Filipe Dos Santos Adami Tcacenco

Otimização de Rede Neural em ARM32

Porto Alegre, RS

Maio de 2021

Filipe Dos Santos Adami Tcacenco

Otimização de Rede Neural em ARM32

Trabalho de Diplomação em Engenharia Física II apresentado à faculdade de Engenharia Física da Universidade Federal do Rio Grande do Sul como requisito para obtençãodo grau de bacharel em Engenharia Física

Universidade Federal do Rio Grande do Sul

Orientador: Prof. Carlo Requião da Cunha, Ph.D.

Porto Alegre, RS
Maio de 2021

Filipe Dos Santos Adami Tcacenco

Otimização de Rede Neural em ARM32

Trabalho de Diplomação em Engenharia Física II apresentado à faculdade de Engenharia Física da Universidade Federal do Rio Grande do Sul como requisito para obtençãodo grau de bacharel em Engenharia Física

Trabalho aprovado. Porto Alegre, RS, 20 de Maio de 2021:

Prof. Carlo Requião da Cunha, Ph.D.
Orientador

Prof. Dr. Luigi Carro
Avaliador

Prof. Dr. Rubem Erichsen Junior
Avaliador

Porto Alegre, RS
Maio de 2021

Resumo

Este trabalho visa o desenvolvimento de um sistema para processamento de modelos de redes neurais do tipo *feedforward* em ponto flutuante e modelos quantizados em ponto fixo de 32, 16, 8 e 4 bits de precisão em um microcontrolador ARM Cortex M4. O trabalho teve como objetivo avaliar as vantagens e desvantagens do processamento do algoritmo de *forward propagation* em diferentes precisões com relação ao consumo energético, alocação de memória, velocidade de processamento e erro de quantização. Além disso foi executado um algoritmo de otimização sobre uma rede neural de regressão para obter hiperparâmetros que minimizam a função custo respeitando os limites de memória estabelecidos. A implementação foi desenvolvida, inicialmente, em um computador e, em sequência, foi adaptada para ser processada em microcontrolador utilizando a biblioteca CMSIS DSP. Os resultados obtidos da implementação foram testados com um *dataset* aberto e foi possível verificar algumas vantagens da inferência em ponto fixo *signed* de 16 e 8 bits em relação ao ponto flutuante. Contudo, o algoritmo implementado em ponto flutuante se mostrou superior em velocidade de processamento e consumo energético.

Palavras-chave: Redes Neurais, Sistemas Embarcados, Microcontroladores, ARM, Deep Learning, Otimização.

Sumário

1 INTRODUÇÃO . . . 7
1.1 Histórico . . . 7
1.2 Estado da Arte . . . 8
1.3 Questão de Pesquisa . . . 8
1.4 Relevância . . . 9
1.5 Objetivo . . . 9
1.6 Organização . . . 10

I REFERENCIAL TEÓRICO 11

2 REDES NEURAIS ARTIFICIAIS . . . 13
2.1 Funcionamento e *Forward propagation* . . . 13
2.1.1 Funções de Ativação . . . 15
2.1.2 ReLU - Rectifier Linear Units . . . 15
2.1.3 Sigmoide - Função Logística . . . 16
2.1.4 Tanh - Tangente Hiperbólica . . . 17
2.1.5 Softmax . . . 17
2.2 Otimização e *backpropagation* . . . 18
2.3 Quantização . . . 20

II METODOLOGIA 23

3 METODOLOGIA . . . 25
3.1 Materiais e Métodos . . . 26
3.1.1 Orçamento . . . 26
3.1.2 Esboço do Sistema . . . 26
3.1.3 Configurações do Projeto e Alocação de Memória . . . 27
3.1.4 Rede *Feedforward* . . . 29
3.1.5 *Simulated Annealing* . . . 30
3.2 Desenvolvimento . . . 31
3.2.1 Processamento em Ponto Flutuante . . . 31
3.2.2 Quantização de Rede Treinada . . . 35
3.2.3 Processamento em Ponto Fixo . . . 35
3.3 Comunicação Serial via USB . . . 41
3.4 Obtenção dos Dados para Avaliação . . . 42

III RESULTADOS 45

4 RESULTADOS 47
4.1 Otimização por *Simulated Annealing* 47
4.2 Treinamento com Hiperparâmetros Selecionados 53
4.3 Quantização dos Modelos Treinados 63
4.4 Avaliação dos Modelos 65

IV CONCLUSÃO 69

5 CONCLUSÃO 71
5.1 Direções Futuras 71

REFERÊNCIAS 73

1 Introdução

1.1 Histórico

Com o recente avanço das tecnologias digitais, a quantidade de dados se tornou tão grande até o ponto de as técnicas tradicionais de aprendizado de máquina para processamento de dados não serem mais tão efetivas para tratamento e classificação de informações (X.; H., 2015)(Rezvani; Wang; Pourpanah, 2019)(WANG; ZHAO, 2020). Para resolver os problemas da análise de conjuntos de dados enormes, complexos e de alta dimensionalidade, modelos de *deep learning* mostraram resultados excepcionais, revolucionando o futuro da inteligência artificial(WANG; ZHAO, 2020). Esses modelos, de fato, receberam uma atenção especial na última década após a crescente popularização da inteligência artificial. Algumas instituições de ensino renomadas e empresas, como Google e Facebook, criaram suas próprias bibliotecas, como Pytorch, Tensorflow e Keras (Wan, 2019)(PATEL, 2017). Contudo, uma das primeiras ideias de utilizar lógica de limiar começou em 1938, quando Rashevsky iniciou os estudos em teoria de campos neurais, representando ativação e propagação em termos de equações diferenciais(MEHROTRA; MOHAN; RANKA, 1997). Em 1969, Minsky e Papert mostraram os pontos positivos e as limitações do algoritmo de *perceptron*, o que resultou em uma redução drástica no financiamento de pesquisas em redes neurais (MEHROTRA; MOHAN; RANKA, 1997). Em 1989, aplicando o algoritmo de *backpropagation*, Yann LeCun desenvolve uma rede neural profunda funcional (uma das primeiras redes neurais convolucionais modernas) para reconhecimento de escrita a mão, porém o tempo de treinamento da rede foi medido em dias (AL, 1989). Nos anos 2000, essa forma de aprendizado de máquina começou a impactar o mercado com a utilização de redes neurais convolucionais para processamento de cheques bancários preenchidos a mão nos Estados Unidos (LECUN, 2016). Em 2011, após a inserção da ferramenta "Google Voice", a Apple lança a "Siri"(assistente virtual que utiliza reconhecimento de voz) e implementa o uso de redes neurais profundas nesse serviço em 2014, aprimorando a performance do serviço com maior precisão no reconhecimento de voz (LEVY, 2016). O tempo de processamento e treinamento foram reduzidos com os avanços tecnológicos em *central processing units* (CPUs) e *graphics processing units* (GPUs) no início da década de 2010 (FOOTE, 2017). Na sequência, com a inserção de *deep learning* cada vez maior no mercado e na academia, juntamente com a demanda por maior velocidade de processamento, a Google introduz em 2016 a *tensor processing unit* (TPU), *hardware* dedicado à álgebra multilinear para inteligência artificial, o que diminuiu consideravelmente os tempos de treinamento de redes neurais (SATO; YOUNG; PATTERSON, 2017)

Apesar de já haver bastante avanço no setor de aprendizado de máquina, há ainda muita motivação quando se trata de técnicas de otimização e processamento de redes neurais. Isto se deve ao fato de as redes neurais e os conjuntos de dados se tornam cada vez maiores e mais complexos e o processo de treinamento poder ser muito lento mesmo quando são utilizadas GPUs modernas para aceleração. Além disso, GPUs consomem muita energia e podem ser um problema em sistemas embarcados ou sistemas alimentados por bateria(QASAIMEH et al., 2019). Existem aplicações em que é vantajoso o processamento de redes neurais em um microcontrolador por diversos motivos, como em um sistema *standalone* em que não há conexão com internet e é necessário reconhecer padrões, em pré-reconhecimento e compactação de dados para reduzir o consumo energético quando forem processados em nuvem ou até mesmo para melhoras em questão de privacidade (TAN, 2018). Porém, este tipo de dispositivo muitas vezes tem certas limitações que restringem ou impossibilitam este modo de aplicação.

1.2 Estado da Arte

Atualmente, o processamento de redes neurais é comumente realizado em CPUs (fase de predição ou classificação) e GPUs (utilizada na fase de treinamento) (Raihan; Goli; Aamodt, 2019). Ele também é realizado, em alguns casos, em placas dedicadas, como chips de inteligência artificial em *smartphones* e em TPUs, para processamento em nuvem. O último dispositivo citado é considerado o mais rápido quando se trata de processamento de redes neurais, pois ele utiliza uma arquitetura diferente de GPUs, em que é possível obter um desempenho muito alto em operações matriciais com consumo energético baixo quando comparado com CPUs e GPUs (SATO; YOUNG; PATTERSON, 2017). Além disso, quando se trata somente de processamento de redes neurais, existem algumas soluções disponíveis para facilmente implementar uma rede neural específica em FPGA (*Field-programmable Gate Array*), como Zynqnet e FBLAS (geração de módulos para *Basic Linear Algebra Subprograms* - BLAS), acelerando o processamento da mesma (GSCHWEND, 2020)(MATTEIS; LICHT; HOEFLER, 2019).

1.3 Questão de Pesquisa

Muitas vezes se torna inviável a compra de dispositivos de ponta, como TPUs; Por este motivo são procurados dispositivos com melhor relação custo benefício. Neste caso, GPUs parecem ser uma boa solução; Todavia, ainda podem ser caras e também consomem muita energia, o que é contrário à tendência de tecnologias modernas e de sistemas embarcados, inviabilizando sua utilização em certas aplicações (NAKUTIS, 2013)(Shawahna; Sait; El-Maleh, 2019). A implementação de um sistema de processamento de redes neurais em FPGA pode suprir a demanda de alta velocidade de processamento

e de baixo consumo energético; entretanto, este dispositivo tem certas limitações. O desempenho depende da forma como o sistema é projetado e seu custo é muito elevado. Sistemas compactos normalmente são compostos de microprocessadores que não possuem módulos de processamento gráfico ou de ponto flutuante em sua arquitetura, portanto muitas vezes podem não ser velozes o suficiente para processar redes neurais profundas em tempo hábil. Além disso, estes sistemas podem ter limitação de memória, o que pode ser um problema para aplicações que utilizam redes neurais. Contudo, o processamento de redes neurais em ponto fixo pode ser uma solução viável para esta implementação em sistemas com microprocessador, visto que pode diminuir o tempo de processamento e a alocação de memória.

1.4 Relevância

Recentemente, vários artigos e trabalhos ambicionam a aplicação de algoritmos de redes neurais em sistemas microcontrolados, pois tal aplicação pode trazer a possibilidade de embutir inteligência artificial em um sistema de baixo custo, de alta integrabilidade e baixo consumo energético (LIBERIS, 2020). Além disso, há formas de implementação de redes neurais (inferência e treinamento) que utilizam ponto fixo com número de bits reduzidos, o que tem mostrado, em pesquisas recentes, que ocorrem aceleração de processamento, diminuição de alocação de memória e redução de consumo energético, sem haver perda significativa de precisão na camada de saída de redes neurais (ZHU et al., 2019)(WU et al., 2018).

1.5 Objetivo

Será projetado um sistema para processamento de redes neurais do tipo *feedforward* em ponto fixo (inferência) em que, por meio de um microcontrolador com comunicação com um *host*, o usuário irá especificar os hiperparâmetros da rede e enviar os valores de pesos e bias da rede treinada e quantizada. O sistema também terá suporte para processamento em ponto flutuante para que seja possível comparar resultados entre os dois tipos de implementação. Foi escolhido um microcontrolador ARM Cortex-M4 para realizar o processamento do algoritmo de *forward propagation*, pois ele está cada vez mais presente na indústria e esse é o modelo mais simples da linha Cortex-M que possui instruções para DSP (*Digital Signal Processor*) e FPU (*Floating-point Unit*) os quais estão embarcados em seu *chip*.

Será utilizado um *dataset* aberto para específicar a rede de regressão cujos hiperparâmetros serão otimizados através do algoritmo de *simulated annealing* para processamento em ponto flutuante. Em seguida, a rede será adaptada para realizar o processamento em ponto fixo de 32, 16, 8 e 4 bits usando o processo de quantização.

A viabilidade da implementação da rede neural em ponto fixo neste microprocessador com arquitetura de 32 bits será avaliada, assim como suas vantagens e desvantagens em relação ao processamento em ponto flutuante. O algoritmo que será desenvolvido para processamento em ponto fixo em DSP pode ter velocidade de processamento maior que a implementação em ponto flutuante, consumo energético menor e menor alocação de memória, alinhando-se às tendências atuais de sistemas embarcados e tornando viável sua utilização em um sistema remoto ou de controle que tem memórias voláteis e não voláteis limitadas. Os custos envolvidos serão muito inferiores à utilização de circuitos integrados de aplicação específica, tais como chips de inteligência artificial e computação neuromórfica, pois o modelo de microcontrolador é amplamente produzido e utilizado no mercado. Além disso, é possível que a redução de alocação de memória seja suficiente para processar redes mais complexas em um sistema cuja capacidade total, se somadas memórias voláteis e não voláteis, é inferior a 2 *Megabytes*.

1.6 Organização

Primeiramente o trabalho apresenta o referencial teórico necessário para compreender o assunto. Em segundo lugar, é apresentada a metodologia que foi utilizada para tratar a questão de pesquisa, bem como a forma como o sistema foi projetado. Nessa parte são encontradas informações sobre os materiais e métodos, esboço do projeto, avaliação de riscos, cronograma e orçamento. Na sequência, são mostrados e discutidos os resultados obtidos após a implementação realizada. Por fim, é apresentada a conclusão a respeito das hipóteses levantadas.

Parte I

Referencial Teórico

2 Redes Neurais Artificiais

Redes Neurais artificiais são sistemas computacionais inspirados nas redes de neurônios biológicos do cérebro animal (CHEN et al., 2019). Elas aprendem e processam informação de uma forma similar às redes neurais biológicas. Uma rede neural artificial é capaz de classificar e gerar predições sobre um conjunto de dados de entrada, ao adaptar o processamento da entrada para que seja gerado o melhor resultado possível.

2.1 Funcionamento e *Forward propagation*

A forma mais simples de uma rede neural é chamada de perceptron multicamadas, ou rede neural *feedforward*. A finalidade dessa rede é aproximar uma função, $y = F(x)$, utilizando parâmetros θ de forma que, pelo do processo de otimização ou treinamento, a rede aprende os valores de θ que melhor aproximam a função $f(x, \theta)$ à F(x) (GOODFELLOW; COURVILLE, 2016). Essa classe de rede neural é formada por camadas compostas de neurônios. Cada um destes realiza o processamento dos dados encaminhados pela camada de neurônios anteriores, ou seja, caso a camada anterior tenha 3 neurônios, cada neurônio da camada seguinte irá processar 3 entradas e retornar um valor como saída. O valor de saída é obtido aplicando uma função de ativação na soma dos produtos dos valores de entrada x_i e pesos w_i com um valor bias.

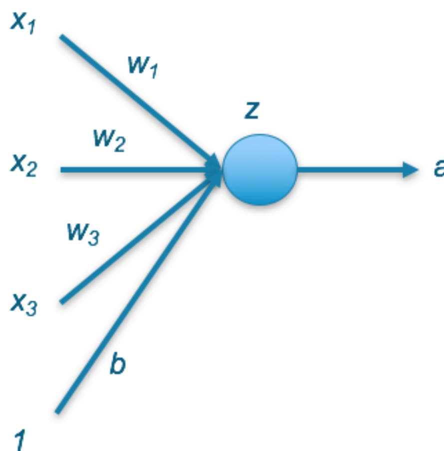

Figura 1 – Neurônio de uma rede neural. Fonte:(RONAGHAN, 2018)

As equações 2.1 e 2.2 descrevem de forma resumida a operação que um neurônio realiza.

$$Z_{(W,X,b)} = b + \sum_i w_i x_i, \tag{2.1}$$

$$a = g_{ativ}(Z_{(w,x,b)}). \tag{2.2}$$

Por meio da conexão de diversos neurônios, que formam camadas de neurônios, com entrada, camadas escondidas (*hidden layers*) e saídas (que são os valores aproximados da função desejada), é possível construir uma rede neural como a da figura 2, em que cada camada pode ter um número distinto de neurônios e a camada seguinte é alimentada com a saída da anterior (*feedforward*).

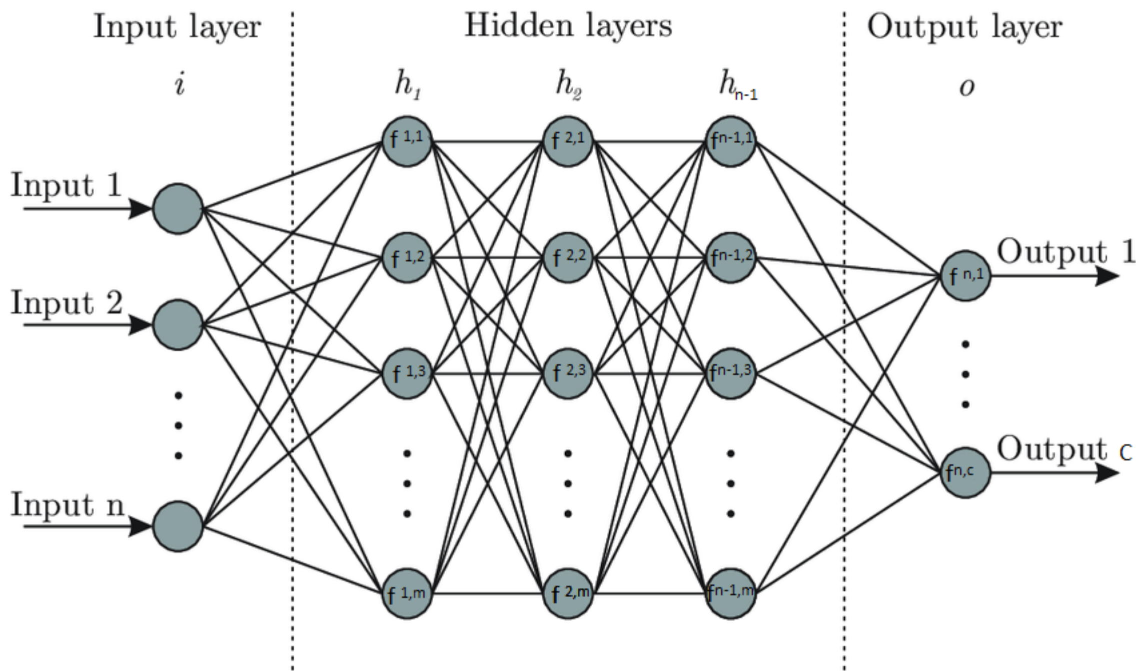

Figura 2 – *Artificial Neural Network* (ANN). Figura modificada de (BRE; GIMENEZ; FACHINOTTI, 2017)

O funcionamento dessa rede neural se dá por meio do processo que é chamado de *forward propagation*, em que normalmente é utilizada a notação matricial para descrever as operações dos neurônios exemplificadas nas equações 2.1 e 2.2. A forma matricial é mais compacta para explicar o funcionamento tanto do *forward propagation*, quanto do treinamento (*backpropagation*) da rede. Na figura 2, o valor de saída de cada neurônio é descrito pela função f_k^l, em que o índice "l"representa a camada do neurônio e o índice "k", o neurônio da camada "l". A equação 2.3 expressa esta representação, em que o vetor $\vec{f_l}$ representa o conjunto de saídas dos neurônios da camada "l", $\vec{b_i}$, o conjunto de valores bias de cada neurônio dessa camada, e W_l é o tensor bidimensional em que cada linha contém o conjunto de pesos de cada neurônio. A função de ativação g_{ativ} é aplicada em cada elemento do vetor resultante entre parênteses.

$$\vec{f_l} = g_{ativ}\left(\vec{b_l} + W_l \cdot \vec{f_{l-1}}\right), \tag{2.3}$$

$$\begin{bmatrix} f_1^l \\ f_2^l \\ \vdots \\ \vdots \\ f_m^l \end{bmatrix} = g_{ativ}\left(\begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ \vdots \\ b_m^l \end{bmatrix} + \begin{bmatrix} w_{1,1}^l & w_{1,2}^l \cdots\cdots & w_{1,n}^l \\ w_{2,1}^l & w_{2,2}^l \cdots\cdots & w_{2,n}^l \\ \vdots & \vdots \ddots & \vdots \\ w_{m,1}^l & w_{m,2}^l \cdots\cdots & w_{m,n}^l \end{bmatrix}^T \begin{bmatrix} f_1^{l-1} \\ f_2^{l-1} \\ \vdots \\ \vdots \\ f_n^{l-1} \end{bmatrix} \right). \quad (2.4)$$

2.1.1 Funções de Ativação

A finalidade de uma função de ativação em redes neurais está relacionada ao modo como os neurônios funcionam dadas as entradas que eles irão processar. De forma breve, pode-se dizer que, se o resultado do somatório de $w_i x_i$ e "b"na equação 2.1 exceder um certo valor, o neurônio é acionado, ou seja, o valor de saída dele é alterado. Contudo, essa função de ativação não é necessariamente booleana. Isso quer dizer que o valor de saída do neurônio pode ser alterado gradativamente quando alteramos também de forma gradativa os valores de entrada. Podemos definir a função de ativação como uma função que determina a saída de um nó ou neurônio dados seus valores de entrada. Funções de ativação não-lineares permitem às redes neurais computar problemas não triviais usando um número pequeno de nós ou neurônios. Algumas das funções de ativação mais utilizadas são: ReLU (rectifier Linear Units), Sigmoide, Tanh e Softmax. Em uma rede neural podem ser utilizadas diferentes funções de ativação em diferentes camadas para otimizar a rede.

2.1.2 ReLU - Rectifier Linear Units

Essa função de ativação se assemelha a uma função linear, exceto pela sua descontinuidade em $z = 0$. Portanto, no processo de otimização ou treinamento ela pode ser facilmente manipulada, pois sua derivada é descontínua, mas constante onde $z \neq 0$. Essa função pode ser descrita como $g_{ReLU}(z) = max\{0, z\}$ ou pela equação 2.5.

$$g_{ReLU}(z) = \begin{cases} z, & \text{se } z \geq 0 \\ 0, & \text{se } z < 0 \end{cases} \quad (2.5)$$

$$\frac{dg_{ReLU}(z)}{dz} = \begin{cases} 1, & \text{se } z > 0 \\ 0, & \text{se } z < 0 \end{cases} \quad (2.6)$$

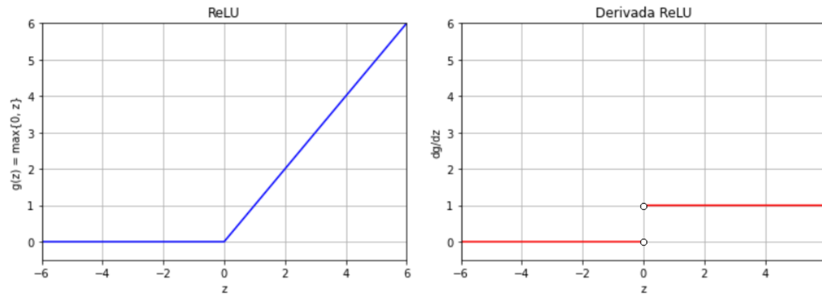

Figura 3 – Função de ativação ReLU e sua derivada (não definida em $z = 0$).

Devido ao fato da sua derivada ser 0 quando $z < 0$, redes neurais que utilizam essa função não podem aprender ou serem otimizadas utilizando métodos baseados em gradiente que serão explicados mais adiante nesse trabalho. Por esse motivo, existem variantes da função ReLU para que a derivada seja diferente de 0 nessa região. Essas variantes normalmente têm uma desempenho similar à ReLU (GOODFELLOW; COURVILLE, 2016). Algumas variantes dessa função utilizam o formato descrito na equação 2.7, em que a função é dividida em duas partes, em $z < 0$ e $z > 0$.

$$g(z, \alpha_i) = max\{0, z\} + \alpha_i min\{0, z\} \tag{2.7}$$

Uma variante interessante é a Leaky ReLU, que utiliza um valor positivo, constante e pequeno para a derivada da função em $z < 0$ (tipicamente 0,01). A equação 2.8 descreve essa função, em que $K \ll 1$.

$$g(z) = \begin{cases} x, & \text{se } z > 0 \\ Kx, & \text{se } z < 0 \end{cases} \tag{2.8}$$

2.1.3 Sigmoide - Função Logística

Antes da função ReLU ser introduzida, as redes neurais utilizavam a função logística, que é uma função sigmoide que tem curva em forma de "S"(GOODFELLOW; COURVILLE, 2016). Contudo, ao contrário da função ReLU, a sigmoide é contínua, diferenciável, tem seus valores de derivada não negativos para todo valor de entrada e um ponto de inflexão que define o valor máximo de sua derivada.

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2.9}$$

$$\frac{d\sigma(z)}{dz} = \sigma(z) \cdot [1 - \sigma(z)] = \frac{1}{1 + e^{-z}} \cdot \left[1 - \frac{1}{1 + e^{-z}}\right]. \tag{2.10}$$

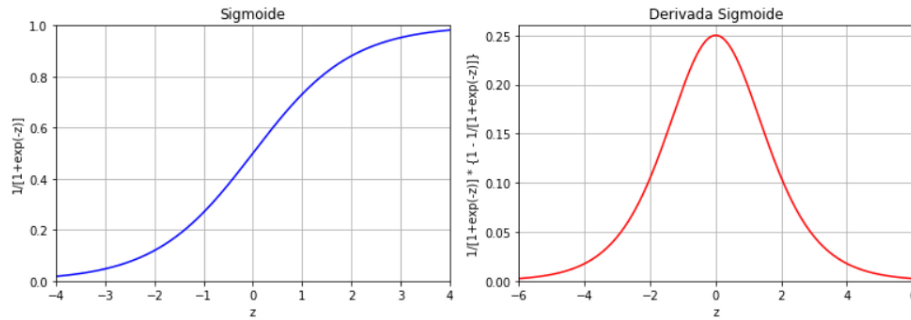

Figura 4 – Função de ativação Logística e sua derivada.

2.1.4 Tanh - Tangente Hiperbólica

A função de ativação tangente hiperbólica tem um formato muito similar à função logística, em forma de "S". Além disso, essas funções são fortemente correlacionadas de forma que uma delas pode ser facilmente expressa em função de outra.

$$tanh(z) = 2\sigma(2z) - 1 = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{2.11}$$

$$\frac{dtanh(z)}{dz} = 1 - tanh^2(z). \tag{2.12}$$

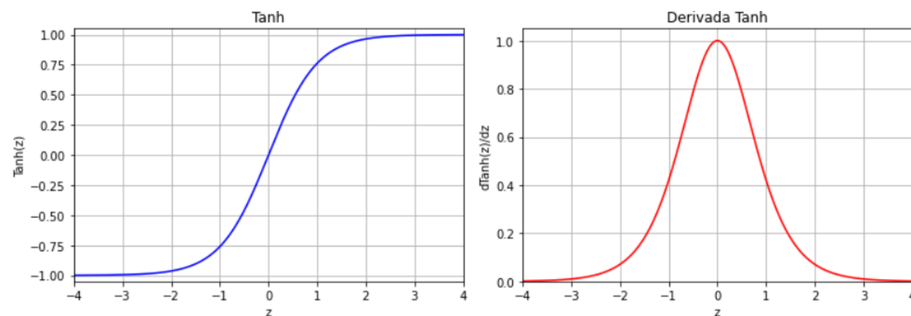

Figura 5 – Função de ativação tangente hiperbólica e sua derivada.

Quando comparada à função logística sigmoide, a tangente hiperbólica tem melhor desempenho em redes neurais *feedforward*, pois se assemelham à identidade quando z é próximo de zero. Entretanto, em redes convolucionais, a função logística apresenta melhor desempenho.

2.1.5 Softmax

A função softmax, também conhecida como função exponencial normalizada, é comumente utilizada na camada de saída de redes neurais de classificação. Isso acontece

porque ela dá um significado para a saída de uma rede neural, ao atribuir probabilidades em cada uma das saídas dos neurônios de uma camada de forma normalizada.

$$g_{sm}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{2.13}$$

A função de ativação utiliza os valores calculados em todos os neurônios de uma camada para gerar a saída em um dos neurônios. Ela normalmente é utilizada em arquiteturas de redes neurais mais complexas, as quais aprendem a manipular memória (GOODFELLOW; COURVILLE, 2016).

2.2 Otimização e *backpropagation*

Primeiramente, devemos entender que, para o treinamento de uma rede neural, é necessário minimizar a função custo (*cost* ou loss do inglês), a qual expressa quão próximo ou distante o resultado de saída de um conjunto de dados inseridos na entrada está de um valor ideal. A otimização, ou minimização da função custo, está relacionada com regressão linear e com o modo pelo qual podemos traçar a melhor reta para representar um conjunto de pontos. Porém, neste caso, não se trata de uma reta, e, sim, algo com maior dimensão. As duas funções custo mais utilizadas são a função erro quadrático médio [MSE - *mean squared error*], utilizada em redes de predição (equação 2.14), e a função de entropia cruzada (*Cross-entropy*), utilizada em redes de classificação (a função de ativação da camada de saída deve resultar em um valor entre 0 e 1) (equação 2.15):

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{Y}_i)^2, \tag{2.14}$$

$$L_{CE} = -\frac{1}{N}\sum_{i=1}^{N}\hat{Y}_i ln(y_i). \tag{2.15}$$

onde $\hat{Y}_i$ é o valor de saída desejado para a i-ésima saída (este valor é binário na função custo entropia cruzada), e y_i é o i-ésimo valor de saída obtido para um conjunto de dados de entrada. Uma forma de minimizar a função custo é aplicar otimizadores, os quais têm a finalidade de achar o mínimo dessa função. Esses otimizadores normalmente utilizam métodos de gradientes, ou seja, a função custo é gradativamente minimizada através da atualização dos parâmetros utilizados para calculá-la. Em uma rede neural, a função custo tem como parâmetro os pesos e valor *bias* que cada neurônio utiliza para gerar sua saída. Portanto, devemos alterar esses valores para obter um menor custo da rede.

Existem várias formas de fazer isso, porém a mais comum é utilizar otimização iterativa através do método de gradiente descendente estocástico. Existem diversas variantes

desse método, porém o fundamento é o mesmo: utilizar a derivada da função custo em relação a um parâmetro para alterá-lo. A equação 2.16 expressa a implementação desse método para uma iteração.

$$new_w_{j,k}^{l} = w_{j,k}^{l} - \eta \cdot \frac{\partial L}{\partial w_{j,k}^{l}}, \tag{2.16}$$

$$new_b_{k}^{l} = b_{k}^{l} - \eta \cdot \frac{\partial L}{\partial b_{k}^{l}}. \tag{2.17}$$

A variável η é chamada de taxa de aprendizado, a qual atribui a velocidade de aprendizado para cada iteração e, dependendo do método de gradiente descendente, pode ser atribuído um valor fixo a ela ou um valor que diminui ao longo das iterações. Isso se deve ao fato de que quanto mais próximo do mínimo, mais cautelosa deve ser cada iteração.

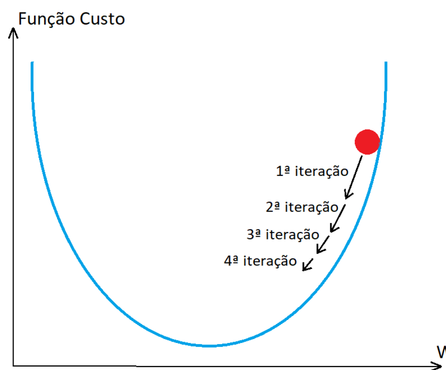

Figura 6 – Método de gradiente descendente.

É possível aplicar a regra da cadeia para obter o valor da derivada da função custo em relação ao **j**-ésimo peso do neurônio **k** da camada **l**. A equação 2.18 expressa esta relação em termos do argumento Z_k^l da função de ativação descrito na equação 2.1.

$$\frac{\partial L}{\partial w_{j,k}^{l}} = \frac{\partial L}{\partial Z_{k}^{l}} \cdot \frac{\partial Z_{k}^{l}}{\partial w_{j,k}^{l}}. \tag{2.18}$$

O termo da direita é reduzido em:

$$\frac{\partial Z_{k}^{l}}{\partial w_{j,k}^{l}} = \frac{\partial}{\partial w_{j,k}^{l}} \left(b_{k}^{l} + \sum_{i} w_{i,k}^{l} f_{i}^{l-1} \right) = f_{k}^{l-1}. \tag{2.19}$$

O termo da esquerda é denominado erro do k-ésimo neurônio na camada l e é expresso pela variável δ_k^l. Deste modo, temos a equação resultante 2.20 e obtemos, de

forma análoga à equação 2.21 (NIELSEN, 2019) para a derivada do custo em relação ao valor *bias*.

$$\frac{\partial L}{\partial w^l_{j,k}} = \delta^l_k f^{l-1}_k, \tag{2.20}$$

$$\frac{\partial L}{\partial b^l_k} = \delta^l_k. \tag{2.21}$$

Através de algumas manipulações algébricas é possível obter as equações 2.22 (NIELSEN, 2019), para o erro da camada de saída e, em forma vetorial e compacta, 2.23 (NIELSEN, 2019), para o erro da camada l em termos da camada seguinte. O símbolo "$\odot$"representa o produto Hadamard, que é o produto elemento a elemento de vetores e resulta em um vetor de mesma dimensão. Assim, é possível calcular o erro a partir da última camada e em seguida calcular o erro das camadas anteriores, de trás para frente. Por este motivo o mecanismo chama-se *backpropagation*.

$$\delta^l_k = \frac{\partial L}{\partial f^l_k}\frac{\partial f^l_k}{\partial Z^l_k} = \frac{\partial L}{\partial f^l_k} g'_{ativ}(Z^l_k), \tag{2.22}$$

$$\vec{\delta^l} = [W^{(l+1)^T} \vec{\delta^{l+1}}] \odot \vec{g_{ativ}}'(Z^l). \tag{2.23}$$

2.3 Quantização

O processo de quantização de uma rede neural pode ser realizado para diminuir a alocação de memória, reduzir o consumo energético e aumentar a velocidade de processamento, dependendo da arquitetura do *hardware* que for utilizado. As variáveis de uma rede neural *feedforward* podem ser quantizadas para ponto fixo do tipo *signed* através das equações 2.24, 2.25, 2.26 e 2.27:

$$f^q = \left\lfloor \frac{f}{S_f} \right\rfloor + z_f, \tag{2.24}$$

$$x^q = \left\lfloor \frac{x}{S_x} \right\rfloor + z_x, \tag{2.25}$$

$$w^q = \left\lfloor \frac{w}{S_w} \right\rfloor, \tag{2.26}$$

$$b^q = \left\lfloor \frac{b}{S_b} \right\rfloor. \tag{2.27}$$

Os valores S e z correspondem à resolução de quantização e ao deslocamento da distribuição da variável, respectivamente, e podem ser calculados através das equações 2.28 e 2.29. Nessa abordagem, os valores de peso e *bias* são assumidos com distribuição simétrica e centralizada em zero. Portanto, não é necessário deslocar a distribuição e a equação $Max = -Min$ deve ser verdade para este caso:

$$S = \frac{Max - Min}{2^{bits} - 1}, \tag{2.28}$$

$$z = -\frac{Min}{S} - 2^{bits-1}. \tag{2.29}$$

Manipulando e substituindo as equações de quantização na equação 2.3, é possível obter a saída quantizada de um neurônio em função de suas entradas, também quantizadas (equação 2.30):

$$f^q \;=\; \frac{g_{ativ.}\left[\left(\sum_i S_x\left(x_i^q - z_x\right) S_w w_i^q\right) \;+\; S_b b^q\right]}{S_f} \;+\; z_f. \tag{2.30}$$

Para a função de ativação ReLU e *leaky* ReLU, é possível aplicar a seguinte propriedade $k \cdot g_{ativ.}[Z] = g_{ativ.}[k \cdot Z]$ que foi utilizada na equação 2.30 para obter a equação 2.31:

$$f^q \;=\; g_{ativ.}\left[\frac{S_x S_w}{S_f}\left(\sum_i x_i^q w_i^q \;-\; z_x \sum_i w_i^q\right) \;+\; \frac{S_b}{S_f} b^q\right] \;+\; z_f, \tag{2.31}$$

$$f^q \;=\; g_{ativ.}\left[\frac{S_x S_w}{S_f}\sum_i x_i^q w_i^q \;+\left(\frac{S_b}{S_f} b^q \;-\; \frac{S_x S_w}{S_f} z_x \sum_i w_i^q\right)\right] \;+\; z_f. \tag{2.32}$$

Separando alguns termos que se resumem em constantes, é obtida a equação 2.33, cujas variáveis b_{comp} e S_{comp} são constantes do neurônio da rede quantizada e não precisam ser calculadas sempre que a rede for processada:

$$f^q \;=\; g_{ativ.}\left[b_{comp} \;+\; S_{comp} \sum_i x_i^q w_i^q\right] \;+\; z_f, \tag{2.33}$$

$$b_{comp} \;=\; \frac{S_b}{S_f} b^q \;-\; \frac{S_x S_w}{S_f} z_x \sum_i w_i^q, \tag{2.34}$$

$$S_{comp} = \frac{S_x S_w}{S_f}. \tag{2.35}$$

Parte II

Metodologia

3 Metodologia

Através da concepção de um sistema para processamento de redes neurais *feedforward* em um microcontrolador contendo um ARM Cortex M4, foi avaliado o processamento em ponto flutuante e em ponto fixo *signed* (32, 16, 8 e 4 bits de precisão) de uma rede neural *feedforward* em sistema embarcado. Os algoritmos desenvolvidos para processamento em ponto fixo utilizaram o DSP embarcado no microcontrolador e foram executados após o processo de quantização da rede. Os principais fatores que foram avaliados entre as formas implementadas foram: consumo energético, latência média da rede (tempo de inferência a partir do início do processamento de um conjunto de entradas), alocação de memória, custo e erro de quantização na camada de saída. O sistema é capaz de processar somente *forward propagation* e a etapa de treinamento e quantização, que utiliza os algoritmos de *backpropagation* e otimização da rede neural através da otimização por gradiente descendente, foi realizada *offline* em um computador pessoal.

Foi utilizado o *dataset* "*Ailerons*" para regressão com o propósito de validar e testar a implementação. Ele foi obtido por Rui Camacho e disponibilizado por Luís Torgo de forma pública em <https://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html>. Esse conjunto de dados contém 41 atributos, sendo 40 entradas com informações de *status* de uma aeronave F16 e 1 saída com a ação de controle sobre seus *ailerons*. O objetivo desse *dataset* é prever a ação de controle a partir dos dados de entrada. Ele possui 13750 exemplos (linhas) e cada variável de entrada e saída foi normalizada individualmente para treinar e processar a rede *feedforward* de regressão. O *dataset* foi dividido em parte de treino e parte de teste de forma randomizada para treinamento. A parte de teste compõe 10% do total do *dataset* e a de treino, 90%.

A fim de encontrar hiperparâmetros que resultam em uma menor função custo da rede, foi executado um algoritmo de *simulated annealing* sobre a rede neural em ponto flutuante para otimização da topologia. O resultado obtido do algoritmo foi utilizado na implementação realizada no microcontrolador. Após essa otimização, a rede obtida foi quantizada para ser processada em ponto fixo do tipo *signed* de 32, 16, 8 e 4 bits. Os parâmetros de quantização foram ajustados manualmente e testados em uma implementação para computador com a finalidade de evitar problemas de *overflow* e *undeflow* de variáveis e obter resultados com menor erro de quantização antes de processar a rede no microcontrolador. Posteriormente, a implementação foi adaptada para o microcontrolador para realizar os cálculos das operações MAC através de instruções SIMD (*Single Instruction Multiple Data*) executadas no DSP do microcontrolador.

3.1 Materiais e Métodos

Foi utilizada uma placa de desenvolvimento contendo o microcontrolador STM32F407 da família ARM Cortex-M4. Esta linha de microprocessadores foi escolhida, pois os dispositivos têm baixo consumo para utilização em aplicações de eficiência energética, o que os torna muito relevantes em sistemas alimentados por bateria. Além disso, esta é a família mais simples de microprocessadores ARM que possui DSP e FPU embarcados na arquitetura, o que pode diminuir drasticamente o tempo de execução de algoritmos de redes neurais. Os microprocessadores ARM possuem um ecossistema amplo de *softwares* e bibliotecas para facilitar o desenvolvimento do *firmware* de um produto final, tornando sua escolha preferível.

A programação do microcontrolador foi realizada em linguagem C através da IDE (*Integrated development environment*) STM32CubeIDE. Para programar, depurar e testar o *firmware* desenvolvido, foi utilizado um programador ST-LINK V2 (SWD - *Serial Wire Debug*) e um cabo USB para comunicação. A execução do projeto iniciou por uma simulação da topologia da rede *feedforward* em Python através da biblioteca TensorFlow para *machine learning* que foi executada em CPU em um computador pessoal contendo um processador Intel Core i7-6500U de modo a encontrar os hiperparâmetros ideais para a aplicação com a limitação de tamanho de rede no microcontrolador. Foi utilizado o algoritmo *simulated annealing* (detalhado nas seções seguintes) para otimização dos hiperparâmetros a partir da função custo (erro quadrático médio) na camada de saída.

3.1.1 Orçamento

Os custos envolvidos no projeto se resumem à placa de desenvolvimento contendo o microcontrolador STM32F407ZGT6 e ao programador ST-LINK V2. É importante notar que a placa de desenvolvimento possui diversas outras funcionalidades que não serão utilizadas nesse projeto. Portanto, caso fosse elaborado um produto final, o custo seria muito inferior, pois seriam utilizados somente o chip contendo o microcontrolador, dispositivos periféricos e componentes eletrônicos de baixo custo.

- Placa de Desenvolvimento STM32F407ZGT6 ARM Cortex M-4: R$219,00
- Programador ST-LINK V2: R$35,90

3.1.2 Esboço do Sistema

O sistema desenvolvido é capaz de obter os modelos da rede e as entradas através de uma comunicação serial iniciada por um *host*. O tratamento de dados recebidos foi implementado em *firmware* para testar e validar o sistema com o auxílio do programador ST-LINK V2.

Os valores de pesos e os hiperparâmetros do modelo podem ser transmitidos para o microcontrolador por uma comunicação serial utilizando o protocolo USB (*Universal Serial BUS*) através de um *driver* USB e um *middleware* em *firmware* utilizado para criar uma porta serial virtual no microcontrolador. Esses valores são armazenados na memória flash do microcontrolador. A parte de teste do *dataset* "*Ailerons*" também poderá ser obtida através dessa comunicação e armazenada na memória flash para realizar os testes e obter os resultados do processamento.

Foi desenvolvida uma interface simples para comunicação serial cujo código para carregamento de dados utilizado na comunicação foi parcialmente implementado no *middleware* gerado pela IDE STM32CubeIDE. Os dados são obtidos via USB e gravados em seções da memória flash. Parte desse código também é responsável por carregar os dados gravados em flash na memória SRAM em *structs* para processamento da rede com maior desempenho. O código utilizado para processamento dos modelos de rede em ponto fixo e ponto flutuante que estão armazenado na memória flash utiliza o DSP e FPU e acessa somente a memória SRAM. Após carregar o modelo em ponto flutuante ou ponto fixo de 32, 16, 8 ou 4 bits, é possível iniciar a inferência da parte de teste do *dataset* através de comandos enviados pelo host. A Figura 7 mostra um esboço do sistema.

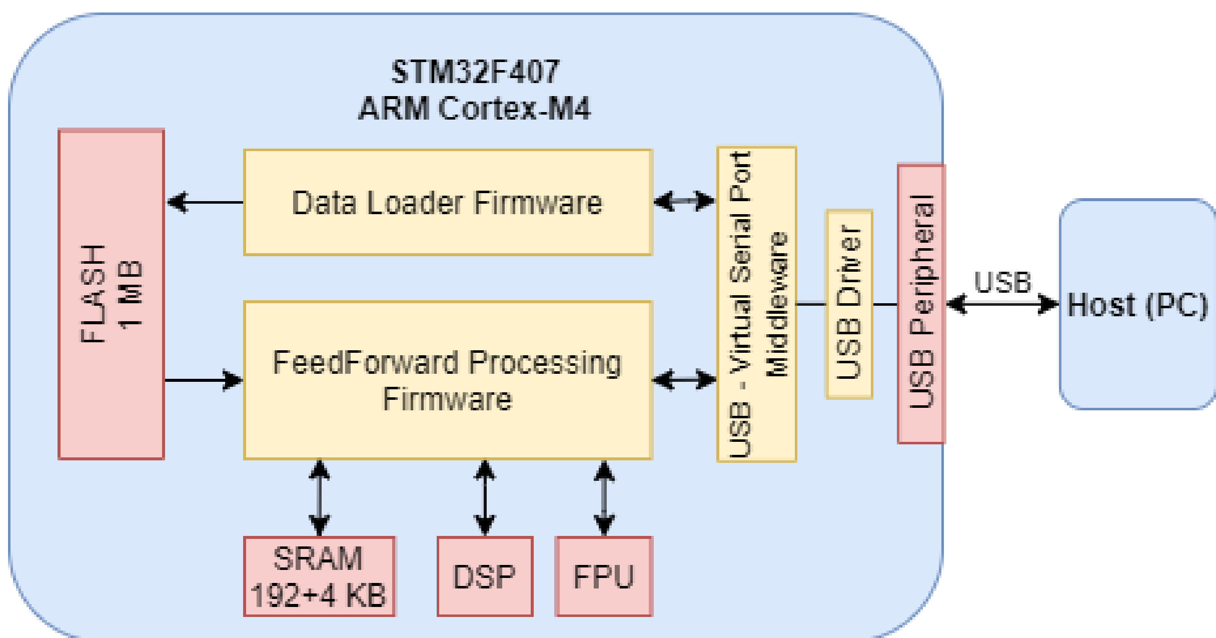

Figura 7 – Esboço do Projeto.

3.1.3 Configurações do Projeto e Alocação de Memória

É importante notar que o microcontrolador tem limitação de memória, e, para fins de maior performance, é ideal que os valores *bias* e pesos sejam armazenados na SRAM. O modelo de microcontrolador ARM Cortex-M4 selecionado possui 192Kbytes

de SRAM para uso geral, no qual 64Kbytes são SRAM CCM (*Core Coupled Memory*) e 128Kbytes são de SRAM regular, cujos barramentos de memória estão indicados na figura 8. De acordo com a STMicroelectronics (STMICROELECTRONICS, 2019), quando o código está localizado na SRAM CCM e os dados são armazenados na SRAM regular, o núcleo do Cortex-M4 está na configuração Harvard ideal. Esse princípio foi seguido e foram atribuídos 80Kbytes da SRAM regular para pesos e valores *bias* e o restante para uso geral dos algoritmos relacionados. Ao considerar o processamento em ponto flutuante, cada valor de peso e *bias* ocupa 32 bits de memória, portanto 80KBytes ($80 \times 1024 \times 8$ bits) permitem o armazenamento de 20480 valores:

$$Número_máximo_de_valores = 20480. \tag{3.1}$$

Para armazenar e executar o código na memória SRAM CCM do microcontrolador, foi necessário realizar duas modificações de arquivos gerados pela IDE STM32Cube:

- *Linker Script* (STM32F407ZGTX.ld): Alteração para indicar a seção de memória em que o *firmware* será armazenado.
- *Startup File* (startup_stm32f407zgtx.s): Alteração para copiar o *firmware* da memória flash na SRAM CCM antes de iniciar sua execução quando o microcontrolador for alimentado.

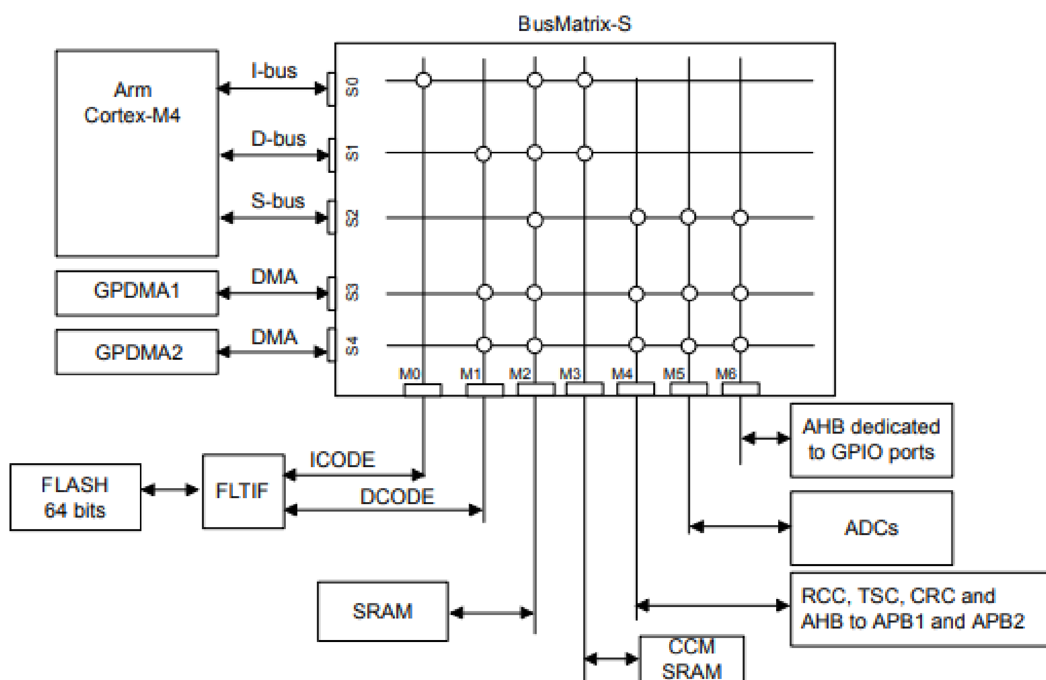

Figura 8 – Matriz BUS dos microprocessadores Cortex-M da STMicroelectronics. Fonte:(STMICROELECTRONICS, 2019)

Foram alocadas seis seções de memória flash do microcontrolador para armazenar os modelos e os exemplos do *dataset* que foram utilizados para avaliação. É possível armazenar um modelo em ponto fixo e um modelo em ponto flutuante simultaneamente em memória não volátil, pois estes são armazenados em seções diferentes. Cada seção possui 128KBytes e foram destinadas 4 seções para a parte de teste do *dataset* (0x08040000-0x080BFFFF), 1 seção para o modelo em ponto flutuante (0x080C0000-0x080DFFFF) e 1 seção para o modelo em ponto fixo e seus parâmetros (0x080E0000-0x080FFFFF).

A frequência de *clock* do microprocessador foi configurada em 168 MHz para atingir maior desempenho em velocidade de processamento do *firmware*. Além disso, a compilação do código foi configurada para ter maior otimização e executar o código no menor tempo possível (figura 9).

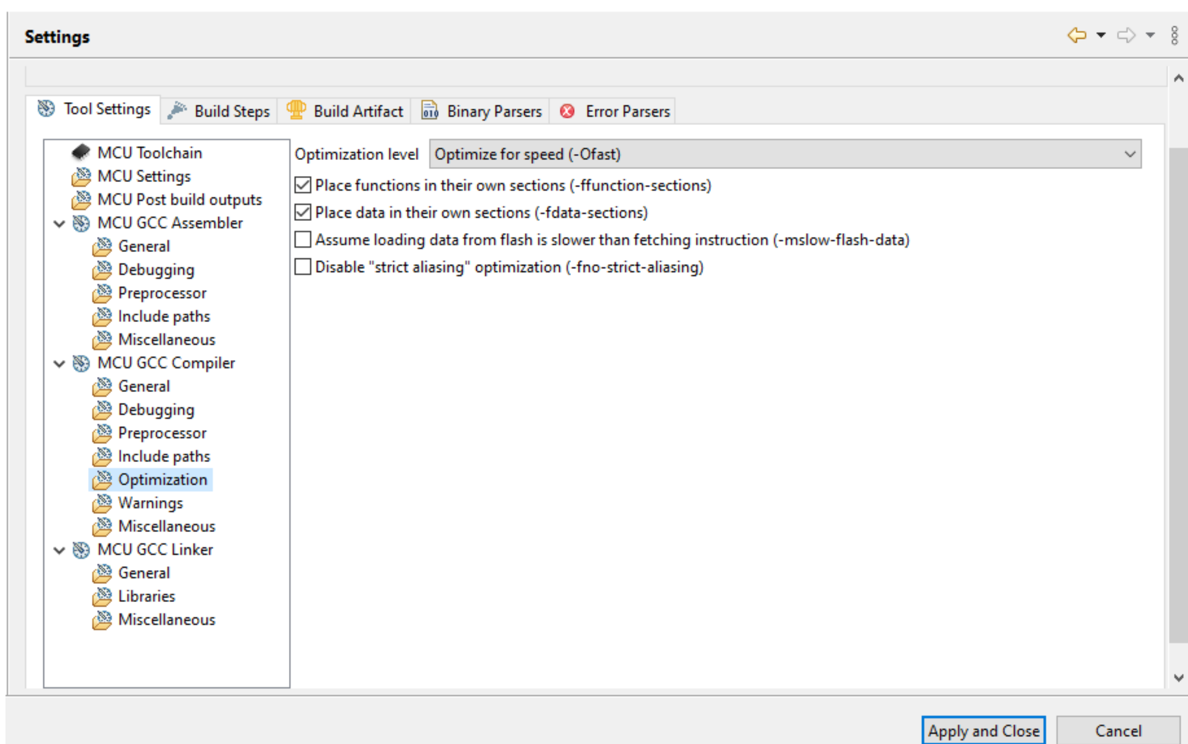

Figura 9 – Configuração de otimização da IDE STM32Cube utilizada para a compilação do código.

3.1.4 Rede *Feedforward*

O sistema para processamento de redes neurais *feedforward* de regressão utiliza, para as camadas escondidas da rede, a função de ativação *Leaky* ReLU com $K = 0,1$ (equação 2.8), pois do ponto de vista de hardware é a mais simples e acelera o treinamento em relação à ReLU devido à sua derivada em $x < 0$ ser diferente de zero (GOODFELLOW; COURVILLE, 2016). A ativação da camada de saída utilizada é uma função linear $g(x) = x$, pois a saída deverá ser, teoricamente, um valor numérico contínuo. Para realizar

o treinamento da rede, foi utilizada a inicialização He & Xavier normal, que determina a variância das distribuições de valores inicializados igual ao inverso do *fan-in* da camada.

Os hiperparâmetros "número de camadas escondidas" e "neurônios de cada camada" foram determinados através do algoritmo de *simulated annealing*, dadas as restrições de quantidade de pesos e *bias*. A saída de uma camada l que contém N_l neurônios é calculada a partir de uma matriz que contém $N_l \cdot N_{l-1}$ pesos e de um vetor *bias* que contém N_l valores. Portanto, dado o número máximo de valores de peso e *bias* (equação 3.1), é possível obter a relação de restrição de neurônios através do somatório de valores das matrizes de pesos e valores *bias* (equação 3.2). N_i é o número de neurônios da camada "i", e n_{out}, o índice da camada de saída:

$$\sum_{i=1}^{n_{out}} N_i \cdot (1 + N_{i-1}) \leq 20480. \tag{3.2}$$

3.1.5 *Simulated Annealing*

Para encontrar hiperparâmetros ótimos, foi executado o algoritmo de *simulated annealing*, que foi adaptado para otimizar o número de neurônios em cada camada da rede neural *feedforward*. Foram simuladas redes, separadamente, com diferentes números de camadas em Python através da biblioteca TensorFlow. Os números de neurônios das camadas de cada rede foram incrementados ou reduzidos por um conjunto de variáveis inteiras pseudoaleatórias representadas por δS, tal que a condição imposta na equação 3.2 seja satisfeita. Essa alteração no número de neurônios das camadas só é efetivamente realizada se uma das seguintes condições for satisfeita:

- A função objetiva (custo da rede) tem um valor menor depois da alteração.
- Uma variável aleatória **R** gerada entre 0 e 1 satisfaz a condição do algoritmo de *simulated annealing*.

A condição do algoritmo de *simulated annealing* é dada pela equação 3.3, derivada do algoritmo utilizado por Rasdi Rere (RERE; FANANY; ARYMURTHY, 2015) para otimização de redes neurais. A variável S representa o conjunto de números de neurônios inicial e L, a função custo médio da rede (erro quadrático médio). O valor k representa uma constante que deve ser ajustada antes de iniciar o algoritmo e T é a temperatura de *annealing* que tem um valor inicial e decresce ao longo das iterações. Esse método de otimização pode ser eficiente para evitar mínimos locais da função custo que não sejam ideais, pois há a probabilidade de alterar a rede mesmo que a função custo aumente. δS representa o conjunto de variáveis aleatórias de inteiros para alteração de número de

neurônios nas camadas. A figura 10 mostra o esquema de funcionamento desse algoritmo, em que a função objetiva é o custo da rede:

$$R < exp\left(-\frac{L_{(S+\delta S)} - L_{(S)}}{kT}\right). \tag{3.3}$$

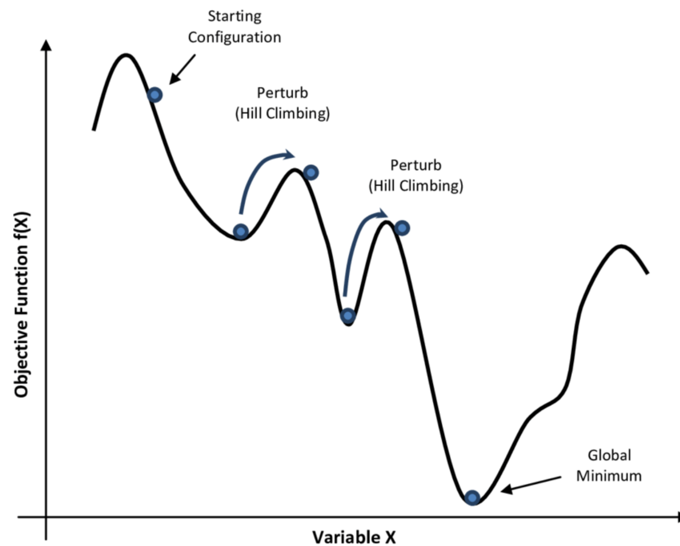

Figura 10 – Esquema de funcionamento do algoritmo de *simulated annealing*. Fonte: (CAPARRINI, 2019)

O algoritmo de otimização utilizou uma rampa de temperatura com descida pausada em uma certa etapa. O passo de descida por iteração utilizado tem valor igual a 1. Dessa forma, a temperatura começa alta e diminui gradativamente até certo ponto, quando ela é pausada. A execução das iterações continua com a mesma temperatura e, próximo ao fim da otimização, a temperatura volta a diminuir até atingir o valor igual a 1.

A inicialização de pesos He & Xavier utilizada resulta, após o treinamento, em uma rede com função custo com variância considerável, o que prejudica o funcionamento correto do algoritmo. Para contornar esse problema e obter um resultado do algoritmo com maior reprodutibilidade, foram realizados vários treinamentos individuais truncados (poucas epochs) para cada iteração do algoritmo e o custo médio resultante dos treinamentos foi considerado para a iteração.

3.2 Desenvolvimento

3.2.1 Processamento em Ponto Flutuante

O processamento em ponto flutuante utiliza a FPU embarcada no chip do microcontrolador. O compilador utilizado na IDE STM32Cube implementa automaticamente

qualquer operação de ponto flutuante para ser realizada por instruções da FPU e não há necessidade de realizar modificações no código escrito em linguagem C para isso. Essa unidade de processamento possui diversas instruções para processar ponto flutuante de precisão simples (32 bits) de acordo com o padrão IEEE 754 de aritmética de ponto flutuante. A maior parte dessas instruções são realizadas em um único ciclo de *clock* na arquitetura do microprocessador, porém a instrução mais relevante para o cálculo do produto da matriz de pesos pela entrada de uma camada de neurônios, FMAC (*fused multiply–accumulate*), é realizada em 3 ciclos de *clock* (LEWIS, 2016). O código para processamento em ponto flutuante foi inicialmente desenvolvido em implementação para computador e, em sequência, transferido para o microcontrolador. Ele utiliza *structs* de parâmetros e alocação dinâmica de variáveis de saídas dos neurônios e de pesos e *bias* para minimizar a alocação de memória. A figura 11 mostra o fluxograma simplificado do algoritmo.

A variável "layers" na figura 11 representa o número de camadas da rede. Nessa abordagem, uma rede *feedforward* com 6 camadas, por exemplo, contém 1 camada de entrada, 4 camadas escondidas e 1 camada de saída.

A função de processamento das entradas de uma determinada camada e a forma com que as variáveis de peso e *bias* foram implementadas seguem a forma matricial da equação 2.4. O número de linhas da matriz peso transposta de uma determinada camada é igual ao número de neurônios da mesma, e o número de colunas é igual ao número de neurônios da camada anterior. Para minimizar a alocação de memória e facilitar a transferência de dados, todos os valores de peso e *bias* foram estruturados e organizados em um único vetor alocado dinamicamente. A figura 12 mostra o fluxograma do algoritmo utilizado para processar as entradas de uma camada de neurônios. Inicialmente, são calculadas as variáveis "index_w", "index_b", "rows" e "columns" que representam os índices em que iniciam os valores de peso e *bias* da camada no vetor alocado, o número de linha e o número de colunas da camada, respectivamente. Uma variável auxiliar é inicializada em cada iteração do loop com o valor de bias do neurônio que será processado. O resultado das operações MAC é acumulado nessa variável e o valor é copiado para outra uma variável do neurônio no fim da iteração, que representa o valor Z na equação 2.1.

A ativação *leaky* ReLU simplesmente realiza uma comparação do resultado do processamento das entradas de uma camada. Caso o resultado para um neurônio seja menor que zero, o mesmo é multiplicado por K (equação 2.8). Por fim, o valor é copiado para o vetor de saídas da camada.

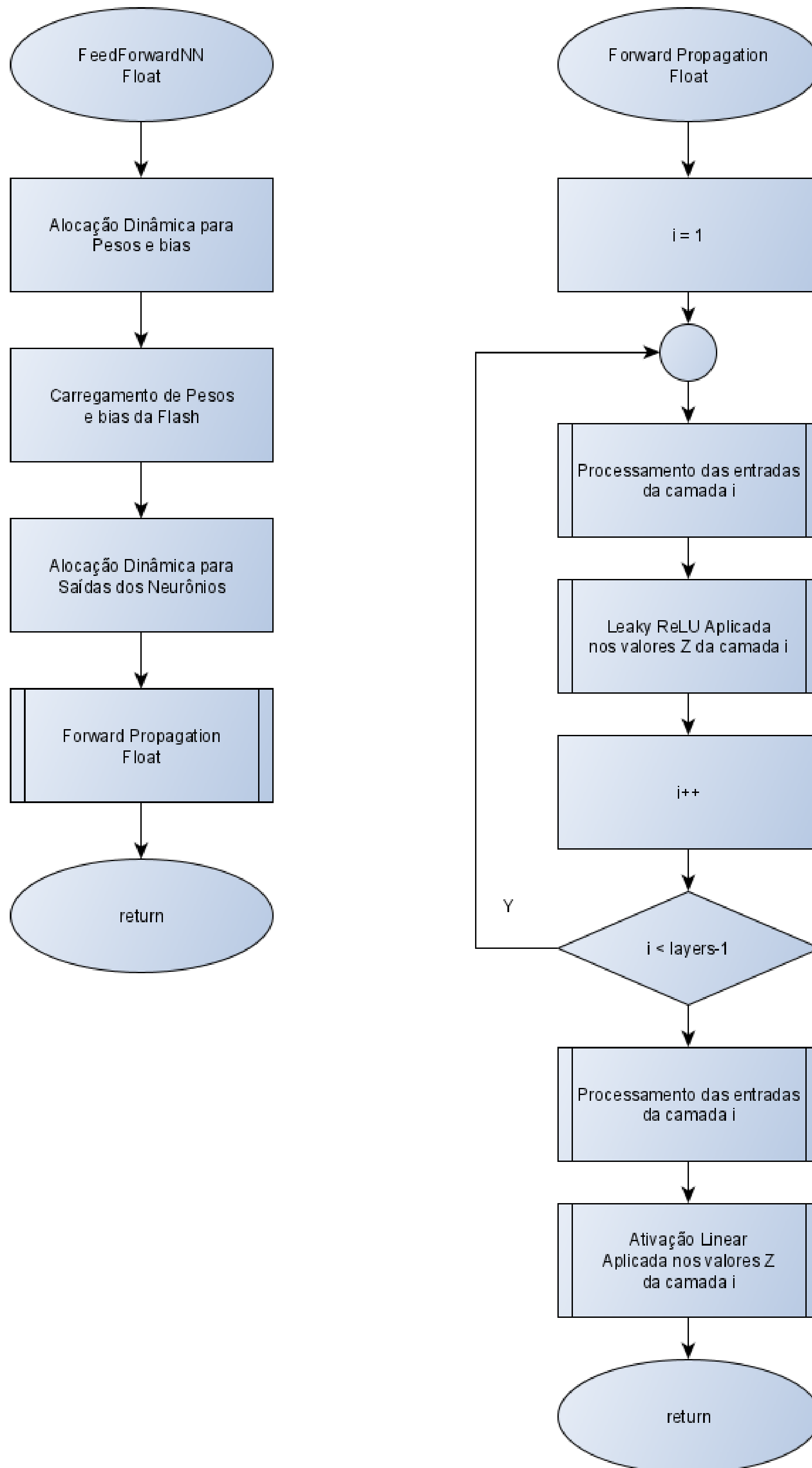

Figura 11 – Estrutura do algoritmo para processamento em ponto flutuante.

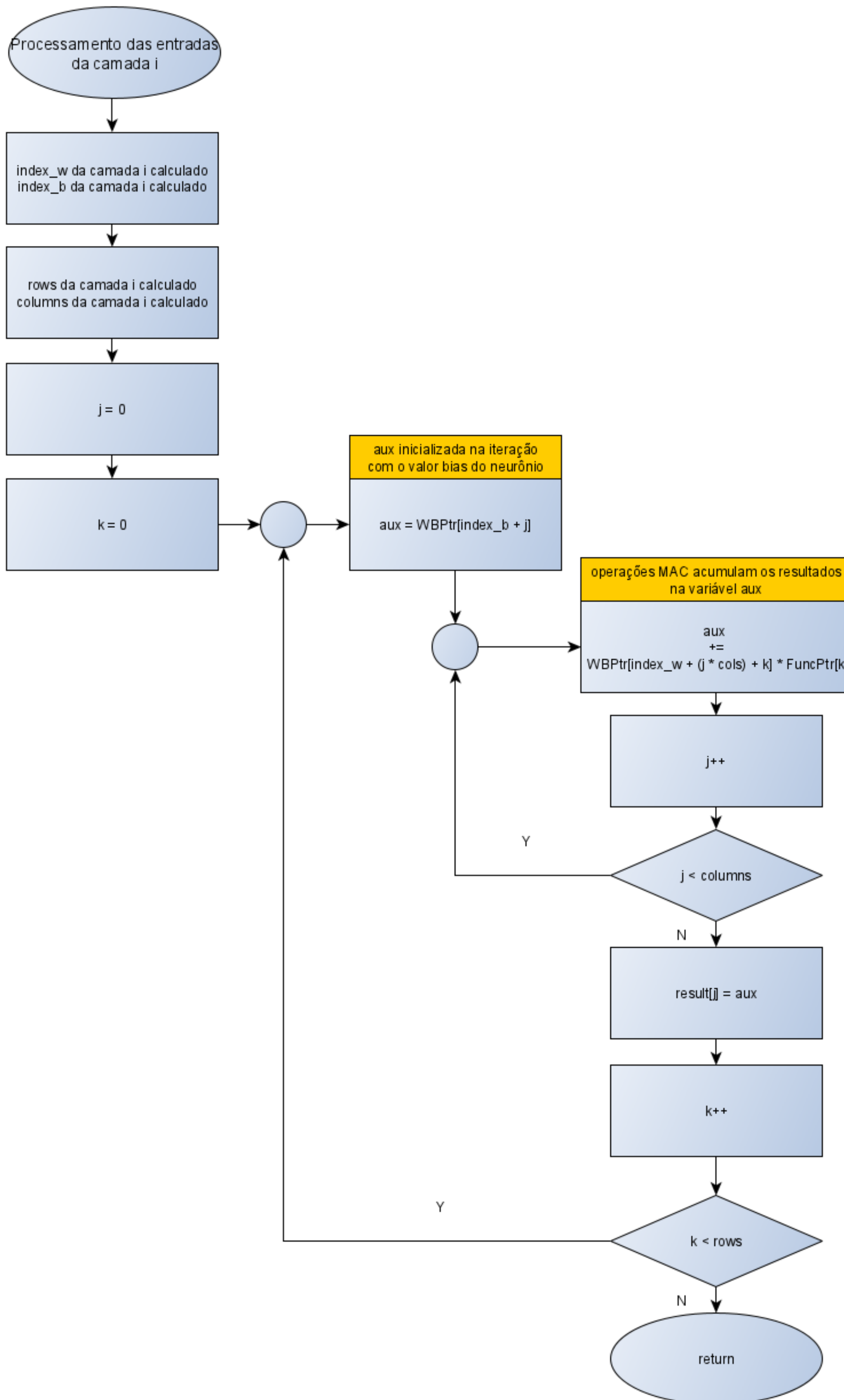

Figura 12 – Fluxograma do algoritmo de processamento das entradas de uma camada de neurônios.

3.2.2 Quantização de Rede Treinada

O processo de quantização da rede otimizada e treinada seguiu as equações 2.24, 2.25, 2.26, 2.27, 2.28 e 2.29 descritas na seção de quantização do referencial teórico. Essa etapa foi realizada em um computador após treinar as redes otimizadas. As distribuições de saídas de cada camada, de pesos e de *bias* da rede foram analisadas para determinar os parâmetros de quantização. Em seguida, o modelo foi quantizado e seus pesos, bias e parâmetros foram salvos em arquivo binário para ser transferido para o microcontrolador via comunicação serial. Então, a rede foi simulada em computador e a parte de teste do *dataset* foi executado para avaliar o erro de quantização médio de cada camada. Para isso, foi utilizada a equação 3.4 que descreve o erro quadrático médio de uma camada L a partir dos valores de saída $f_{i,j}$ dos neurônios dessa camada processados em ponto flutuante e dos valores $f^q_{i,j}$ processados em ponto fixo, que podem ser desquantizados através da equação 2.24. A variável N_L representa o número de neurônios na camada L, e $N_{dataset}$, o tamanho da parte de teste do *dataset*.

$$MSE_L = \frac{1}{N_L N_{dataset}} \sum_{j}^{N_{dataset}} \sum_{i}^{N_L} \left[S^L_f (f^q_{i,j} - z^L_f) - f_{i,j} \right]^2. \tag{3.4}$$

3.2.3 Processamento em Ponto Fixo

O algoritmo para processamento da rede quantizada em ponto fixo utiliza as equações 2.33 e 2.34. A partir dele, é possível processar a rede quantizada em ponto fixo do tipo *signed* com precisões de 32, 16, 8 e 4 bits. Nessa implementação, todas as variáveis da rede possuem a mesma precisão em ponto fixo, exceto alguns parâmetros de quantização e variáveis auxiliares temporárias utilizadas para evitar underflow e *overflow* nas operações MAC. O algoritmo foi implementado em computador e, após sua validação, foi feita uma adaptação do código para executar as operações MAC no DSP do microcontrolador. A figura 13 mostra o fluxograma resumido do algoritmo de processamento em ponto fixo da rede quantizada. Inicialmente, é feita a alocação dinâmica de um ponteiro do tipo void que armazenará os valores de pesos e *bias* na precisão que será utilizada. Em seguida, os valores armazenados na memória flash são carregados na SRAM através desse ponteiro. Antes de iniciar o processamento da rede, há uma alocação dinâmica para as variáveis de saída das camadas da rede e é efetuada a quantização dos valores da camada de entrada da rede. Sempre que uma operação for realizada com essas variáveis alocadas de forma dinâmica, é realizado um *typecast* para a precisão utilizada em ponto fixo.

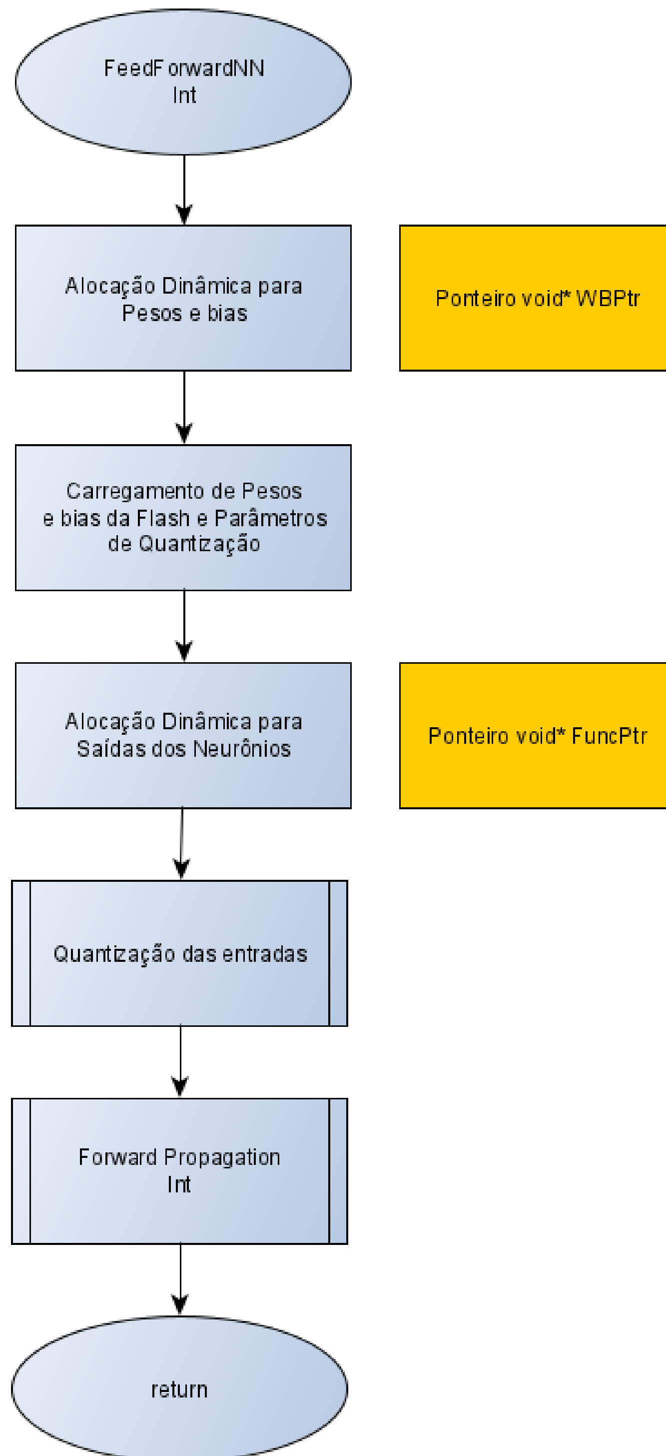

Figura 13 – Fluxograma resumido do algoritmo de processamento em ponto fixo da rede quantizada.

A quantização das entradas é realizada de acordo com a equação 2.25. Para evitar *overflow* e *underflow* das entradas nesse processo, é realizado um teste de saturação em que os valores que estiverem acima do valor máximo ou abaixo do de valor mínimo de quantização serão saturados com esses valores de extremos. A figura 14 mostra o fluxograma desse algoritmo.

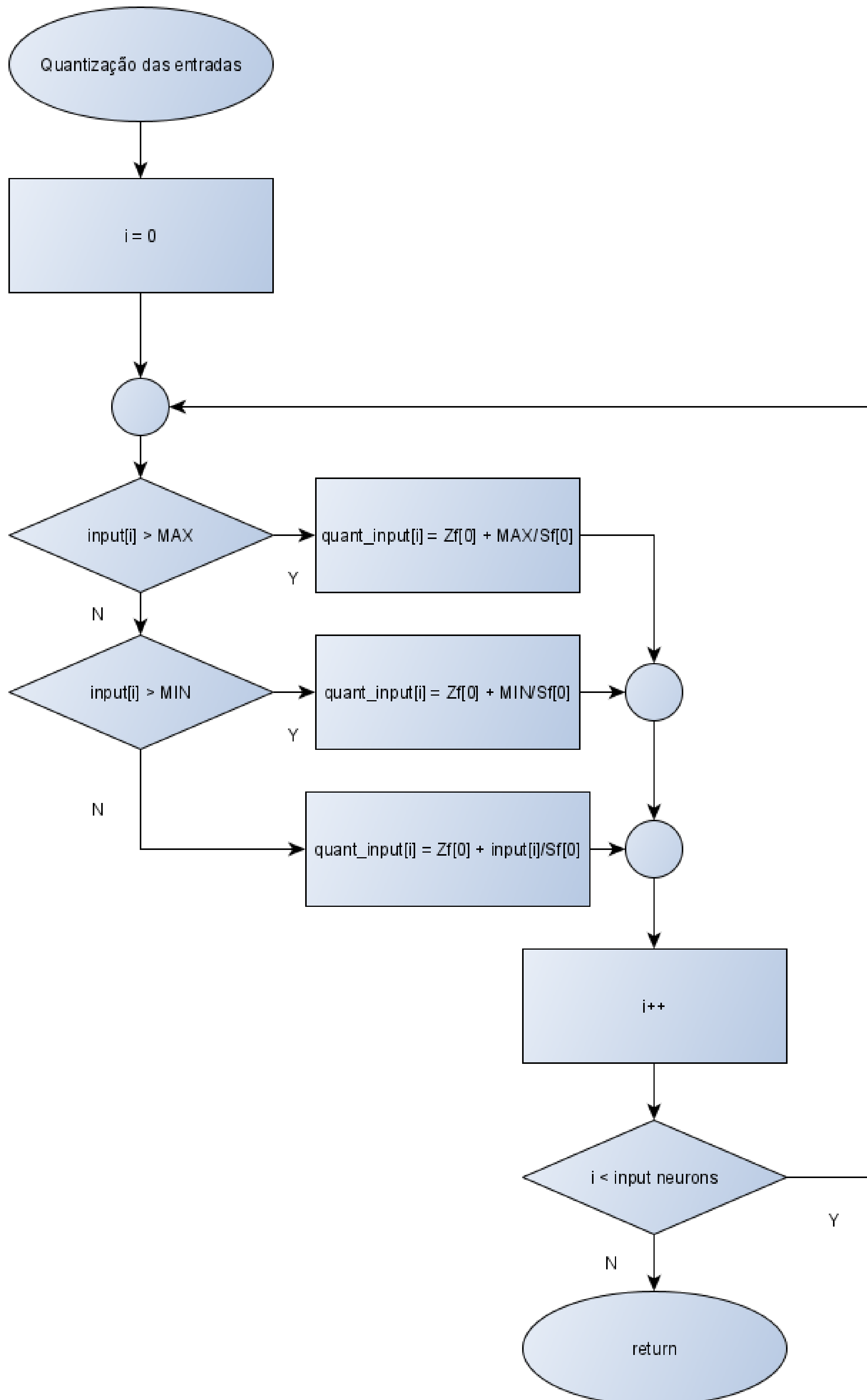

Figura 14 – Fluxograma do algoritmo de quantização.

O processamento em ponto fixo *signed* do algoritmo de *forward propagation* foi implementado em computador e adaptado para microcontrolador em sequência. Foram implementadas instruções de DSP na seção do código responsável pelas operações MAC. A figura 15 mostra as instruções SIMD da biblioteca CMSIS realizadas no DSP que foram utilizadas para processamento da rede em ponto fixo:

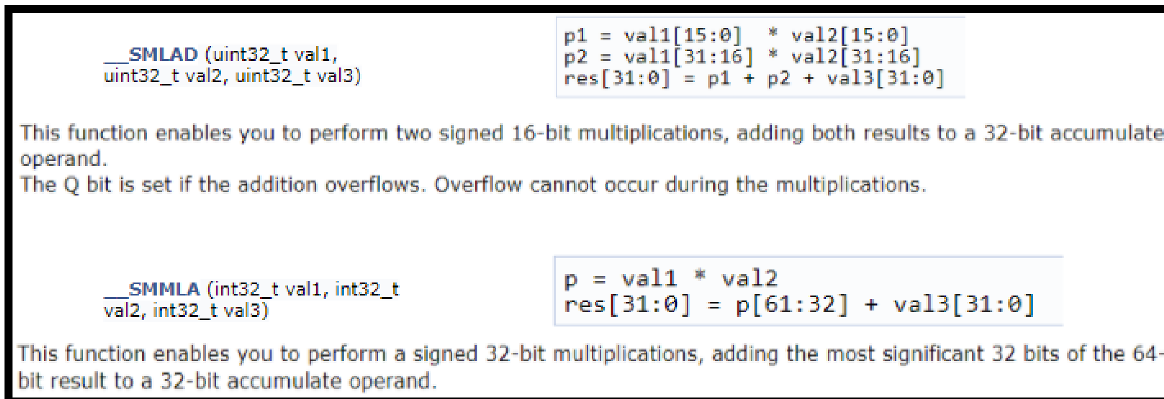

Figura 15 – Instruções SIMD da biblioteca CMSIS para ARM Cortex-M utilizadas no processamento de rede *feedforward* em ponto fixo. Fonte: (ARM, 2020)

A mesma estrutura apresentada no fluxograma da figura 11 foi utilizada para o processamento em ponto fixo de *forward propagation*. O processamento das entradas de uma camada através de operações MAC é responsável por efetuar o cálculo do argumento da função de ativação da equação 2.33. Para as entradas de uma camada em 32 bits, foi utilizada a instrução SIMD SMMLA. Já para 16 e 8 bits, a foi utilizada a instrução SMLAD. A implementação para ponto fixo *signed* de 4 bits foi tratada de forma especial, pois a linguagem C não suporta esse tipo de variável não padronizada. Portanto, nessa abordagem, foi definido seu valor mínimo (b1111) e máximo (b0111) iguais a -7 e 8, respectivamente. Além disso, foi utilizado o tipo de variável int8_t e a instrução ___SMLAD para tratar operações desse tipo de variável. Visando minimizar alocação de memória, duas variáveis de 4 bits são lidas e armazenadas no mesmo byte através de uma função que organiza e manipula esses 8 bits para leitura e escrita dessas duas variáveis. Em caso de *underflow* e *overflow* do valor acumulado no resultado das operações MAC, ao fim do processamento das entradas de uma camada, é efetuada a saturação desse valor para o máximo ou mínimo da precisão utilizada. A figura 16 mostra o fluxograma do algoritmo utilizado para realizar o processamento das entradas de uma camada em 32 bits através da instrução SMMLA. Para utilizar corretamente a instrução SMLAD com variáveis em ponto fixo de 16, 8 e 4 bits, foi necessário manipular e concatenar os pares de variáveis. Dessa forma, são realizadas duas operações MAC em uma única instrução que é executada em 1 ciclo de *clock*. A figura 17 mostra o fluxograma do algoritmo utilizado para o processamento das entradas de uma camada em ponto fixo de 16, 8 ou 4 bits.

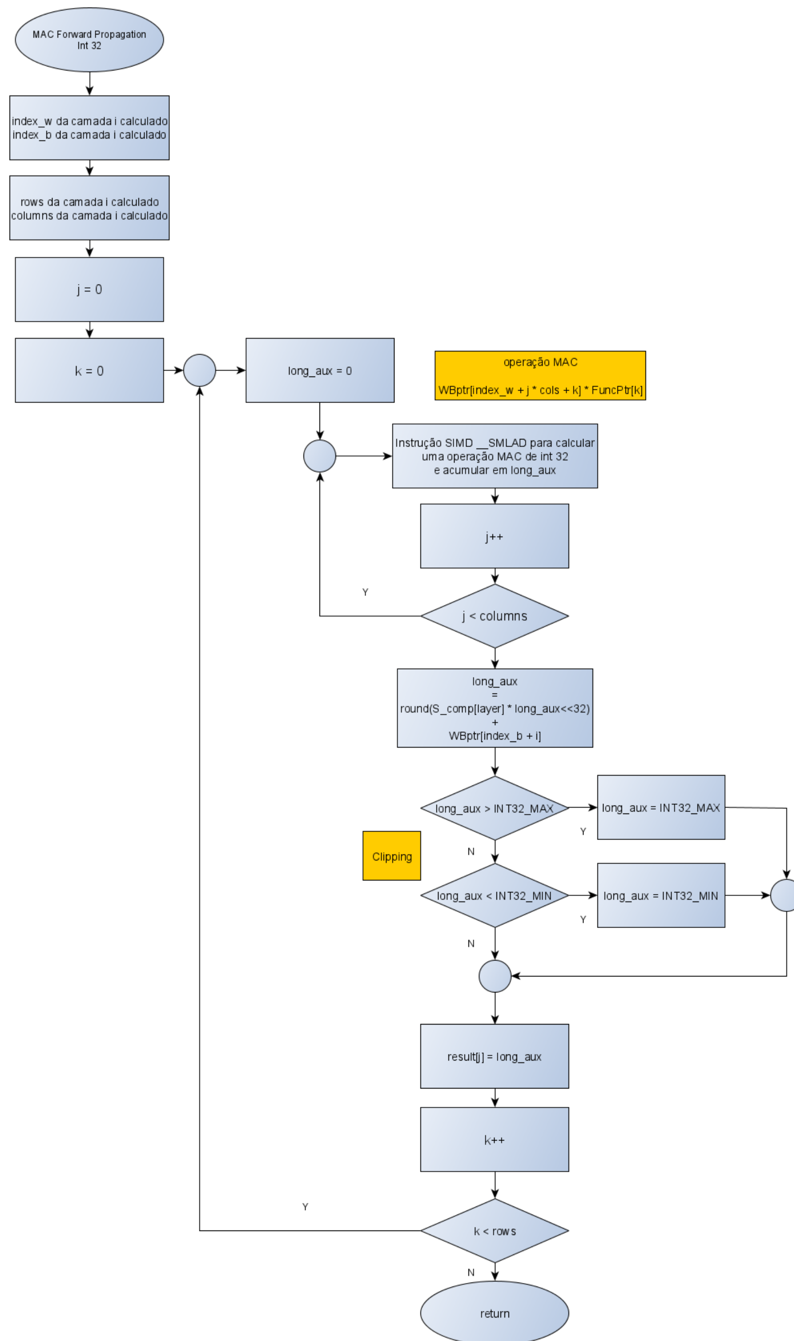

Figura 16 – Fluxograma do processamento de entradas de uma camada em ponto fixo de 32 bits.

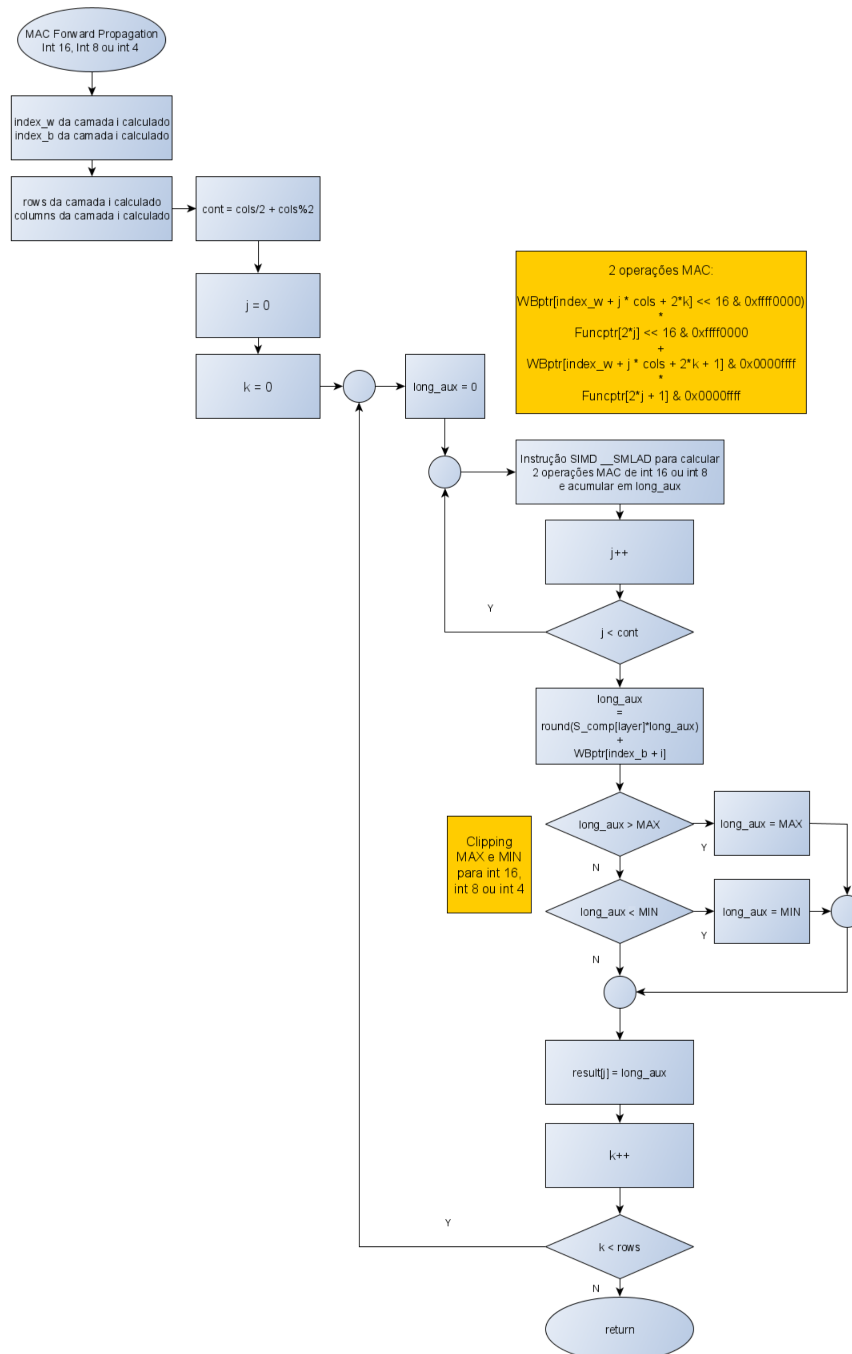

Figura 17 – Fluxograma do processamento de entradas de uma camada em ponto fixo de 16 ou 8 bits.

Após processar as entradas de uma camada, é efetuada a aplicação da função de ativação *leaky* ReLU através de uma comparação do sinal do resultado. Em sequência, o termo z_f (equação 2.33) é somado na saída do neurônio. A figura 18 mostra o fluxograma dessa etapa, em que a saída é saturada para os extremos da precisão utilizada se houver *overflow* ou *undeflow*. A aplicação da função de ativação linear na camada de saída da rede segue a mesma lógica do fluxograma:

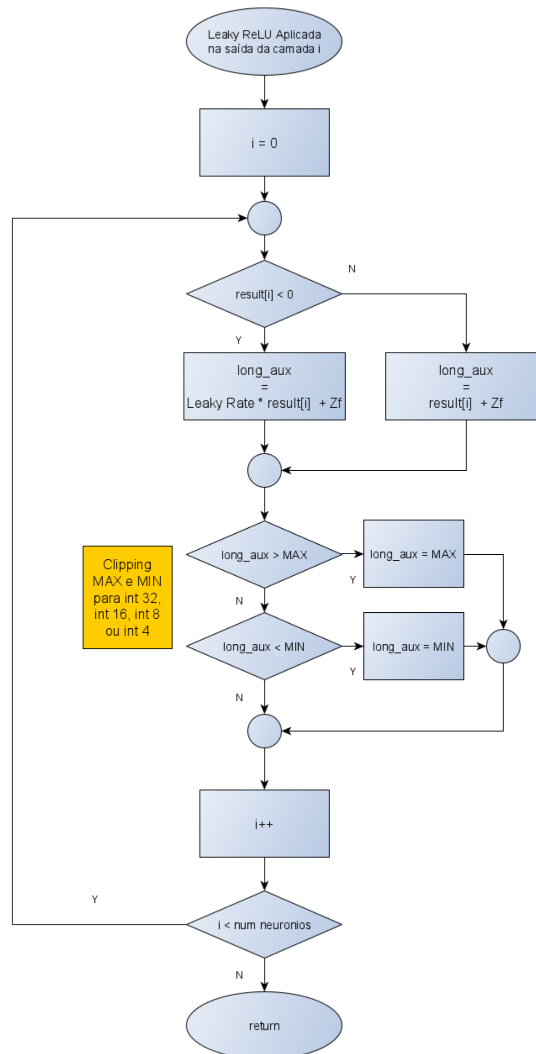

Figura 18 – Fluxograma do processamento da ativação *leaky* ReLU na saída de uma camada da rede.

3.3 Comunicação Serial via USB

Uma interface simples de comunicação serial foi desenvolvida com comandos para envio de modelo em ponto fixo ou flutuante, envio de parte do dataset, seleção de modelo e inicio de processamento. Os comandos disponíveis estão listados e descritos na tabela 1. Eles são enviados através de um pacote de mensagem USB e são codificados em bytes de caracteres da tabela ASCII. O primeiro byte da mensagem representa o comando enviado e o dado do comando começa a partir do terceiro byte da mensagem. Foi utilizado o programa RealTerm para envio de comandos seriais e para validar o sistema.

Byte (ASCII)	Dado	Descrição	Resposta
S	Tamanho em Bytes	Envia ao microcontrolador a informação de tamanho do arquivo que será transferido	Tamanho de arquivo em Bytes
D	Tipo de Dado: float model int model dataset	Envia ao microcontrolador o tipo de dado que será transferido	Tipo de Dado Reconhecido
I	-	Sinaliza início de envio de arquivo	Seções da flash Apagadas e Mensagem de Arquivo Recebido.
M	Tipo de Modelo: float int 32 int 16 int 8 int 4	Informa ao microcontrolador o tipo de modelo que se deseja processar	-
R	Índice inicial e número de inferências	Informa ao microcontrolador o índice do exemplo a partir do qual o dataset será processado e o número de inferências que serão processadas	Tempo Decorrido

Tabela 1 – Tabela de comandos da interface implementada no sistema.

3.4 Obtenção dos Dados para Avaliação

O custo da rede quantizada (erro quadrático médio), a alocação de memória e os dados de erro de quantização nas camadas foram obtidos na implementação em computador antes de ser adaptada para o microcontrolador. O custo da rede nas duas implementações deve ser o mesmo, pois os valores de saída inferidos foram verificados para conjuntos de entradas da rede em ponto flutuante e em ponto fixo de 32, 16 8 e 4 bits tanto no *firmware* executado no microcontrolador quanto na implementação em computador e os valores foram idênticos para as duas implementações.

Os dados da implementação no microcontrolador foram obtidos ao executar os algoritmos desenvolvidos e adaptados para a arquitetura utilizada. O *firmware* foi executado no microcontrolador para processar a parte de teste do *dataset* com as diferentes redes *feedfoward* (quantizadas e em ponto flutuante). O tempo de execução por inferência foi medido através de um timer implementado em *firmware* e os dados de consumo energético foram obtidos ao medir a corrente elétrica média consumida e a tensão de alimentação com um multímetro de bancada Keithley 2110. Para a medida de consumo, a placa

foi alimentada por um pino de alimentação de 3,3 Volts do programador ST-LINK V2 utilizado para programar e depurar o *firmware* e o multímetro foi ligado em série no modo amperímetro digital. A potência média pôde ser obtida através do produto da corrente elétrica média pela tensão de alimentação. A partir do tempo de execução e potência média, foi possível obter o consumo energético em Joules por inferência processada pelo microcontrolador.

Parte III

Resultados

4 Resultados

Inicialmente, foram obtidos os resultados da etapa de otimização da rede neural através do algoritmo de *simulated annealing*. Esses dados foram analisados e, após, foram selecionados dois conjuntos de hiperparâmetros para realizar os testes da implementação no microcontrolador. As redes foram, então, treinadas e processadas em computador em ponto flutuante e as distribuições de valores de pesos, *bias* e saídas das camadas foram analisadas para determinar os parâmetros de quantização. Em seguida, foi efetuada a quantização da mesma. A parte de teste do *dataset* e os modelos em ponto flutuante e em ponto fixo de 32, 16, 8 e 4 bits foram transferidos para a implementação no microcontrolador e os algoritmos de inferência foram executados para obter os dados para avaliação.

4.1 Otimização por *Simulated Annealing*

O algoritmo de otimização realizou o treinamento truncado por SGD (*stochastic gradient descent*) das redes por 50 epochs com *batch size* igual a 256 e taxa de aprendizado igual a 0,01 com função custo erro quadrático médio. Ele foi executado em Python através da biblioteca TensorFlow. Para cada iteração foram realizados 100 treinamentos com os mesmos hiperparâmetros e a média dos custos resultantes da parte de teste do *dataset* foi considerada para a iteração. O parâmetro k da equação 3.3 utilizado possui valor igual a 0,000025 e a temperatura iniciou em 500. Foi realizada uma rampa de descida da temperatura até atingir 100, quando, então, a descida foi pausada. Após 500 iterações, a temperatura voltou a diminuir até chegar a 1, quando o algoritmo chega ao fim.

As redes iniciaram com 16 neurônios em cada camada no algoritmo e o conjunto de variáveis inteiras δS da equação 3.3, que representa as variações de número de neurônios em cada iteração do algoritmo, foi gerado de forma aleatória com valores de -3 a 3 para cada camada com probabilidades iguais.

As figuras 19, 21, 23, 25 e 27 mostram os resultados obtidos através desse algoritmo. Analisando os gráficos das funções custos, é possível notar que, quanto maior o número de camadas escondidas, mais eficiente o algoritmo tende a ser para otimizar o custo em função do número de neurônios em cada camada. A tabela 2 mostra os resultados da execução do algoritmo para as redes com 2, 3, 4, 5 e 6 camadas escondidas.

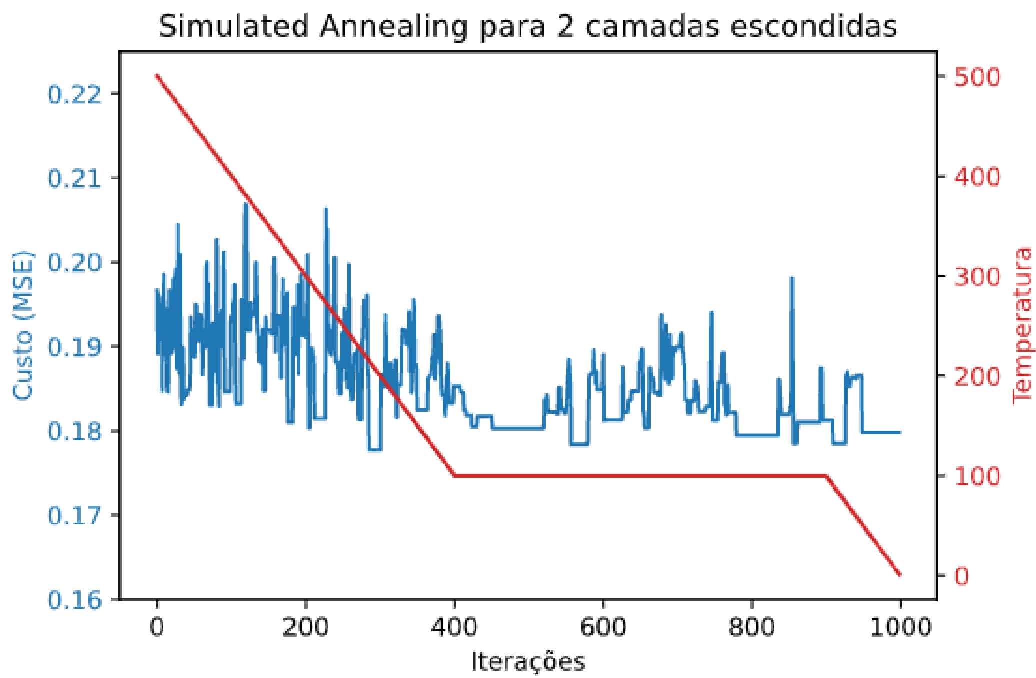

Figura 19 – Otimização do custo da rede com 2 camadas escondidas por *simulated annealing*.

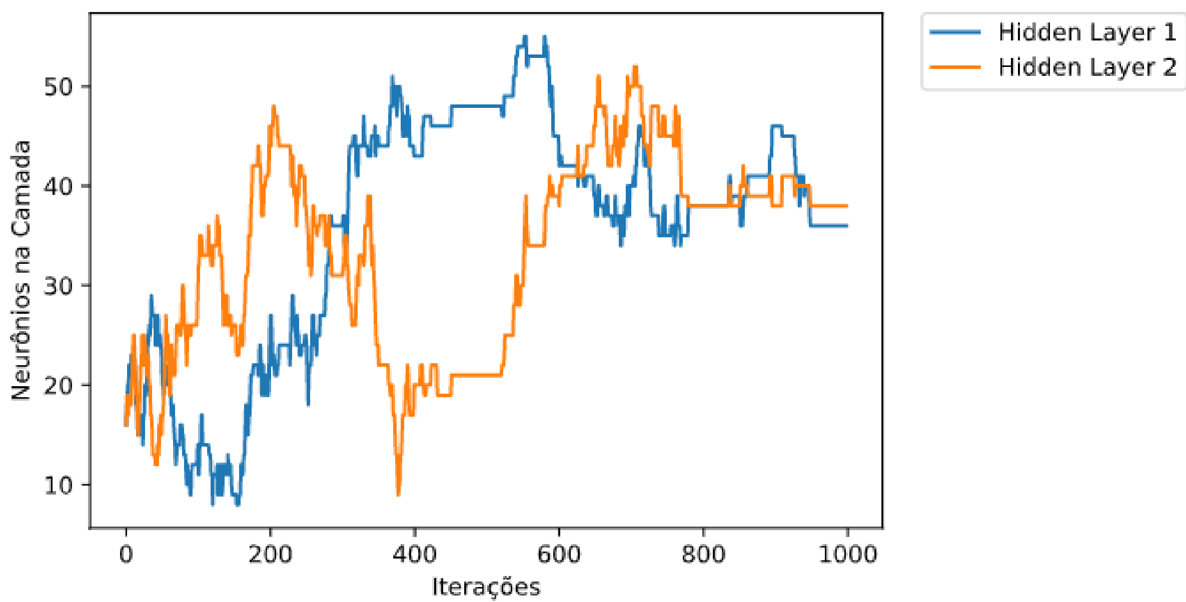

Figura 20 – Número de neurônios na rede com 2 camadas escondidas ao longo da execução do algoritmo de *simulated annealing*.

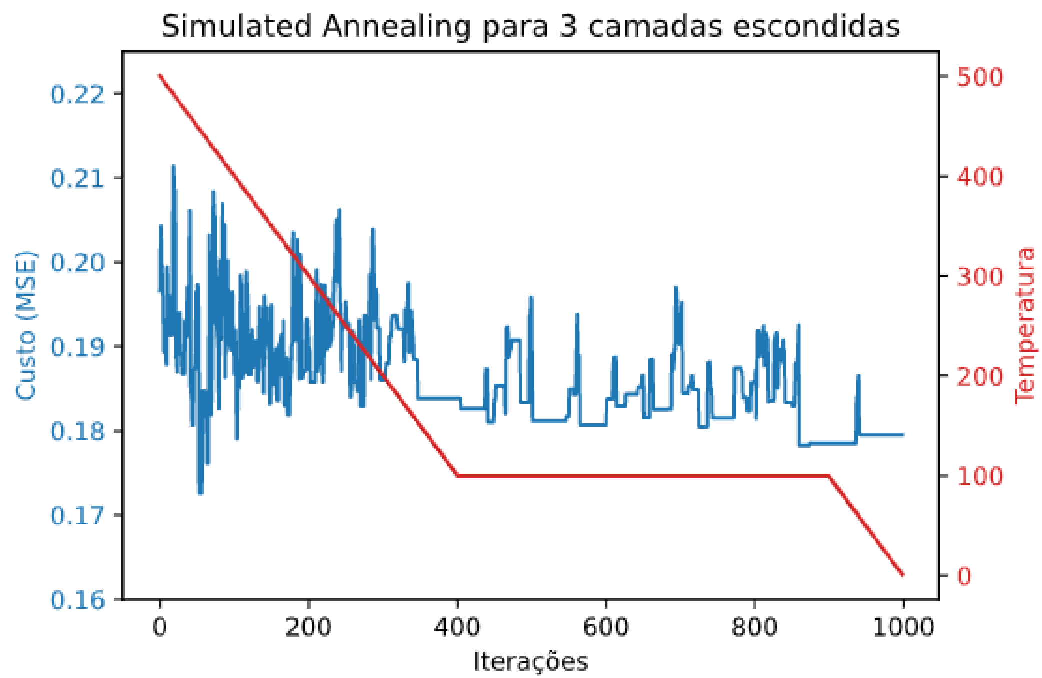

Figura 21 – Otimização do custo da rede com 3 camadas escondidas por *simulated annealing*.

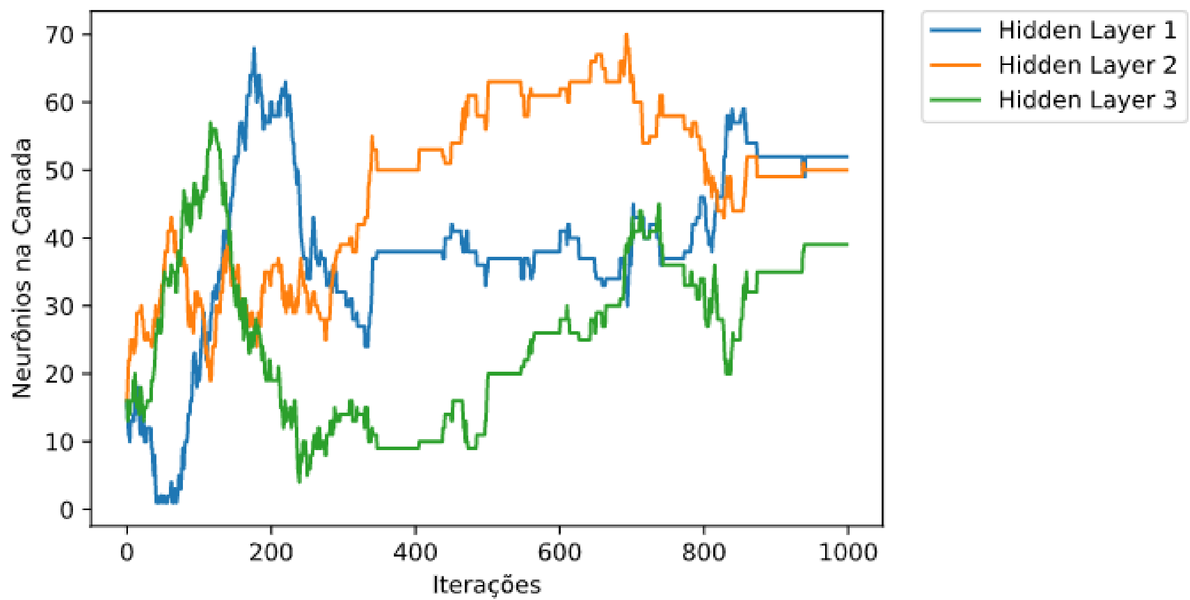

Figura 22 – Número de neurônios na rede com 3 camadas escondidas ao longo da execução do algoritmo de *simulated annealing*.

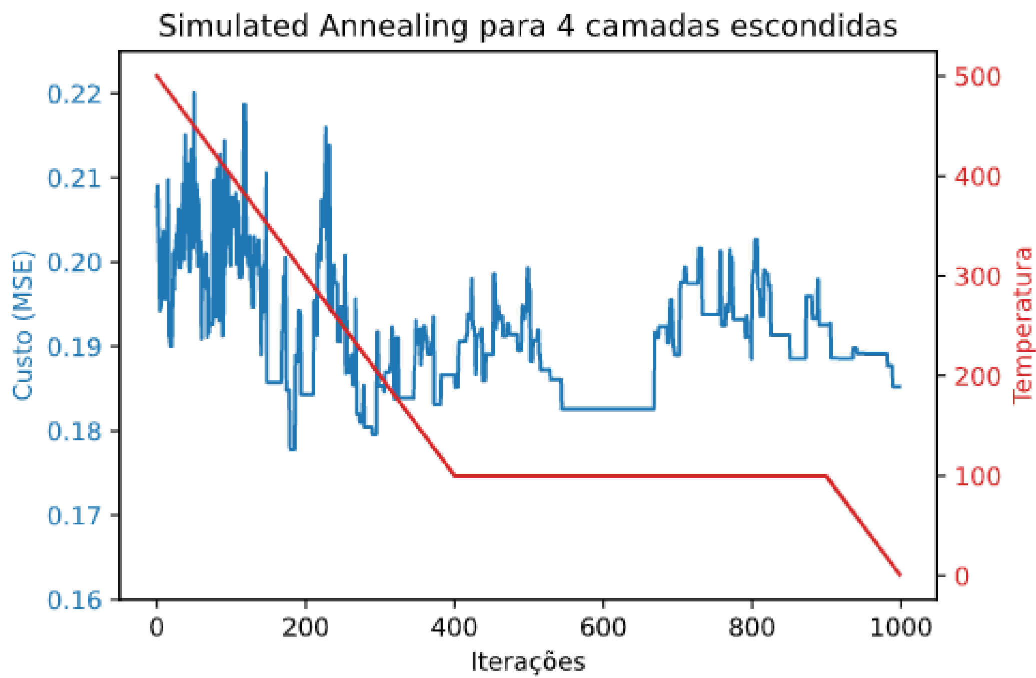

Figura 23 – Otimização do custo da rede com 4 camadas escondidas por *simulated annealing*.

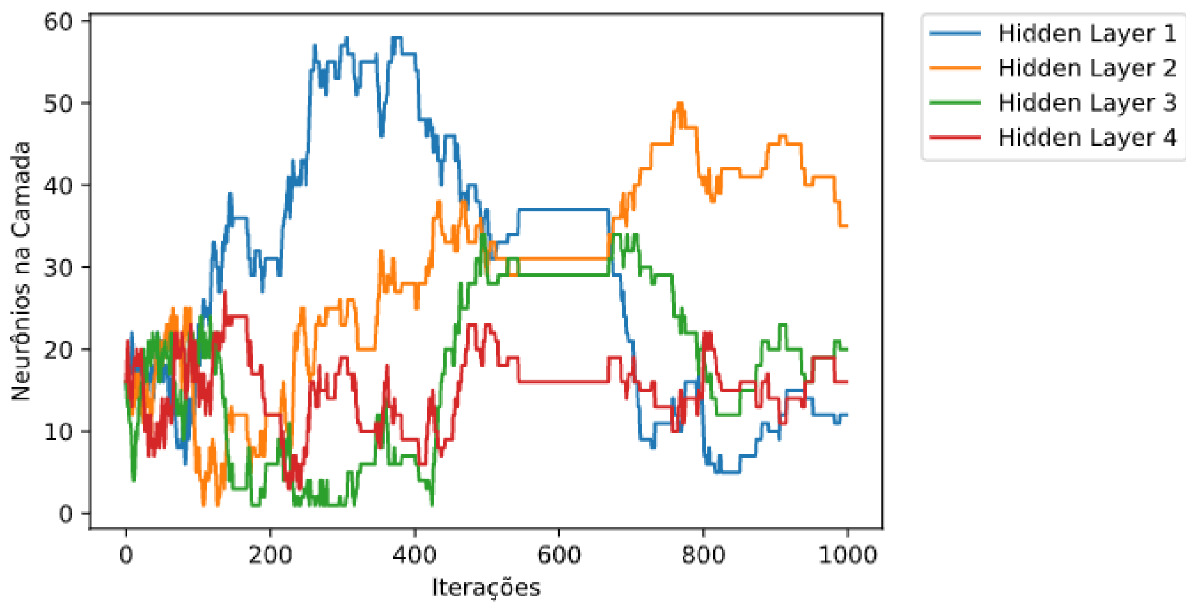

Figura 24 – Número de neurônios na rede com 4 camadas escondidas ao longo da execução do algoritmo de *simulated annealing*.

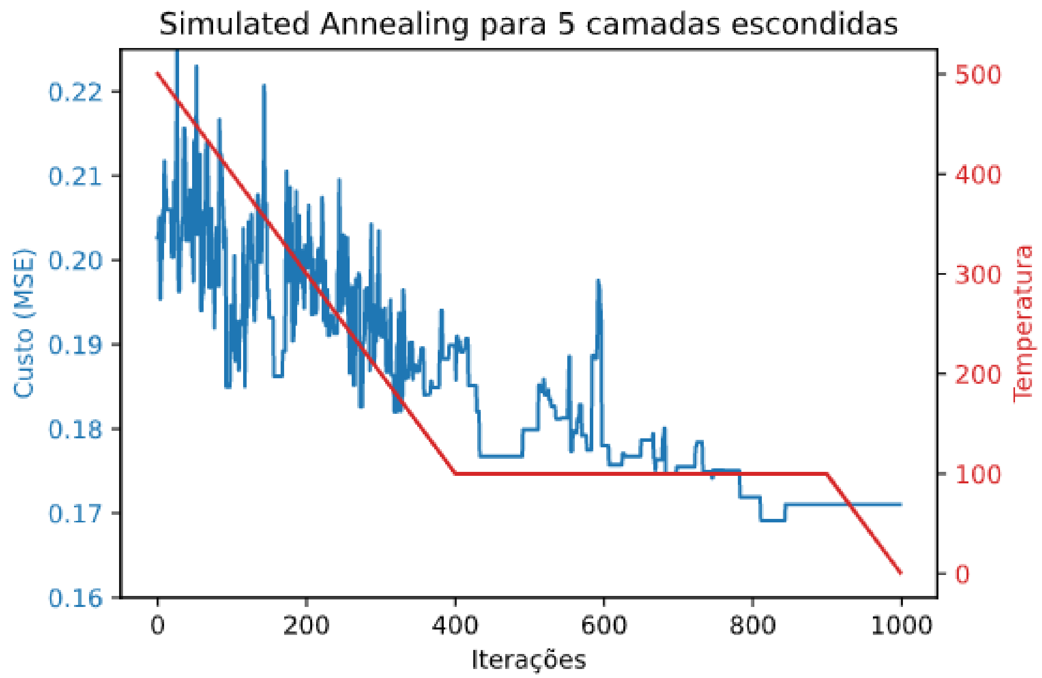

Figura 25 – Otimização do custo da rede com 5 camadas escondidas por *simulated annealing*.

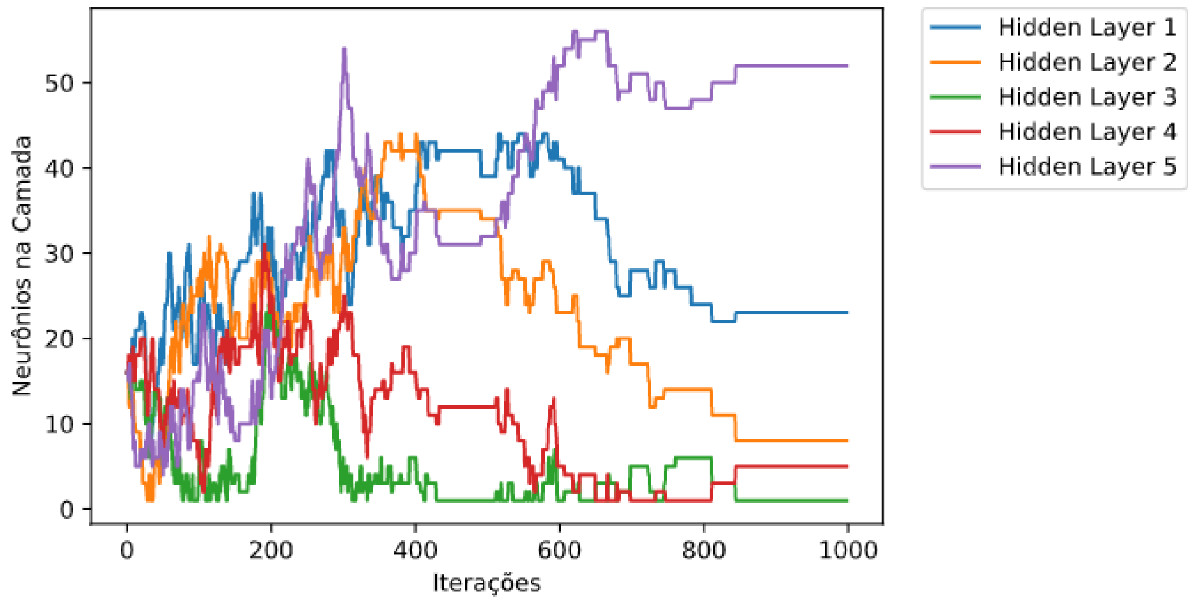

Figura 26 – Número de neurônios na rede com 5 camadas escondidas ao longo da execução do algoritmo de *simulated annealing*.

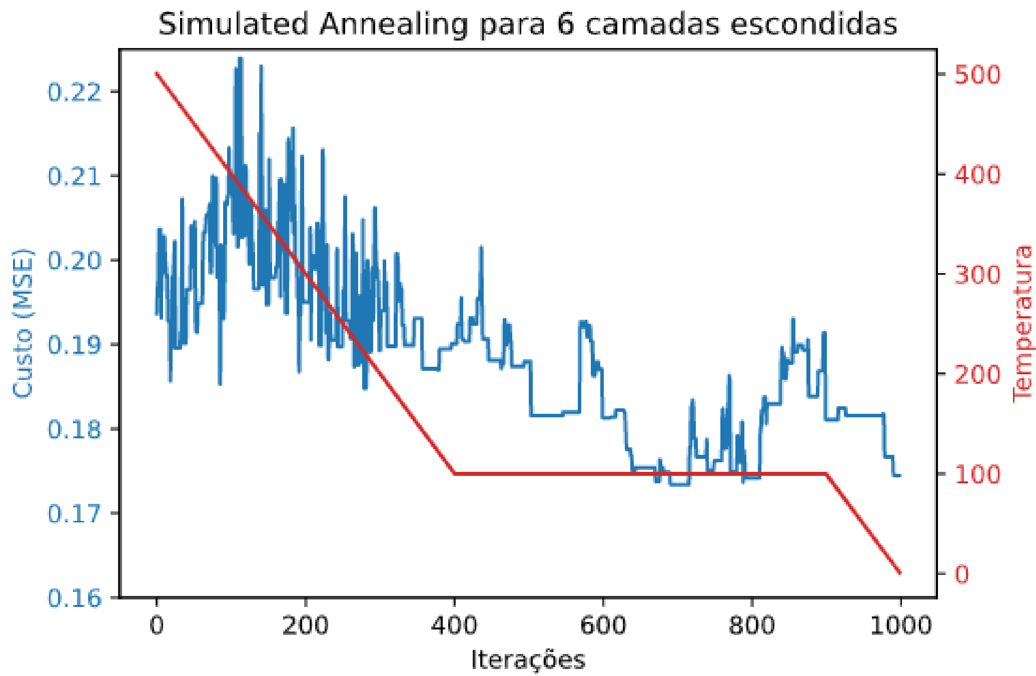

Figura 27 – Otimização do custo da rede com 6 camadas escondidas por *simulated annealing*.

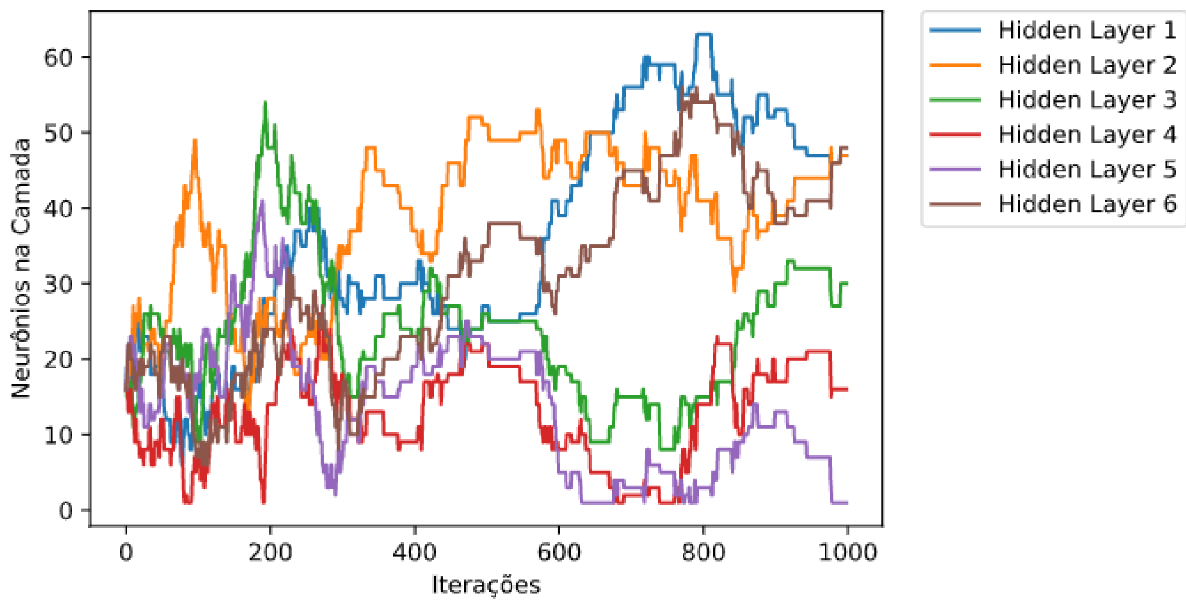

Figura 28 – Número de neurônios na rede com 6 camadas escondidas ao longo da execução do algoritmo de *simulated annealing*.

Camadas escondidas	Custo Final	Neurônios nas Camadas Escondidas	Alocação de memória do modelo (float)
2	0.17979	[36, 38]	11684 Bytes
3	0.17953	[52, 50, 39]	27244 Bytes
4	0.18526	[12, 35, 20, 16]	8080 Bytes
5	0.17103	[23, 8, 1, 5, 52]	6076 Bytes
6	0.17447	[47, 47, 30, 16, 1, 48]	25124 Bytes

Tabela 2 – Dados da execução do algoritmo de *simulated annealing* para diferentes números de camadas escondidas.

A partir destes resultados obtidos na otimização, foram selecionadas 2 conjuntos de hiperparâmetros resultantes para quantizar as redes a partir deles e processá-las no microcontrolador. A rede com 5 camadas escondidas foi selecionada, pois resultou em uma menor função custo e menor alocação de memória no fim da otimização. Além dela, a rede com 4 camadas também foi selecionada, pois também resultou em baixa alocação de memória e possui número de neurônios mais uniformes nas camadas.

4.2 Treinamento com Hiperparâmetros Selecionados

As rede com 4 e 5 camadas escondidas foram inicialmente treinadas em ponto flutuante através da biblioteca TensorFlow em Python por 2048 epochs com *batch size* igual a 1024 com os hiperparâmetros resultantes da otimização. Foi utilizado o otimizador SGD no treinamento e taxa de aprendizado igual a 0,01. A função de ativação utilizada para as camadas escondidas foi a *leaky* ReLU com K=0,1 e, para a saída, ativação linear. As figuras 29 e 30 mostram o treinamento com o valor da função custo para a parte de treino e de teste do *dataset*. Os custos finais para a parte de teste foram iguais a 0,152605 e 0,150080 para 4 camadas e 5 camadas, respectivamente.

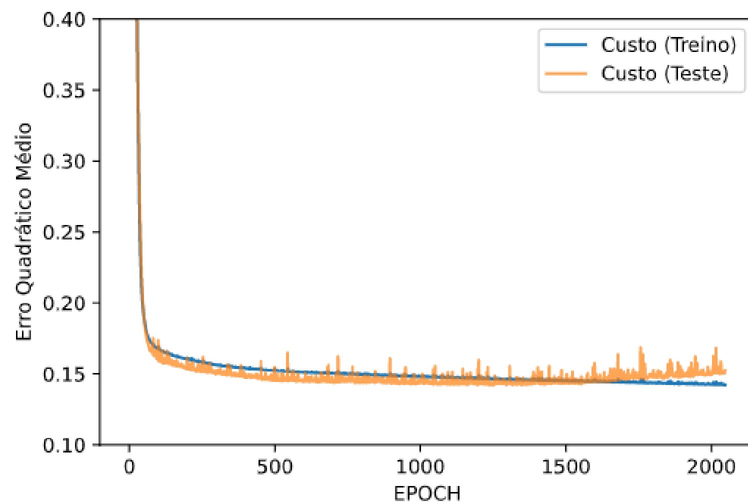

Figura 29 – Treinamento da rede com 4 camadas escondidas.

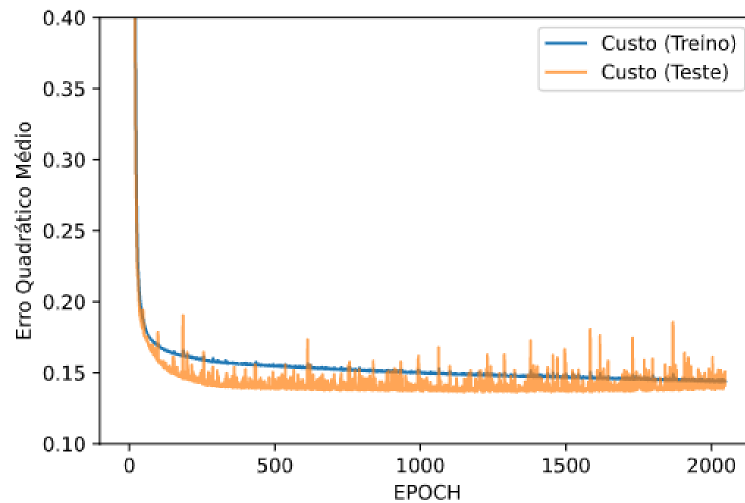

Figura 30 – Treinamento da rede com 5 camadas escondidas.

Em seguida, foram extraídas as distribuições de valores de saída de cada camada da rede treinada para determinar os parâmetros de quantização. As figuras 31 e 32 mostram o histograma das entradas da rede (variáveis de entrada da parte de treino do *dataset*). As figuras 33, 34, 35, 36 e 37 mostram as distribuições de valores de saída das camadas para a rede com 4 camadas escondidas. As distribuições da rede com 5 camadas são apresentadas nas figuras 43, 44, 45, 46, 47 e 48.

Além disso, foram analisadas as distribuições dos valores de peso de cada camada para realizar sua quantização. As figuras 38, 39, 40, 41 e 42 mostram os histogramas das distribuições de pesos da rede com 4 camadas escondidas. As distribuições de valores de pesos da rede com 5 camadas escondidas estão apresentadas nas figuras 49, 50, 51, 52, 52 e 54

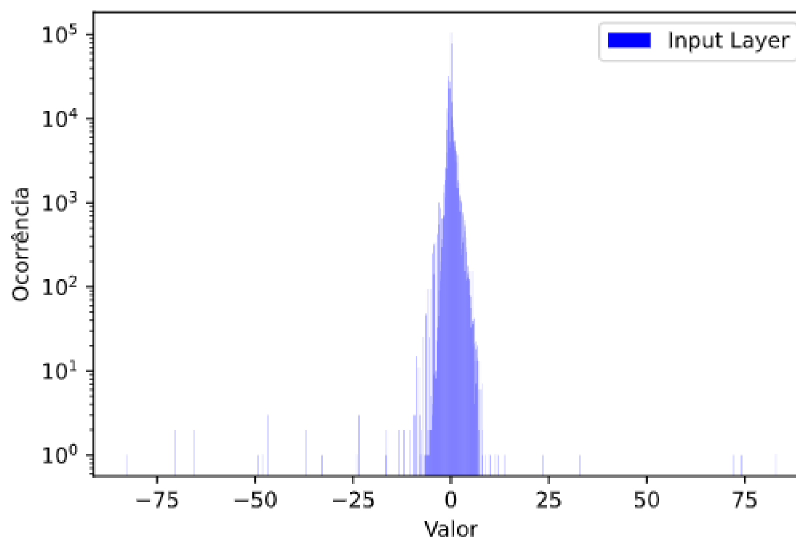

Figura 31 – Histograma de valores de entrada da rede.

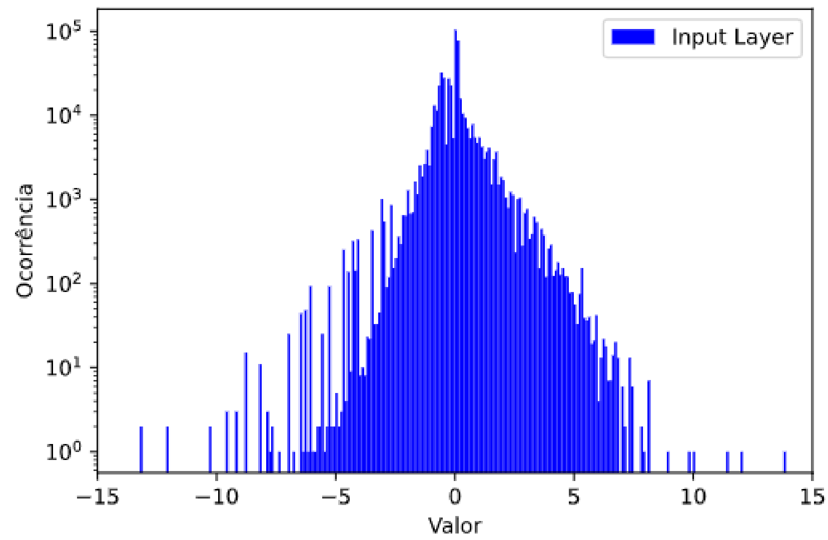

Figura 32 – Histograma de valores de entrada da rede (ampliado).

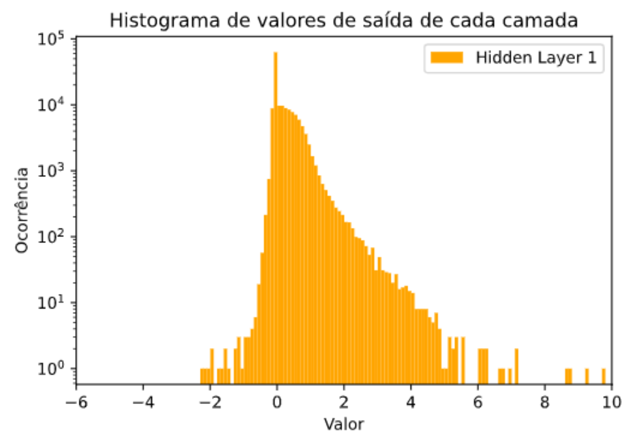

Figura 33 – Histograma de valores de saida da primeira camada escondida da rede com 4 camadas escondidas.

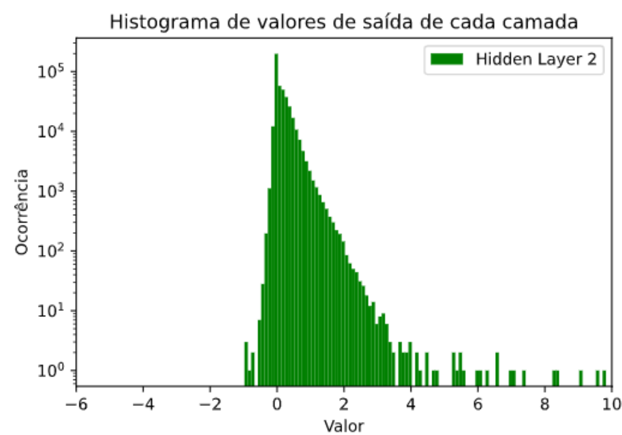

Figura 34 – Histograma de valores de saida da segunda camada escondida da rede com 4 camadas escondidas.

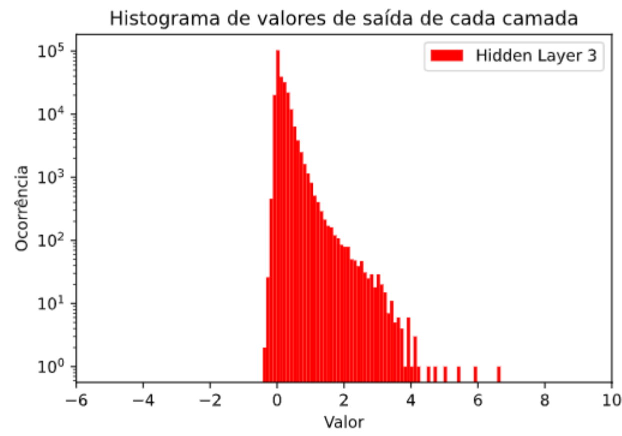

Figura 35 – Histograma de valores de saida da terceira camada escondida da rede com 4 camadas escondidas.

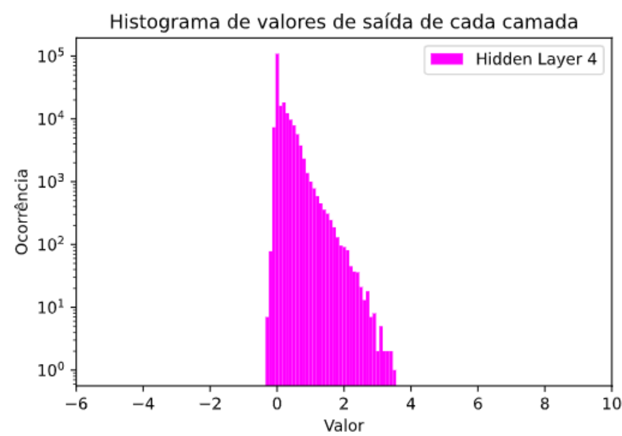

Figura 36 – Histograma de valores de saida da quarta camada escondida da rede com 4 camadas escondidas.

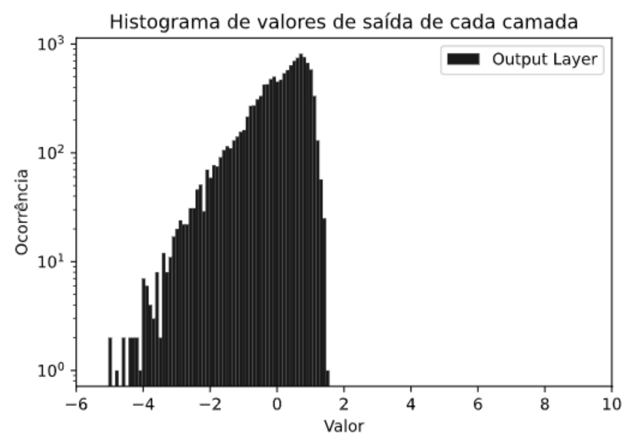

Figura 37 – Histograma de valores de saida da camada de saída da rede com 4 camadas escondidas.

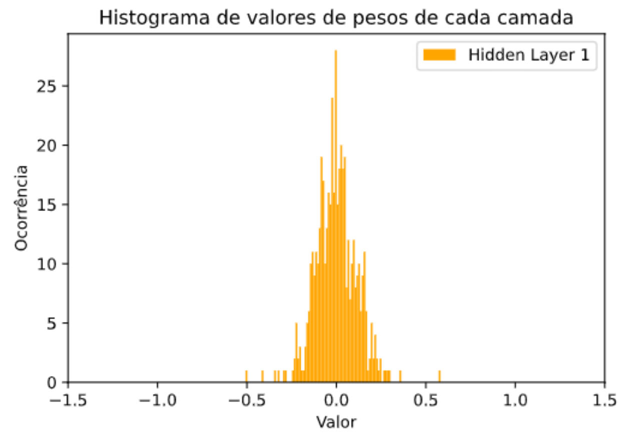

Figura 38 – Histograma de valores de pesos da primeira camada escondida da rede com 4 camadas escondidas.

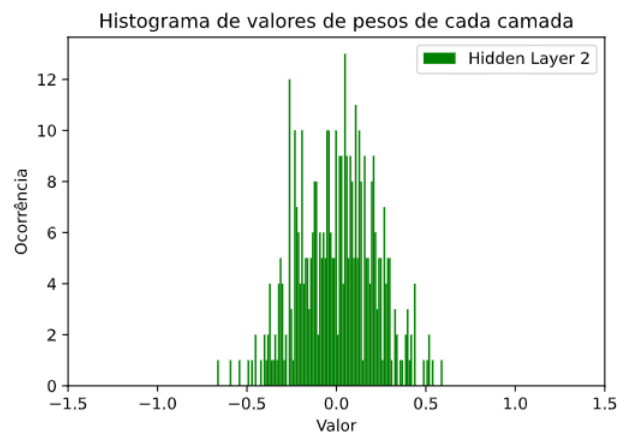

Figura 39 – Histograma de valores de pesos da segunda camada escondida da rede com 4 camadas escondidas.

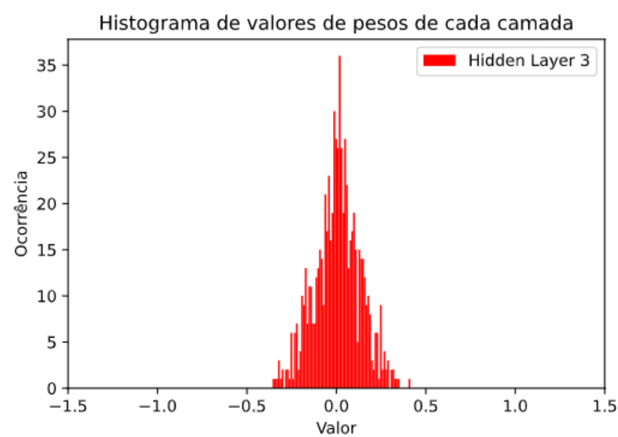

Figura 40 – Histograma de valores de pesos da terceira camada escondida da rede com 4 camadas escondidas.

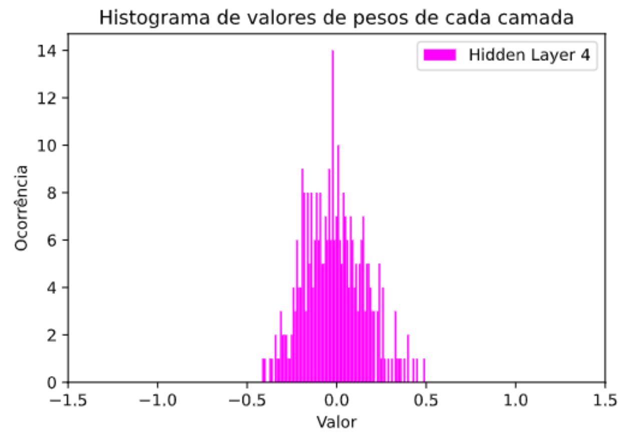

Figura 41 – Histograma de valores de pesos da quarta camada escondida da rede com 4 camadas escondidas.

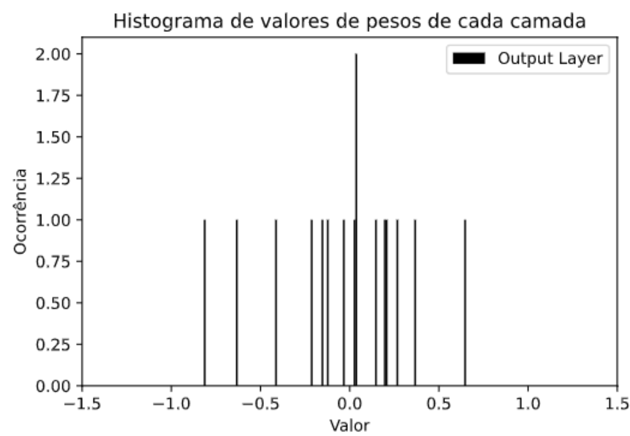

Figura 42 – Histograma de valores de pesos da camada de saída da rede com 4 camadas escondidas.

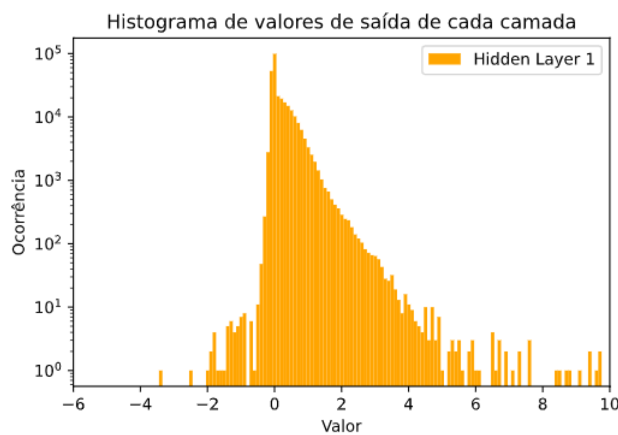

Figura 43 – Histograma de valores de saida da primeira camada escondida da rede com 5 camadas escondidas.

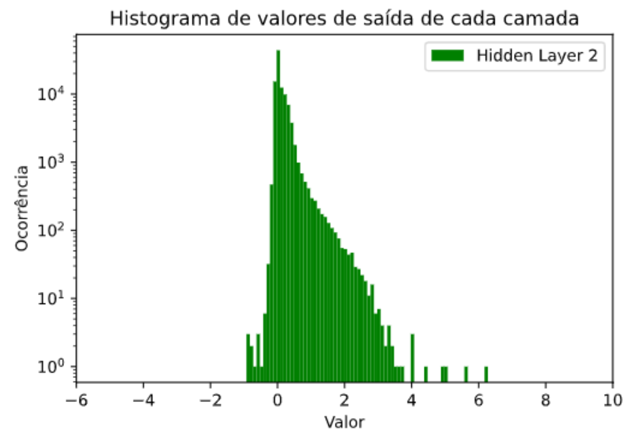

Figura 44 – Histograma de valores de saida da segunda camada escondida da rede com 5 camadas escondidas.

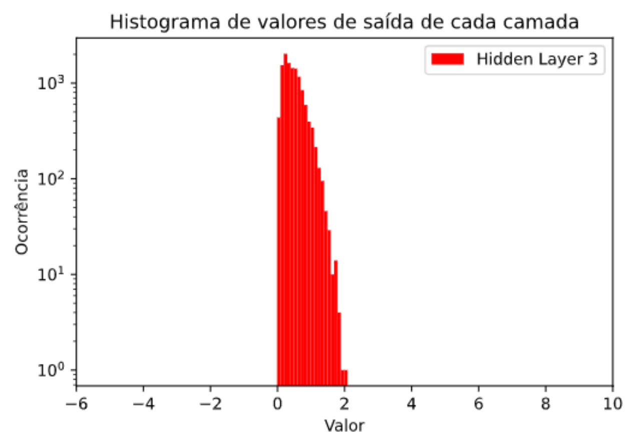

Figura 45 – Histograma de valores de saida da terceira camada escondida da rede com 5 camadas escondidas.

Figura 46 – Histograma de valores de saida da quarta camada escondida da rede com 5 camadas escondidas.

Figura 47 – Histograma de valores de saida da quinta camada escondida da rede com 5 camadas escondidas.

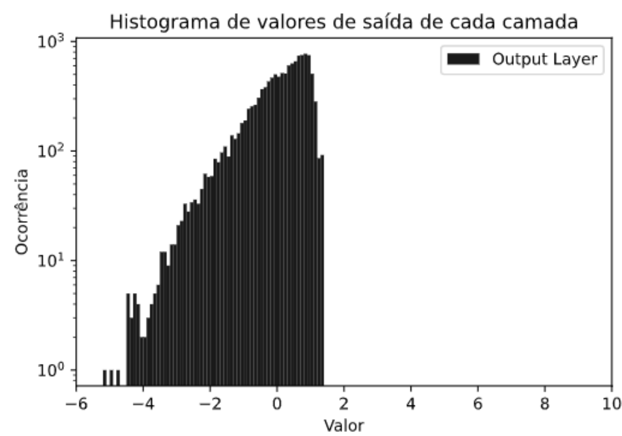

Figura 48 – Histograma de valores de saida da camada de saída da rede com 5 camadas escondidas.

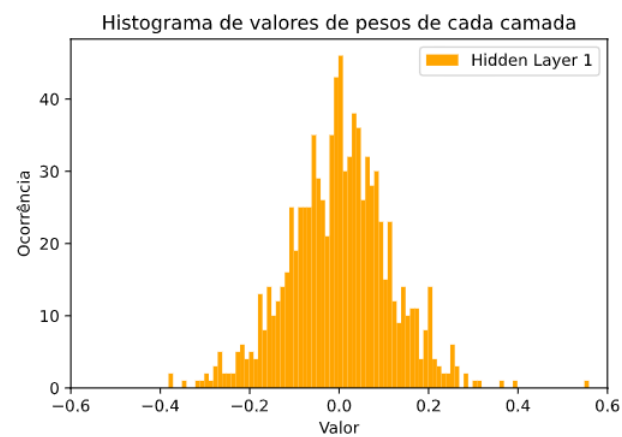

Figura 49 – Histograma de valores de pesos da primeira camada escondida da rede com 5 camadas escondidas.

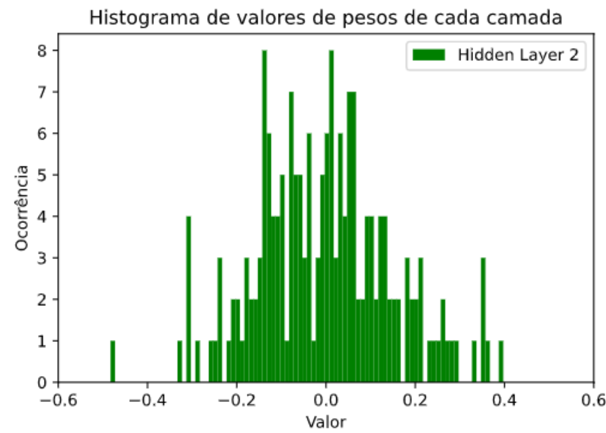

Figura 50 – Histograma de valores de pesos da segunda camada escondida da rede com 5 camadas escondidas.

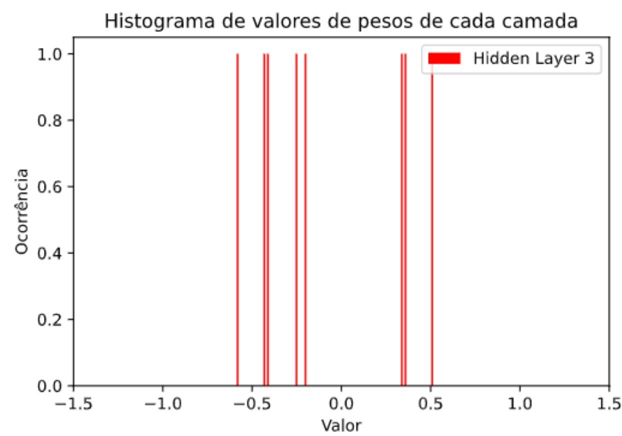

Figura 51 – Histograma de valores de pesos da terceira camada escondida da rede com 5 camadas escondidas.

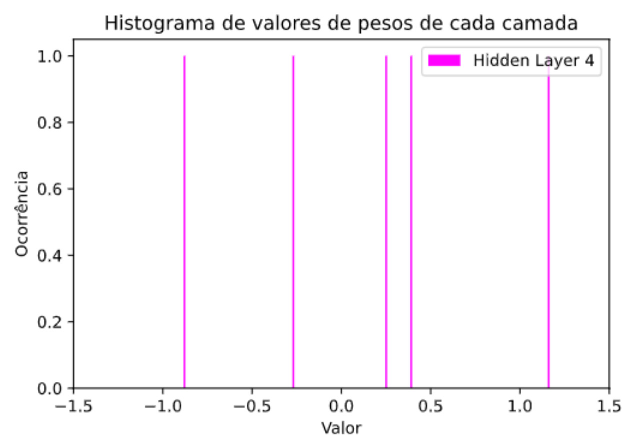

Figura 52 – Histograma de valores de pesos da quarta camada escondida da rede com 5 camadas escondidas.

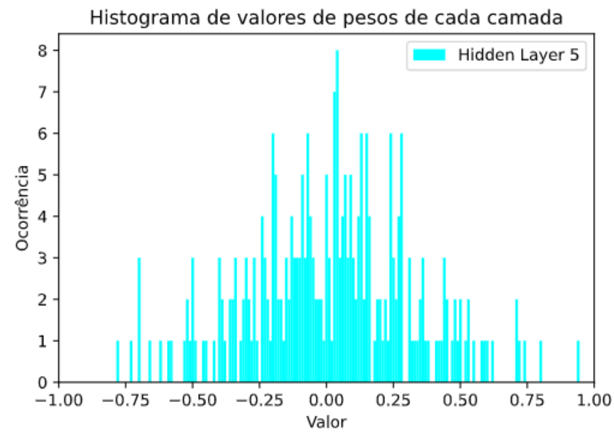

Figura 53 – Histograma de valores de pesos da quinta camada escondida da rede com 5 camadas escondidas.

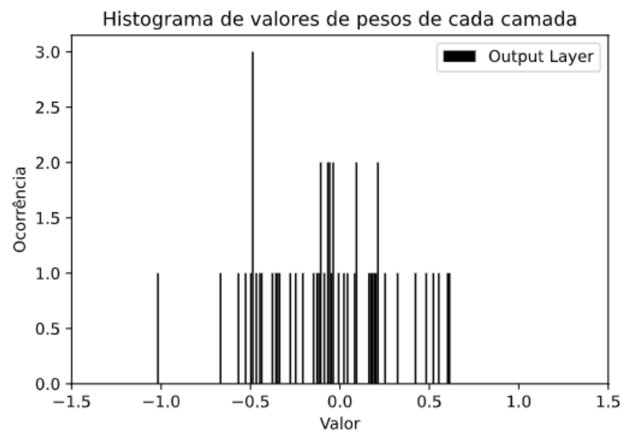

Figura 54 – Histograma de valores de pesos da camada de saída da rede com 5 camadas escondidas.

4.3 Quantização dos Modelos Treinados

A partir das distribuições, foram determinados os parâmetros de quantização para os modelos. Contudo, alguns ajustes foram aplicados nesses parâmetros para evitar *underflow* e *overflow* no processamento em ponto fixo. Estes ajustes levaram em consideração o valor máximo e mínimo de quantização entre camadas adjacentes, tal que a razão entre o valor $MAX - MIN$ entre camadas adjacentes seja próxima de 1 sem causar aumento no erro de quantização. Essa razão impacta diretamente no termo $\frac{S_x}{S_f}$ da equação 2.34 que pode contribuir com a probabilidade de *underflow* e *overflow*, visto que ele multiplica um somatório de pesos. Os parâmetros de quantização ajustados que foram utilizados para as redes de 4 e 5 camadas escondidas estão listados nas tabelas 3 e 4, respectivamente.

Camada	Saída (Mínimo)	Saída (Máximo)	Peso (Mínimo)	Peso (Máximo)	bias (Mínimo)	bias (Máximo)
Entrada	-7,5	7,5	-	-	-	-
1	-5,0	7,5	-0,6	0,6	-0,5	0,5
2	-3,0	6,0	-0,7	0,7	-0,5	0,5
3	-2,5	6,0	-0,5	0,5	-0,5	0,5
4	-2,0	6,0	-0,5	0,5	-0,5	0,5
Saída	-6,0	2,5	-1,2	1,2	-0,5	0,5

Tabela 3 – Parâmetros de quantização da rede com 4 camadas escondidas.

Camada	Saída (Mínimo)	Saída (Máximo)	Peso (Mínimo)	Peso (Máximo)	bias (Mínimo)	bias (Máximo)
Entrada	-7,5	7,5	-	-	-	-
1	-6,5	7,0	-0,6	0,6	-0,5	0,5
2	-5,0	6,0	-0,5	0,5	-0,5	0,5
3	-1,0	3,0	-0,6	0,6	-0,5	0,5
4	-1,5	3,0	-1,2	1,2	-0,5	0,5
5	-2,0	3,5	-1,0	1,0	-0,5	0,5
Saída	-6,0	5,75	-1,1	1,1	-0,5	0,5

Tabela 4 – Parâmetros de quantização da rede com 5 camadas escondidas.

Os modelos quantizados em 32, 16, 8 e 4 bits foram testados na implementação em computador e foram obtidos os erros de quantização de cada camada com relação ao modelo em ponto flutuante (equação 3.4). Devido à distribuição de cauda pesada das entradas (figura 31), ocorreu saturação de alguns exemplos do *dataset* que impossibilitaram uma medida concisa do erro de quantização, pois o erro quadrático aumentou de forma considerável quando estes exemplos foram incluídos no cálculo do erro. Para contornar esse problema, o cálculo do erro foi realizado utilizando os exemplos da parte de teste do *dataset* cujas entradas não foram saturadas no processo de quantização de entradas.

As figuras 55 e 56 mostram os erros de quantização de cada camada para 4 e 5 camadas escondidas, respectivamente.

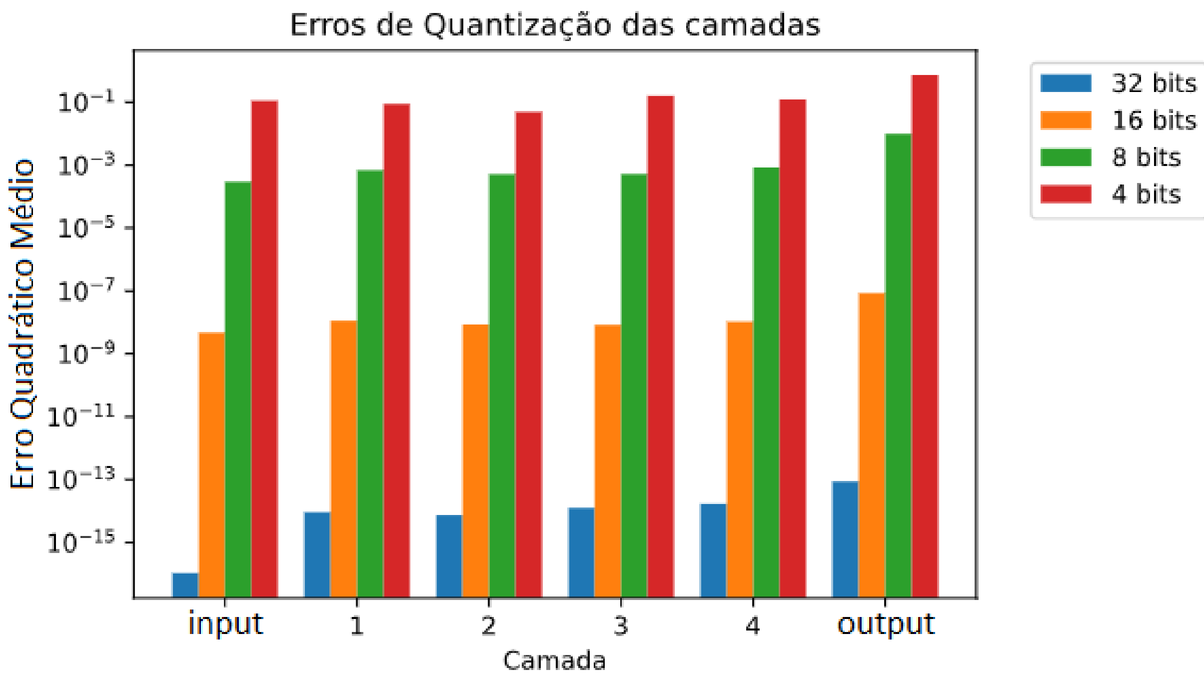

Figura 55 – Erro quadrático médio de quantização de cada camada em diferentes precisões de quantização para a rede com 4 camadas escondidas.

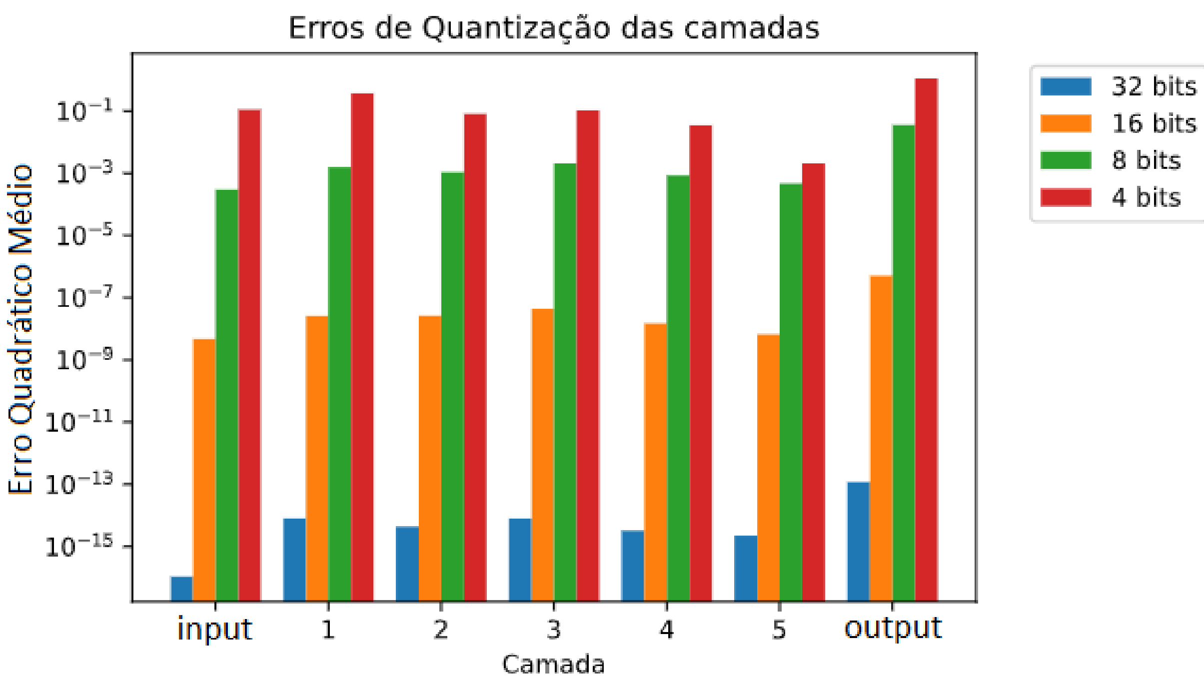

Figura 56 – Erro quadrático médio de quantização de cada camada em diferentes precisões de quantização para a rede com 5 camadas escondidas.

4.4 Avaliação dos Modelos

O custo das redes foi avaliado através da parte de teste do *dataset* e foi seguida a mesma restrição para exclusão de exemplos cujas entradas são saturadas nos modelos quantizados. Em sequência, os modelos foram transferidos para o microcontrolador através da interface desenvolvida com comunicação serial. Eles foram executados na implementação feita para o microcontrolador e os dados de velocidade de processamento e consumo energético foram coletados.

A figura 57 mostra um exemplo da transferência do modelo de 4 camadas escondidas em ponto fixo de 32 bits com resposta do microcontrolador (tabela 1) ao enviar os comandos listados abaixo em sequência e o arquivo contendo o modelo através do programa RealTerm:

- S 8500
- D int model
- I

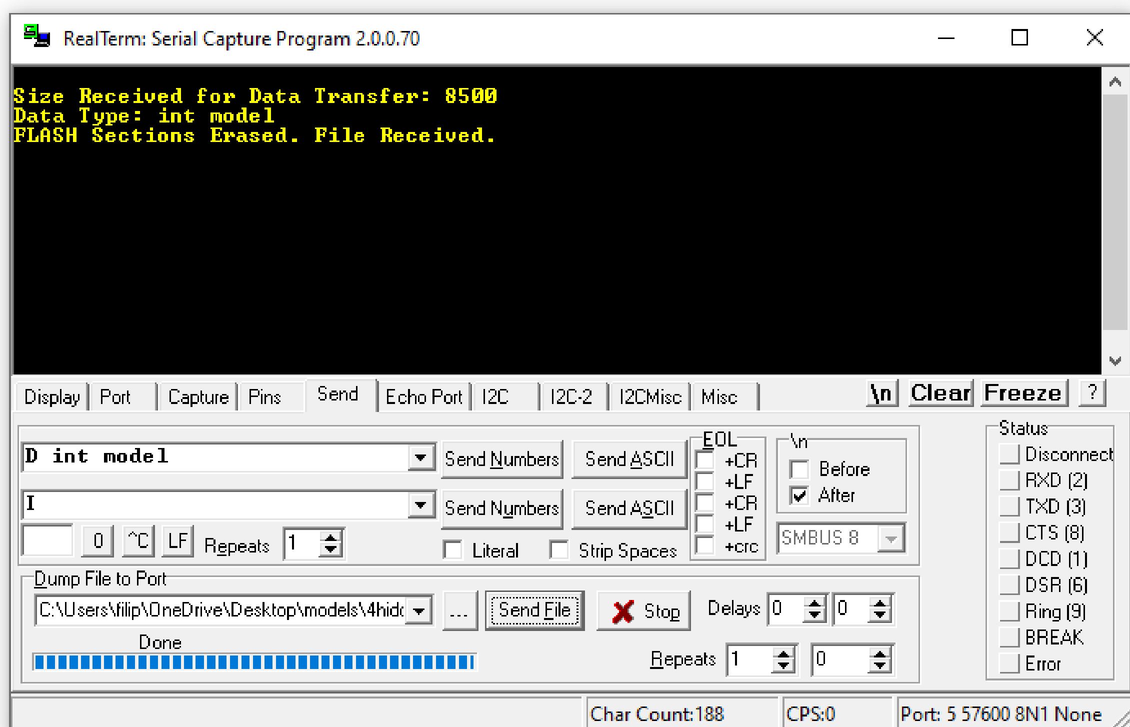

Figura 57 – Exemplo de envio do modelo de 4 camadas escondidas quantizada em ponto fixo de 32 bits através do RealTerm.

A figura 58 mostra a placa de desenvolvimento que foi utilizada conectada aos pinos do programador ST-LINK V2, em que o multímetro de bancada está conectado em série com a alimentação 3V3 fornecida pelo programador. A alimentação foi medida com o multímetro para obter a tensão com maior exatidão. O valor obtido foi de 3,28 Volts, que foi utilizado posteriormente para obter a potência na execução do código. Para realizar a medida de tempo de processamento, foram carregados os primeiros 100 exemplos da parte de teste do *dataset* através dos comandos da interface serial USB (tabela 1). Após o carregamento, um timer inicia a contagem de tempo e o resultado de cada exemplo é inferido 500 vezes em um laço, ou seja, são executadas 50000 inferências. Ao final, o timer é pausado e o tempo decorrido é enviado via USB. Para medir o consumo energético, foi aferida a corrente elétrica sem a conexão USB, garantindo, assim, que a única fonte de alimentação do microcontrolador proviesse do programador. A corrente foi medida em diferentes precisões e foi verificado que seu valor é aproximadamente constante durante o processamento (as oscilações ocorreram na segunda casa decimal da corrente em mA). As tabelas 5 e 6 mostram os dados coletados para o modelo de 4 e 5 camadas escondidas, respectivamente. Todos os dados coletados foram reunidos nas tabelas 7 e 8.

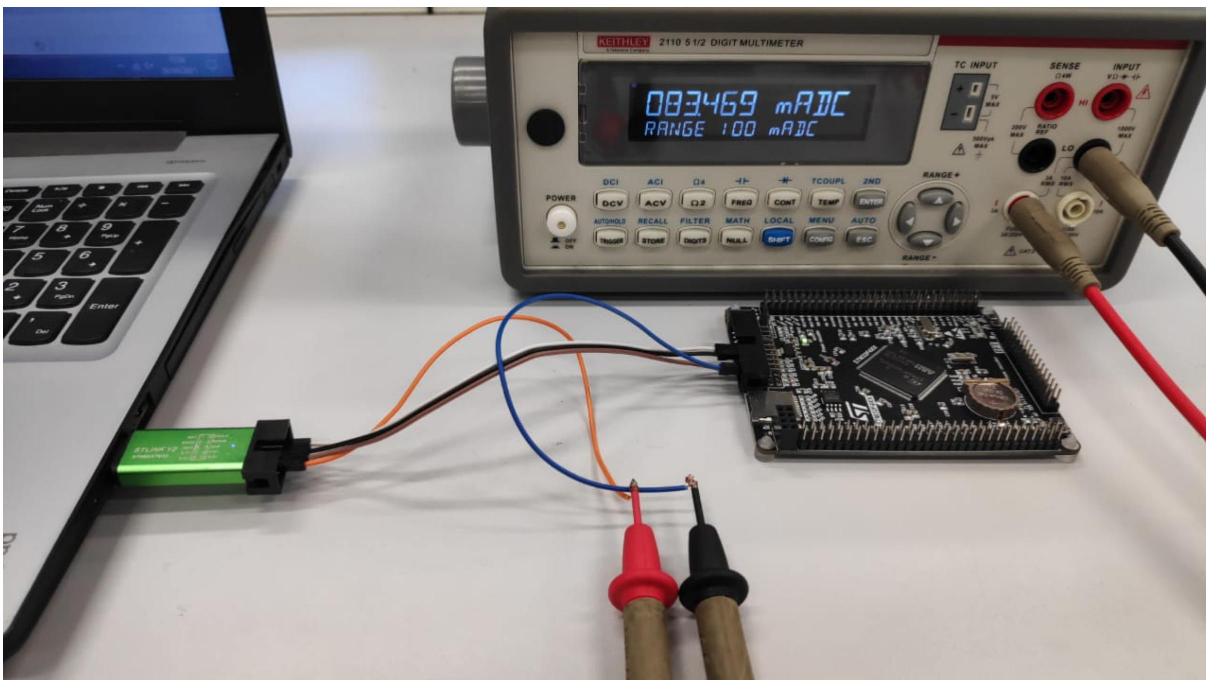

Figura 58 – Placa de desenvolvimento alimentada pelo programador ST-LINK V2 com multímetro de bancada conectado em série para realizar medida de corrente elétrica. O valor mostrado no multímetro corresponde à corrente máxima de consumo do *firmware* quando é executado um laço while(1) em estado de espera.

Precisão	Corrente Elétrica Média	Potência	Tempo Decorrido
float	59,9 mA	196 mW	6462 ms
int 32	67,6 mA	222 mW	39260 ms
int16	65,2 mA	214 mW	38518 ms
int 8	64,7 mA	212 mW	38498 ms
int 4	63,9 mA	210 mW	59830 ms

Tabela 5 – Valores de corrente elétrica, potência e tempo decorrido ao executar 50000 inferências na rede com 4 camadas escondidas quantizada em diferentes precisões.

Precisão	Corrente Elétrica Média	Potência	Tempo Decorrido
float	59,2 mA	194 mW	5478 ms
int 32	66,7 mA	219 mW	39684 ms
int16	64,9 mA	213 mW	39518 ms
int 8	64,6 mA	212 mW	39027 ms
int 4	63,5 mA	208 mW	54959 ms

Tabela 6 – Valores de corrente elétrica, potência e tempo decorrido ao executar 50000 inferências na rede com 5 camadas escondidas quantizada em diferentes precisões.

Precisão	Custo	Erro de Quantização da Saída (MSE)	Tamanho do modelo (com parâmetros)	Tempo Médio de Inferência	Consumo Médio /Inferência
float	0,129320	-	8164 Bytes	129,24 us	25,3 uJ
int 32	0,129320	8.69167 e-14	8500 Bytes	785,20 us	174,3 uJ
int 16	0,129324	8.54340 e-08	4460 Bytes	770,36 us	164,9 uJ
int 8	0,140280	9.74995 e-03	2440 Bytes	769,96 us	163,2 uJ
int 4	1,155162	7.26597 e-01	1430 Bytes	1196,60 us	251,3 uJ

Tabela 7 – Resumo de dados obtidos da rede com 4 camadas escondidas.

Precisão	Custo	Erro de Quantização da Saída (MSE)	Tamanho do modelo (com parâmetros)	Tempo Médio de Inferência	Consumo Médio /Inferência
float	0,139951	-	6160 Bytes	109,56 us	21,3 uJ
int 32	0,139951	1,18155 e-13	6552 Bytes	793,68 us	173,8 uJ
int 16	0,139925	5,03929 e-07	3514 Bytes	790,36 us	168,3 uJ
int 8	0,170288	3,52892 e-02	1995 Bytes	780,54 us	165,5 uJ
int 4	0,921179	1,12959	1236 Bytes	1099,18 us	228,6 uJ

Tabela 8 – Resumo de dados obtidos da rede com 5 camadas escondidas.

Claramente, os modelos processados em ponto flutuante apresentaram resultados muito atrativos, pois seus tempos de processamento e de consumo energético por inferência mostraram-se muito superiores aos valores apresentados pelos modelos processados em ponto fixo. O processamento em ponto fixo de 32 bits mostrou um erro de quantização muito pequeno e custo da rede praticamente idêntico ao custo apresentado nos modelos processados em ponto flutuante. Contudo, todos os demais dados coletados para essa precisão mostram que não há vantagens em realizar o processamento em 32 bits nessa arquitetura através do *firmware* desenvolvido. O processamento em ponto fixo de 4 bits mostrou erro de quantização muito elevado, o que pode descartá-lo em muitas aplicações. Além disso, esta precisão resultou no mais alto tempo médio de inferência, pois são realizadas manipulações de bits para leitura e escrita de variáveis de 4 bits que não são suportadas na linguagem C. Essas manipulações resultam em maior número de instruções, o que aumenta o tempo de execução do código e o consumo por inferência.

Os modelos processados em ponto fixo de 16 e 8 bits mostraram resultados que podem ser favoráveis para utilização em sistemas com limites críticos de memória, pois, apesar de mostrarem maior consumo e tempo médio por inferência na implementação, apresentam erro de quantização razoável e menor tamanho total do modelo.

Os altos valores de tempo médio de inferência nos modelos processados em ponto fixo podem estar relacionados ao fato de que as instruções SIMD de DSP não foram utilizadas de forma otimizada com o acesso à memória. Isto é, foram realizadas manipulações e *typecast* para acesso às variáveis em ponto fixo alocadas de forma dinâmica, o que pode resultar em maior tempo de execução do código e, consequentemente, maior consumo energético. Além disso, no algoritmo de *forward propagation* em ponto fixo, foi utilizada a variável auxiliar "long_aux" do tipo *int* de 64 bits (figuras 16 e 17), o que pode ter causado maior tempo de execução do código e maior consumo energético por inferência, pois o microprocessador tem arquitetura de 32 bits.

Parte IV

Conclusão

5 Conclusão

O trabalho enfocou o desenvolvimento de um sistema de processamento de redes neurais em ponto flutuante e ponto fixo em um microprocessador ARM32. A finalidade dessa implementação foi avaliar as vantagens e desvantagens de diferentes precisões utilizadas no processamento do algoritmo de *forward propagation* desenvolvido para um microcontrolador ARM Cortex M4. A otimização de um modelo e do processamento é muito relevante para sistemas embarcados, pois eles têm memória limitada e devem minimizar o consumo energético em sistemas alimentados por bateria. O trabalho mostrou que o *firmware* desenvolvido apresentou tempo de inferência e consumo energético, no processamento em ponto fixo de 32, 16 e 8 bits, em torno de 6 a 7 vezes maiores que o processamento em ponto flutuante. Para o caso de 4 bits, estes valores foram em torno de 10 vezes maiores que a inferência em ponto flutuante. Estes dados mostram que o *firmware* desenvolvido para o microcontrolador da família ARM Cortex M4 não obteve os resultados esperados de menor consumo energético e menor tempo de processamento nas redes quantizadas em comparação com as redes processadas em ponto flutuante. Contudo, a alocação de memória, o erro de quantização e o custo para ponto fixo de 16 e 8 bits podem ser relevantes para casos em que o principal fator crítico do sistema é a quantidade de memória. Em 16 bits, o custo na camada de saída se manteve próximo do valor na rede em ponto flutuante, e, no caso de 8 bits, houve ligeiro aumento da função custo, porém, a redução de alocação de memória pode ser muito interessante para este caso. Os resultados do algoritmo utilizado para processamento de *forward propagation* em ponto flutuante se mostraram muito superiores ao processamento em ponto fixo com relação ao tempo de execução e consumo energético. É possível que a otimização utilizada na compilação tenha impactado positivamente no processamento em ponto flutuante. Todavia, o mesmo pode não ter ocorrido no código para processamento em ponto fixo devido à utilização de instruções de DSP, cujo acesso à memória SRAM não foi otimizado no código.

O algoritmo de *simulated annealing* se mostrou pouco eficiente na otimização do hiperparâmetros da rede. Contudo, para maior número de camadas escondidas, o algoritmo se mostrou mais funcional, o que pode indicar que deveriam ser utilizadas mais camadas para otimização do custo.

5.1 Direções Futuras

O *firmware* desenvolvido pode ser adaptado de forma a otimizar o acesso à memória com o uso da biblioteca CMSIS DSP, o que pode mostrar menor tempo de execução do código e menor consumo energético por inferência para o processamento em ponto fixo.

Além disso, é importante avaliar um *dataset* com relação às suas distribuições de valores normalizados para poder avaliar um sistema de processamento de forma ideal sem ocorrer anomalias de quantização que comprometam a avaliação.

Outros algoritmos podem ser utilizados ao invés do *simulated annealing* para otimizar hiperparâmetros da rede com limitações de alocação de memória. Dessa forma é possível verificar se os hiperparâmetros utilizados foram inicializados de forma sobre-ajustada e avaliar a eficiência de outros algoritmos com relação ao *simulated annealing* para a mesma finalidade.

Referências

AL, L. et. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1989. Citado na página 7.

ARM. Cmsis dsp software library version 1.8.0. 2020. Disponível em: <https://www.keil.com/pack/doc/cmsis/DSP/html/index.html>. Acesso em: 05 mai. 2021. Citado na página 38.

BRE, F.; GIMENEZ, J.; FACHINOTTI, V. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, v. 158, 11 2017. Citado na página 14.

CAPARRINI, F. S. *Simulated Annealing in NetLogo*. 2019. Citado na página 31.

CHEN, Y.-Y. et al. Design and implementation of cloud analytics-assisted smart power meters considering advanced artificial intelligence as edge analytics in demand-side management for smart homes. 2019. Citado na página 13.

FOOTE, K. D. *A Brief History of Deep Learning*. 2017. Disponível em: <https://www.dataversity.net/brief-history-deep-learning>. Acesso em: 05 mai. 2021. Citado na página 7.

GOODFELLOW, Y. B. I.; COURVILLE, A. *Deep Learning*. [s.n.], 2016. Book in preparation for MIT Press. Disponível em: <http://www.deeplearningbook.org>. Acesso em: 05 mai. 2021. Citado 4 vezes nas páginas 13, 16, 18 e 29.

GSCHWEND, D. *ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network*. 2020. Citado na página 8.

LECUN, Y. *Deep Learning and the Future of AI*. 2016. Disponível em: <https://indico.cern.ch/event/510372/>. Acesso em: 05 mai. 2021. Citado na página 7.

LEVY, S. *An exclusive inside look at how artificial intelligence and machine learning work at Apple*. 2016. Disponível em: <https://www.wired.com/2016/08/an-exclusive-look-at-how-ai-and-machine-learning-work-at-apple/>. Acesso em: 05 mai. 2021. Citado na página 7.

LEWIS, D. W. *ARM Assembly for Embedded Applications*. [S.l.: s.n.], 2016. Citado na página 32.

LIBERIS, N. D. L. E. Neural networks on microcontrollers: saving memory at inference via operator reordering. *On-device Intelligence Workshop at MLSys 2020: 3rd Conference on Machine Learning and Systems.*, 2020. Citado na página 9.

MATTEIS, T. D.; LICHT, J. de F.; HOEFLER, T. *FBLAS: Streaming Linear Algebra on FPGA*. 2019. Citado na página 8.

MEHROTRA, K.; MOHAN, C. K.; RANKA, S. *Elements of artificial neural networks*. [S.l.]: MIT Press, 1997. 4-7 p. (Complex adaptive systems). Citado na página 7.

NAKUTIS, A current consumption measurement approach for fpga-based embedded systems. *Instrumentation and Measurement, IEEE Transactions on*, v. 62, 05 2013. Citado na página 8.

NIELSEN, M. *Neural Networks and Deep Learning*. [S.l.: s.n.], 2019. Citado na página 20.

PATEL, M. *When two trends fuse: PyTorch and recommender systems*. 2017. Citado na página 7.

QASAIMEH, M. et al. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. *IEEE*, 2019. Citado na página 8.

Raihan, M. A.; Goli, N.; Aamodt, T. M. Modeling deep learning accelerator enabled gpus. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [S.l.: s.n.], 2019. p. 79–92. Citado na página 8.

RERE, L. R.; FANANY, M. I.; ARYMURTHY, A. M. Simulated annealing algorithm for deep learning. *Procedia Computer Science*, v. 72, p. 137 – 144, 2015. ISSN 1877-0509. The Third Information Systems International Conference 2015. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915035759>. Acesso em: 05 mai. 2021. Citado na página 30.

Rezvani, S.; Wang, X.; Pourpanah, F. Intuitionistic fuzzy twin support vector machines. *IEEE Transactions on Fuzzy Systems*, v. 27, n. 11, p. 2140–2151, 2019. Citado na página 7.

RONAGHAN, S. Deep learning: Overview of neurons and activation functions. Medium, 2018. Disponível em: <https://medium.com/@srnghn/deep-learning-overview-of-neurons-and-activation-functions-1d98286cf1e4>. Acesso em: 05 mai. 2021. Citado na página 13.

SATO, K.; YOUNG, C.; PATTERSON, D. An in-depth look at google's first tensor processing unit (tpu). *Google Cloud Blog*, 2017. Citado 2 vezes nas páginas 7 e 8.

Shawahna, A.; Sait, S. M.; El-Maleh, A. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, v. 7, p. 7823–7859, 2019. Citado na página 8.

STMICROELECTRONICS. *Use STM32F3/STM32G4 CCM SRAM with IAR™ EWARM, Keil® MDK-ARM and GNU-based toolchains*. 2019. Citado na página 28.

TAN, N. *Why Machine Learning on The Edge?* 2018. Disponível em: <https://towardsdatascience.com/why-machine-learning-on-the-edge-92fac32105e6>. Acesso em: 05 mai. 2021. Citado na página 8.

Wan, H. Deep learning:neural network, optimizing method and libraries review. In: *2019 International Conference on Robots Intelligent System (ICRIS)*. [S.l.: s.n.], 2019. p. 497–500. Citado na página 7.

WANG, X.; ZHAO, Y. Recent advances in deep learning. *International Journal of Machine Learning and Cybernetics*, 2020. Citado na página 7.

WU, S. et al. *Training and Inference with Integers in Deep Neural Networks*. 2018. Citado na página 9.

X., W.; H., J. *Uncertainty in learning from big data*. [S.l.: s.n.], 2015. 1-4 p. Citado na página 7.

ZHU, F. et al. *Towards Unified INT8 Training for Convolutional Neural Network*. 2019. Citado na página 9.