UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE CARAÚBAS
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

FELIPE WILLIAM DA COSTA

DESENVOLVIMENTO DE SISTEMAS EMBARCADOS INTELIGENTES PARA APLICAÇÕES EM INTERNET DAS COISAS UTILIZANDO TINY MACHINE LEARNING

CARAÚBAS
2023

FELIPE WILLIAM DA COSTA

DESENVOLVIMENTO DE SISTEMAS EMBARCADOS INTELIGENTES PARA APLICAÇÕES EM INTERNET DAS COISAS UTILIZANDO TINY MACHINE LEARNING

Monografia apresentada à Universidade Federal Rural do Semi-Árido, como requisito à obtenção do título bacharel em Engenharia Elétrica.

Orientador: D.Sc. Francisco de A. Brito Filho.

CARAÚBAS

2023

C837d Costa, Felipe William da.
Desenvolvimento de Sistemas Embarcados Inteligentes para Aplicações em Internet das Coisas Utilizando Tiny Machine Learning / Felipe William da Costa. – 2023.
63 f. : il.

Orientador: Francisco de Assis Brito Filho.
Monografia (graduação) – Universidade Federal Rural do Semi-árido, Curso de Engenharia Elétrica, 2023.

1. TinyML. 2. Co-design. 3. TensorFlow Lite Micro. 4. FPGA. I. Brito Filho, Francisco de Assis , orient. II. Título.

Ficha catalográfica elaborada por sistema gerador automáto em conformidade com AACR2 e os dados fornecidos pelo) autor(a).
Biblioteca Campus Caraúbas
Bibliotecária: Dalvanira Brito Rodrigues
CRB: 15/700

FELIPE WILLIAM DA COSTA

DESENVOLVIMENTO DE SISTEMAS EMBARCADOS INTELIGENTES PARA APLICAÇÕES EM INTERNET DAS COISAS UTILIZANDO TINY MACHINE LEARNING

Monografia apresentada à Universidade Federal Rural do Semi-Árido, como requisito à obtenção do título bacharel em Engenharia Elétrica.

Defendida em: 10 de outubro de 2023

BANCA EXAMINADORA

FRANCISCO DE ASSIS BRITO FILHO [redacted]
Assinado de forma digital por FRANCISCO DE ASSIS BRITO FILHO: [redacted]
Dados: 2023.10.17 13:26:12 -03'00'

D.Sc. Francisco de A. Brito Filho (Orientador)
Universidade Federal Rural do Semi-Árido
(UFERSA)

Documento assinado digitalmente
gov.br
PEDRO THIAGO VALERIO DE SOUZA
Data: 17/10/2023 18:18:01-0300
Verifique em https://validar.iti.gov.br

D.Sc. Pedro Thiago Valério de Souza
Universidade Federal Rural do Semi-Árido
(UFERSA)

Documento assinado digitalmente
gov.br
SILVAN FERREIRA DA SILVA JUNIOR
Data: 19/10/2023 16:48:24-0300
Verifique em https://validar.iti.gov.br

M.Sc. Silvan Ferreira da Silva Júnior
Universidade Federal do Rio Grande do Norte
(UFRN)

AGRADECIMENTOS

Aos meus pais Antônio João da Costa Neto e Maria Benigna Costa por me proporcionar esta oportunidade.

Ao Prof. D.Sc. Francisco de Assis Brito Filho pela orientação durante este período e por me apresentar novos horizontes.

Aos amigos Werly Lucena, Guilherme Brasil, Luan Nunes, José Lira, Janneson Ferreira, John Lennon, Fábio Caetano, Edson Alves, Édney Viana, Monaliza Soares e Naara Melo pela ajuda e pelos momentos de descontração.

RESUMO

O *Tiny Machine Learning* (TinyML) é um novo paradigma que surge para possibilitar a execução de modelos de *Machine Learning* (ML) em plataformas onde os recursos de *hardware* são altamente restringidos, como os microcontroladores, que são usualmente empregados no desenvolvimento de sistemas embarcados. Muitos *frameworks* surgiram para facilitar a inferência de modelos de ML em microcontroladores, entre eles, o TensorFlow Lite Micro (TFLM) ganhou notoriedade por ser de código aberto e compatível com uma ampla gama de dispositivos e arquiteturas. Para alcançar um processamento mais eficiente dos modelos de ML deu-se origem aos chamados aceleradores de *hardware* que consistem em *hardware* adicionais e especializados encarregados de desempenhar operações designadas pelo desenvolvedor de forma otimizada através do *co-design hardware/software*. O CFU Playground é um *framework full-stack* de código aberto voltado para o *design* de aceleradores de ML para sistemas embarcados em *Field-Programmable Gate Array* (FPGA) no qual foi-se utilizado para acelerar a inferência do modelo através de uma *Custom Function Unit* (CFU). Neste trabalho, utiliza-se um modelo de visão computacional embarcado baseado em Redes Neurais Convolucionais para inferência nas plataformas Kosagi Fomu e ESP32-C3 onde aplicam-se otimizações por meio de bibliotecas com *kernel* otimizado e do *co-design* para tecer comparações de desempenho entre as plataformas. Após aplicação das otimizações, obteve-se uma diminuição de 33,19% no tempo de inferência do modelo no ESP32-C3 por meio da biblioteca ESP-NN e 65,74% no FPGA Fomu através do *co-desing* e implementação de uma CFU ambas em comparação com a inferência sem nenhuma otimização implementada.

Palavras-chave: TinyML; Co-design; TensorFlow Lite Micro; FPGA.

ABSTRACT

Tiny Machine Learning (TinyML) is a new paradigm that emerges to enable the execution of Machine Learning (ML) models on platforms where hardware resources are highly constrained, such as microcontrollers, which are usually used in the development of embedded systems. Many frameworks have emerged to facilitate the inference of ML models on microcontrollers, among them, TensorFlow Lite Micro (TFLM) gained prominence for being open source and compatible with a wide range of devices and architectures. To achieve more efficient processing of ML models, hardware accelerators were created, which consist of additional and specialized hardware responsible for performing operations designated by the developer in an optimized way through hardware/software co-design. The CFU Playground is a full-stack open source framework aimed at the design of ML accelerators for embedded systems on Field-Programmable Gate Array (FPGA) in which it was used to accelerate the inference of the model through a Custom Function Unit (CFU). In this work, we use an embedded computer vision model based on Convolutional Neural Networks for inference on the Kosagi Fomu and ESP32-C3 platforms where we apply optimizations using libraries with optimized kernel and co-design to make performance comparisons between the platforms. After applying the optimizations, we obtained a decrease of 33.19% in the inference time of the model on the ESP32-C3 using the ESP-NN library and 65.74% on the FPGA Fomu through co-design and implementation of a CFU both compared to the inference without any optimization implemented.

Keywords: TinyML; Co-design; TensorFlow Lite Micro; FPGA.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Adaline	*Adaptive Linear Neuron*
API	Interface de Programação de Aplicativos
ASIC	*Application Specific Integrated Circuits*
BRAM	*Block* RAM
CFU	*Custom Function Unit*
CLB	*Configurable Logic Block*
CNN	*Convolutional Neural Networks*
CPU	*Central Processing Unit*
DFF	*D Flip-Flop*
DL	*Deep Learning*
DSL	*Domain-Specific Language*
DSP	*Digital Signal Processor*
FPGA	*Field-Programmable Gate Array*
GUI	*Graphical User Interface*
HDL	*Hardware Description Language*
IA	Inteligência Artificial
IDC	*International Data Corporation*
IoT	*Internet of Things*
ISPU	*Intelligent Sensor Processing Unit*
LUT	*look-up table*
MCP	Neurônio de McCulloch-Pitts
ML	*Machine Learning*
MLP	*Multilayer Perceptron*
PLB	*Programmable Logic Blocks*
PnR	*Place-and-Route*
RAM	*Random Access Memory*
ReLU	*Rectified Linear Units*
RISC-V	*Reduced Instruction Set Computer - Five*
RNA	Rede Neural Artificial
ROM	*Read-Only Memory*

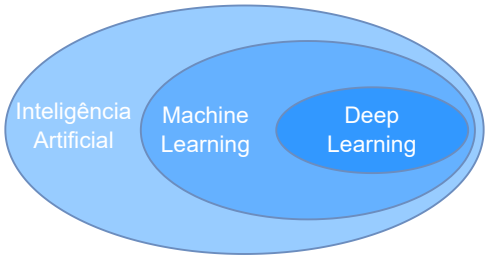
RTL	*Register-Transfer Level*
SIMD	*Single Instruction, Multiple Data*
SoC	*System-on-Chip*
SPI	*Serial Peripheral Interface*
SRAM	*Static Random-Access Memory*
TFLM	TensorFlow Lite Micro
TinyML	*Tiny Machine Learning*
UART	*Universal Asynchronous Receiver/Transmitter*

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, estamos presenciando um grande avanço no que diz respeito à Inteligência Artificial (IA), ao *Machine Learning* (ML) e ao *Deep Learning* (DL). São inúmeras as aplicações que só foram possíveis serem implementadas por meio de aplicações utilizando IA. Podem ser citados como exemplos: *chatbots*, carros autônomos, assistentes virtuais, etc. A Figura (1) elucida o campo da IA e seus sub-campos.

Figura 1 – IA vs. *Machine Learning* vs. *Deep Learning*.



Fonte: Adaptado de Chollet (2018).

Para Chollet (2018, p. 4) podemos definir o campo da IA como o esforço para automatizar tarefas intelectuais que normalmente são executadas por humanos. Há uma vasta quantidade de microcontroladores em dispositivos eletrônicos capazes de executar um modelo de *Machine Learning*, porém, a maioria deles possuem uma capacidade reduzida de poder computacional como memória e frequência do sinal de *clock*. Deste modo, um novo paradigma teve que ser criado para contornar a disponibilidade de recursos restritos dos sistemas embarcados — o TinyML.

Com o acelerado aumento da quantidade de dispositivos relacionados à *Internet of Things* (IoT), a quantidade de dados enviados para a nuvem também está em constante crescimento dado o aumento em ritmo acelerado dos dispositivos IoT presente na vida cotidiana. Ao mesmo tempo, o número de plataformas que oferecem serviços de computação na nuvem cresce e tornam-se cada vez mais accessíveis aos usuários. Segundo Shirer e MacGillivray (2019) uma nova previsão da *International Data Corporation* (IDC) estima que haverão cerca de 41,6 bilhões de dispositivos de IoT conectados gerando cerca de 79,4 zettabytes (ZB) de dados em 2025. No entanto, não há necessidade de transferir todos esses dados para a nuvem quando pode-se tirar proveito da computação de borda para os dispositivos IoT.

O termo TinyML é o mais utilizado no contexto de um modelo leve de *Machine Learning* para dispositivos embarcados. Além disso, estes dispositivos estão frequentemente

localizados na borda do mundo físico adquirindo dados relacionados a alguma grandeza física. Para Banbury *et al.* (2020) os *hardwares* de TinyML são definidos pelo seu ultra-baixo consumo de potência no qual é frequentemente na ordem de 1 mW e abaixo. Para tal critério, englobam-se os processadores 32 bits ARM Cortex M7, RISC-V PULP e abaixo.

A inteligência artificial na borda também está modificando os sensores. Recentemente os fabricantes tem adicionado recursos de IA através de circuitos integrados ou *Application Specific Integrated Circuits* (ASIC), um exemplo dessa nova tecnologia emergente são as *Intelligent Sensor Processing Unit* (ISPU) da STMicroelectronics.

Entre os desafios a serem enfrentados pelo TinyML, podem-se elencar: o reduzido poder computacional presente na maioria dos microcontroladores; a falta de uma estrutura unificada que possa ser utilizada em uma ampla gama de *hardwares*, dado que existem ferramentas distintas que possibilitam a implementação de um modelo de TinyML; a ausência de um conjunto de dados para TinyML de código aberto que corresponda aos mesmos dados adquiridos pelos sensores e que seja grande o suficiente para *benchmarking* e pesquisa. Como vantagens advindas da utilização de modelos de TinyML, têm-se: algumas aplicações que realizam inferência em modelos de inteligência artificial na nuvem podem ser muito centradas nos dados, o que eleva o seu custo. Portanto, realizar a inferência na origem dos dados diminui o custo de uma aplicação; economia de energia, tendo em vista que não há a necessidade de realizar a transmissão dos dados, que é um processo que demanda um consumo energético considerável; redução da latência e aumento da privacidade e segurança.

O crescimento dos dispositivos embarcados de baixo consumo juntamente com a otimização dos algoritmos de ML proporcionaram uma nova perspectiva para o IoT, a possibilidade de implementar modelos de ML em dispositivos IoT através do TinyML. O TinyML no âmbito da Internet das Coisas visa promover baixa latência, melhor utilização da largura de banda, fortalecer a segurança dos dados, aumentar a privacidade e reduzir custos, dentre os campos de aplicações pode-se citar: reconhecimento de imagens (Wong *et al.*, 2020), reconhecimento de gestos (Prasanna *et al.*, 2022), detecção de face (Giordano *et al.*, 2020), detecção de anomalias (Siang *et al.*, 2021), veículos autônomos (Prado *et al.*, 2021), gestão de tráfico (Roshan *et al.*, 2021), aquisição de sinais médicos (Kim *et al.*, 2023) e reconhecimento de fala (Prasanna *et al.*, 2022). O TinyML permite que aplicações IoT inteligentes operem de maneira adequada sem que haja acessos constantes a serviços na nuvem o que torna-se uma solução de baixo custo. Como antes mencionado, o poder de processamento e memória são alguns dos desafios que o TinyML

deve enfrentar. Para otimizar a desempenho dos dispositivos foram desenvolvidas bibliotecas contendo *kernels* otimizados para execução de modelos em dispositivos IoT como exemplos destas bibliotecas pode-se citar a CMSIS NN[1] desenvolvida para maximizar o desempenho e minimizar o consumo de memória de uma Rede Neural Artificial (RNA) em processadores ARM Cortex-M, outro exemplo é a ESP-NN[2] para plataformas ESP32 contendo processadores com arquitetura *Reduced Instruction Set Computer - Five* (RISC-V).

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é o desenvolvimento de sistemas embarcados inteligentes em diferentes plataformas voltado para aplicações de IoT por meio de modelos de ML utilizando a abordagem fornecida pelo TinyML e aplicar otimizações através de bibliotecas que possuem algorítimos otimizados para desempenhar as operações requisitadas e do *co-design* através da descrição, utilizando Verilog, de um *hardware* adicional para uma operação específica.

1.1.2 Objetivos Específicos

Dentre os objetivos específicos, pode-se citar:

- Implementar o modelo em plataformas *System-on-Chip* (SoC) e FPGA;
- Avaliar o desempenho da inferência em ambas as plataformas;
- Avaliar o consumo energético de ambas;
- Aplicar otimizações através de bibliotecas e do *co-design hardware/software*;
- Fornecer um comparativo de desempenho entre as placas utilizadas.

[1] CMSIS NN: https://github.com/ARM-software/CMSIS-NN.
[2] ESP-NN: https://github.com/espressif/esp-nn.

2 FUNDAMENTAÇÃO TEÓRICA

A primeira RNA criada foi a rede Perceptron que teve como base o Neurônio de McCulloch-Pitts (MCP) de onde surgiu a primeira regra de aprendizado. Os conceitos fornecidos pela rede Perceptron foram importantes para o avanço da IA, na Seção 2.1 aborda-se a temática das RNAs de forma analítica. Para executar a inferência de um modelo de ML em dispositivos com recursos limitados, como os microcontroladores, surge uma nova abordagem denominada de TinyML. Na Seção 2.2 discuti-se acerca das ferramentas utilizadas na inferência e otimização de modelos de TinyML.

2.1 Inteligência Artificial

Nas subseções a seguir, aborda-se a temática das RNAs, mais especificamente as redes abrangidas pelos campos do *Machine Learning* e *Deep Learning* que se referem à capacidade das máquinas realizarem tarefas que normalmente são feitas por humanos.

2.1.1 Neurônio Artificial: Perceptron

Com o objetivo de captar, de forma simplificada, o funcionamento de um neurônio biológico, Warren McCulloch e Walter Pitts descreveram o MCP. Segundo Raschka e Mirjalili (2019, p. 20), o MCP foi descrito como uma porta lógica com saídas binárias, onde múltiplos sinais chegam até o neurônio, acumulam-se no corpo do mesmo e, se ultrapassarem um certo limite, um sinal de saída é gerado. Posteriormente, Frank Rosenblatt publicou o primeiro conceito de regra de aprendizado baseado no modelo do neurônio MCP, no qual foi proposto um algoritmo onde os pesos seriam ajustados automaticamente e multiplicados pelas entradas, a fim de decidir se um neurônio transmite ou não o sinal de saída. Idealizando um neurônio cujas as saídas binárias são 1 e 0, pode-se definir uma função de ativação $(\sigma(z))$ que recebe a combinação linear entre os valores de entrada e seus respectivos coeficientes ou pesos, tal que $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n$, tal que:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

onde a função de ativação pode ser determinada em sentenças de modo que θ é um limite estipulado, como segue:

$$\sigma(z) = \begin{cases} 1, & \text{se } z \geq \theta, \\ 0, & \text{se } z < \theta. \end{cases}$$

Por simplicidade podemos definir $w_0 = -\theta$ e $x_0 = 1$, onde $w_0 = -\theta$ é chamado de unidade de *bias*, logo, pode-se escrever que:

$$z = w_0 x_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \sum_{j=0}^{n} w_j x_j = \mathbf{w}^T \mathbf{x} \tag{2.1}$$

e a nova função de ativação $\sigma(z)$:

$$\sigma(z) = \begin{cases} 1, & \text{se } z \geq 0, \\ 0, & \text{se } z < 0. \end{cases}$$

A função em questão é conhecida como degrau unitário e não é diferenciável em todos os pontos do seu domínio o que pode acarretar problemas para o treinamento da rede. Desta forma, pode-se utilizar a função sigmoide como função de ativação para evitar possíveis erros:

$$S(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

O treinamento da rede segue dois princípios básicos, o primeiro deles é que deve-se inciar o vetor de pesos ($\mathbf{w}$) com valores nulos ou valores randômicos muito próximos a zero e para cada entrada $x^{(i)}$ deve-se obter sua saída ($\hat{y}$) e atualizar os pesos (w_j). De maneira mais formal podemos escrever a atualização dos pesos w_j como sendo:

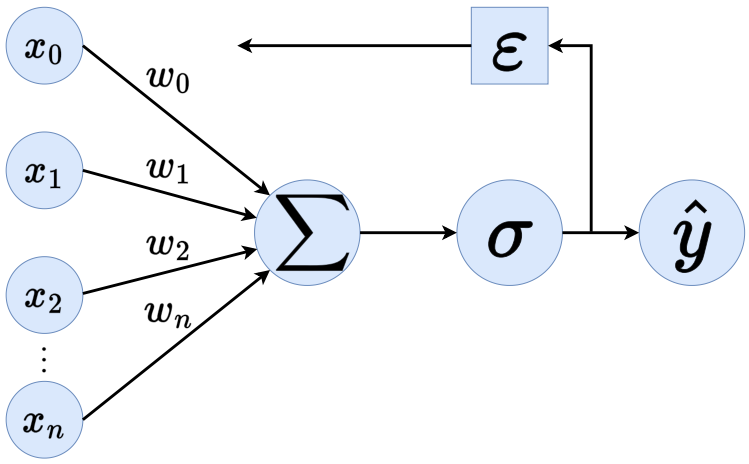
$$w_j := w_j + \Delta w_j \tag{2.3}$$

onde Δw_j pode ser calculado através de:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)} \tag{2.4}$$

em que η é o coeficiente de aprendizado, $y^{(i)}$ é o valor real, $\hat{y}^{(i)}$ é o valor predito e $x_j^{(i)}$ é a entrada da rede. A representação de uma rede neural Perceptron é apresentada na Figura (2).

Figura 2 – Representação de uma rede neural Perceptron simples onde ε representa o erro da rede.



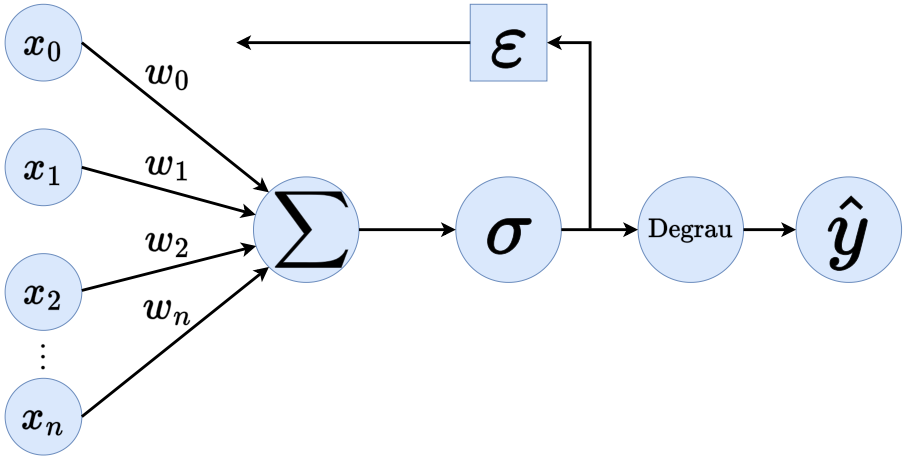
Fonte: Adaptado de Raschka e Mirjalili (2019).

2.1.2 Neurônio Linear Adaptativo: Adaline

Nesta Seção, aborda-se outro tipo de rede neural simples: a *Adaptive Linear Neuron* (Adaline). O algoritmo da rede Adaline aborda conceitos relacionados à minimização das funções de custo, os quais, por sua vez, servem de base para algoritmos de classificação, regressão logística, modelos de classificação, entre outros.

Em comparação à RNA apresentada na Seção 2.1.1, a Adaline usa uma regra distinta para atualizar os pesos, que é executado com base em uma função de ativação linear, enquanto na rede Perceptron se utiliza de uma função degrau para tal. Conforme emprega-se uma função de ativação linear para ajustar os pesos, uma função de limiar, semelhante a função degrau, é utilizada para realizar de predição da rede. Desta forma, a Figura (3) apresenta a rede neural Adaline.

Figura 3 – Representação da rede neural Adaline onde σ representa uma função linear.



Fonte: Autor (2023).

Nos algoritmos de *Machine Learning* supervisionados é definida uma função de custo que deverá ser minimizada durante o processo de aprendizagem. Para a rede Adaline, pode-se definir a função de custo $J(\mathbf{w})$ para otimizar o ajuste dos pesos como sendo a soma dos erros quadrados entre a saída predita pela rede e pelo o seu real valor, como segue:

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left[y^{(i)} - \sigma\left(z^{(i)}\right)\right]^2 \tag{2.5}$$

onde o termo $\frac{1}{2}$ é adicionado para facilitar a derivação do gradiente da função de custo ou de perda com relação ao peso. Segundo Raschka e Mirjalili (2019, p. 38) a principal vantagem de uma função de ativação linear, em contraste a função degrau, é que a função de custo se torna diferenciável. Outro aspecto notável é de que a mesma é uma função quadrática e podemos usar o algoritmo de otimização conhecido por gradiente descendente para determinar os pesos que minimizam a função de custo.

Utilizando o gradiente descendente, pode-se atualizar os pesos indo na direção oposta do gradiente $\nabla J(\mathbf{w})$ da função de custo $J(\mathbf{w})$, como segue:

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w} \tag{2.6}$$

onde $\Delta\mathbf{w}$ é definido como sendo o gradiente negativo multiplicado pelo coeficiente de aprendizado, como é mostrado a seguir:

$$\Delta\mathbf{w} = -\eta\nabla J(\mathbf{w}) \tag{2.7}$$

Para determinar o gradiente da função de custo é necessário que seja calculado a derivada parcial da função de custo em relação a cada peso, tal que:

$$\frac{\partial J}{\partial w_j} = -\sum_i \left[y^{(i)} - \sigma\left(z^{(i)}\right)\right] x_j^{(i)} \tag{2.8}$$

Pode-se escrever que os pesos são atualizados na forma:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_i \left[y^{(i)} - \sigma\left(z^{(i)}\right)\right] x_j^{(i)} \tag{2.9}$$

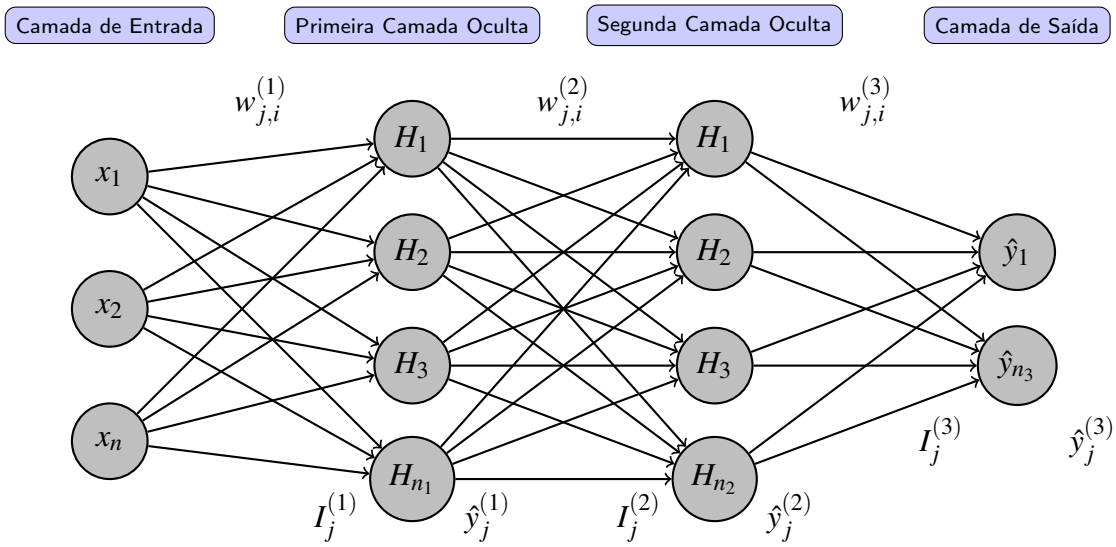
Contanto que todos os pesos sejam atualizados de forma simultânea, a regra de aprendizado da rede Adaline é dada pela Equação (2.6).

2.1.3 Redes Neurais de Múltiplas Camadas

Nas seções anteriores, foram abordadas as redes neurais simples como a rede Perceptron e Adaline, agora volta-se a atenção para as redes neurais de múltiplas camadas. A rede

Perceptron de múltiplas camadas — do inglês *Multilayer Perceptron* (MLP) — consiste em camadas formadas por neurônios Perceptron sem realimentação como mostra a Figura (4), onde tem-se uma camada de entrada, duas camadas ocultas, sendo que a quantidade de camadas ocultas pode variar, e uma camada de saída. Cada camada está totalmente conectada a camada seguinte e se a rede possuir mais de uma camada oculta, pode-se defini-lá como uma rede neural artificial profunda. Uma das motivações para a utilização de redes MLP é a capacidade de maior abstração, isto é, a capacidade de manipular e representar conceitos que não estão associados diretamente aos objetos e tal capacidade de abstração permite lidar com problemas não-lineares e complexos.

Figura 4 – Rede Perceptron de múltiplas camadas totalmente conectada.



Fonte: Adaptado de Silva *et al.* (2016).

De acordo com Segundo (2018, p. 37) o sinal de entrada se propaga até a camada de saída, neste caso, o treinamento da rede é feita de forma supervisionada no qual o algoritmo mais famoso é o *backpropagation*. Para determinar analiticamente o processo de ajuste dos pesos sinápticos de uma rede MLP através do algoritmo *backpropagation*, considere a rede apresentada na Figura (4) onde $w_{j,i}^{(L)}$ representa uma matriz de pesos sinápticos onde estão conectados o j-ésimo neurônio da camada L ao i-ésimo neurônio da camada $L-1$, logo, podemos definir que:

- $w_{j,i}^{(3)}$ representa a matriz de pesos que conecta j-ésimo neurônio da camada de saída ao i-ésimo neurônio da segunda camada oculta ou intermediária;
- $w_{j,i}^{(2)}$ representa a matriz de pesos que conecta j-ésimo neurônio da segunda camada oculta ao i-ésimo neurônio da primeira camada oculta;
- $w_{j,i}^{(1)}$ representa a matriz de pesos que conecta o j-ésimo neurônio da primeira camada

oculta ao i-ésimo neurônio da camada de entrada;

- $I_J^{(L)}$ representam vetores contendo as entradas ponderadas em relação ao j-ésimo neurônio da camada L;
- $\hat{y}_j^{(L)}$ representam vetores contendo a saída do j-ésimo neurônio da camada L.

Desta forma, podemos escrever que:

$$I_j^{(1)} = \sum_{i=0}^{n} w_{j,i}^{(1)} \cdot x_i = w_{j,0}^{(1)} \cdot x_0 + w_{j,1}^{(1)} \cdot x_1 + w_{j,2}^{(1)} \cdot x_2 + \ldots + w_{j,n}^{(1)} \cdot x_n \tag{2.10}$$

$$I_j^{(2)} = \sum_{i=0}^{n} w_{j,i}^{(2)} \cdot \hat{y}_i^{(1)} = w_{j,0}^{(2)} \cdot \hat{y}_0^{(1)} + w_{j,1}^{(2)} \cdot \hat{y}_1^{(1)} + w_{j,2}^{(2)} \cdot \hat{y}_2^{(1)} + \ldots + w_{j,n}^{(2)} \cdot \hat{y}_{n_1}^{(1)} \tag{2.11}$$

$$I_j^{(3)} = \sum_{i=0}^{n} w_{j,i}^{(3)} \cdot \hat{y}_i^{(2)} = w_{j,0}^{(3)} \cdot \hat{y}_0^{(2)} + w_{j,1}^{(3)} \cdot \hat{y}_1^{(2)} + w_{j,2}^{(3)} \cdot \hat{y}_2^{(2)} + \ldots + w_{j,n}^{(3)} \cdot \hat{y}_{n_2}^{(2)} \tag{2.12}$$

De tal forma que as saídas são dadas por $\hat{y}_j^{(1)} = \sigma\left(I_j^{(1)}\right)$, $\hat{y}_j^{(2)} = \sigma\left(I_j^{(2)}\right)$ e $\hat{y}_j^{(3)} = \sigma\left(I_j^{(3)}\right)$. Ao final, é necessário quantificar o quão assertiva a rede é, essa quantificação fica a cargo do cálculo do erro quadrático considerando a k-ésima amostra utilizada para o treinamento, pode-se escrever que:

$$E(k) = \frac{1}{2} \sum_{j=1}^{n_3} \left(d_j(k) - \hat{y}_j^{(3)}(k)\right)^2 \tag{2.13}$$

onde $d_j(k)$ é o valor desejado e $\hat{y}_j^{(3)}(k)$ é o valor na camada de saída da rede para a k-ésima amostra. Dado um conjunto formado por p amostras, o desempenho do algoritmo *backpropagation* pode ser medido através do erro quadrático médio:

$$E_M = \frac{1}{p} \sum_{k=1}^{p} E(k) \tag{2.14}$$

O processo de treinamento visa minimizar o erro entre a saída produzida pela rede e as saídas desejadas através do ajuste dos pesos sinápticos das matrizes $w_{j,i}^{(L)}$. Segundo Silva *et al.* (2016, p. 100) considerando a k-ésima amostra de treinamento referente ao j-ésimo neurônio da camada, a regra de ajuste se torna parecida a utilizada na rede Adaline. Pode-se então minimizar o erro da camada de saída aplicando a definição de gradiente que pela regra de derivação em cadeia é dado por:

$$\nabla E^{(3)} = \frac{\partial E}{\partial w_{j,i}^{(3)}} = \frac{\partial E}{\partial \hat{y}_j^{(3)}} \frac{\partial \hat{y}_j^{(3)}}{\partial I_J^{(3)}} \frac{\partial I_J^{(3)}}{\partial w_{j,i}^{(3)}} \tag{2.15}$$

Pode-se então determinar as derivadas como sendo:

$$\frac{\partial E}{\partial \hat{y}_j^{(3)}} = -\left(d_j - \hat{y}_j^{(3)}\right) \tag{2.16}$$

$$\frac{\partial \hat{y}_j^{(3)}}{\partial I_j^{(3)}} = \sigma'\left(I_j^{(3)}\right) \tag{2.17}$$

$$\frac{\partial I_j^{(3)}}{\partial w_{j,i}^{(3)}} = \hat{y}_i^{(2)} \tag{2.18}$$

Substituindo as Equações (2.18), (2.17) e (2.16) em (2.15), determina-se $\frac{\partial E}{\partial w_{j,i}^{(3)}}$:

$$\frac{\partial E}{\partial w_{j,i}^{(3)}} = -\left(d_j - \hat{y}_j^{(3)}\right)\sigma'\left(I_j^{(3)}\right)\hat{y}_j^{(2)} \tag{2.19}$$

Portanto, o ajuste da matriz de pesos $w_{j,i}^{(3)}$ deve ser efetuado na direção contrária ao gradiente, tem-se que:

$$\Delta w_{j,i}^{(3)} = -\eta \frac{\partial E}{\partial w_{j,i}^{(3)}} = \eta \delta_j^{(3)} \hat{y}_j^{(2)} \tag{2.20}$$

onde $\delta_j^{(3)} = \left(d_j - \hat{y}_j^{(3)}\right)\sigma'\left(I_j^{(3)}\right)$ e é definido como sendo o gradiente local em relação ao j-ésimo neurônio. Pode-se então definir a seguinte relação iterativa para o ajuste dos pesos sinápticos:

$$w_{j,i}^{(3)} := w_{j,i}^{(3)} + \eta \delta_j^{(3)} \hat{y}_j^{(2)} \tag{2.21}$$

Para calibrar os pesos sinápticos das camadas intermediárias não tem-se acesso ao valor desejado, sendo assim, os pesos são corrigidos por meio do erro retro-propagado e ponderado pelos pesos sinápticos dos neurônios da camada posterior que foram previamente ajustados. Para a segunda camada oculta, tem-se:

$$\nabla E^{(2)} = \frac{\partial E}{\partial w_{j,i}^{(2)}} = \frac{\partial E}{\partial \hat{y}_j^{(2)}} \frac{\partial \hat{y}_j^{(2)}}{\partial I_j^{(2)}} \frac{\partial I_j^{(2)}}{\partial w_{j,i}^{(2)}} \tag{2.22}$$

Tal qual o procedimento anterior, pode-se determinar que:

$$\frac{\partial E}{\partial \hat{y}_j^{(2)}} = \sum_{k=1}^{n_3} \frac{\partial E}{\partial I_k^{(3)}} \frac{\partial I_k^{(3)}}{\partial \hat{y}_j^{(2)}} = \sum_{k=1}^{n_3} \frac{\partial E}{\partial I_k^{(3)}} \cdot \frac{\partial \left(\sum_{k=1}^{n_3} w_{k,j}^{(3)} \cdot \hat{y}_j^{(2)}\right)}{\partial \hat{y}_j^{(2)}} = \sum_{k=1}^{n_3} \frac{\partial E}{\partial I_k^{(3)}} \cdot w_{k,j}^{(3)} \tag{2.23}$$

$$\frac{\partial \hat{y}_j^{(2)}}{\partial I_j^{(2)}} = \sigma' \left(I_j^{(2)} \right) \tag{2.24}$$

$$\frac{\partial I_j^{(2)}}{\partial w_{j,i}^{(2)}} = \hat{y}_i^{(1)} \tag{2.25}$$

O termo $w_{k,j}^{(3)}$ na Equação (2.23) refere-se a todos os pesos sinápticos da camada de saída que estão conectados um neurônio j da segunda camada oculta. Levando-se em consideração as Equações (2.17) e (2.18), pode-se escrever que:

$$\frac{\partial E}{\partial \hat{y}_j^{(2)}} = -\sum_{k=1}^{n_3} \delta_k^{(3)} \cdot w_{k,j}^{(3)} \tag{2.26}$$

Assim, substituindo as Equações (2.26), (2.25) e (2.24) em (2.22), tem-se:

$$\frac{\partial E}{\partial w_{j,i}^{(2)}} = -\left(\sum_{k=1}^{n_3} \delta_k^{(3)} \cdot w_{k,j}^{(3)} \right) \sigma' \left(I_j^{(2)} \right) \hat{y}_i^{(1)} \tag{2.27}$$

portanto, pode-se definir o ajuste da matriz de pesos sinápticos $w_{j,i}^{(2)}$ como:

$$\Delta w_{j,i}^{(2)} = -\eta \frac{\partial E}{\partial w_{j,i}^{(2)}} = \eta \delta_j^{(2)} \hat{y}_i^{(1)} \tag{2.28}$$

onde $\delta_j^{(2)} = -\left(\sum_{k=1}^{n_3} \delta_k^{(3)} \cdot w_{k,j}^{(3)} \right) \sigma' \left(I_j^{(2)} \right)$, levando a expressão algorítmica:

$$w_{j,i}^{(2)} := w_{j,i}^{(2)} + \eta \delta_j^{(2)} \hat{y}_i^{(1)} \tag{2.29}$$

O procedimento para determinar a regra de ajuste dos pesos da primeira camada oculta é análogo ao realizado para a segunda camada, tal que:

$$\nabla E^{(1)} = \frac{\partial E}{\partial w_{j,i}^{(1)}} = \frac{\partial E}{\partial \hat{y}_j^{(1)}} \frac{\partial \hat{y}_j^{(1)}}{\partial I_j^{(1)}} \frac{\partial I_j^{(1)}}{\partial w_{j,i}^{(1)}} \tag{2.30}$$

Onde:

$$\frac{\partial E}{\partial \hat{y}_j^{(1)}} = \sum_{k=1}^{n_2} \frac{\partial E}{\partial I_k^{(2)}} \frac{\partial I_k^{(2)}}{\partial \hat{y}_j^{(1)}} = \sum_{k=1}^{n_2} \frac{\partial E}{\partial I_k^{(2)}} \cdot \frac{\partial \left(\sum_{k=1}^{n_2} w_{k,j}^{(2)} \cdot \hat{y}_j^{(1)} \right)}{\partial \hat{y}_j^{(1)}} = \sum_{k=1}^{n_2} \frac{\partial E}{\partial I_k^{(2)}} \cdot w_{k,j}^{(2)} \tag{2.31}$$

$$\frac{\partial \hat{y}_j^{(1)}}{\partial I_j^{(1)}} = \sigma' \left(I_j^{(1)} \right) \tag{2.32}$$

$$\frac{\partial I_j^{(1)}}{\partial w_{j,i}^{(1)}} = x_i \tag{2.33}$$

De modo que:

$$\frac{\partial E}{\partial \hat{y}_j^{(1)}} = -\sum_{k=1}^{n_2} \delta_k^{(2)} \cdot w_{k,j}^{(2)} \tag{2.34}$$

Por conseguinte, encontra-se:

$$\frac{\partial E}{\partial w_{j,i}^{(1)}} = -\left(\sum_{k=1}^{n_2} \delta_k^{(2)} \cdot w_{k,j}^{(2)}\right) \sigma'\left(I_j^{(1)}\right) x_i \tag{2.35}$$

Para enfim determinar a regra de ajuste dos pesos da matriz $w_{j,i}^{(1)}$, como segue:

$$\Delta w_{j,i}^{(1)} = -\eta \frac{\partial E}{\partial w_{j,i}^{(1)}} = \eta \delta_j^{(1)} x_i \tag{2.36}$$

sendo que $\delta_j^{(1)} = -\left(\sum_{k=1}^{n_2} \delta_k^{(2)} \cdot w_{k,j}^{(2)}\right) \sigma'\left(I_j^{(1)}\right)$, gerando a relação:

$$w_{j,i}^{(1)} := w_{j,i}^{(1)} + \eta \delta_j^{(1)} x_i \tag{2.37}$$

O processo de ajuste das matrizes de pesos sinápticos através da retro-propagação do erro pode ser generalizado para quaisquer topologia de rede MLP independente da quantidade de camadas ocultas.

2.1.4 Redes Neurais Convolucionais

Nas seções anteriores foi dado enfase nos princípios básicos para a construção de RNAs simples como as redes Perceptron e Adaline e as redes neurais de múltiplas camadas e como é efetuado o aprendizado dessas redes — o aprendizado refere-se ao ajuste dos pesos sinápticos utilizando uma regra de aprendizado que pode ou não envolver métodos para a minimização do erro ou algoritmos mais sofisticados, caso da retro-propagação do erro para redes MLPs — que fornece os principais conceitos das RNAs. Nesta Seção aborda-se as *Convolutional Neural Networks* (CNN) que são amplamente utilizadas para o reconhecimento de imagens, reconhecimento de objetos e de áudio.

Para O'Shea e Nash (2015, p. 2) as CNNs são análogas às RNAs, pois são compostas por neurônios que se auto-otimizam por meio do aprendizado. O neurônio realiza uma operação seguida de uma função não-linear, que serve de base para inúmeras RNAs. Pode-se elencar as

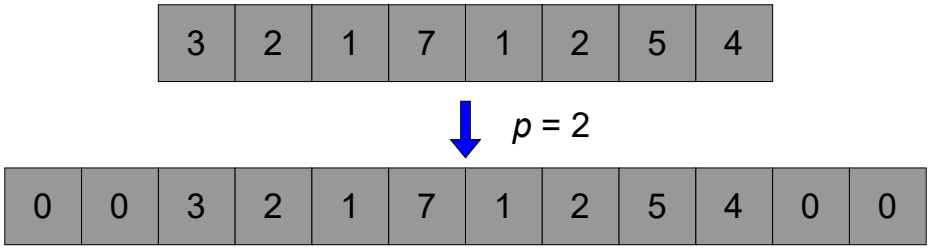
quatro principais camadas de uma CNN como sendo: convolução; Unidades Lineares Retificadas, do inglês *Rectified Linear Units* (ReLU); *pooling* e camadas totalmente conectadas ou *fully connected layers*.

A convolução discreta é uma das mais importantes operações de uma CNN, portanto, é essencial conhecer o seu funcionamento. Para tanto, apresenta-se a seguir operações de convolução unidimensionais (vetores) e bi-dimensionais (matrizes) e no espaço tridimensional. Para a operação de convolução unidimensional, considere os vetores $\mathbf{x}$ e $\mathbf{w}$ tal que $\mathbf{y} = \mathbf{x} * \mathbf{w}$, onde $\mathbf{x}$ é a entrada, $\mathbf{w}$ é o filtro que também pode ser chamado de *kernel* e $\mathbf{y}$ é a saída. Pode-se determinar matematicamente a operação de convolução como sendo:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \implies y[n] = \sum_{k=-\infty}^{\infty} x[n-k]w[k] \tag{2.38}$$

Considerando o vetor $\mathbf{x} = [x_0, x_1, \ldots, x_9]$. Neste caso, todos os valores fora do limite de $\mathbf{x}$ são preenchidos com o valor zero resultando em uma saída $\mathbf{y}$ também infinita, no entanto, na prática uma operação de convolução no intervalo $[-\infty, \infty]$ não é palpável tal que apenas um número finito de posições para $\mathbf{x}$ são preenchidas com o valor zero, esta ação é conhecida como *zero-padding* ou simplesmente *padding*, considerando uma imagem qualquer, *pixels* com o valor zero são utilizados para preencher a borda e preservar as informações nela contida. A Figura (5) elucida o processo de *padding*.

Figura 5 – Processo de *padding* em um vetor com $p = 2$.



Fonte: Adaptado de Raschka e Mirjalili (2019).

Assumindo que os vetores $\mathbf{x}$ e $\mathbf{w}$ possuem i e j elementos, respectivamente, onde $j \leq i$ tal que após o processo de *zero-padding* o tamanho é dado por $s = i + 2p$ e a operação de convolução pode ser escrita como:

$$y[n] = \sum_{k=0}^{j-1} x^p[n+j-k]w[k] \tag{2.39}$$

Onde x^p representa o vetor após o processo de *zero-padding*. É importante perceber que $\mathbf{x}$ e $\mathbf{w}$ estão indexados em direções diferentes do somatório, calcular o somatório em direções

opostas é equivalente a calcular o somatório na mesma direção com a ordem dos elementos de um dos vetores alterada. Para exemplificar, considere $\mathbf{w} = [1,1,0,1]$ e $\mathbf{x} = [3,2,1,7,1,2,5,4]$ sendo que a convolução é obtida através do produto escalar $\mathbf{x}[n:n+j] \cdot \mathbf{w}^r$ onde $\mathbf{w}^r = [1,0,1,1]$ é o vetor $\mathbf{w}$ rotacionado, tem-se então:

$$y[0] = \mathbf{x}[0:3] \cdot \mathbf{w}^r = 3 \cdot 1 + 2 \cdot 0 + 1 \cdot 1 + 7 \cdot 1 = 11$$

$$y[1] = \mathbf{x}[2:5] \cdot \mathbf{w}^r = 1 \cdot 1 + 7 \cdot 0 + 1 \cdot 1 + 2 \cdot 1 = 4$$

$$y[2] = \mathbf{x}[4:7] \cdot \mathbf{w}^r = 1 \cdot 1 + 2 \cdot 0 + 5 \cdot 1 + 4 \cdot 1 = 10$$

Percebe-se que $p = 0$ e também que desloca-se o vetor $\mathbf{w}^r$ duas posições a cada operação definindo assim um novo hiper-parâmetro da convolução, o *stride*, que é equivalente a $s = 2$ neste caso. O tamanho da saída de uma convolução pode ser determinado pelo número total de vezes que deslocamos o filtro $\mathbf{w}$ ao longo do vetor de entrada $\mathbf{x}$:

$$o = \left[\frac{i + 2p - j}{s} \right] + 1 \tag{2.40}$$

Onde i é o tamanho de vetor de entrada e j é a largura do filtro ou *kernel*. Pode-se estender os conceitos da convolução utilizados para 1D (caso unidimensional) para o 2D (caso bidimensional), considere uma matriz $\mathbf{X}_{i_1 \times i_2}$ como matriz de entrada e uma matriz de filtro $\mathbf{W}_{j_1 \times j_2}$ de forma que $j_1 \leq i_1$ e $j_2 \leq i_2$ onde $\mathbf{Y} = \mathbf{X} * \mathbf{W}$ é resultado da convolução 2D, logo, tem-se que:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} \implies Y[n,m] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} X[n-k_1, m-k_2] W[k_1,k_2] \tag{2.41}$$

Para esclarecer como a processo de convolução 2D ocorre, considere uma matriz de entrada $\mathbf{X}_{3\times3}$ que após o *padding* de $p = (1,1)$ torna-se a matriz $\mathbf{X}_{5\times5}$ a seguir:

$$\mathbf{X}_{5\times5} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 5 & 0 & 1 & 0 \\ 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

E uma matriz de filtro $\mathbf{W}_{3\times3}$ como segue:

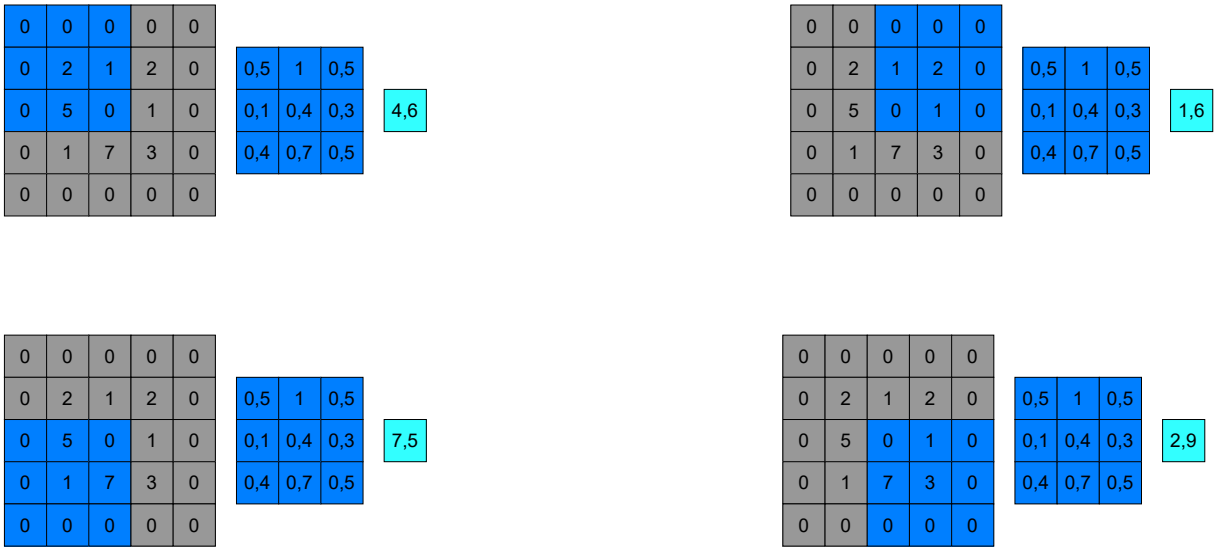
$$\mathbf{W}_{3\times3} = \begin{bmatrix} 0,5 & 0,7 & 0,4 \\ 0,3 & 0,4 & 0,1 \\ 0,5 & 1 & 0,5 \end{bmatrix}$$

Como no caso anterior, é necessário rotacionar a matriz de filtro tornando-a igual a:

$$\mathbf{W}^{r}_{3\times3} = \begin{bmatrix} 0,5 & 1 & 0,5 \\ 0,1 & 0,4 & 0,3 \\ 0,4 & 0,7 & 0,5 \end{bmatrix}$$

Deste modo a convolução $\mathbf{Y} = \mathbf{X}_{5\times5} * \mathbf{W}^{r}_{3\times3}$ com $s = (2,2)$ resultará nos valores apresentados na Figura (6).

Figura 6 – Cálculo da operação de convolução.



Fonte: Autor (2023).

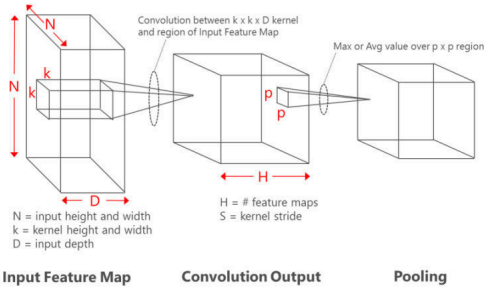
A convolução resulta na matriz $\mathbf{Y}_{2,2}$ a seguir:

$$\mathbf{Y}_{2,2} = \begin{bmatrix} 4,6 & 1,6 \\ 7,5 & 2,9 \end{bmatrix}$$

De acordo com Hijazi *et al.* (2015, p. 4) a operação de convolução extrai diferentes características, a primeira camada extrai dados de baixo nível como arestas, linhas e cantos. A Figura (7) ilustra o processo de convolução 3D (tridimensional) onde a dimensão da entrada é $N \times N \times D$ é convoluída por H *kernels* independentes de dimensão $k \times k \times D$ produzindo H saídas. Iniciado o processo no canto superior esquerdo, o *kernel* se desloca um elemento por vez até o canto superior direito e então movido um elemento para baixo, o processo se repete até que o *kernel* alcance o canto inferior direito da entrada. Para cada elemento $N \times N \times D$ da entrada e $k \times k \times D$ do *kernel* ocorre a multiplicação e acumulação desses elementos necessitando de operações *multiply-and-accumulate*.

As camadas de *pooling* são encarregadas de reduzir a resolução das *features* tornando-as mais robustas contra ruído e distorção. Existem duas maneiras de executar o *pooling*, através

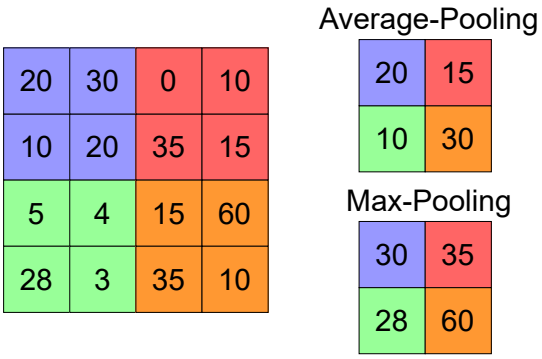
Figura 7 – Processo de Convolução 3D.



Fonte: Ovtcharov *et al.* (2015).

do *max pooling* ou do *average pooling*, em que ambos os casos a entrada é dividida em espaços bi-dimensionais não sobrepostos. A Figura (8) exemplifica a ação de *pooling* em uma matriz de entrada 4×4 com uma subamostragem de dimensão 2×2, no *max pooling* o maior valor da sub-matriz 2×2 é tomado e para o *average pooling* utiliza-se a média entre os elementos da sub-matriz.

Figura 8 – Representação do *max pooling* e do *average pooling*.



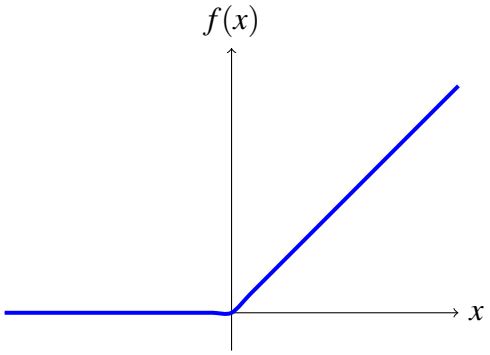
Fonte: Autor (2023)

A ReLU implementa a função $f(x) = \max(0,x)$ com o objetivo de aumentar a não-linearidade da rede e da função de decisão sem que os campos receptivos da camada de convolução sejam afetados pois os dados utilizados para o aprendizado da rede são, na maioria das vezes, não-lineares. Quando comparada com outras funções não-lineares utilizadas em CNNs, como a tangente hiperbólica, sigmoide, entre outras, a ReLU possibilita que a rede realize o treinamento de forma mais eficiente dado que a mesma evita o problema do desvanecimento do gradiente, isto é, a diminuição exponencial ao passar pelas camadas dado que ela mantém o gradiente constante para entradas positivas. A Figura (9) mostra a função ReLU.

Para exemplificar a atuação da ReLU, considere a Figura (10) onde mostra uma matriz de dados, que para $x < 0$ o valor zero é atribuído e para $x \geq 0$ o valor é mantido.

A camada totalmente conectada (*fully connected layer*) é normalmente utilizada

Figura 9 – Gráfico da função ReLU.



Fonte: Autor (2023).

Figura 10 – Aplicação da função não-linear ReLU.

-1	4	5	-10
11	-2	7	3
4	7	-15	0
28	3	75	-25

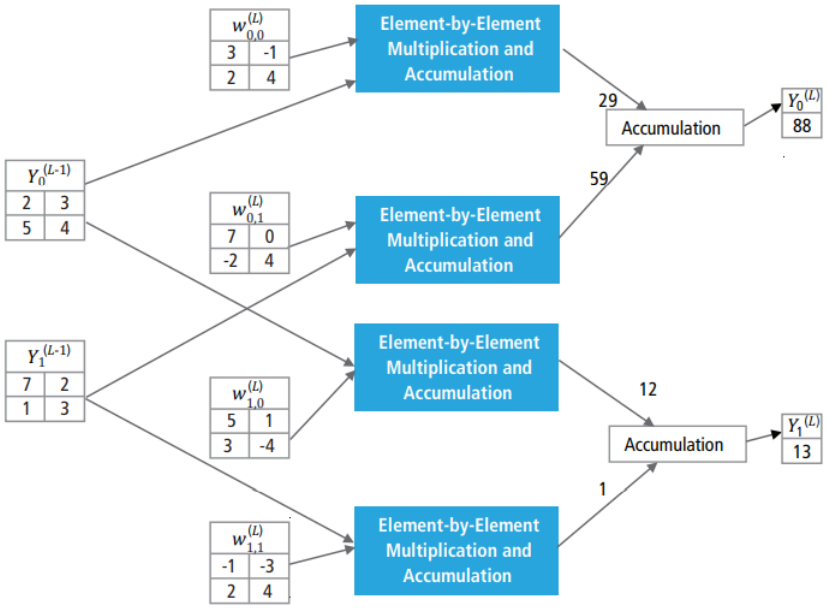
ReLU →

0	4	5	0
11	0	7	3
4	7	0	0
28	3	75	0

Fonte: Autor (2023).

como camada final de uma CNN, semelhante a uma RNA tradicional, cada elemento de uma camada está totalmente conectado aos elementos da camada anterior e posterior. Nesta camada todos os recursos dos elementos da camada anterior são utilizados para calcular cada recurso dos elementos de saída tal qual é mostrado na Figura (11).

Figura 11 – Processamento de uma camada totalmente conectada.



Fonte: Hijazi *et al.* (2015).

Nota-se que em uma camada totalmente conectada há um uso considerável das operações de multiplicação e acumulação, que dependendo da quantidade de elementos contidos nesta camada, torna o processamento da mesma consideravelmente demorado.

2.1.5 MobileNets

Os modelos pré-treinados e o *transfer learning* são técnicas de aprendizado de máquina que permitem aproveitar de modelos treinados em grandes conjuntos de dados para resolver tarefas relacionadas com menos dados e recursos. Como exemplo destes modelos pré-treinados pode-se citar o ImageNet (Deng *et al.*, 2009) e dentre estes está o MobileNet.

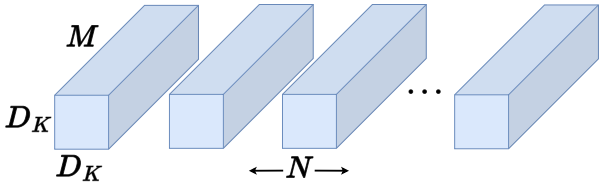
MobileNets é uma família de arquiteturas com o objetivo de prover uma boa exatidão com o menor número de parâmetros e operações aritméticas para dispositivos móveis e visão embarcada. O conceito principal por trás desta família de arquiteturas são as *depthwise separable convolution* (convolução separável em profundidade), uma variante das convoluções 2D clássicas que funcionam de forma mais eficiente sem que haja redução significativa na precisão.

Segundo Howard *et al.* (2017) o modelo MobileNet é baseado em convoluções separáveis profundas, uma forma de convolução fatorada que fatoriza a convolução padrão em uma convolução em profundidade e uma convolução 1×1 chamada de convolução pontual. A convolução comum filtra e combina as entradas em novas saídas em uma única etapa já a *depthwise separable convolution* as divide em duas operações, uma camada separada para filtragem e uma para combinação. Esta fatoração tem o efeito de reduzir consideravelmente o processamento necessário e o tamanho do modelo. A Figura (12) mostra como a convolução padrão é fatorada em *depthwise convolution* e *pointwise convolution*.

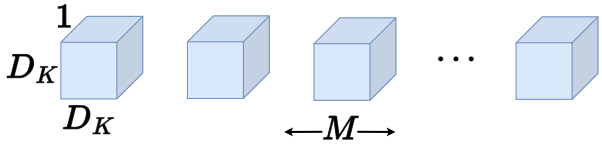
A convolução padrão é parametrizada pelo *kernel* de tamanho $D_K \times D_K \times M \times N$, onde D_K é a dimensão espacial do *kernel*, M é o número de canais de entrada e N é número de canais de saída. A camada de convolução *depthwise* aplica um único filtro para cada canal de entrada e então a *pointwise convolution* calcula uma combinação linear da saída da camada *depthwise* gerando novos recursos, as arquiteturas MobileNets utilizam as funções não lineares *batchnorm* e ReLU em ambas as camadas.

Todas as camadas são seguidas por um *batchnorm* e uma função não-linear ReLU com exceção da camada totalmente conectada final que não possui não-linearidade e alimenta um camada de classificação *softmax*. MobileNet utiliza 3×3 *depthwise separable convolutions*, a Figura (13) contrasta-a com a convolução padrão.

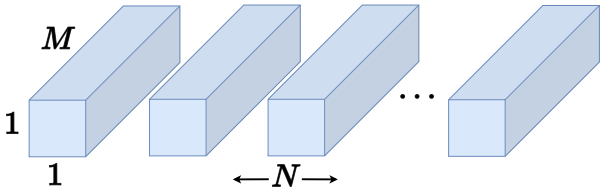
Figura 12 – Convolução padrão (a) substituída por pela *depthwise convolution* (b) e *pointwise convolution* (c).



(a) Filtros de convolução padrão.



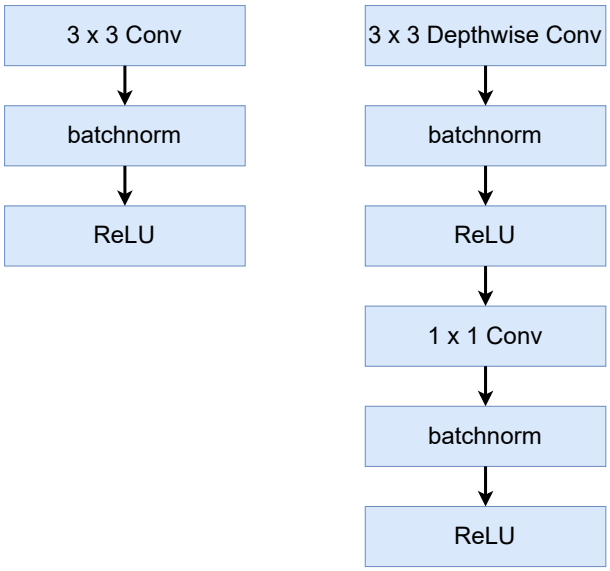
(b) Filtros convolucionais *depthwise*.



(c) Filtros convolucionais 1×1 chamados de *pointwise convolution*.

Fonte: Adaptado de Howard *et al.* (2017).

Figura 13 – Fluxo de operações para convolução padrão e para *depthwise separable convolutions*, respectivamente.



Fonte: Adaptado de Howard *et al.* (2017).

A MobileNet v1 (primeira versão) contém ao todo 31 camadas, sendo 14 delas camadas de *depthwise separable convolution* e 14 convoluções convencionais, um *average pooling* reduz a resolução espacial para um antes de camada totalmente conectada que segue para uma camada de decisão (*softmax*).

2.2 Tiny Machine Learning

A seguir, nas subseções, discute-se acerca das ferramentas utilizadas para a inferência de um modelo de ML, para o *co-design hardware/software* por meio do CFU Playground e, por fim, do modelo utilizado, o qual está disponível no repositório do TFLM no GitHub[1].

2.2.1 TensorFlow Lite Micro

Um dos desafios mencionado anteriormente foi a falta de um *framework* padrão para o TinyML que implica, mais especificamente, no requerimento de otimizações manuais para cada tipo de *hardware*, o que dificulta a sua portabilidade para uma quantidade maior de plataformas. Mais um desafio limitante proveniente da falta de uma ferramenta padrão é que os desenvolvedores de *hardware* possuem necessidades semelhantes, porém não idênticas, o que torna difícil avaliar o desempenho dos *hardwares* de maneira isolada, impossibilitando determinar se há necessidade de melhorias no *hardware* ou no *software*.

O *framework* deve oferecer meios para treinar os modelos em plataformas com poder de processamento maior para depois importá-los em microcontroladores. O TFLM é um *framework* de inferência de código aberto para executar modelos de ML em sistemas embarcados. O ambiente de execução principal cabe em 16 KB em um ARM Cortex M3 e pode executar muitos modelos básicos. Ele não requer suporte a sistemas operacionais, bibliotecas C ou C++ padrão nem a alocação de memória dinâmica (TensorFlow, 2021).

O TFLM é escrito na linguagem C++ 11 e requer que a plataforma seja de 32 *bits*, testado em processadores baseado na arquitetura ARM Cortex-M Series e portado para outras arquiteturas tal como o ESP32. As plataformas de desenvolvimento que são compatíveis com o TFLM estão listadas na Tabela (1).

Existe um fluxo de trabalho específico para executar um modelo do TensorFlow em um microcontrolador, como segue:

- Treinamento do modelo
  - Gerar um modelo pequeno do TensorFlow que se adéque ao dispositivo de destino e contenha as operações compatíveis;
  - Converter em um modelo do TensorFlow Lite;
  - Converter o modelo que está no formato de arquivo FlatBuffer em uma matriz de

[1] TensorFlow Lite Micro: https://github.com/tensorflow/tflite-micro

Tabela 1 – Listagem das plataformas compatíveis com o TFLM.

Placa	**Fabricante**
Arduino Nano 33 BLE Sense	Arduino
SparkFun Edge	SparkFun Electronics
STM32F746 Discovery kit	STMicroelectronics
Adafruit EdgeBadge	Adafruit Industries
Adafruit TensorFlow Lite for Microcontrollers Kit	Adafruit Industries
Adafruit Circuit Playground Bluefruit	Adafruit Industries
Espressif ESP32-DevKitC	Espressif Systems
Espressif ESP-EYE	Espressif Systems
Wio Terminal: ATSAMD51	Seeed Studio
Himax WE-I Plus EVB Endpoint AI Development Board	Himax Technologies, Inc.
Coral Dev Board Micro	Google LLC
Renesas Boards	Renesas Electronics Corporation
Silicon Labs Dev Kits	Silicon Laboratories Inc.
Texas Instruments Dev Boards	Texas Instruments Incorporated
Synopsys DesignWare ARC EM Software Development Platform	Synopsys, Inc.
Sony Spresense	Sony Corporation

Fonte: Autor (2023).

bytes C usando a ferramenta `XXD`, que é padrão para sistemas baseados em Unix, para armazená-lo em uma memória de programa somente para leitura do dispositivo.

- Inferência
  - Execute o modelo utilizando a biblioteca C++ do TFLM.

Um arquivo no formato FlatBuffer, ao qual foi mencionado anteriormente, é um binário contento objetos aninhados. São organizados usando deslocamentos para que os dados possam ser percorridos semelhante a estruturas de dados baseadas em ponteiros. Apenas um grupo restrito de operações do TensorFlow estão presentes no TFLM. A título de informação, existem outras ferramentas voltadas para aplicação de algoritmos de ML em microcontroladores, a Tabela (2) faz uma comparação entre os diferentes *frameworks*.

2.2.2 CFU Playground

A necessidade por um processamento eficiente de redes neurais artificiais em dispositivos com recursos limitados deu origem aos aceleradores de *hardware*. O aumento na utilização de *hardware* especializado evidenciou a necessidade de fluxos de *design* mais ágeis para o *co-desing* de *hardware* e *software*. O CFU Playground é um *framework full-stack* de código aberto que permite um rápido *design* de aceleradores de ML para sistemas embarcados no qual fornece um fluxo *end-to-end* para o *co-design* em FPGA.

Dada a necessidade de reduzir o consumo energético enquanto executa a inferência

Tabela 2 – Comparativo entre as diferentes ferramentas voltadas ao TinyML.

Ferramenta	Algoritmos	Plataformas Compatíveis	Linguagem	Bibliotecas Externas	Disponibilidade
TFLM	Redes Neurais	ARM Cortex-M RISC-V Xtensa	C++ 11	TensorFlow	Código Aberto
STM Cube IA	Redes Neurais	STM32	C	Keras TensorFlow Lite Caffe ConvNetJs Lasangne	Dispositivos STM32
ELL	Redes Neurais	ARM Cortex-M ARM Cortex-A	C/C++	CNTK ARMDarknet ONNX	Código Aberto
ARM-NN AI	Redes Neurais	ARM Cortex-A ARM Mali ARM Ethos	C	TensorFlow Caffe ONNX	Código Aberto
AIfES	Redes Neurais	Raspberry Pi Windows (DLL) ARM Cortex-M4	C	TensorFlow Keras	Licença Privada
MicroMLGen	SVM RVM	Arduino ESP32 Arduino ESP8266	C	Scikit-learn	Licença Privada
m2cgen	Regressão Linear Regressão Logística Redes Neurais SVM Árvore de Decisão	Múltiplas plataformas restritas e não restringidas	Python C C# Java	Scikit-learn	Licença Privada

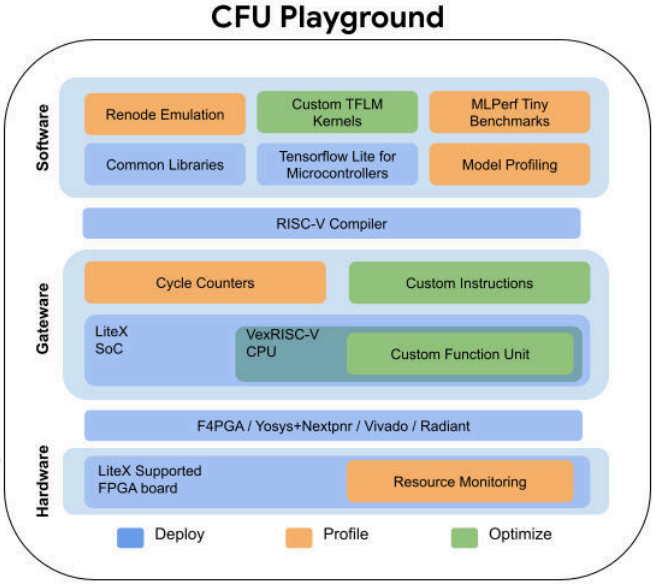
Fonte: Adaptado de Osman *et al.* (2021).

de um modelo de ML em plataformas embarcadas, aceleradores de *hardware* podem apresentar uma solução viável. Como o campo da inteligência artificial aplicada a sistemas embarcados está em fase inicial e em constante mudança, é desejável que se evitem soluções com valores elevados como construir ASICs, por exemplo, que possuem um alto custo e tempo de fabricação. Tendo em vista que tais sistemas são desenvolvidos para uma finalidade específica, pode-se evitar de construir aceleradores de uso geral e centrar esforços na implementação de aceleradores específicos para cada tipo de aplicação e operação.

O CFU Playground é um *framework* interativo para implementação, quantização de performance e otimização. Segundo Prakash *et al.* (2023) o CFU Playground combina ferramentas de código aberto de *software* como TFLM e GCC, para geração de SoCs como LiteX, VexRiscv, Migen e Amaranth, e ferramentas para síntese tais como yosys, nextpnr e F4PGA/SymbiFlow. Por utilizar um fluxo de trabalho que contém apenas ferramentas *open-source* evita-se problemas com restrições de licenciamento, as ferramentas que estão presentes (ou podem ser utilizadas em conjunto) no CFU Playground são apresentadas na Figura (14).

Uma CFU, ou Unidade de Função Customizada em português, é uma pequena parte de *hardware* adicionado a CPU para acelerar uma função discreta determinada pelo desenvolvedor. A CFU pode otimizar a execução de um modelo de ML de muitas formas, tais como: especializar operações em que os operandos são constantes; performar várias operações distintas ao mesmo tempo por meio do paralelismo; executar uma fusão de operações dentro de um ciclo e segmentar várias operações através de vários ciclos.

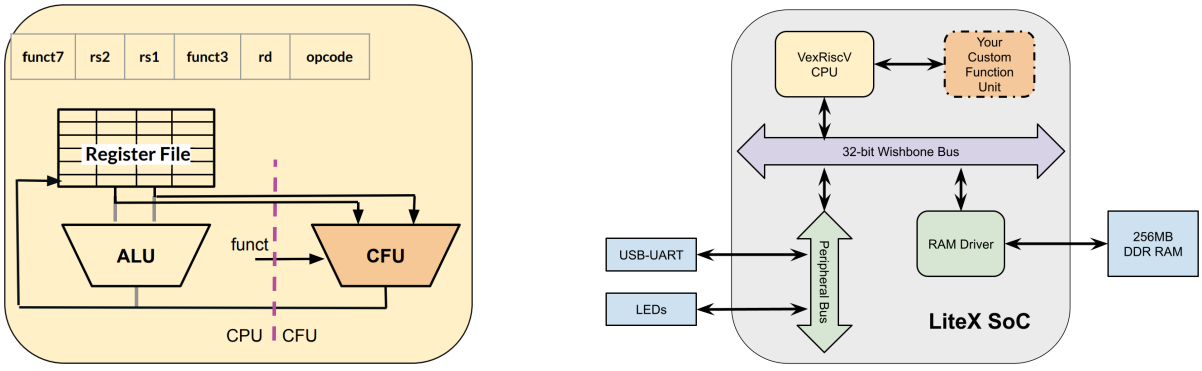
Figura 14 – Ferramentas contidas no CFU Playground.



Fonte: Prakash *et al.* (2023).

O armazenamento flexível e configurável também permite que os dados sejam armazenados e reutilizados localmente evitando a transferência desnecessária de dados. A CFU segue o *R-format* da arquitetura RISC-V, isto é, recebe dois operandos do arquivo de registradores e retorna um resultado. A comunicação entre CFU e CPU é estritamente lógica, a Figura (15) mostra como a CFU está disposta perante a CPU e no SoC.

Figura 15 – Representação da CFU diante do restante da CPU (a) e CFU incorporada em um SoC utilizando FPGA por meio do LiteX (b).



(a) *Custom Function Unit*. (b) LiteX *System-on-Chip*.

Fonte: Prakash *et al.* (2023).

Como apresentado na Figura (15b), o CFU Playground incorpora tanto CPU quanto CFU em um SoC utilizando FPGA, ele utiliza o *framework* LiteX que fornece uma infraestrutura eficiente destinada a síntese de CPUs *softcore* e SoCs em FPGAs para construir o *gateware*. De acordo com Kermarrec *et al.* (2020) o LiteX é um construtor SoC e biblioteca de componentes

IP descritos no nível de *Register-Transfer Level* (RTL), juntamente com vários utilitários que facilitam a construção de projetos SoC.

Para utilizar-se uma placa no CFU Playgorund primeiramente ela deve estar descrita no LiteX, as placas comerciais mais populares possuem descrição na biblioteca de placas do mesmo e a CPU *softcore* utilizada, e a única suportada no CFU Playground, é a VexRiscv que é altamente configurável permitindo adicionar ou remover diferentes recursos para desempenho e funcionalidade, como exemplos, tem-se: estágios de *pipelining*, memórias cache e unidades de ponto flutuante.

O *gateware* para o CFU Playground é adaptado para uma ampla quantidade de FPGAs, o mesmo pode ser alocado em uma placa tão pequena quanto a Kosagi Fomu que cabe em uma porta USB. Entretanto, à medida que mais recursos de *hardware* se encontram disponíveis, *soft* CPUs e CFUs mais robustas podem ser implementadas, com uma maior capacidade de memória pode-se executar modelos maiores. Um resumo das placas suportadas atualmente pelo *framework* é apresentado na Tabela (3):

Tabela 3 – Placas FPGA testadas e suportadas pelo CFU Playground.

Placa	**LUTs**	**DSPs**	**RAM**	**ROM**	**Freq. de *Clock***
Arty A7-100T	101.440	240	256 MB	16 MB	100 MHz
Arty A7-35T	33.280	90	256 MB	16 MB	100 MHz
OrangeCrab	24.000	28	128 MB	16 MB	75 MHz
ULX3S (12F)	12.000	12	32 MB	4 MB	50 MHz
iCEBreaker	5280	8	128 kB	16 MB	24 MHz
Kosagi Fomu	5280	8	128 kB	2 MB	12 MHz

Fonte: Prakash *et al.* (2023).

Os recursos mínimos que uma placa e seu FPGA precisa ter é: a capacidade de criar uma conexão *Universal Asynchronous Receiver/Transmitter* (UART) para interagir com o *software* na placa, o FPGA deve ter recursos suficientes para sintetizar uma das versões da CPU VexRiscv, deve conter *Random Access Memory* (RAM) suficiente para fornecer memória de trabalho para o *software* e deve haver RAM ou *Read-Only Memory* (ROM) capaz de armazenar código e dados constantes como o modelo do TFLM.

Para utilizar o *hardware* da CFU, novas instruções devem ser criadas e adicionadas ao conjunto de instruções da CPU. Pode-se utilizar o RISC-V GCC *toolchain* juntamente com uma macro que é fornecida e que expande-se em diretivas “asm” para gerar essas novas instruções. A macro requer quatro argumentos e retorna um resultado:

```
acc = cfu_op(funct7, funct3, a, b)
```

Onde “`funct7`” e “`funct3`” são campos de 7 *bits* e 3 *bits*, respectivamente, que são reservados para especificar o *opcode* da instrução. “`a`” e “`b`” são variáveis C/C++ inteiras de 32 *bits* utilizadas como operandos da instrução que por sua vez retorna uma saída de 32 *bits*.

2.2.3 Otimização: Co-design Hardware/Software

As CNNs são largamente aplicadas em tarefas de processamento de imagens combinando filtros com redes neurais assemelhando-se aos campos receptivos do cortex visual. Redes neurais convolucionais em geral requerem um alto poder computacional devido as suas várias camadas de neurônios que, por sua vez, realizam muitas multiplicações com variáveis do tipo ponto flutuante.

Pode-se focar a otimização de uma CNN tanto em algoritmos do *software* ou no *hardware* da plataforma de execução. Deste modo, é desejável tirar proveito do paralelismo para acelerar a execução porém é raro que um algoritmo seja totalmente executado de forma paralela sem um controle sequencial do fluxo e isto causa um efeito direto na velocidade de execução alcançada. De maneira geral, instruções paralelas são mais adequadas para execuções de *hardware* enquanto instruções sequenciais são adequadas para execução de *software*.

Segundo Chen *et al.* (2018) *hardware/software co-design* oferece equilíbrio entre velocidade e flexibilidade, combinando a execução tradicional de *software* em CPUs ou GPUs com aceleradores de *hardware* customizados em FPGAs, DSP ou ASICs. Tendo o CFU Playground como ferramenta para implementar o acelerador, contando que o mesmo e suas dependências estejam devidamente instaladas[2], um novo projeto pode ser iniciado copiando o *template* padrão fornecido pelo *framework* com:

```
$ cp -r proj/proj_template proj/accel_name
$ cd proj/accel_name
```

No diretório do projeto criado é possível adicionar outros modelos de ML modificando o arquivo `Makefile`, por padrão o modelo de detecção de pessoas quantizado em 8 *bits* é o único incluído. O passo seguinte é executar o modelo, é necessário compilar todo o *kernel* do TFLM e gerar os arquivos binários que serão carregados no FPGA, para a placa Kosagi Fomu esta etapa pode ser executada com a seguinte linha de comando:

```
$ make TARGET=kosagi_fomu TTY=\dev\ttyACM0 prog load
```

[2] O CFU Playground pode ser instalado apenas nas distribuições Linux: Debian e Ubuntu.

Uma versão customizada da *soft* CPU VexRiscv para atender os requisitos mínimos com os recursos disponibilizados pelo FPGA será sintetizada juntamente com a CFU. A inferência do modelo é feita através de um menu interativo que após concluída imprime na tela as seguintes informações: "`Event`", "`Tag`" e "`Ticks`" que são a numeração de cada camada, identificação nominal da camada e quantos ciclos de máquina foram necessários para que a execução da camada fosse concluída, respectivamente. Cada "ticks" que é contado equivale a 1024 ciclos de *clock*.

A operação de convolução demanda muita capacidade de processamento tornando-a alvo para otimizações, pode-se obter mais informações sobre a performance de execução da convolução copiando o arquivo fonte da mesma na pasta de projeto e incluindo os contadores de performance com a biblioteca `"perf.h"`. O versão da CPU VexRiscv sintetizada no Kosagi Fomu não tem suporte para a biblioteca de contadores de performance, porém, é possível obter a contagem de ciclos por meio da função `perf_get_mcycle()` que retorna um valor de um contador de 32 *bits*.

Até agora foram mencionados meios de como quantificar a performance da execução do modelo de ML através de contagem de ciclos, para aplicar as otimizações de *software* à operação da convolução necessita-se primeiramente conhecer alguns parâmetros da mesma, para isso é necessário inserir a biblioteca `"playground_util/print_params.h"`. Após uma nova inferência, pode-se observar que os parâmetros apresentados na Tabela (4) são contantes.

Tabela 4 – Parâmetros constantes da convolução.

Parâmetro	**Valor**
`stride_width`	1
`stride_height`	1
`dilation_width_factor`	1
`dilation_height_factor`	1
`filter_height`	1
`filter_width`	1
`pad_width`	0
`pad_height`	0
`input_offset`	128

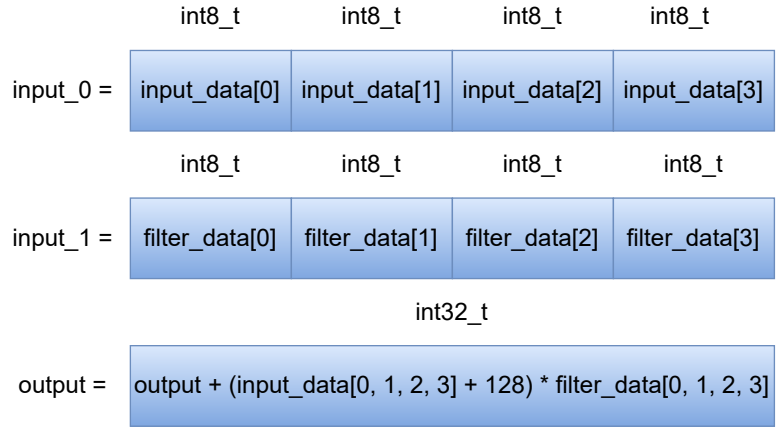
Fonte: Callahan *et al.* (2020).

Pode-se substituir a função que adquire esses parâmetros no código fonte pelos seus valores literais, concluí-se a primeira parte da otimização do *software*. O laço de repetição mais interno da operação de convolução tem como variável de controle `input_filter_depth`, sabendo que a mesma é sempre um múltiplo de quatro pode-se fazer com que o *loop* mais interno

realize quatro operações antes de testar a sua condição, esta otimização é conhecida como *loop unrolling*.

Implementadas as técnicas de otimização de *software* pertinentes segue-se para a descrição do *hardware* adicional (CFU). Nota-se que o *loop* mais interno realiza exclusivamente as operações de soma, multiplicação e acumulação. Percebe-se que no laço de repetição é carregado, multiplicado e acumulado 8 valores do tipo `int8_t` que implica em uma utilização pouco proveitosa dos recursos dado que os registradores possuem 32 *bits* de largura. Tem-se como objetivo criar uma CFU que performe uma operação *Single Instruction, Multiple Data* (SIMD) de multiplicação e acumulação. De acordo com Santana (2020) uma instrução SIMD refere-se a um tipo de instrução que agrupa uma certa quantidade de dados e realiza operações com os mesmos de forma paralela. O perfil das variáveis é apresentado na Figura (16).

Figura 16 – Perfil dos parâmetros de entrada e saída da CFU.



Fonte: Callahan *et al.* (2020).

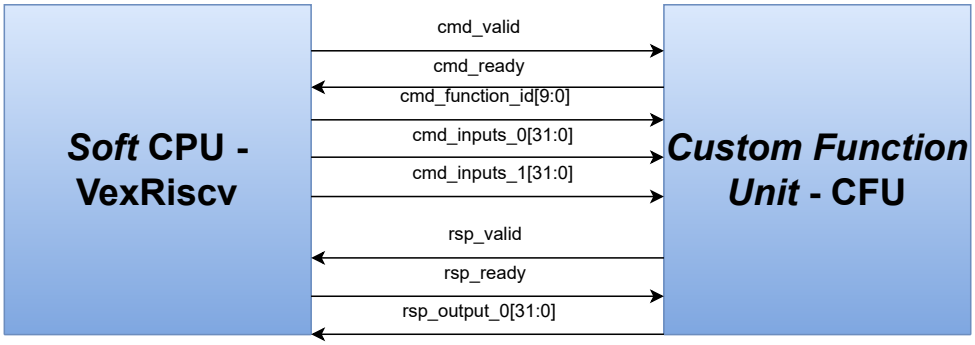
A CFU pode ser implementadas nas seguintes *Hardware Description Language* (HDL): Verilog, SystemVerilog e Amaranth. A quando descrita em Verilog ou SystemVerilog necessita que o *handshaking* com a *soft* CPU, isto é, a comunicação através de sinais de controle seja implementada manualmente. Quando utiliza-se a Amaranth — que é incorporada na linguagem Python — para criar a CFU não é necessário implementar a comunicação entre as partes pois já está feita na classe base. O barramento de comunicação entre CPU e possui dois diferentes fluxos, são eles:

- A CPU usa o fluxo de comando (`cmd`) para enviar os operandos e 10 *bits* de *opcode*, iniciando a operação da CFU;
- A CFU usa o fluxo de resposta (`rsp`) para retornar o resultado para a CPU.

Cada fluxo conta com *handshaking* bidirecional indicados por "`*_valid`" e "`*_ready`".

Pode-se indicar que não é possível realizar uma nova transferência de dados colocando seu sinal "`ready`" em nível lógico baixo, o envio dos dados apenas ocorre quando o sinal de "`valid`" da unidade que está enviando e de "`ready`" da unidade receptora estão ambos em nível lógico alto. A Figura (17) detalha os sinais de controle e o barramento de dados entre CPU e CFU.

Figura 17 – Sinais de controle e barramento de dados entre e CPU.



Fonte: Adaptado de Callahan *et al.* (2020).

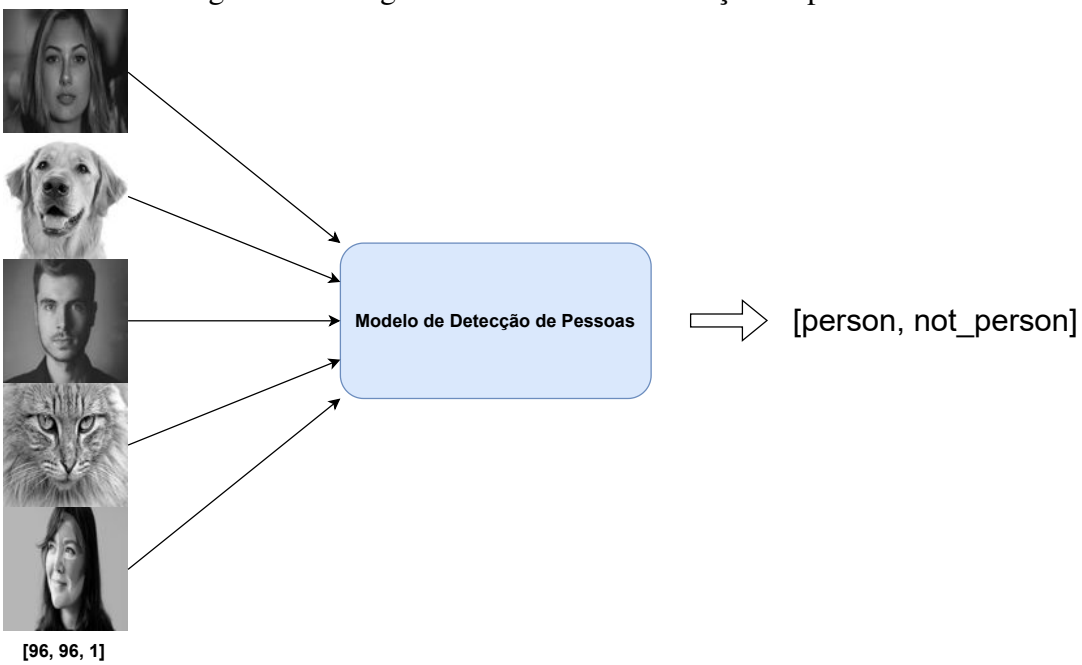
2.2.4 Modelo de Detecção de Pessoas

Até muito recentemente a capacidade de "ver" não estava disponível para as máquinas, a maioria dos robôs apenas vasculhavam áreas através de sensores de proximidade. Um ser humano pode facilmente identificar e classificar um objeto qualquer sem estar em contato com ele, capacidade esta que as máquinas não tinham até pouco tempo atrás pois as informações visuais eram muito confusas, desestruturadas e de difícil interpretação.

Com a evolução das CNNs foi possível filtrar uma imagem extremamente complexa em um mapa de padrões e formas conhecidas o que permitiu a criação de algoritmos inteligentes capazes de identificar diferentes padrões em imagens digitais. Nesta Seção será abordada a construção de uma aplicação embarcada para classificar imagens capturadas por uma câmera, o modelo é treinado para identificar se uma pessoa está ou não presente na imagem que foi enviada por uma câmera, por exemplo, para assim gerar um sinal de controle.

De acordo com Warden e Situnayake (2019, p. 224) o modelo de detecção de pessoas é uma rede neural convolucional treinada utilizando o conjunto de dados *Visual Wake Words dataset* que contém 115 000 imagens classificadas. O modelo tem o tamanho de 250 kB, aceita como entrada imagens com dimensão de 96×96 *pixels* em *grayscale* onde cada imagem é fornecida como um tensor 3D com dimensão $96 \times 96 \times 1$, sendo que a última dimensão é um valor de 8 *bits* que representa um único *pixel* de forma que 0 equivale a um *pixel* totalmente

Figura 18 – Diagrama do modelo de detecção de pessoas.



Fonte: Autor (2023).

preto e 255, um totalmente branco. A Figura (18) apresenta uma representação do modelo.

Segundo Jain *et al.* (2021) o *Visual Wake Words dataset* é projetado para ser útil para *benchmarking* e teste de visão computacional embarcada uma vez que representa uma tarefa simples, a classificação binária que precisa realizar-se com restrições de recursos. O *download* do *dataset* em questão requer um espaço de memória na ordem de $\sim$40 GB.

O modelo é treinado utilizando MobileNets que é uma arquitetura para CNNs focada em visão móvel projetada para oferecer uma boa acurácia com o menor número possível de parâmetros de pesos e operações aritméticas, existem várias versões desta arquitetura, porém, a versão utilizada neste modelo é a v1 que requer menor quantidade de memória RAM em tempo de execução. Após o treinamento da rede, inicia-se o processo de quantização que, resumidamente, é converter valores 32 *bits* de ponto flutuante para 8 *bits* inteiro, por exemplo, sem que haja perda significativa na precisão do modelo.

De maneira geral, uma rede neural é muito sensível à precisão numérica, erros podem se acumular e culminar em um resultado impreciso. Os modelos de aprendizado profundo são diferentes — eles são capazes de lidar com grandes perdas na precisão numérica durante os cálculos intermediários e ainda produzir resultados finais que são precisos em geral (Warden; Situnayake, 2019, p. 404). Esta robustez provem do processo de treinamento na qual as entradas são muito grandes e ruidosas de modo que os modelos são debilmente afetados por pequenas variações e focam no aprendizado dos padrões que são importantes.

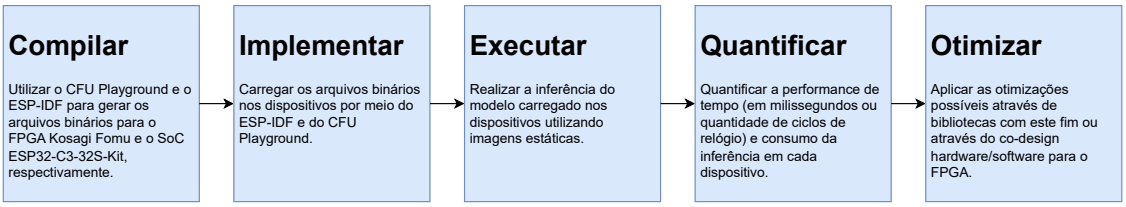
A maioria das inferências realizadas por aplicações consegue produzir resultados indistinguíveis do equivalente de ponto flutuante, utilizando apenas 8 *bits* para armazenar pesos e valores de ativação. A quantização é benéfica para os dispositivos embarcados tendo em vista que estes possuem amplo suporte para instruções *multiply-and-accumulate* da qual os modelos dependem.

3 MATERIAIS E MÉTODOS

Após uma revisão da literatura acerca do tema utiliza-se um modelo de ML para averiguar a evolução das otimizações implementadas por *software* e através do *co-design hardware/software*. O modelo de ML selecionado é desenvolvido para identificar se existe ou não pessoas em uma imagem estática e está disponível no repositório do TensorFlow Lite Micro no GitHub e está disponível em ambas as ferramentas.

Para realizar a inferência do modelo faz-se uso de imagens estáticas, imagens no formato bitmap ou outro que foram convertidas em um *array* de *bytes* da linguagem C para ser interpretada pelo processador. O processo utilizado para implementar as otimizações de performance é apresentado na Figura (19).

Figura 19 – Processo sistemático para otimizar a performance.



Fonte: Autor (2023).

De maneira recorrente, aplicam-se as etapas apresentadas na Figura (19) onde o primeiro passo é compilar em ambas as plataformas para então corrigir possíveis erros e para aplicar as otimizações; implementar, ou seja, carregar os arquivos binários em ambas as placas utilizando o respectivo *framework* preparando para inferência; executar a inferência fornecendo imagens estáticas (ou *arrays*) ao modelo; quantificar a performance e identificar as otimizações necessárias e implementar estas otimizações no *hardware* ou *software*.

Para a placa ESP-C3-32S-Kit, pode-se fornecer a imagem estática através do ESP-IDF com o comando `detect_image <image_number>` no terminal e como saída é fornecido o tempo em milissegundos (ms) que é utilizado para quantificar a performance da inferência. O CFU Playground não evidência, depois de uma inferência, o tempo total gasto ao invés disso é fornecida uma quantidade denominada como “ticks” que equivale a 1024 ciclos de relógio da *soft* CPU, logo, o tempo gasto pode ser encontrado por meio de:

$$t = \frac{1024}{f_{\text{clk}}} \cdot \text{ticks} \tag{3.1}$$

As frequências das CPUs sintetizadas nas diferentes placas suportadas pelo *fra-*

mework estão listadas na Tabela (3). A performance pode ser quantificada por:

$$P_{\%} = \frac{t_i - t_f}{t_i} \cdot 100\% \tag{3.2}$$

Onde:

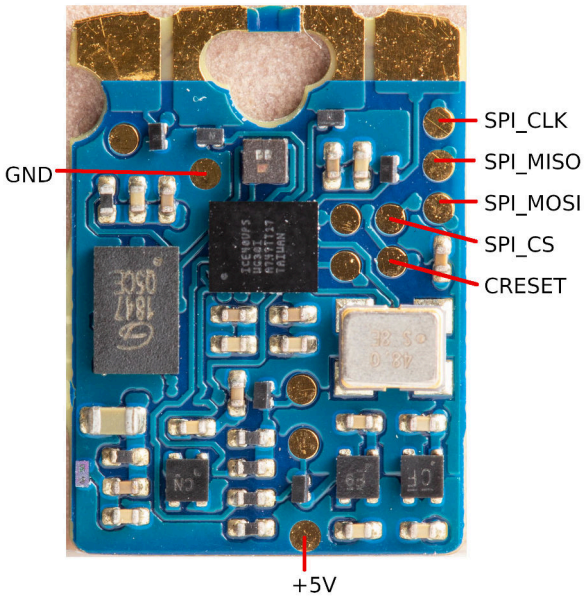
- $P_{\%}$: percentual de otimização;
- t_i: tempo inicial;
- t_f: tempo após a otimização.

Para a obtenção dos dados relacionados ao consumo energético de cada placa faz-se uso de um medidor USB de tensão e corrente cujo os valores são dados em volts e amperes, respectivamente. Nas seções 3.1 e 3.2 aborda-se especificações de *hardware* e as ferramentas utilizadas em conjunto com as placas Kosagi Fomu e ESP-C3-32S-Kit, utilizadas para a implementação e execução do modelo.

3.1 Kosagi Fomu

O Kosagi Fomu, ou apenas Fomu, é uma placa FPGA programável e é comportado em uma porta USB. O Fomu possuí quatro *pads* (superfícies condutoras expostas que são utilizadas para conectar componentes eletrônicos) de cobre que atuam como entradas, um LED RGB para atuar como um elemento de saída e é compatível com uma cadeia de ferramentas de código aberto capaz de executar um *core* RISC-V. A Figura (20) exibe a placa.

Figura 20 – Placa FPGA Kosagi Fomu.



Fonte: Cross *et al.* (2023).

Com 128 kB de memória RAM e uma boa capacidade de armazenamento, é possível executar o Python de forma nativa. Abaixo do interpretador Python está um *softcore* RISC-V rodando no FPGA. Segundo Cross e Ansell (2023) o *firmware* padrão do Fomu expõe um *bootloader* USB executando um *softcore* RISC-V. O *chip* principal do Fomu é o FPGA Lattice ICE40UP5K com aproximadamente 5 000 *look-up table* (LUT), mais especificações acerca da placa podem ser encontrados na Tabela (5).

Tabela 5 – Especificações do Fomu.

Parâmetro	**Especificação**
FPGA	Lattice ICE40UP5K
Velocidade	Oscilador externo de 48 MHz
RAM	128 kB
Armazenamento	1 MB *Serial Peripheral Interface* (SPI) *flash*
Conectividade	USB 2.0 FS (12 Mbps)
Entradas	Quatro *pads* de cobre
Saídas	Um LED RGB

Fonte: Adaptado de Cross e Ansell (2023).

Além dos 128 kB de memória RAM o FPGA dispõe de mais 1024 kB de memória *flash* disponível. Um novo *design* de *hardware* descrito em Verilog ou VHDL pode ser gerado e carregado na placa usando as ferramentas *open-source* Nextpnr[1], IceStorm[2], Yosys[3] e dfu-util[4].

O Nextpnr é uma ferramenta de código aberto para *Place-and-Route* (PnR) que são etapas de decisão de onde colocar os elementos lógicos e circuitos no FPGA e conexão entre estes elementos com o objetivo de otimizar o desempenho, consumo energético e a utilização de recursos. Possui *Graphical User Interface* (GUI) e está disponível para os sistemas operacionais Linux, Windows e macOS. IceStorm é um projeto que visa fornecer ferramentas para analisar e criar arquivos *bitstream* no formato dos FPGAs Lattice iCE40. Segundo Shah *et al.* (2019) o Yosys é um *framework* de código aberto para síntese e verificação Verilog, ele oferece suporte a todos os recursos sintetizáveis comumente utilizados do Verilog-2005 e pode ser direcionado tanto para FPGAs quanto para ASICs. Por fim, o dfu-util é um utilitário que é destinado a fazer *download* e *upload* de *firmware* em dispositivos conectados via USB.

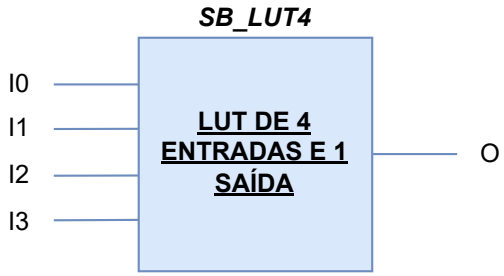
Um FPGA é, mais especificamente, um circuito integrado contendo matrizes de portas que são programáveis, pode-se alterar suas conexões internas apenas carregando novas

[1] Nextpnr: https://github.com/YosysHQ/nextpnr.
[2] IceStorm: https://github.com/YosysHQ/icestorm.
[3] Yosys: https://github.com/YosysHQ/yosys.
[4] dfu-util: https://github.com/Stefan-Schmidt/dfu-util.

configurações. Essas novas configurações são obtidas por meio de tabelas de consulta do programa de configuração — denominada por LUT — que formam os blocos lógicos básicos.

De acordo com Cross e Ansell (2023) as LUTs de um FPGA possuem quase sempre *n*-entradas e uma saída, a família ICE da Lattice contém LUTs de quatro entradas. O bloco básico do FPGA da placa Fomu é a `SB_LUT4` representada na Figura (21).

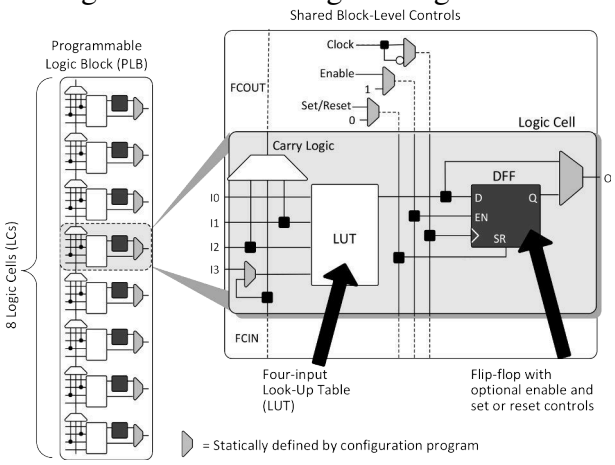
Figura 21 – LUT do Lattice ICE40UP5K.



Fonte: Adaptado de Cross e Ansell (2023).

Os dispositivos FPGA modernos não são compostos apenas por LUTs. Usualmente, as LUTs são agrupadas com registradores *D Flip-Flop* (DFF) e alguma lógica de *carry* nos chamados *Programmable Logic Blocks* (PLB) ou *Configurable Logic Block* (CLB). Blocos de finalidade especifica como RAMs, *Digital Signal Processor* (DSP), transmissores e receptores serial de alta performance, entre outros, estão incluídos nos dispositivos. Os blocos citados são conhecidos como *hard blocks* e permitem o uso eficiente de área e energia. A Figura (22) mostra o diagrama de um Bloco Lógico Programável.

Figura 22 – Bloco Lógico Programável.



Fonte: Cross e Ansell (2023).

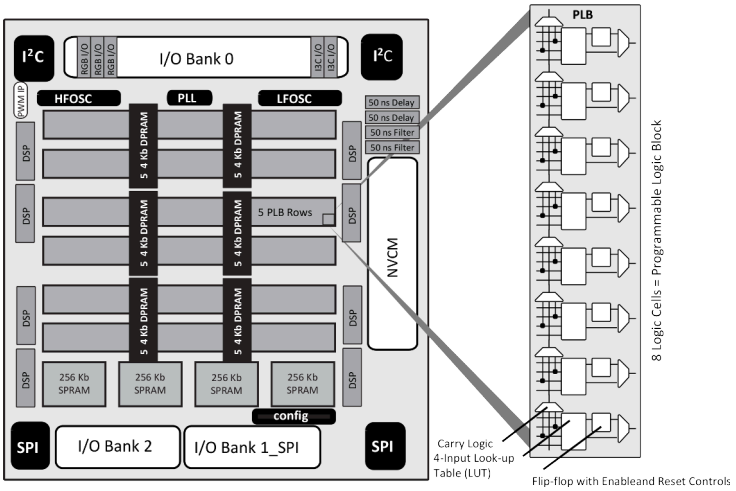
Configurar todos os elementos lógicos presentes em um FPGA manualmente é uma tarefa quase que impossível, para isso foi-se desenvolvida as HDLs que permite ao usuário

descrever seu *hardware* em níveis de abstração mais elevados. As HDLs mais comuns são Verilog e VHDL, no entanto, há uma tendencia para utilização das *Domain-Specific Language* (DSL) que são incorporadas em uma linguagem de programação já conhecida, como exemplos de DSLs, pode-se citar: Migen, nMigen e Amaranth encorporadas na linguagem Python; Clash é encorporada na linguagem Haskell e Chisel e SpinalHDL são incorporadas na linguagem Scala.

Pode-se executar um *hardware* descrito por alguma HDL em um sintetizador e gerar blocos de LUTs e DFFs que formam pilhas que precisam ser agrupadas e isto é feito por uma ferramenta de PnR — no caso do Fomu, o Nextpnr — que atribui PLBs físicos para cada LUT e DFF definido pelo sintetizador e em seguida realiza as conexões entre as partes.

Uma vez finalizado o processo de PnR, é gerado um arquivo abstrato que precisa ser convertido em um formato que o *hardware* possa reconhecer, isto é feito por uma ferramenta de empacotamento de *bitstream* que para FPGAs da família ICE da Lattice é o IceStorm. Os FPGAs são voláteis, i. e., perdem a configuração quando são desconectados. Para evitar que as configuração se percam, as placas FPGAs inclui memórias *flash* que normalmente são programadas por meio de interface SPI. A Figura (23) apresenta um diagrama com a arquitetura do FPGA Lattice iCE40 UltraPlus 5K.

Figura 23 – Arquitetura do Lattice iCE40 UltraPlus 5K.

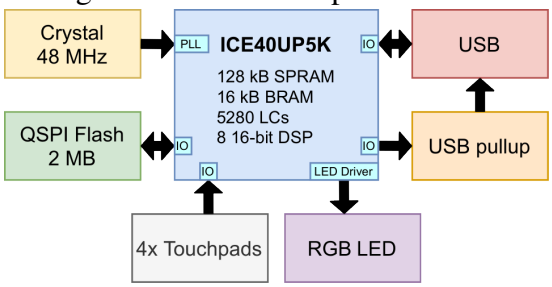


Fonte: Cross e Ansell (2023).

Elencando-se os recursos presentes no FPGA Lattice iCE40 UltraPlus 5K: 5280 LUTs de 4 entradas; 16 kB de *Block* RAM (BRAM); 128 kB de *Static Random-Access Memory* (SRAM) interna; duas interfaces I2C e SPI; 8 unidades DSPs de 16 bits, um *driver* de LED de 3 canais com limitação de corrente e até 1024 kB de memória *flash*. A família iCE40 ainda dispõe de um recurso chamado de *warmboot* que permite ter múltiplos projetos ativos no mesmo FPGA

e alternar entre eles. Um diagrama de blocos da placa Kosagi Fomu que resume os recursos internos do FPGA Lattice ICE40UP5K e demais componentes é apresentado na Figura (24).

Figura 24 – Diagrama de blocos da placa FPGA Kosagi Fomu.



Fonte: Cross e Ansell (2023).

Ressalta-se que o Fomu implementa sua comunicação USB através de um *softcore* o que consome espaço de armazenamento.

3.2 ESP-C3-32S-Kit

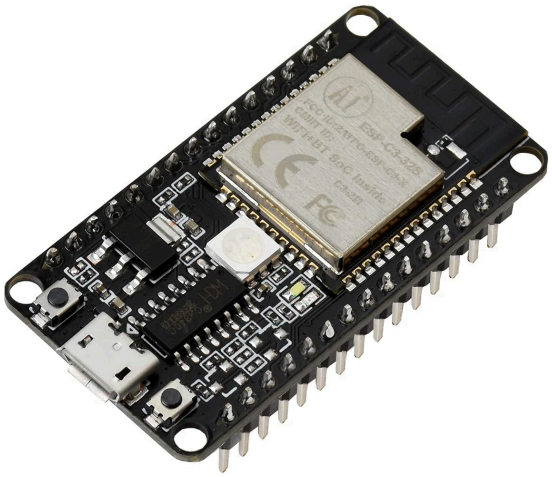
A placa de desenvolvimento ESP-C3-32S-Kit é projetada pela empresa Shenzhen Ai-Thinker Technology Co. e dispõe dos seguintes recursos: suporte para os protocolos Wi-Fi IEEE802.11b/g/n e BLE 5.0; equipada com um processador RISC-V 32 bits *single-core* com uma frequência de operação de 160 MHz; o *chip* possuí uma memória SRAM de 400 kB; 384 kB de memória ROM; RTC SRAM de 8 kB; memória *Flash* embutida de 4 MB e suporte para memória *Flash* externa.

Dentre os periféricos disponíveis na placa, tem-se: UART, PWM, SPI, I2S, I2C, ADC, sensor de temperatura e até 21 GPIOs. O *chip* suporta uma variedade de estados de trabalho de baixo consumo de energia para atender diversos cenários de aplicação. A placa de desenvolvimento é apresentada na Figura (25).

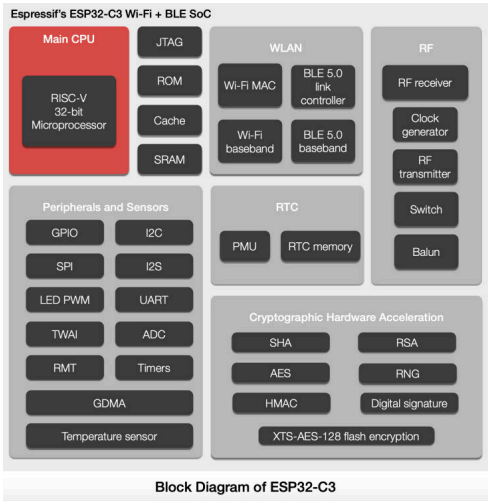
O ESP32-C3 é programado através do *Espressif IoT Development Framework*[5] (ESP-IDF) que é o *framework* de desenvolvimento oficial para as series de SoCs ESP32, ESP32-S, ESP32-C e ESP32-H. Para utilizar o ESP-IDF é necessário que tenha-se: uma *toolchain* para compilar códigos para o ESP32, ferramentas de construção como o Cmake e Ninja para construir a aplicação e o ESP-IDF que é essencialmente uma Interface de Programação de Aplicativos (API) que contém as bibliotecas de *software* e os códigos fonte para o ESP32 e *scripts* para operar a *toolchain*.

[5] ESP-IDF: https://github.com/espressif/esp-idf.

Figura 25 – Placa física do ESP-C3-32S-Kit (a) e suas partes integrantes (b).



(a) ESP-C3-32S-Kit.



(b) Resumo dos componentes.

Fonte: Amazon (2021) (a); Thinker (2021) (b).

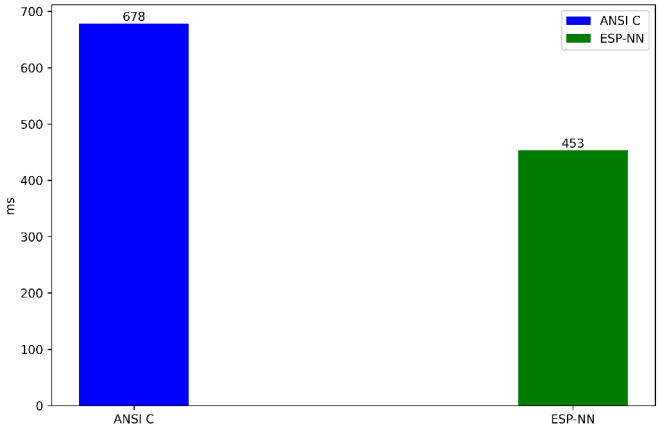
A Espressif também fornece uma biblioteca destinada a otimização do *software* utilizado na execução das operações do TFLM, a ESP-NN. Segundo Dattu *et al.* (2023) o ESP-NN contém implementações de *kernel* otimizadas para os *kernels* usados no TensorFlow Lite Micro. Atualmente os *chips* ESP que possuem suporte são: ESP32-S3 com versões de montagem otimizadas para se beneficiar das instruções vetoriais; ESP32 com otimizações genéricas e o ESP32-C3 também com otimizações genéricas.

4 RESULTADOS

4.1 Inferência da Performance no ESP-C3-32S-Kit

O tempo total gasto em milissegundos para cada inferência é apresentado na Figura (26), vale ressaltar que os resultados apresentados são determinísticos, ou seja, ao efetuar a inferência do modelo mais de uma vez os valores se mantém constantes.

Figura 26 – Tempo da execução do modelo de detecção de pessoas no ESP-C3-32S.



Fonte: Autor (2023).

Pela Equação (3.2) pode-se calcular a evolução da performance para otimizações de *software* implementadas pela biblioteca ESP-NN:

$$P_{\%_{\text{esp32}}} = \frac{678 - 453}{678} \cdot 100\% = 33,19\% \tag{4.1}$$

Logo, tem-se um decréscimo de 33,19% no tempo de execução de uma inferência.

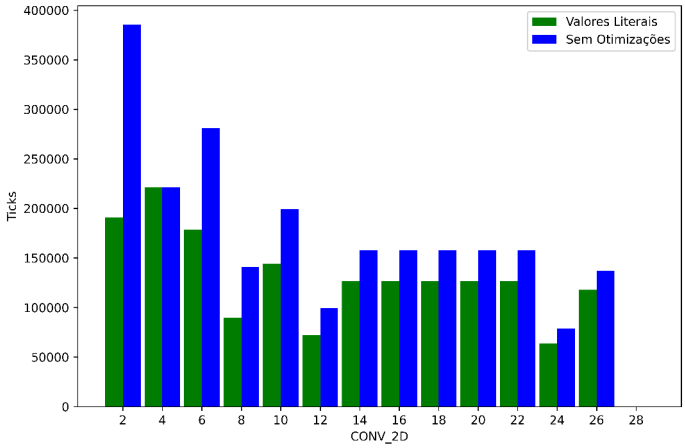
4.2 Inferência da Performance no FPGA Kosagi Fomu

A ferramenta de *co-design* CFU Playground permite que se faça otimizações de *software* e *hardware*. A inferência do modelo no Kosagi Fomu sem qualquer tipo de otimização é apresentada na Tabela (7). Para a inferência realizada sem otimizações foi-se necessário aproximadamente $2,708929536 \times 10^9$ ciclos, onde $2,38754816 \times 10^9$ ciclos são utilizados para realizar a operação de convolução (CONV_2D) que representa cerca de 88,1% dos ciclos totais em uma inferência. Pela Equação (3.1) tem-se que o tempo total é:

$$t = \frac{2,708929536 \times 10^9}{12 \times 10^6} = 225,74\,\text{s} \tag{4.2}$$

Pode-se então substituir as funções que adquirem os valores constantes apresentados na Tabela (4) pelos seus valores literais no código fonte da convolução, a Tabela (8) mostra o resultado da otimização de *software*, tem-se que o tempo de execução reduziu para 172,88 s, a Figura (27) mostra uma comparação entre as camadas de CONV_2D sem otimizações e com a otimização de *software* implementada.

Figura 27 – Comparativo entre a inferência sem otimizações e com valores constantes substituídos pelos valores literais.



Fonte: Autor (2023).

A próxima otimização é denominada de *loop unrolling* ou desenrolamento de laço. O laço de repetição mais interno no código fonte da operação de convolução do TFLM é apresentado na Figura (28).

Figura 28 – Laço de repetição mais interno da operação de convolução do TFLM.

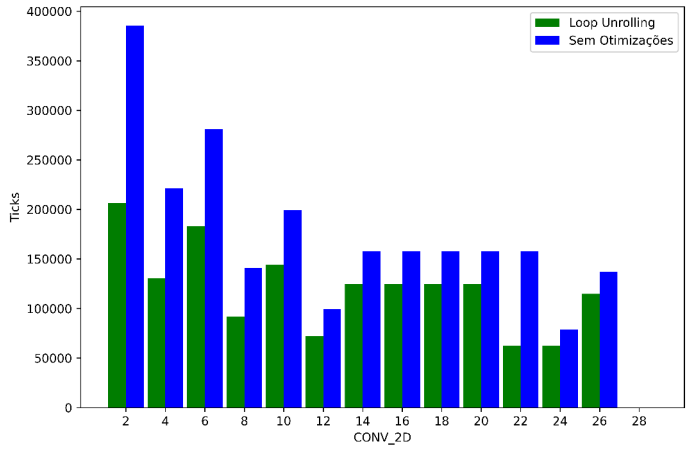
```
for (int in_channel = 0; in_channel < filter_input_depth;
    ++in_channel) {
  int32_t input_val = input_data[Offset(input_shape, batch,
      in_y, in_x, in_channel + group * filter_input_depth)];
  int32_t filter_val = filter_data[Offset(filter_shape,
      out_channel, filter_y, filter_x, in_channel)];
  acc += filter_val * (input_val + input_offset);
}
```

Fonte: Autor (2023).

Sabe-se que a variável `filter_input_depth` que define a profundidade do filtro é sempre um múltiplo de quatro, logo, pode-se implementar quatro operações de acumulação antes de verificar a condição do laço. O resultado depois da implementação do *loop unrolling* é

apresentado na Tabela (9) onde o tempo de execução para uma inferência após a implementação foi reduzido para 159,57 s. A Figura (29) apresenta um comprativo da quantidade de "ticks" entre as camadas de convolução 2D do modelo após implementação do desenrolamento de laço.

Figura 29 – Comparativo entre a inferência sem otimizações e com o *loop unrolling*.



Fonte: Autor (2023).

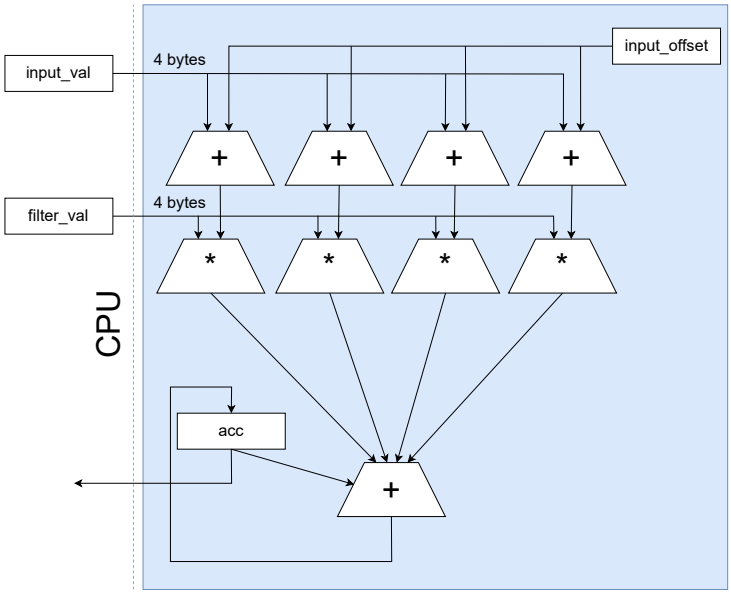
Para otimizar a execução do modelo através do *hardware*, faz-se o uso de uma unidade SIMD *multiply-and-accumulate* para operar em cada *byte* de cada entrada como descrito na Figura (16). Utiliza-se a linguagem de descrição de *hardware* Verilog para implementar *hardware* adicional e performar a operação:

```
acc += filter_val * (input_val + input_offset);
```

Para a descrição, defini-se os barramentos de dados e dos sinais de controle do modulo da CFU apresentados na Figura (17), estabelece-se um parâmetro local chamado de `InputOffset` para armazenar o valor do *offset* e evitar o seu transporte através de um barramento. Em seguida é descrito o *hardware* responsável por executar a instrução SIMD apresentada na Figura (30) para realizar a operação e multiplicação de acumulação. Por fim, é implementado o controle entre CFU e CPU onde tem-se um fluxo de controle para o *reset*, para aguardar até que a resposta seja entregue a CPU e para realizar a acumulação ou resetar o acumulador dependendo do *opcode* enviado.

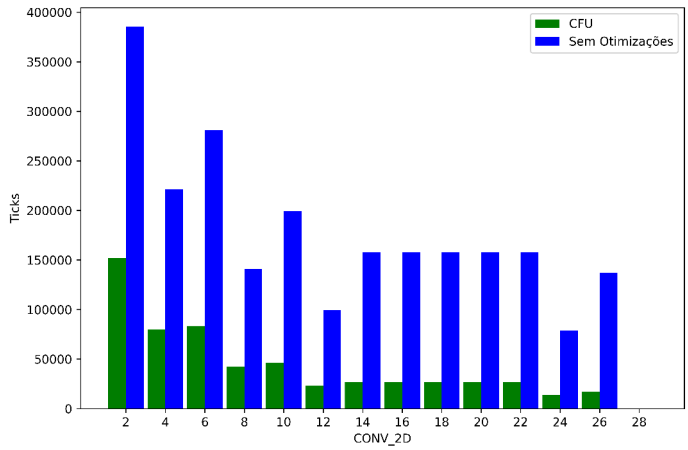
Uma comparação da inferência sem otimização e com a utilização da CFU para aplicar as otimizações de *hardware* é apresentado na Figura (31).

Figura 30 – Diagrama do bloco *multiply-and-accumulate* da CFU.



Fonte: Autor (2023).

Figura 31 – Comparativo entre a inferência sem otimizações e com a CFU implementada.



Fonte: Autor (2023).

Ao final da implementação das otimizações de *hardware* e *software* por meio do *co-design* obteve-se uma redução no tempo de inferência para cerca de 77,33 s, que representa uma diminuição de 65,74% no tempo total de uma inferência. Ao contrário do ESP32-C3, a contagem de ciclos pode sofrer uma pequena alteração mas que não afeta de forma significativa o resultado das otimizações.

4.3 Consumo Energético e Custo

Utiliza-se um aparelho de medição USB para adquirir os dados relacionados ao consumo de cada placa tais como a tensão e corrente da porta USB durante a execução do modelo, a Tabela (6) apresenta os dados obtidos na medição.

Tabela 6 – Dados do consumo energético de cada placa.

Placa	Corrente (mA)	Tensão (V)
Kosagi Fomu	≤ 9	5,12
ESP-C3-32S-Kit	30	5,12

Fonte: Autor (2023).

A placa FPGA kosagi Fomu por ter uma área reduzida e consequentemente menos componentes, requer uma corrente menor ou igual a 9 mA que é aproximadamente $3\times$ menor quando comparada com a corrente requisitada pelo SoC ESP32-C3 vale salientar que os parâmetros se mantém inalterados ao analisar com as otimizações e sem elas. Com relação ao preço de cada placa, o FPGA pode ser adquirido por, em média, $50,00 enquanto a placa ESP-C3-32S-Kit da Ai Thinker custa aproximadamente $13,00. O preço elevado do Fomu pode estar associado ao processo de fabricação que contém componentes mais caros de serem fabricados como o FPGA Lattice ICE40UP5K, outro fator de encarecimento da placa FPGA Kosagi Fomu é de que o mesmo não é produzido em larga escala e consequentemente não é comercializado em larga escala implicando em um aumento do preço.

5 CONCLUSÃO

Realizados os processos de otimização em ambas as plataformas de desenvolvimento, a inferência do modelo no ESP-C3-32S-Kit é cerca de 171× mais rápida quando comparada com o FPGA Kosagi Fomu. Muito desta discrepância no tempo de inferência se dá por conta das CPUs que encontra-se em cada plataforma, no Kosagi Fomu tem-se uma CPU *softcore* (VexRiscv) com 12 MHz de frequência máxima de *clock* enquanto o ESP32-C3 possuí um processador *hardcore* de 160 MHz que permite executar mais operações em um menor espaço de tempo.

O Fomu ainda traz outro fator limitante, sua comunicação com o *host* é implementada através de um *softcore* o que reduz os recursos disponíveis para implementação da *soft* CPU e CFU acarretando em um *co-desing hardware/software* simplificado, i. e., sem recursos sofisticados presentes na CFU. Com relação ao consumo, o Fomu tem vantagem dado sua área reduzida e consequentemente menos componentes *on-chip* quando comparado ao ESP32-C3 e este se torna mais atrativo dado a quantidade de recursos voltados ao IoT presentes na placa e custo para aquisição.

Relato as otimizações presentes em cada plataforma, fica evidente a eficiência do *co-design* quando comparado a otimizações individuais de *software*. Alcançou-se 65,74% de otimização no tempo de inferência do modelo implicando que o Fomu é uma plataforma viável para aplicações onde tem-se modelos com poucas camadas de convolução.

5.1 Trabalhos Futuros

Este trabalho aborda o *co-design hardware/software* através do CFU Playground como ferramenta para diminuir o tempo de inferência do modelo otimizando o *kernel* do TFLM em FPGAs e compara a outras plataformas com *software* otimizado, como sugestões para trabalhos futuros pode-se elencar:

- Mover operações do *software* para o *gateware* por completo utilizando o *framework* CFU Playground;
- Otimizar outras operações presentes no *kernel* do TFLM;
- Investigação e otimização de outros modelos de ML, um exemplo de modelo é o *keyword spotting*.

REFERÊNCIAS

AMAZON. **TOP1 ESP-C3-32S Kit Development Board for IoT Applications Industrial Control Board**. 2021. Disponível em: https://www.amazon.com/TOP1-ESP-C3-32S-Kit-Development-Applications-Industrial/dp/B09BTKNFGH. Acesso em: 07 de Ago. de 2023.

BANBURY, Colby R. *et al.* Benchmarking tinyml systems: Challenges and direction. **CoRR**, 2020.

CALLAHAN, Timothy *et al.* **The CFU Playground: Accelerate ML models on FPGAs**. 2020. Disponível em: https://cfu-playground.readthedocs.io/en/latest/index.html. Acesso em: 14 de Ago. de 2023.

CHEN, Andrew Tzer-Yeu *et al.* Convolutional neural network acceleration with hardware/software co-design. **Applied Intelligence**, v. 48, p. 1288–1301, 2018.

CHOLLET, François. **Deep Learning with Python**. Shelter Island, NY: Manning Publications, 2018.

CROSS, Sean; ANSELL, Tim. **Fomu: an FPGA in your USB port**. 2023. Disponível em: https://www.crowdsupply.com/sutajio-kosagi/fomu. Acesso em: 19 de Ago. de 2023.

CROSS, Sean *et al.* **Fomu Hardware**. 2023. Disponível em: https://github.com/im-tomu/fomu-hardware/tree/master. Acesso em: 19 de Ago. de 2023.

DATTU, Vikram *et al.* **TensorFlow Lite Micro for Espressif Chipsets**. 2023. Disponível em: https://github.com/espressif/tflite-micro-esp-examples. Acesso em: 31 de Ago. de 2023.

DENG, Jia *et al.* Imagenet: A large-scale hierarchical image database. **2009 IEEE conference on computer vision and pattern recognition**, p. 248–255, 2009.

GIORDANO, Marco; MAYER, Philipp; MAGNO, Michele. A battery-free long-range wireless smart camera for face detection. **Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems**, p. 29–35, 2020.

HIJAZI, Samer *et al.* Using convolutional neural networks for image recognition. **Cadence Design Systems Inc.: San Jose, CA, USA**, v. 9, p. 1, 2015.

HOWARD, Andrew G *et al.* Mobilenets: Efficient convolutional neural networks for mobile vision applications. **arXiv preprint arXiv:1704.04861**, 2017.

JAIN, Advait *et al.* **Person Detection Training**. 2021. Disponível em: https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/person_detection/training_a_model.md. Acesso em: 18 de Ago. de 2023.

KERMARREC, Florent *et al.* Litex: an open-source soc builder and library based on migen python DSL. **CoRR**, 2020.

KIM, Eunchan *et al.* Tinyml-based classification in an ecg monitoring embedded system. **Computers, Materials and Continua**, Tech Science Press, v. 75, n. 1, p. 1751–1764, 2023.

O'SHEA, Keiron; NASH, Ryan. An introduction to convolutional neural networks. **CoRR**, 2015.

OSMAN, Anas *et al.* Tinyml platforms benchmarking. **International Conference on Applications in Electronics Pervading Industry, Environment and Society**, Springer, p. 139–148, 2021.

OVTCHAROV, Kalin *et al.* Accelerating deep convolutional neural networks using specialized hardware. **Microsoft Research Whitepaper**, Citeseer, v. 2, n. 11, p. 1–4, 2015.

PRADO, Miguel de *et al.* Robustifying the deployment of tinyml models for autonomous mini-vehicles. **Sensors**, MDPI, v. 21, n. 4, p. 1339, 2021.

PRAKASH, Shvetank *et al.* Cfu playground: Full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas. In: IEEE. **2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [*S. l.*], 2023. p. 157–167.

PRASANNA, Raghavendra *et al.* Implementation of tiny machine learning models on arduino 33 ble for gesture and speech recognition. **arXiv preprint arXiv:2207.12866**, 2022.

RASCHKA, Sebastian; MIRJALILI, Vahid. **Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2**. Birmingham, UK: Packt Publishing, 2019.

ROSHAN, A Navaas *et al.* Adaptive traffic control with tinyml. **2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)**, p. 451–455, 2021.

SANTANA, Gabriel Dias. **ArchLearn: Implementação de acelerador em hardware baseado em FPGA para redes neurais artificiais**. 61 p. Monografia (Graduação em Engenharia da Computação) – Departamento de Computação, Universidade Federal de Sergipe, São Cristovão, Brasil, 2020.

SEGUNDO, Francisco Carlos Gurgel da Silva. **Aplicação de inteligência computacional em projetos de superfície seletiva em frequência multibanda e/ou banda larga para aplicações comerciais**. 141 p. Tese (Doutorado em Engenharia Elétrica e de Computação) – Centro de Tecnologia, Universidade Federal do Rio Grande do Norte, Natal, Brasil, 2018.

SHAH, David *et al.* Yosys+nextpnr: an open source framework from verilog to bitstream for commercial fpgas. In: IEEE. **2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [*S. l.*], 2019. p. 1–4.

SHIRER, Michael; MACGILLIVRAY, Carrie. The growth in connected iot devices is expected to generate 79.4 zb of data in 2025, according to a new idc forecast. **IDC. com. https://www. idc. com/getdoc. jsp**, 2019.

SIANG, Yap Yan; AHAMD, Mohd Ridzuan; ABIDIN, Mastura Shafinaz Zainal. Anomaly detection based on tiny machine learning: A review. **Open International Journal of Informatics**, v. 9, n. Special Issue 2, p. 67–78, 2021.

SILVA, Ivan Nunes; SPATTI, Danilo Hernane; FLAUZINO, Rogério Andrade. **Redes Neurais Artificiais Para Engenharia e Ciências Aplicadas**. 2. ed. São Paulo: Artliber, 2016.

TENSORFLOW. **TensorFlow Lite para microcontroladores**. 2021. Disponível em: https://www.tensorflow.org/lite/microcontrollers?hl=pt-br. Acesso em: 24 de Jul. de 2023.

THINKER, AI. **ESP-C3-32S Kit V1.0 Specification**. 2021. Disponível em: https://docs.ai-thinker.com/_media/esp32/docs/esp-c3-32s-kit-v1.0_specification.pdf. Acesso em: 19 de Ago. de 2023.

WARDEN, Pete; SITUNAYAKE, Daniel. **Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers**. [*S. l.*]: O'Reilly Media, 2019.

WONG, Alexander; FAMOURI, Mahmoud; SHAFIEE, Mohammad Javad. Attendnets: Tiny deep image recognition neural networks for the edge via visual attention condensers. **CoRR**, abs/2009.14385, 2020. Disponível em: https://arxiv.org/abs/2009.14385.

ANEXOS

ANEXO A – INFERÊNCIA DO MODELO NO FOMU PARTE I

Tabela 7 – Inferência sem otimizações.

Event	Tag	Ticks
0	DEPTHWISE_CONV_2D	56 337
1	DEPTHWISE_CONV_2D	55 983
2	CONV_2D	206 443
3	DEPTHWISE_CONV_2D	27 965
4	CONV_2D	130 641
5	DEPTHWISE_CONV_2D	54 873
6	CONV_2D	183 176
7	DEPTHWISE_CONV_2D	13 726
8	CONV_2D	91 951
9	DEPTHWISE_CONV_2D	26 416
10	CONV_2D	144 372
11	DEPTHWISE_CONV_2D	6 625
12	CONV_2D	72 297
13	DEPTHWISE_CONV_2D	12 202
14	CONV_2D	124 667
15	DEPTHWISE_CONV_2D	1 229
16	CONV_2D	124 667
17	DEPTHWISE_CONV_2D	12 228
18	CONV_2D	124 659
19	DEPTHWISE_CONV_2D	12 236
20	CONV_2D	124 667
21	DEPTHWISE_CONV_2D	12 222
22	CONV_2D	62 396
23	DEPTHWISE_CONV_2D	3 094
24	CONV_2D	62 396
25	DEPTHWISE_CONV_2D	5 179
26	CONV_2D	114 761
27	AVERAGE_POOL_2D	2 289
28	CONV_2D	156
29	RESHAPE	14
30	SOFTMAX	83

Fonte: Autor (2023).

Tabela 8 – Inferência com valores literais.

Event	Tag	Ticks
0	DEPTHWISE_CONV_2D	56 337
1	DEPTHWISE_CONV_2D	55 984
2	CONV_2D	190 787
3	DEPTHWISE_CONV_2D	27 965
4	CONV_2D	221 040
5	DEPTHWISE_CONV_2D	54 874
6	CONV_2D	178 755
7	DEPTHWISE_CONV_2D	13 726
8	CONV_2D	89 813
9	DEPTHWISE_CONV_2D	26 417
10	CONV_2D	144 315
11	DEPTHWISE_CONV_2D	6 626
12	CONV_2D	72 289
13	DEPTHWISE_CONV_2D	12 202
14	CONV_2D	126 744
15	DEPTHWISE_CONV_2D	12 210
16	CONV_2D	126 744
17	DEPTHWISE_CONV_2D	12 228
18	CONV_2D	126 737
19	DEPTHWISE_CONV_2D	12 237
20	CONV_2D	126 743
21	DEPTHWISE_CONV_2D	12 222
22	CONV_2D	126 748
23	DEPTHWISE_CONV_2D	3 095
24	CONV_2D	63 441
25	DEPTHWISE_CONV_2D	5 180
26	CONV_2D	117 892
27	AVERAGE_POOL_2D	2 371
28	CONV_2D	161
29	RESHAPE	5
30	SOFTMAX	84

Fonte: Autor (2023).

ANEXO B – INFERÊNCIA DO MODELO NO FOMU PARTE II

Tabela 9 – Inferência com o *loop unrolling*.

Event	Tag	Ticks
0	DEPTHWISE_CONV_2D	56 337
1	DEPTHWISE_CONV_2D	55 983
2	CONV_2D	206 443
3	DEPTHWISE_CONV_2D	27 965
4	CONV_2D	130 641
5	DEPTHWISE_CONV_2D	54 873
6	CONV_2D	183 176
7	DEPTHWISE_CONV_2D	13 726
8	CONV_2D	91 951
9	DEPTHWISE_CONV_2D	26 416
10	CONV_2D	144 372
11	DEPTHWISE_CONV_2D	6 625
12	CONV_2D	72 297
13	DEPTHWISE_CONV_2D	12 202
14	CONV_2D	124 667
15	DEPTHWISE_CONV_2D	1 229
16	CONV_2D	124 667
17	DEPTHWISE_CONV_2D	12 228
18	CONV_2D	124 659
19	DEPTHWISE_CONV_2D	12 236
20	CONV_2D	124 667
21	DEPTHWISE_CONV_2D	12 222
22	CONV_2D	62 396
23	DEPTHWISE_CONV_2D	3 094
24	CONV_2D	62 396
25	DEPTHWISE_CONV_2D	5 179
26	CONV_2D	114 761
27	AVERAGE_POOL_2D	2 289
28	CONV_2D	156
29	RESHAPE	14
30	SOFTMAX	83

Fonte: Autor (2023).

Tabela 10 – Inferência com a CFU.

Event	Tag	Ticks
0	DEPTHWISE_CONV_2D	56 337
1	DEPTHWISE_CONV_2D	55 982
2	CONV_2D	152 015
3	DEPTHWISE_CONV_2D	27 965
4	CONV_2D	79 620
5	DEPTHWISE_CONV_2D	54 873
6	CONV_2D	83 533
7	DEPTHWISE_CONV_2D	13 726
8	CONV_2D	42 247
9	DEPTHWISE_CONV_2D	26 416
10	CONV_2D	46 060
11	DEPTHWISE_CONV_2D	6 626
12	CONV_2D	23 168
13	DEPTHWISE_CONV_2D	12 203
14	CONV_2D	26 935
15	DEPTHWISE_CONV_2D	12 210
16	CONV_2D	26 936
17	DEPTHWISE_CONV_2D	12 229
18	CONV_2D	26 928
19	DEPTHWISE_CONV_2D	12 237
20	CONV_2D	26 934
21	DEPTHWISE_CONV_2D	12 222
22	CONV_2D	26 940
23	DEPTHWISE_CONV_2D	3 095
24	CONV_2D	13 518
25	DEPTHWISE_CONV_2D	5 180
26	CONV_2D	17 281
27	AVERAGE_POOL_2D	2 328
28	CONV_2D	63
29	RESHAPE	14
30	SOFTMAX	84

Fonte: Autor (2023).

ANEXO C – DESCRIÇÃO DA CFU EM VERILOG

Figura 32 – Código fonte da CFU.

```
module Cfu (
  input               cmd_valid,
  output              cmd_ready,
  input      [9:0]    cmd_payload_function_id,
  input      [31:0]   cmd_payload_inputs_0,
  input      [31:0]   cmd_payload_inputs_1,
  output reg          rsp_valid,
  input               rsp_ready,
  output reg [31:0]   rsp_payload_outputs_0,
  input               reset,
  input               clk
);
  localparam InputOffset = $signed(9'd128);

  // Operacao SIMD de multiplicacao:
  wire signed [15:0] prod_0, prod_1, prod_2, prod_3;
  assign prod_0 =  ($signed(cmd_payload_inputs_0[7 : 0]) + InputOffset)
                 * $signed(cmd_payload_inputs_1[7 : 0]);
  assign prod_1 =  ($signed(cmd_payload_inputs_0[15: 8]) + InputOffset)
                 * $signed(cmd_payload_inputs_1[15: 8]);
  assign prod_2 =  ($signed(cmd_payload_inputs_0[23:16]) + InputOffset)
                 * $signed(cmd_payload_inputs_1[23:16]);
  assign prod_3 =  ($signed(cmd_payload_inputs_0[31:24]) + InputOffset)
                 * $signed(cmd_payload_inputs_1[31:24]);

  wire signed [31:0] sum_prods;
  assign sum_prods = prod_0 + prod_1 + prod_2 + prod_3;

  // Sinal para obter a resposta e autorizar o comando:
  assign cmd_ready = ~rsp_valid;

  always @(posedge clk) begin
    if (reset) begin
      rsp_payload_outputs_0 <= 32'b0;
      rsp_valid <= 1'b0;
    end else if (rsp_valid) begin
      // Aguarda para entregar a resposta para CPU:
      rsp_valid <= ~rsp_ready;
    end else if (cmd_valid) begin
      rsp_valid <= 1'b1;
      // Passo de acumulacao:
      rsp_payload_outputs_0 <= |cmd_payload_function_id[9:3]
          ? 32'b0
          : rsp_payload_outputs_0 + sum_prods;
    end
  end
endmodule
```

Fonte: Adaptado de Callahan *et al.* (2020).

ANEXO D – INFERÊNCIA UTILIZANDO O ESP-IDF

Utiliza-se o *framework* da Espressif, ESP-IDF, para carregar o modelo na placa e executar a inferência do mesmo. Primeiramente navega-se até o diretório onde se encontra a pasta do modelo e executa-se o comando `idf.py set-target esp32c3` para configurar o SoC utilizado, segue-se com o comando `idf.py build` que compila o código fonte e os transforma em binários para execução no microcontrolador. Para carregar o modelo e monitorar a saída serial, utiliza-se o comando:

```
idf.py --port /dev/ttyUSB0 flash monitor
```

A janela do monitor permite que realize-se uma inferência através do comando:

```
detect_image 0
```

Por padrão o ESP-IDF utiliza a biblioteca contendo as otimizações do *kernel* (ESP-NN), para selecionar a biblioteca padrão (ANSI C) executa-se o comando `idf.py menuconfig` antes de compilar o modelo e seleciona-se a opção ANSI C, como segue:

```
ESP-NN > NN_OPTIMIZATIONS > ANSI C
```