
257 "01001001",-- (11101011) = -21.0    || 73.01444066897737 = 73.0
258 "01001011",-- (11101100) = -20.0    || 75.31903287372512 = 75.0
259 "01001101",-- (11101101) = -19.0    || 77.6654922598253 = 77.0
260 "01010000",-- (11101110) = -18.0    || 80.05266426951576 = 80.0
261 "01010010",-- (11101111) = -17.0    || 82.47927472283249 = 82.0
262 "01010100",-- (11110000) = -16.0    || 84.94393032494948 = 84.0
263 "01010111",-- (11110001) = -15.0    || 87.44511985620915 = 87.0
264 "01011001",-- (11110010) = -14.0    || 89.98121606683623 = 89.0
265 "01011100",-- (11110011) = -13.0    || 92.55047829204331 = 92.0

```

```
266 "01011111",-- (11110100) = -12.0    || 95.15105579638626 = 95.0
267 "01100001",-- (11110101) = -11.0    || 97.78099184889408 = 97.0
268 "01100100",-- (11110110) = -10.0    || 100.43822852277974 = 100.0
269 "01100111",-- (11110111) = -9.0     || 103.12061220554642 = 103.0
270 "01101001",-- (11111000) = -8.0     || 105.82589979716352 = 105.0
271 "01101100",-- (11111001) = -7.0     || 108.55176556583123 = 108.0
272 "01101111",-- (11111010) = -6.0     || 111.29580862282243 = 111.0
273 "01110010",-- (11111011) = -5.0     || 114.05556097013061 = 114.0
274 "01110100",-- (11111100) = -4.0     || 116.82849606730782 = 116.0
275 "01110111",-- (11111101) = -3.0     || 119.61203785708545 = 119.0
276 "01111010",-- (11111110) = -2.0     || 122.40357018327 = 122.0
277 "01111101",-- (11111111) = -1.0     || 125.20044652911506 = 125.0
278
279 -- 2 => "11111111" , --255
280 -- 3 => "11010101" ,
281 others => "00000000"
282 );
283 begin
284 -----
285 data_out <= myrom(to_integer(unsigned(address))) ;
286 end architecture ;
```

Código A.9 – LUT para Função de Ativação Sigmoid de 8 bits