Trabalho de Conclusão de Curso

Aplicação de TinyML em Sensores Virtuais para Monitoramento da Qualidade do Ar em Ambientes Industriais

Tayco Murilo Santos Rodrigues

Orientadores:

Prof. Dr. Thiago Damasceno Cordeiro

Prof. Dr. Frede de Oliveira Carvalho

Maceió, Novembro de 2024

Tayco Murilo Santos Rodrigues

Aplicação de TinyML em Sensores Virtuais para Monitoramento da Qualidade do Ar em Ambientes Industriais

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Prof. Dr. Thiago Damasceno Cordeiro

Prof. Dr. Frede de Oliveira Carvalho

Maceió, Novembro de 2024

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Prof. Dr. Thiago Damasceno Cordeiro - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Prof. Dr. Frede de Oliveira Carvalho - Orientador
Instituto de Química e Biotecnologia
Universidade Federal de Alagoas

Prof. Dr. Erick de Andrade Barboza - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Prof.ª M.ª Andressa Martins Oliveira - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, Novembro de 2024

Agradecimentos

A realização deste Trabalho foi possível graças ao apoio que tive de diferentes pessoas durante os anos de graduação, a quem dedico meus sinceros agradecimentos.

Inicialmente, gostaria de deixar registrada minha gratidão ao meu orientador, Thiago Cordeiro, por toda a sua paciência (muita paciência, mas muita mesmo), disponibilidade e disposição em contribuir de diferentes formas para o meu crescimento tanto acadêmico quanto pessoal. O cara é bom!

Aos meus amigos, Aldemir Filho, Sandoval Júnior, João Victor Holanda, Rebeca Brandão e Marcos André, agradeço do fundo do meu coração por todos os momentos em que vocês estiveram e estarão ao meu lado tornando a minha vida melhor.

Em especial, agradeço ao meu companheiro de projetos e amigo, Rafael Durelli, por compartilhar comigo momentos de aprendizado e por todo o companheirismo ao longo de quase dois anos.

Aos professores: Ícaro de Araújo, Erick Barboza e Bruno Nogueira, possuo gratidão pelas diferentes oportunidades que me foram dadas durante estes últimos anos que de diferentes formas, contribuíram direta ou indiretamente para a minha formação.

Por fim, para todos aqueles que acreditaram em mim, meu muito obrigado!

"A vida não tem que ser perfeita, só tem que ser vivida."

– Dexter Morgan

Resumo

O monitoramento contínuo da qualidade do ar, especialmente no setor de mineração, é essencial para garantir que os níveis de poluentes estejam dentro dos padrões regulatórios. As soluções tradicionais para este monitoramento enfrentam limitações técnicas e econômicas e, neste contexto, as técnicas de aprendizagem de Máquina, do termo em inglês machine learning (ML), aplicadas a sistemas embarcados (TinyML), oferecem uma alternativa promissora para a previsão e análise de poluentes atmosféricos. Este trabalho tem como objetivo desenvolver um sistema de baixo custo e com recursos computacionais limitados para monitoramento contínuo da qualidade do ar que utilize técnicas de inteligência artificial (IA) para prever a concentração de cloro no ar com base em variáveis de entrada facilmente acessíveis: potássio e sódio. O modelo de IA foi treinado, otimizado e embarcado em um microcontrolador ESP32-S3. A quantização dos modelos reduziu o tamanho dos mesmos sem comprometer a precisão, garantindo a viabilidade de processamento em dispositivos com recursos limitados, com média de tempo para o processo de inferência sendo aproximadamente de 49,01 µs. No que diz respeito à precisão nas previsões de cloro, o modelo apresentou um erro quadrático médio de 3,96, um erro absoluto médio de 1,14 e um coeficiente de determinação de 0,95, demonstrando alta eficiência confirmando a viabilidade do sistema proposto.

Palavras-chave: Inteligência artificial, TinyML, monitoramento ambiental, poluição atmosférica, mineração, ESP32-S3, quantização de modelos, redes neurais artificiais.

Abstract

Continuous air quality monitoring, especially in the mining sector, is essential to ensure that pollutant levels remain within regulatory standards. Traditional solutions for this monitoring face technical and economic limitations. In this context, Machine Learning (ML) techniques applied to embedded systems (TinyML) offer a promising alternative for forecasting and analyzing atmospheric pollutants. This work aims to develop a low-cost system with limited computational resources for continuous air quality monitoring that uses artificial intelligence (AI) techniques to predict chlorine concentration in the air based on easily accessible input variables: potassium and sodium. The AI model was trained, optimized, and embedded in an ESP32-S3 microcontroller. Model quantization reduced the model size without compromising accuracy, ensuring processing feasibility on resource-limited devices, with an average inference process time of approximately 49.01 µs. Regarding chlorine prediction accuracy, the model presented a mean squared error of 3.96, a mean absolute error of 1.14, and a determination coefficient of 0.95, demonstrating high efficiency and confirming the viability of the proposed system.

Keywords: Artificial intelligence, TinyML, environmental monitoring, air pollution, mining, ESP32-S3, model quantization, artificial neural networks.

Sumário

Lista de Abreviaturas e Siglas ix
Lista de Figuras x

1 Introdução **1**
1.1 Motivação 1
1.2 Objetivos 3
1.3 Estrutura do Trabalho 3

2 Fundamentação Teórica **4**
2.1 Aprendizado de máquina 4
2.1.1 Técnicas de Aprendizado de Máquina 4
2.1.2 Redes Neurais Artificiais 6
2.1.3 Redes *Perceptron* Multicamadas 7
2.2 Métricas de Desempenho 8
2.2.1 Erro Quadrático Médio (MSE) 9
2.2.2 Raiz do Erro Quadrático Médio (RMSE) 9
2.2.3 Erro Médio Absoluto (MAE) 9
2.2.4 Coeficiente de Determinação (R^2) 9
2.2.5 Erro Mediano Absoluto (MedAE) 10
2.2.6 Erro Percentual Absoluto Médio (MAPE) 10
2.3 TinyML 10
2.4 Implementação de Modelos em Dispositivos Embarcados 11
2.4.1 *TensorFlow* 11
2.4.2 *TensorFLow Lite* 12
2.5 Monitoramento da Qualidade do Ar 14
2.5.1 Amostrador de Grande Volume (AGV) 14
2.5.2 Análise por Fotometria de Chama 15
2.5.3 Análise por Titulação 16
2.5.4 Exemplo prático para comparação de custos 17
2.6 Conclusão do Capítulo 17

3 Metodologia **19**
3.1 Tecnologias Utilizadas . . . 19
3.2 Escolha e Análise dos Dados . . . 20
3.3 Pré-processamento dos dados . . . 22
3.4 Desenvolvimento e especialização do Modelo . . . 23
3.4.1 Construção do Modelo . . . 24
3.4.2 Especialização do Modelo . . . 26
3.5 Exportação do modelo para TensorFlow Lite . . . 26
3.5.1 Quantização de faixa dinâmica . . . 27
3.5.2 Quantização *Float16* . . . 27
3.5.3 Quantização inteira completa . . . 28
3.5.4 Quantização somente inteiro . . . 28
3.6 Execução dos modelos *TFLite* no *Python* . . . 29
3.7 Exportação do modelo para o microcontrolador . . . 30
3.7.1 Incorporação de Modelo *TensorFlow Lite* em Código C/C++ . . . 30
3.7.2 Desenvolvimento do código-fonte em C/C++ . . . 31
3.8 Conclusão do Capítulo . . . 34

4 Resultados **35**
4.1 Volume dos modelos quantizados . . . 35
4.2 Precisão dos Modelos em *Python* . . . 36
4.2.1 Modelo Original . . . 37
4.2.2 Modelos Quantizados . . . 37
4.2.3 Métricas de Desempenho . . . 39
4.3 Resultados Obtidos do Microcontrolador . . . 40
4.3.1 Análise dos valores obtidos . . . 41
4.3.2 Desempenho do Modelo no Microcontrolador . . . 41
4.4 Comparação Entre Metodologia usual e proposta . . . 44
4.5 Conclusão do Capítulo . . . 47

5 Conclusão **48**

Lista de Tabelas

2.1 Técnicas de Quantização 14
2.2 Cores das chamas e comprimentos de onda de diferentes elementos 16
3.1 Tecnologias utilizadas separadas por categoria. 20
3.2 Matriz de correlação entre as variáveis presentes na base. 20
3.3 Estatísticas descritivas das variáveis presentes na base. 21
3.4 Matriz de Correlação entre Sódio, Potássio e Cloro Após Normalização. 23
3.5 Resumo do modelo. 25
3.6 Técnicas de Quantização 26
3.7 Métricas de desempenho para os modelos otimizados 30
4.1 Métricas de desempenho para os modelos otimizados 36
4.2 Comparativo de Métricas de Desempenho para Modelos Quantizados e Keras . 39
4.3 Equipamentos Utilizados 44
4.4 Custo 45
4.5 Tempo de Análise 45
4.6 Complexidade Operacional 46
4.7 Precisão e Sensibilidade 46

Lista de Códigos

3.1 Construção do Modelo 25
3.2 Compilação e treinamento do modelo 25
3.3 Quantização de faixa dinâmica. 27
3.4 Quantização Float16. 28
3.5 Quantização inteira completa. 28
3.6 Quantização somente inteiro. 29
3.7 Compilação e exportação do modelo para a ESP32. 33
3.8 Pseudocódigo para o fluxo principal. 34
4.1 Exemplo de Saída do sensor virtual. 40

Lista de Abreviaturas e Siglas

TinyML	*Tiny Machine Learning*
IA	*Inteligência Artificial*
ESP32-S3	*Plataforma de Microcontrolador ESP32-S3*
TensorFlow Lite	*Biblioteca para Execução de Modelos em Dispositivos de Baixo Custo*
MLP	*Multi-Layer Perceptron*
MSE	*Mean Squared Error*
MAE	*Mean Absolute Error*
RMSE	*Root Mean Squared Error*
MEDAE	*Median Absolute Error*
MAPE	*Mean Absolute Percentage Error*
GCC	*GNU Compiler Collection*
PSRAM	*Pseudo Static Random Access Memory*
SPI	*Serial Peripheral Interface*
API	*Application Programming Interface*
MB	*Megabyte*
kB	*Kilobyte*
SRAM	*Static Random Access Memory*
PTS	*Partículas totais em suspensão*
AGV	*Amostrador de grande volume*
GPU	*Graphics Processing Unit*
CPU	*Central Processing Unit*
Na	*Sódio*
K	*Potássio*
Cl	*Cloro*

Lista de Figuras

1.1 Metodologia usual. 2
1.2 Metodologia proposta. 2
2.1 Técnicas de aprendizado de máquina. 5
2.2 Modelo de neurônio de McCulloch e Pitts com n entradas. 7
2.3 Configuração de uma rede MLP usual. 8
2.4 Paradigma AIoT 11
2.5 Pipeline TensorFlow Lite 12
2.6 Amostrador de grande volume. 15
3.1 Distribuição dos Dados Originais. 22
3.2 Distribuição dos Dados Após Normalização 23
3.3 Pipeline de execução para o moelo TFLite no Python. 29
3.4 Arquitetura do modelo otimizado. 32
4.1 Tamanho dos modelos quantizados e sua porcentagem de redução. 36
4.2 Gráfico de dispersão - Modelo Keras. 37
4.3 Gráfico de dispersão - Modelos Quantizados. 38
4.4 Gráfico de dispersão - Sensor virtual. 41
4.5 Consumo de Corrente ao Longo do Tempo. 42

1

Introdução

A poluição atmosférica, caracterizada pela presença de substâncias nocivas no ar, é um problema crítico que impacta negativamente a saúde pública, o bem-estar da comunidade e o meio ambiente. Os poluentes atmosféricos são classificados em primários, emitidos diretamente pelas fontes de emissão; e secundários, formados na atmosfera por meio de reações químicas.

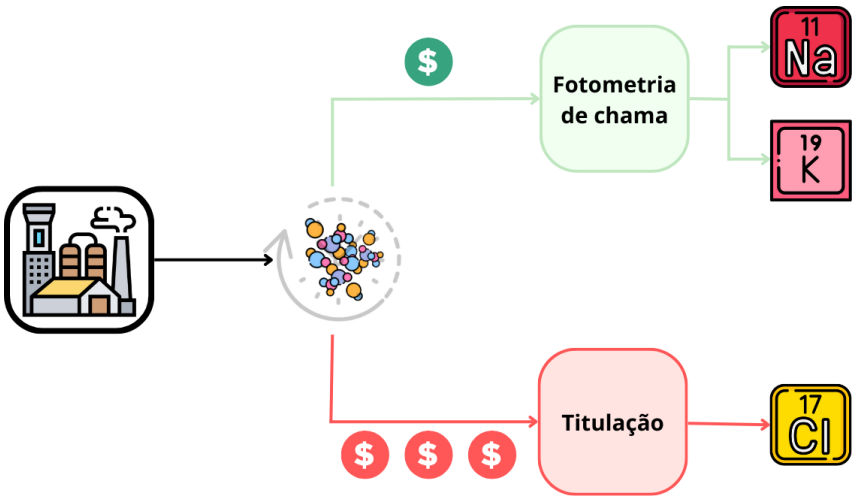
Em setores como a mineração, a emissão de partículas é significativa, tornando essencial o monitoramento rigoroso para garantir que as concentrações de poluentes estejam dentro dos padrões ambientais estabelecidos pela legislação. Neste contexto, as técnicas de inteligência artificial surgem como ferramentas importantes para aprimorar o processo de previsão e análise desses poluentes, oferecendo soluções eficazes para o monitoramento ambiental.

1.1 Motivação

O setor industrial, principalmente as industrias de mineração, destacam-se como uma das principais responsáveis pela emissão de partículas no ar, decorrente das escavações e movimentações de terra necessárias para a extração de minérios. Apesar da adoção de medidas mitigadoras, como a aspersão de vias e a implementação de faixas arbóreas, o volume de poluentes gerado ainda é expressivo (CARVALHO et al., 2007).

Os métodos tradicionais de monitoramento da qualidade do ar, ilustrados na Figura 1.1, baseiam-se em técnicas laboratoriais como a fotometria de chama, para a medição dos níveis de sódio e potássio, e a titulação argentimétrica, utilizada na análise de cloro (KOTZ et al., 2014). Embora precisos, esses métodos apresentam limitações consideráveis, incluindo o alto custo operacional, a necessidade de equipamentos laboratoriais e mão de obra especializada, além do tempo demandado para a coleta, transporte e processamento das amostras. Essas dificuldades tornam o monitoramento contínuo um processo oneroso e, muitas vezes, inviável em grande escala.

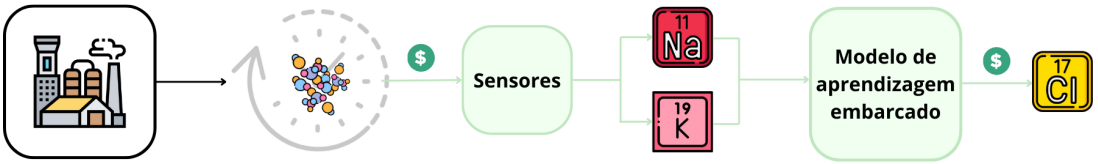
Figura 1.1: Metodologia usual. O fluxo em verde indica um processo mais simples e econômico ($), enquanto o fluxo em vermelho representa um processo mais caro ($$$), com maior complexidade e custo operacional.



Fonte: Autor.

Diante disso, a busca por soluções alternativas e mais acessíveis torna-se imperativa. Neste contexto, os sensores virtuais, integrados às técnicas de aprendizado de máquina embarcado do termo inglês *Tiny Machine Learning* ou simplesmente *TinyML*, despontam como uma solução promissora. A imagem 1.2 apresenta o fluxo proposto para ser possível a realização do barateamento do processo.

Figura 1.2: Metodologia proposta. O fluxo apresenta medições diretas de sódio (Na) e potássio (K), seguidas de análise preditiva com inteligência artificial embarcada (*TinyML*) para estimar os níveis de cloro (Cl). O processo é simplificado e otimizado em termos de custo ($) em comparação com métodos tradicionais.



Fonte: Autor.

Além de tornar o monitoramento mais acessível, a aplicação do *TinyML* em sensores embarcados possibilita uma análise em tempo real dos poluentes, permitindo respostas rápidas e assertivas em ambientes industriais, como o setor de mineração. Resolver o problema na borda é importante, pois reduz a latência, aumenta a eficiência energética, garante maior privacidade e proporciona escalabilidade ao sistema.

Fazer a inferência diretamente no dispositivo assegura que o monitoramento seja contínuo, mesmo em condições adversas, como a auxenia de internet, promovendo não apenas a eficiência no monitoramento da qualidade do ar, mas também uma gestão ambiental mais proativa e eficaz.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver uma metodologia que integre técnicas de inteligência artificial embarcadas (*TinyML*) para criar um sensor virtual capaz de monitorar, de maneira eficaz e precisa, as concentrações de poluentes, especificamente o cloro no ar, utilizando dados obtidos de sensores de potássio e sódio, que realizam medições diretas e em tempo real. Com essa abordagem, busca-se eliminar a necessidade de análises laboratoriais, como a titulação, automatizando o processo de monitoramento, reduzindo custos a longo prazo e favorecendo a escalabilidade da solução.

1.3 Estrutura do Trabalho

Este trabalho está organizado em cinco capítulos, conforme descrito a seguir: No Capítulo 1, apresentamos a introdução, onde contextualizamos o problema e discutimos a importância do monitoramento da qualidade do ar em ambientes industriais, especialmente no setor de mineração, além dos objetivos do trabalho. O Capítulo 2 aborda o referencial teórico, onde exploramos conceitos como aprendizado de máquina e redes neurais, discutindo também o uso de *TinyML* e inteligência artificial embarcada em dispositivos de baixo custo, como o ESP32-S3, para monitoramento ambiental. No Capítulo 3, detalhamos a metodologia utilizada neste trabalho, incluindo a escolha das variáveis de entrada, como potássio e sódio, o desenvolvimento dos modelos preditivos para estimar a concentração de cloro e as etapas de treinamento e quantização dos modelos para execução em dispositivos embarcados. O Capítulo 4 é dedicado à apresentação e discussão dos resultados obtidos, com foco na redução do tamanho dos modelos após a quantização, a análise do desempenho preditivo, e a comparação entre a metodologia proposta e os métodos tradicionais de monitoramento da qualidade do ar. Finalmente, no Capítulo 5, apresentamos as conclusões do trabalho, bem como as principais contribuições, limitações e sugestões para trabalhos futuros.

2

Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais que sustentam o desenvolvimento do trabalho, iniciando com uma introdução ao aprendizado de máquina e ao modelo tradicional de monitoramento da qualidade do ar. Em seguida, explora as redes neurais, suas arquiteturas e aplicações na predição de variáveis ambientais. O conceito de *TinyML* é destacado, enfatizando a relação entre aprendizado de máquina e sistemas embarcados, com a necessidade de otimização de modelos para dispositivos de recursos limitados, como microcontroladores. O foco teórico está na aplicação prática do *TinyML*, especialmente no processo de quantização de modelos e execução em sistemas embarcados, com recomendações para consulta de referências para maior aprofundamento.

2.1 Aprendizado de máquina

Aprendizado de máquina é um campo da da computação que permite aos computadores adquirirem a capacidade de aprender a partir de dados, sem a necessidade de serem explicitamente programados para cada tarefa (SAMUEL, 1959).

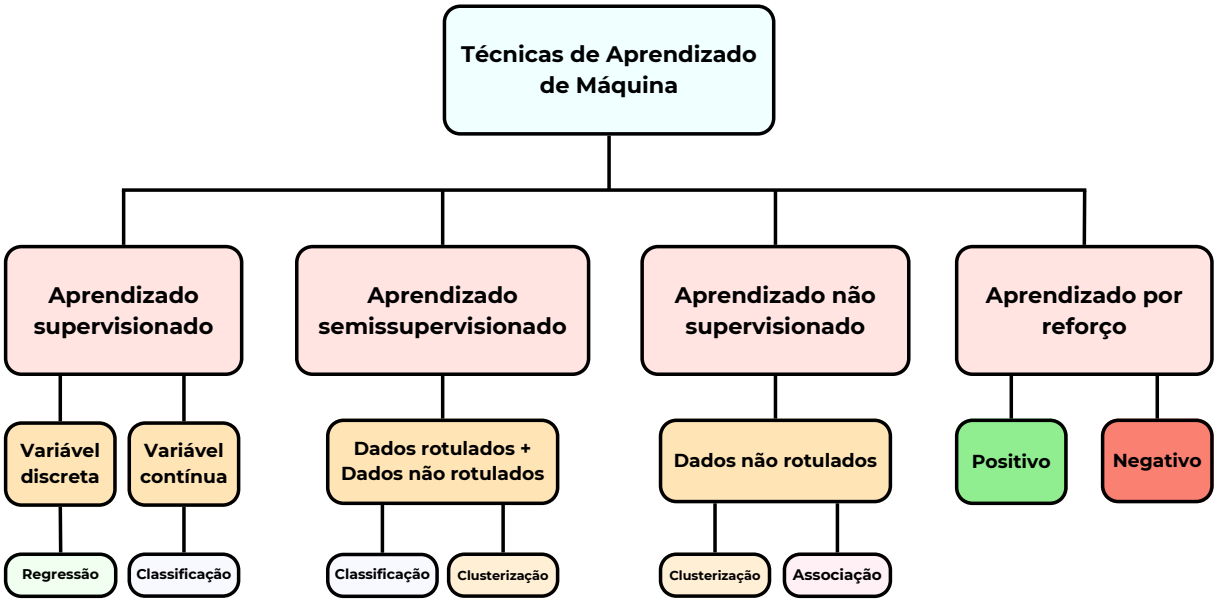
Esse processo envolve o treinamento de modelos utilizando conjuntos de dados, permitindo que eles identifiquem padrões e relações subjacentes. Após o treinamento, o modelo é capaz de aplicar o conhecimento adquirido para interpretar novos dados e executar tarefas como classificação, previsão, reconhecimento de padrões e tomada de decisões de forma autônoma. Dessa forma, o aprendizado de máquina amplia as possibilidades de automação e otimização em diversas áreas, incluindo o monitoramento ambiental (ZANNETTI, 1990).

2.1.1 Técnicas de Aprendizado de Máquina

As técnicas de aprendizado de máquina podem ser classificados em quatro categorias principais: aprendizado supervisionado, aprendizado não supervisionado, aprendizado semissupervisionado e aprendizado por reforço (SARKER, 2021), conforme mostrado na Figura 2.1. Cada

uma dessas categorias tem uma aplicação distinta, dependendo do tipo de problema e da natureza dos dados disponíveis. A seguir, cada técnica será descrita brevemente, com destaque para suas características e aplicações.

Figura 2.1: Técnicas de aprendizado de máquina.



Fonte: Adaptado (SARKER, 2021).

Aprendizagem supervisionada

O aprendizado supervisionado é utilizado para mapear entradas a saídas com base em pares de dados rotulados (HAN; PEI; TONG, 2022). O processo envolve o treinamento de um modelo em um conjunto de dados em que as entradas estão associadas a saídas conhecidas, permitindo que o modelo generalize para novos dados. As tarefas supervisionadas mais comuns são a "classificação", usada para categorizar dados, e a "regressão", aplicada para prever valores contínuos. Um exemplo de classificação seria determinar se a qualidade do ar em uma área é "boa" ou "ruim" com base nos dados dos sensores, enquanto a regressão pode prever a concentração de um poluente no ar com base em variáveis ambientais.

Aprendizagem não supervisionada

O aprendizado não supervisionado trabalha com dados não rotulados, analisando-os para identificar padrões ou estruturas subjacentes (HAN; PEI; TONG, 2022). Suas principais aplicações incluem agrupamento (clusterização), redução de dimensionalidade e detecção de anomalias. Por exemplo, essa técnica pode ser usada para segmentar diferentes zonas em um ambiente industrial, sem necessidade de rotulação prévia.

Aprendizagem semi-supervisionada

A aprendizagem semi-supervisionada combina aspectos do aprendizado supervisionado e não supervisionado, operando com dados rotulados e não rotulados (SARKER et al., 2020). É particularmente útil em situações em que há poucos dados rotulados e uma abundância de dados não rotulados, aproveitando essas informações adicionais para melhorar a precisão do modelo. Um exemplo prático é a predição de poluentes, onde o método usa grandes volumes de dados não rotulados, coletados por sensores, juntamente com um conjunto menor de medições rotuladas de poluentes. O modelo aprende com os dados rotulados e aplica esse conhecimento para inferir a concentração de poluentes nos dados não rotulados.

Aprendizado por reforço

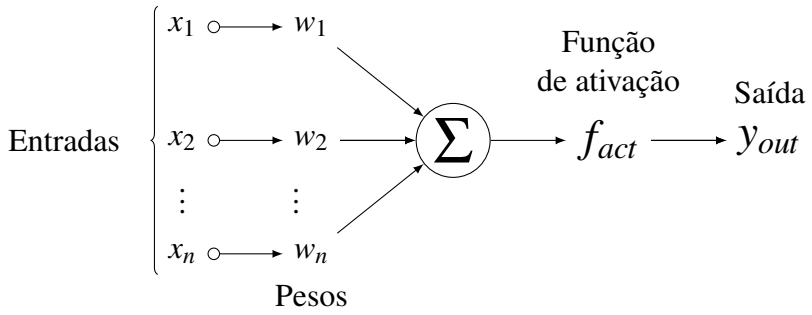
O aprendizado por reforço permite que agentes aprendam a tomar decisões com base em interações com o ambiente, utilizando recompensas e penalidades para ajustar seu comportamento (KAELBLING; LITTMAN; MOORE, 1996). O objetivo é maximizar recompensas ao longo do tempo ou minimizar riscos associados às ações (MOHAMMED; KHAN; BASHIER, 2016). Como exemplo, pode ser citado o desenvolvimento de um sistema de controle para ajustar dinamicamente a ventilação em ambientes fechados, com base nos níveis de poluentes detectados. O agente de aprendizado por reforço recebe recompensas positivas quando consegue reduzir a concentração de poluentes para níveis aceitáveis, e penalidades quando a qualidade do ar se deteriora.

2.1.2 Redes Neurais Artificiais

As redes neurais artificiais (RNAs) desempenham um papel central no aprendizado de máquina, especialmente em problemas complexos de reconhecimento de padrões (BISHOP, 1994). Sua importância se deve à sua flexibilidade estrutural, permitindo que sejam adaptadas para diferentes tipos de problemas em todas as principais categorias de aprendizado de máquina, incluindo aprendizado supervisionado, não supervisionado, semi-supervisionado e por reforço (SARKER, 2021). A maleabilidade das RNAs as torna adequadas para uma ampla gama de aplicações, desde a classificação de dados até a predição de valores contínuos.

Inspiradas pela estrutura neural de organismos biológicos, as RNAs simulam a maneira como o cérebro humano processa informações e adquire conhecimento por meio de experiências (NEGNEVITSKY, 2011). O primeiro modelo de neurônio artificial foi proposto por McCulloch e Pitts em 1943, em uma tentativa de replicar o comportamento de um neurônio biológico (MCCULLOCH; PITTS, 1943). Este modelo serviu de base para o desenvolvimento posterior das redes neurais, compostas por múltiplos neurônios artificiais conectados em camadas.

Figura 2.2: Modelo de neurônio de McCulloch e Pitts com *n* entradas.



Fonte: Adaptado (MCCULLOCH; PITTS, 1943; JUNIOR, 2020)

O modelo de neurônio de McCulloch e Pitts é ilustrado na Figura 2.2, e consiste em um conjunto de entradas $X = \{x_1, x_2, ..., x_n\}$, cada uma associada a um peso w_i. O neurônio processa essas entradas ponderadas e, em seguida, aplica uma função de ativação, que decide se o neurônio irá "disparar" ou não, similar ao processo de sinapse no cérebro (RASCHKA; MIRJALILI, 2017). A equação que descreve o funcionamento desse neurônio é mostrada a seguir.

$$z = x_0 w_0 + x_1 w_1 + \cdots + x_n w_n = \sum_{i=0}^{n} x_i w_i = w^T x \tag{2.1}$$

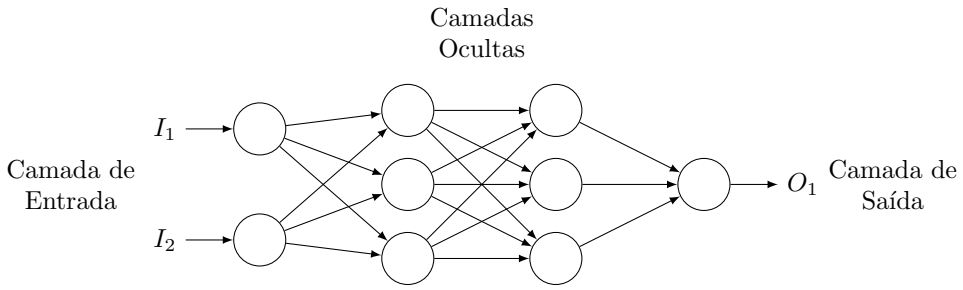
Nesta equação, z representa a combinação linear das entradas x_i e dos pesos w_i, resultando em uma soma ponderada das entradas. O valor z é então utilizado como entrada para uma função de ativação, que determina a saída do neurônio. A notação compacta $w^T x$ facilita o cálculo, representando o produto escalar entre os vetores de pesos e entradas, o que permite uma implementação eficiente em algoritmos de redes neurais.

2.1.3 Redes *Perceptron* Multicamadas

As redes *Perceptron* Multicamadas (do termo em inglês *multi layer perceptro*n, MLP) são conhecidas por sua capacidade de processar informações de forma sequencial, no que é chamado de propagação direta (do termo em inglês *feedforward*). Neste processo, as informações percorrem as camadas da rede, indo da camada de entrada até a camada de saída, passando pelas camadas intermediárias, chamadas de camadas ocultas. Essa estrutura permite a formação de conexões sinápticas entre os neurônios de camadas adjacentes, o que possibilita o aprendizado de diferentes tipos de relações entre os dados. O algoritmo de retropropagação do erro (do termo em inglês *backpropagation*), que se popularizou no treinamento de redes multicamadas, é a técnica usada para ajustar os pesos sinápticos com base nos erros observados, refinando assim a capacidade da rede de generalizar.

Uma das características mais marcantes das MLPs é sua capacidade de aproximar funções contínuas arbitrárias, mesmo com apenas uma camada oculta e uma função de ativação adequada. Isso as tornam extremamente flexíveis e eficazes em diversas tarefas de aprendizado, como classificação, regressão e reconhecimento de padrões.

Figura 2.3: Configuração de uma rede MLP usual.



Fonte: Autor.

A configuração de uma MLP típica é ilustrada na Figura 2.3, com duas variáveis de entrada, duas camadas ocultas e uma variável de saída. Nesta arquitetura, o fluxo de propagação é representado pela fase *forward*, enquanto a fase de correção, ou *backward*, ocorre quando os pesos sinápticos são ajustados após o cálculo dos erros.

Por definição, as redes MLP são compostas por pelo menos três camadas, cada uma desempenhando um papel específico no processamento dos dados:

- **Camada de entrada:** Responsável por receber as variáveis de entrada (também conhecidas como variáveis independentes). Os neurônios dessa camada distribuem essas informações para as camadas subsequentes, iniciando o processo de propagação de dados.
- **Camadas ocultas:** Os neurônios dessas camadas processam as informações recebidas, aplicando regras de propagação e funções de ativação. Dessa forma, as camadas ocultas transformam as entradas em representações que serão úteis para a camada de saída gerar o resultado.
- **Camada de saída:** Esta camada contém os neurônios que produzem as respostas finais da rede, ou seja, as variáveis de saída. Esses neurônios consolidam o processamento realizado pelas camadas anteriores, fornecendo o resultado da inferência realizada pela rede.

O treinamento eficiente das redes MLP, por meio da retropropagação, permite que elas aprendam representações úteis dos dados de forma automática.

2.2 Métricas de Desempenho

Para avaliar a qualidade de um modelo de regressão, é necessário utilizar métricas adequadas que quantifiquem o desempenho do modelo em relação aos dados observados. Nesta seção, serão apresentadas as principais métricas empregadas neste trabalho: Erro Quadrático Médio (do termo em inglês *Mean Squared Error*, MSE), Raiz do Erro Quadrático Médio (do termo em inglês *Root Mean Squared Error*, RMSE), Erro Médio Absoluto (do termo em inglês *Mean*

Absolute Error, MAE), Coeficiente de Determinação (R^2), Erro Mediano Absoluto (do termo em inglês *Median Absolute Error*, MedAE) e Erro Percentual Absoluto Médio (do termo em inglês *Mean Absolute Percentage Error*, MAPE).

2.2.1 Erro Quadrático Médio (MSE)

O Erro Quadrático Médio (MSE) é uma das métricas mais utilizadas para medir a diferença entre os valores preditos pelo modelo ($\hat{y}_i$) e os valores reais (y_i). O MSE penaliza erros maiores de forma mais significativa, pois eleva ao quadrado as diferenças entre os valores preditos e observados. Sua fórmula é dada por:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.2}$$

onde: n é o número de observações, y_i são os valores observados e $\hat{y}_i$ são os valores preditos pelo modelo.

2.2.2 Raiz do Erro Quadrático Médio (RMSE)

O RMSE (Root Mean Squared Error) é a raiz quadrada do MSE, o que traz a métrica para a mesma escala dos dados originais. A fórmula para o cálculo do RMSE é a seguinte:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2.3}$$

O RMSE é mais intuitivo de interpretar em muitos casos, pois expressa o erro médio de forma linear, facilitando a comparação direta com os valores preditos.

2.2.3 Erro Médio Absoluto (MAE)

O Erro Médio Absoluto (MAE) é uma métrica que calcula a média das diferenças absolutas entre os valores preditos e os valores observados. Ao contrário do MSE, o MAE não eleva as diferenças ao quadrado, o que significa que ele é menos sensível a grandes erros. A fórmula é dada por:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.4}$$

2.2.4 Coeficiente de Determinação (R^2)

O Coeficiente de Determinação (R^2) mede a proporção da variância dos dados observados explicada pelo modelo. Um valor de R próximo de 1 indica um bom ajuste do modelo aos dados,

enquanto um valor próximo de 0 indica que o modelo não explica bem a variância dos dados. A fórmula do R é dada por:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{2.5}$$

onde: $\bar{y}$ é a média dos valores observados.

2.2.5 Erro Mediano Absoluto (MedAE)

O Erro Mediano Absoluto (MedAE) é uma métrica que calcula a mediana das diferenças absolutas entre os valores observados e preditos. Esta métrica é menos afetada por outliers, sendo particularmente útil quando há valores discrepantes nos dados:

$$MedAE = \text{mediana}(|y_i - \hat{y}_i|) \tag{2.6}$$

2.2.6 Erro Percentual Absoluto Médio (MAPE)

O Erro Percentual Absoluto Médio (MAPE) calcula o erro percentual médio das previsões em relação aos valores reais. Ele é utilizado para medir a precisão em modelos de séries temporais, por exemplo. A fórmula é dada por:

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{2.7}$$

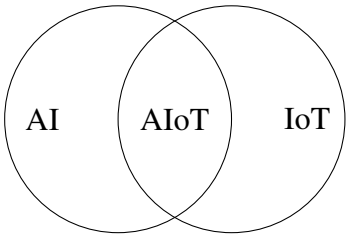
O MAPE expressa o erro em termos percentuais, o que facilita a interpretação quando se quer comparar o desempenho do modelo em diferentes contextos.

2.3 TinyML

O *TinyML*, procura adaptar modelos de aprendizado de máquina para serem suficientemente compactos e possam ser implantados em dispositivos de borda com capacidades limitadas de armazenamento e processamento (WARDEN; SITUNAYAKE, 2020). Esta abordagem surgiu da necessidade de realizar o processamento e a interpretação de dados em tempo real, diretamente em dispositivos de baixo custo, sem a necessidade de transmissão contínua para a nuvem (IODICE, 2022).

A aplicação da Inteligência Artificial integrada a dispositivos IoT destaca o surgimento de uma nova geração de dispositivos inteligentes, denominada AIoT (do termo em inglês *Artificial Intelligence of Things*) (DONG et al., 2021), como podemos ver na Figura 2.4. Essa combinação de técnicas de coleta de dados por meio de dispositivos IoT com a capacidade de tomada de decisão local, utilizando TinyML, proporciona maior eficiência e independência desses sistemas, eliminando a necessidade de redes externas.

Figura 2.4: Paradigma AIoT



Fonte: Autor.

Dentro deste cenário, Banbury et al. (2021) exploraram as limitações associadas à implementação de modelos TinyML, evidenciando as seguintes restrições:

- **Baixo consumo de energia**: O consumo energético é um dos fatores críticos em sistemas TinyML, sendo preciso otimizar o uso de energia para viabilizar a implantação desses sistemas em dispositivos de borda, que possuem recursos energéticos limitados.
- **Limitações de memória**: enquanto os sistemas tradicionais de aprendizado de máquina operam com grandes volumes de memória, na ordem de gigabytes, os sistemas TinyML devem ser implementados em dispositivos com restrições de memória que variam na ordem de megabytes, o que exige otimizações para garantir seu funcionamento adequado.

2.4 Implementação de Modelos em Dispositivos Embarcados

Os dispositivos TinyML podem ser implementados tanto em sistemas centralizados quanto distribuídos (IODICE, 2022). No entanto, até o momento, não há uma estrutura unificada para a implementação de *TinyML* em sistemas embarcados. A implantação de redes neurais nesses sistemas requer a criação de arquiteturas específicas, que precisam ser otimizadas manualmente para cada plataforma de hardware. Estas arquiteturas personalizadas geralmente são desenvolvidas com um foco restrito, adaptadas a um único aplicativo e sem portabilidade para diferentes dispositivos. Isso aumenta a complexidade do processo de desenvolvimento, uma vez que os ajustes manuais são necessários para garantir que o modelo de aprendizado de máquina funcione corretamente em um hardware específico (DAVID et al., 2021). Neste cenário, o uso do *TensorFlow Lite* surge como uma solução promissora, facilitando a implementação de modelos *TinyML* em dispositivos de borda.

2.4.1 *TensorFlow*

O *TensorFlow* é uma plataforma de código aberto voltada para o desenvolvimento de modelos de aprendizado de máquina (ABADI et al., 2015). Oferecendo suporte para uma ampla gama de aplicações, com foco em treinamento e inferência de redes neurais profundas, o *TensorFlow* foi construído com base em grafos computacionais e realiza operações de alta escalabilidade. Para

isso, utiliza tensores como estrutura fundamental para representar e consolidar as informações dos modelos (ABADI et al., 2015).

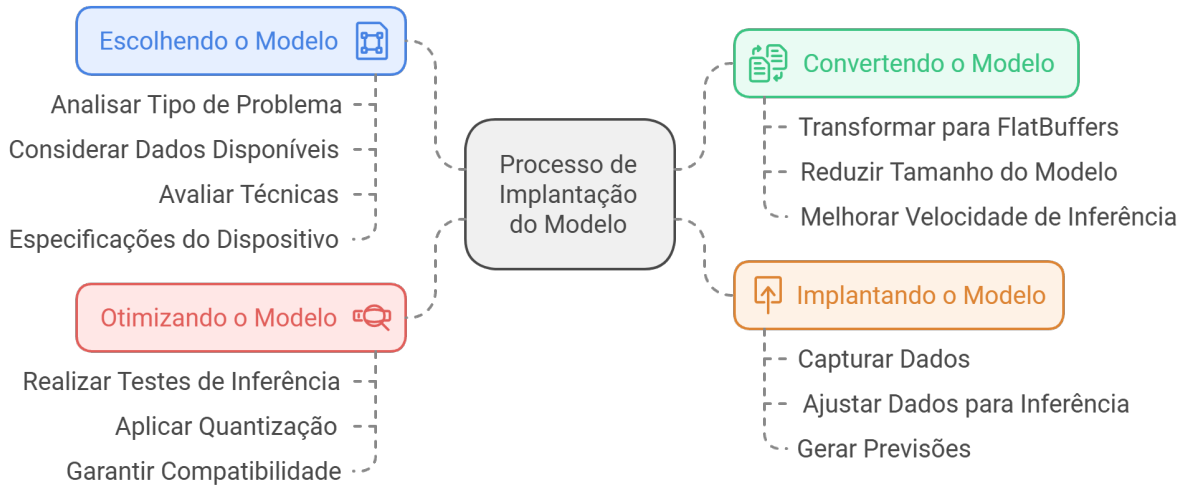
Matematicamente, os tensores podem ser vistos como uma generalização de objetos como escalares, vetores e matrizes. No contexto do *TensorFlow*, o tensor é tratado como um identificador simbólico que representa as entradas e saídas das operações (RASCHKA, 2015). A *API* do *TensorFlow* trata todos os tensores como matrizes de n-dimensões, permitindo que os cálculos sejam executados de forma eficiente e escalável (ABADI et al., 2015).

2.4.2 *TensorFLow Lite*

TensorFlow Lite representa uma extensão do *TensorFlow*, projetada especificamente para rodar modelos de aprendizado de máquina em dispositivos com capacidade de processamento e memória limitados (DAVID et al., 2021). Essa característica se faz necessária, pois permite a execução de modelos de inteligência artificial em sistemas embarcados, que requerem um intérprete adaptado para operar dentro das restrições de hardware do dispositivo (WARDEN; SITUNAYAKE, 2020).

A documentação do *TensorFlow Lite* fornece um guia detalhado para a implementação de modelos em dispositivos de borda. Este processo é ilustrado pela Figura 2.5, que descreve as etapas fundamentais para a implantação eficaz de um modelo *TinyML*.

Figura 2.5: Pipeline TensorFlow Lite



Fonte: Adaptado (ABADI et al., 2015).

Escolha do modelo de aprendizagem

Ao escolher uma arquitetura de modelo, é preciso analisar o tipo de problema que se pretende resolver, bem como os dados disponíveis e as técnicas adequadas para processá-los antes de serem utilizados no modelo (WARDEN; SITUNAYAKE, 2020). A relação entre os dados e

a arquitetura do modelo é profundamente interdependente. Portanto, se faz necessário levar em consideração as especificidades do dispositivo de destino para garantir que a solução seja eficiente e viável dentro das limitações de hardware (IODICE, 2022).

Conversão do modelo de aprendizagem

A conversão de um modelo para o formato *TensorFlow Lite* envolve transformá-lo em um formato *FlatBuffers*[1], representado pela extensão *TensorFlow Lite* (`.tflite`). Esse processo oferece vantagens em relação ao formato padrão de *buffer* de protocolo do *TensorFlow*, incluindo a redução significativa do tamanho do modelo e uma inferência mais rápida. Essas melhorias tornam o *TensorFlow Lite* especialmente adequado para dispositivos com restrições de memória e processamento limitado (ABADI et al., 2015).

Implantação do modelo de aprendizagem

A implantação envolve a execução de um modelo TensorFlow Lite diretamente no dispositivo, visando realizar previsões a partir dos dados de entrada. Nessa fase, é necessário desenvolver códigos que capturem os dados, ajustando-os para que o modelo possa realizar a inferência no dispositivo de maneira eficiente. O resultado desse processo é uma saída contendo as previsões geradas pelo modelo (ABADI et al., 2015).

Otimização do modelo de aprendizagem

Após a implantação do modelo, são realizados testes de inferência com dados reais para avaliar o desempenho do modelo no dispositivo de borda. Como esses dispositivos possuem memória e capacidade de processamento limitadas, otimizações podem ser aplicadas para garantir que o modelo funcione dentro dessas restrições. Essas otimizações, chamadas neste contexto de quantização[2], pretendem reduzir o tamanho do modelo, diminuir a latência e garantir a compatibilidade com o hardware de destino (WARDEN; SITUNAYAKE, 2020).

É importante observar que técnicas de quantização podem resultar em uma leve perda de precisão no modelo. Existe um balanço entre otimizar e manter sua precisão. Assim, cabe à equipe de desenvolvimento definir até que ponto essas otimizações podem ser aplicadas sem comprometer o desempenho geral do modelo (YANG et al., 2019). A Tabela 2.1 detalha fatores a serem considerados para escolha da técnica de quantização, disponibilizadas no *Tensorflow Lite* e apresenta os hardwares compatíveis conforme as quantizações, afirmando que cada forma

[1] Um FlatBuffer é um buffer binário contendo objetos aninhados (structs, tabelas, vetores...) organizados usando deslocamentos para que os dados possam ser percorridos no local como qualquer estrutura de dados baseada em ponteiro. Maiores detalhes<https://flatbuffers.dev/>

[2] Busca reduzir a precisão dos números usados para representar os parâmetros de um modelo, que por padrão apresenta pesos e operações em formato de ponto flutuante de 32 bits. Entre os tipos disponibilizados na documentação do Tensorflow Lite: <https://ai.google.dev/edge/litert/models/model_optimization?hl=pt-br>

de quantização apresenta alterações na precisão do modelo quando comparado ao modelo original. Mais detalhes sobre as quantizações serão discutas no Capítulo 3, seção 3.5.

Tabela 2.1: Técnicas de Quantização

Técnica	Requisitos de Dados	Redução de Tamanho	Precisão	Hardware Compatível
Quantização float16 pós-treinamento	Não há dados	Até 50%	Perda de acurácia insignificante	CPU, GPU
Quantização de intervalo dinâmico pós-treinamento	Não há dados	Até 75%	Menor perda de precisão	CPU, GPU (Android)
Quantização de números inteiros pós-treinamento	Amostra representativa sem rótulo	Até 75%	Pequena perda de precisão	CPU, GPU (Android), EdgeTPU
Treinamento com reconhecimento de quantização	Dados de treinamento rotulados	Até 75%	Menor perda de precisão	CPU, GPU (Android), EdgeTPU

Fonte: Adaptado(ABADI et al., 2015)

2.5 Monitoramento da Qualidade do Ar

O monitoramento da qualidade do ar é fundamental para medir a concentração de poluentes e avaliar seus impactos na saúde pública e no ambiente. Diversos métodos e equipamentos são usados para coletar e analisar amostras de ar, cada um adequado a diferentes tipos de poluentes e condições ambientais. A seguir, são descritos os métodos ilustrados na Figura 1.1.

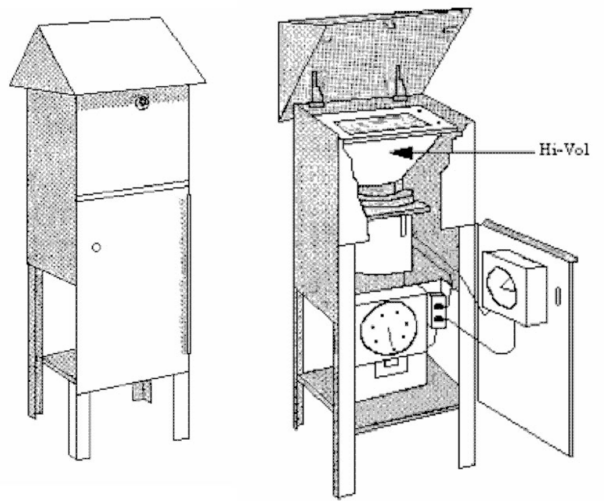
2.5.1 Amostrador de Grande Volume (AGV)

Para o monitoramento em questão é feito uso, com frequência, do equipamento chamado amostrador de grande volume (AGV). Esse equipamento coleta amostras de ar ambiente durante períodos definidos, utilizando filtros instalados em abrigos específicos. O ar é passado através dos filtros, que capturam partículas em suspensão. Após o período de amostragem, os filtros são pesados antes e depois da coleta para calcular o ganho líquido de massa (DIAS, 2016).Um exemplo de um amostrador de grande volume é ilustrado na Figura 2.6.

A concentração de partículas totais em suspensão (PTS) é calculada dividindo-se a massa das partículas coletadas pelo volume de ar amostrado, e expressa em microgramas por metro cúbico ($\mu g/m^3$), conforme a Equação 2.8:

$$PTS = \left(10^6\right) \frac{M_L}{V_p} \tag{2.8}$$

Figura 2.6: Amostrador de grande volume. Na imagem, à esquerda, é mostrado o AGV em sua forma fechada, enquanto à direita, vemos uma visão interna com detalhes de seu funcionamento. O dispositivo possui um compartimento superior onde o ar é filtrado, sendo direcionado para os filtros em um funil identificado como "Hi-Vol". Esses filtros capturam as partículas em suspensão, permitindo que, posteriormente, as amostras sejam analisadas para verificar o conteúdo de poluentes no ar.



Fonte: (CESAR et al., 2004) p. 10.

onde M_L é o ganho líquido de massa no filtro em gramas; V_p volume total de ar amostrado em m^3; e 10^6 é o fator de conversão, $\mu g/g$. O volume de ar amostrado, V_p, é obtido pela soma do produto entre a vazão média corrigida (Q_{pi}) e o tempo decorrido de amostragem (Δt), conforme descrito na Equação 2.9:

$$V_p = \sum_{i=1}^{n} Q_{pi} \Delta t \tag{2.9}$$

onde Q_{pi} representa a vazão média no intervalo i, corrigida para condições padrão (m^3/min); Δt representa tempo de amostragem em cada intervalo (min); e n é o número de intervalos durante a amostragem.

O custo de um amostrador de grande volume pode variar significativamente dependendo do fabricante, especificações técnicas e funcionalidades adicionais. Em geral, o valor de mercado de um AVG básico para coleta de ar em aplicações ambientais é estimado entre R$ 20.000 a R$ 50.000.

2.5.2 Análise por Fotometria de Chama

A fotometria de chama é uma técnica eficiente e de baixo custo usada para analisar quantitativamente certos metais presentes nas amostras de ar, como sódio (Na) e potássio (K). Esse método é amplamente utilizado devido à sua simplicidade e rapidez. O princípio da fotometria de chama envolve a excitação dos átomos em uma chama, com posterior emissão de luz em comprimentos de onda específicos quando os átomos retornam ao estado fundamental. A intensidade da luz

emitida é então utilizada para determinar a concentração dos elementos (HOLLER et al., 2009). Os comprimentos de onda emitidos por diferentes elementos e as respectivas cores da chama são apresentados na Tabela 2.2.

Tabela 2.2: Cores das chamas e comprimentos de onda de diferentes elementos

Elemento	Comprimento de onda emitido (nm)	Cor da chama
Potássio (K)	766	Violeta
Lítio (Li)	670	Vermelho
Cálcio (Ca)	622	Laranja
Sódio (Na)	589	Amarelo
Bário (Ba)	554	Verde-lima

Fonte: Autor.

A fotometria de chama é particularmente econômica quando comparada a outros métodos analíticos. O investimento inicial em um fotômetro de chama básico varia entre R$ 15.000 e R$ 50.000, dependendo das funcionalidades e da sensibilidade do equipamento. Modelos mais avançados e automatizados podem ultrapassar esse valor, podendo custar até R$ 100.000 em alguns casos. Além disso, os custos de manutenção e operação envolvem a calibração periódica do equipamento e a reposição de gases combustíveis (como propano ou acetileno), que podem gerar despesas mensais de R$ 1.000 a R$ 2.500, dependendo do volume de uso e da região.

2.5.3 Análise por Titulação

A titulação é uma técnica laboratorial usada para determinar a concentração de compostos em uma amostra por meio de uma reação química com um reagente de concentração conhecida, chamado titulante (KOTZ et al., 2018). Embora precisa, é uma técnica mais demorada e com maior custo quando comparada à fotometria de chama (BENVENUTO, 2022).

No contexto do monitoramento da qualidade do ar em ambientes industriais, a titulação é comumente utilizada para medir íons cloretos (Cl^-). O processo envolve a adição de uma solução padrão de nitrato de prata ($AgNO_3$), que reage com os íons cloreto para formar cloreto de prata (AgCl), um precipitado branco, conforme a reação química descrita a seguir:

$$Ag^+ + Cl^- \leftrightarrow AgCl$$

Quando o cloreto é totalmente precipitado, o excesso de íons prata reage com cromato ($CrO_4{}^{2-}$), formando um precipitado avermelhado de cromato de prata (Ag_2CrO_4), que indica o ponto final da titulação (KOTZ et al., 2018).

A titulação, embora altamente precisa, é mais custosa em termos de reagentes e equipamentos. Um kit de titulação básico, incluindo bureta, balança analítica, vidrarias e soluções padrão

(como nitrato de prata e indicadores químicos), pode custar entre R$ 5.000 e R$ 20.000, dependendo da precisão e da qualidade dos instrumentos (BENVENUTO, 2022). Além disso, o custo contínuo com reagentes para titulação, como o nitrato de prata ($AgNO_3$), varia entre R$ 500 a R$ 1.500 por litro, dependendo da pureza e da concentração da solução (KOTZ et al., 2018).

Apesar desses custos, a titulação continua sendo amplamente utilizada em análises laboratoriais devido à sua precisão e confiabilidade, especialmente em análises de compostos iônicos, como o cloreto (KOTZ et al., 2018).

2.5.4 Exemplo prático para comparação de custos

Embora ambas as técnicas sejam precisas, a fotometria de chama apresenta custos operacionais significativamente menores, especialmente em aplicações que envolvem um grande volume de amostras, devido à redução no consumo de reagentes e à maior rapidez na execução das análises.

Por exemplo, considere um cenário em que é necessário realizar cem análises de sódio e potássio em um laboratório. **No caso da fotometria de chama**, o investimento inicial em um fotômetro básico varia entre R$ 15.000 e R$ 50.000, com modelos mais avançados chegando a R$ 100.000. O custo operacional por amostra é relativamente baixo, envolvendo apenas o consumo de gases combustíveis, como propano ou acetileno, e energia elétrica, resultando em um custo aproximado de R$ 5,00 por análise. **Para cem análises, o custo operacional total seria de aproximadamente R$ 500,00, excluindo o investimento inicial no equipamento.**

Por outro lado, na titulação, o investimento inicial em um kit básico, composto por bureta, balança analítica e vidrarias, varia entre R$ 5.000 e R$ 20.000. No entanto, o custo operacional por análise é mais elevado, devido à necessidade de reagentes, como o nitrato de prata, e indicadores químicos. **Cada análise pode consumir cerca de R$ 15,00 em reagentes, resultando em um custo total de aproximadamente R$ 1.500,00 para cem análises.** Além disso, o processo manual da titulação exige maior tempo de execução por amostra, aumentando também os custos de mão de obra.

Nesse cenário, o custo total da titulação é cerca de três vezes maior do que o da fotometria de chama para o mesmo número de análises. Essa diferença é especialmente significativa em aplicações que demandam um grande volume de medições, nas quais a fotometria de chama é mais vantajosa devido à sua rapidez, simplicidade e menores custos operacionais. A titulação, embora mais cara e demorada, continua sendo preferida em situações que requerem maior precisão em medições específicas, como a determinação de íons cloreto.

2.6 Conclusão do Capítulo

Neste capítulo, foram abordados os conceitos fundamentais para o desenvolvimento do trabalho, incluindo técnicas de aprendizado de máquina, redes neurais e monitoramento da qualidade

do ar. Foram discutidos também os sistemas embarcados e a computação de borda, partes fundamentais para a execução eficiente de modelos em tempo real em dispositivos com recursos limitados. A aplicação de *TinyML* foi detalhada, com a implementação de modelos utilizando *TensorFlow* e *TensorFlow Lite*, destacando técnicas de quantização. No próximo capítulo, serão descritas as etapas metodológicas para o desenvolvimento e implantação do sistema de monitoramento da qualidade do ar.

3

Metodologia

Este capítulo foca nas etapas realizadas para a criação do sensor virtual, desde a preparação e integração dos dados de entrada até a implementação do modelo embarcado. Dada a necessidade de um sistema eficiente em termos de consumo de recursos, o ESP32-S3 foi selecionado por seu baixo custo, facilidade de uso e suporte extensivo da comunidade de desenvolvedores, além de oferecer o desempenho necessário para o processamento de inferências em tempo real.

3.1 Tecnologias Utilizadas

As tecnologias empregadas no desenvolvimento de cada etapa do sensor virtual foram organizadas em categorias, como mostrado na tabela 3.1, para facilitar a compreensão do leitor em relação às ferramentas e plataformas utilizadas em cada etapa deste trabalho.

Para o desenvolvimento do modelo na linguagem de programação *Python*, foi utilizada uma máquina com alto desempenho, garantindo a eficiência durante o treinamento do modelo. Na etapa de implementação em dispositivos embarcados, foram mantidas as versões de *Python* e *TensorFlow* já utilizadas, adicionando o *TensorFlow Lite Micro*, que permite a execução do modelo em dispositivos de baixa potência, como o ESP32-S3.

Tabela 3.1: Tecnologias utilizadas separadas por categoria.

Modelo em Python	**Modelo Micro**	**Execução**
Notebook Acer, 16,0GiB, 11th Gen Intel®Core™i5-1135G7 @ 2.40GHz × 8	Python v3.10.12	Base de dados proprietária
Python v3.10.12	TensorFlow v2.9.1	ESP32-S3, Xtensa®Dual-Core 32-bit LX7, Flash: 16 MB (Quad SPI), PSRAM: 8 MB (SPI Octal), SRAM: 512 Kbytes, Clock: 240MHz
TensorFlow v2.9.1	Visual Studio Code v1.88.1	Fonte de alimentação: 3,0 ∼ 3,6 V
scikit-learn v1.0.2	Espressif v5.3	
Visual Studio Code v1.88.1	GCC/G++ v11.4.0	
	TensorFlow Lite Micro	

Fonte: Autor.

3.2 Escolha e Análise dos Dados

Dada a natureza do problema que foi descrita no capítulo 1 a base de dados utilizada para especialização do modelo contém um total de 198 registros de dados numéricos, cada um com três características distintas. Suas variáveis quantitativas representam propriedades de amostras específicas, são elas: sódio, potássio e cloro. A matriz de correlação entre essas variáveis é mostrada na Tabela 3.2.

Tabela 3.2: Matriz de correlação entre as variáveis presentes na base.

	Sódio	**Potássio**	**Cloro**
Sódio	1	0,25	0,94
Potássio	0,25	1	0,48
Cloro	0,94	0,48	1

Fonte: Autor.

Os valores da matriz indicam uma forte correlação positiva entre sódio e cloro (0,94), sugerindo que aumentos na concentração de sódio estão fortemente associados a aumentos na concentração de cloro. A correlação entre potássio e cloro (0,48) é moderada, indicando uma relação significativa, mas menos intensa. Estes resultados sugerem que tanto o sódio quanto o potássio podem ser bons preditores para o cloro. As estatísticas descritivas das variáveis sódio, potássio e cloro na base de dados são apresentadas na Tabela 3.3.

Tabela 3.3: Estatísticas descritivas das variáveis presentes na base.

	Sódio	**Potássio**	**Cloro**
Contagem	198	198	198
Média	8,37	4,48	16,55
Desvio Padrão	5,75	2,71	9,40
Mínimo	0,00	0,00	0,00
1º Quartil	4,70	2,70	10,00
Mediana	7,90	4,40	15,30
3º Quartil	11,70	6,10	22,50
Máximo	37,20	10,70	65,50

Fonte: Autor.

Análise dos dados de entrada

- **Amplitude**: Os valores de sódio variam de 0,9 a 37,2 e os de potássio de 0,4 a 17,2, indicando uma amplitude significativa. Esta variação pode afetar o desempenho do modelo de aprendizado de máquina, tornando difícil para o algoritmo ajustar-se corretamente às diferenças de escala entre as variáveis.

- **Distribuição**: As médias e medianas de sódio e potássio estão relativamente próximas, sugerindo uma distribuição razoavelmente simétrica. No entanto, o desvio padrão é relativamente alto, especialmente em relação às médias, o que indica uma dispersão significativa nos dados.

- ***Outliers***: A diferença considerável entre o 75º percentil e os valores máximos sugere a presença de *outliers*, particularmente para o sódio.

Análise dos dados de saída

- **Amplitude**: Os valores de cloro variam de 2 a 65,5, representando uma faixa ampla. Assim como nos dados de entrada, esta amplitude pode interferir no desempenho do modelo, que pode dar mais importância aos valores maiores.

- **Distribuição**: A média dos valores de cloro é consideravelmente maior que a mediana, sugerindo uma distribuição assimétrica, com uma cauda longa à direita.

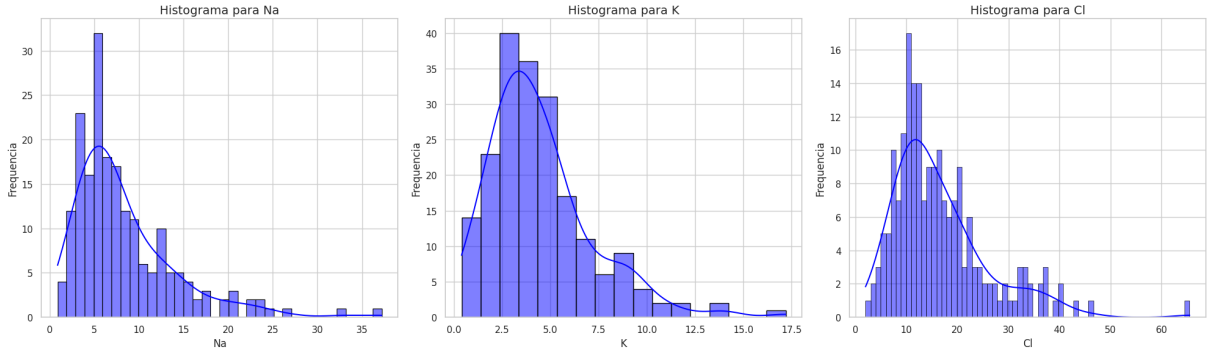
- ***Outliers***: Similar aos dados de entrada, há uma grande diferença entre o 75º percentil e o valor máximo, indicando a presença de *outliers*.

3.3 Pré-processamento dos dados

Dada as características, tanto nos dados de entrada quanto nos dados de saída, descritas se seção 3.2, optou-se por normalizar os dados. Sem esta etapa, o modelo poderia priorizar variáveis com valores maiores, desbalanceando o aprendizado e comprometendo a precisão das predições. A normalização assegura que todas as variáveis estejam na mesma escala, melhorando o desempenho e a capacidade de generalização do modelo.

A distribuição dessas variáveis na forma de histogramas é apresentada na Figura 3.1, permitindo uma análise visual das estatísticas descritivas apresentadas na Tabela 3.3.

Figura 3.1: Distribuição dos Dados Originais.



Fonte: Autor.

Inicialmente, foi aplicada uma transformação logarítmica tanto nos dados de entrada quanto nos de saída. A transformação logarítmica, conforme mostrada na Equação 3.1, suaviza os dados, aplicando o logaritmo natural a cada valor x_i de entrada.

$$y = \log(1+x) \tag{3.1}$$

Após, foi aplicada a padronização, com o uso da classe `StandardScaler`, da biblioteca `sklearn.preprocessing`. Isso garante que todas as variáveis contribuam igualmente para o modelo, melhorando o processo de otimização. Esta técnica reescala as variáveis de modo que tenham média zero e desvio padrão unitário, como mostrado na Equação 3.2.

$$z = \frac{x-\mu}{\sigma} \tag{3.2}$$

de forma que na equação: z representa o valor padronizado da variável; x o seu valor original; μ a sua média; e σ, seu desvio padrão.

A matriz de correlação após a normalização é apresentada na Tabela 3.4. As correlações entre as variáveis normalizadas são semelhantes às da matriz original, indicando que a normalização não alterou significativamente as relações entre as variáveis. Por exemplo, a correlação entre sódio e cloro (0,95) permanece alta. Da mesma forma, as correlações entre potássio e cloro (0,48) também mantêm valores próximos aos observados na Tabela 3.2.

Tabela 3.4: Matriz de Correlação entre Sódio, Potássio e Cloro Após Normalização.

	Sódio	**Potássio**	**Cloro**
Sódio	1	0,26	0,95
Potássio	0,26	1	0,48
Cloro	0,95	0,48	1

Fonte: Autor.

A distribuição das variáveis após o pré-processamento é mostra na Figura 3.2. Os histogramas indicam que as distribuições estão agora mais simétricas e centradas em torno de zero, devido à padronização. Esta transformação aproxima os dados de uma distribuição normal (gaussiana), com a maioria dos valores agrupados ao redor da média.

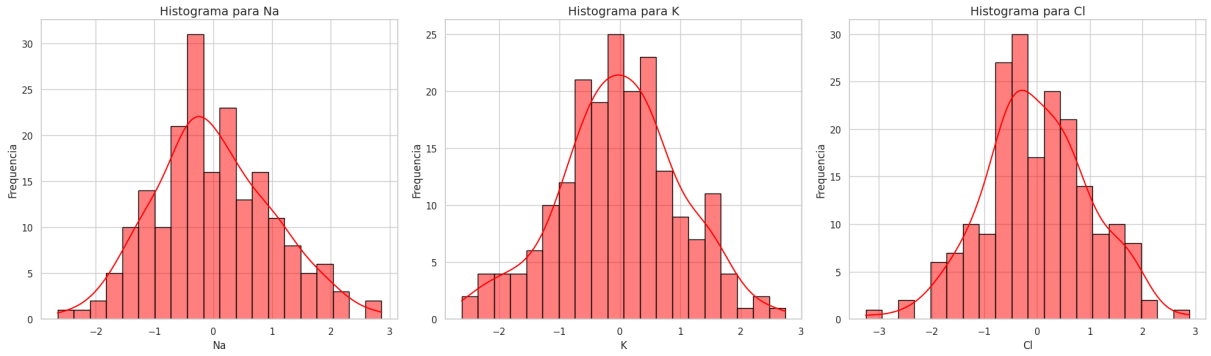


Figura 3.2: Distribuição dos Dados Após Normalização
Fonte: Autor.

Após o pré-processamento, os dados foram divididos em dois conjuntos: 70% para treino e 30% para teste. Os dados processados estão disponíveis no repositório que armazena os resultados deste trabalho [3]. Dentro do conjunto de treino, 20% foram separados para validação.

3.4 Desenvolvimento e especialização do Modelo

A escolha do modelo de aprendizado como sendo um *multilayer perceptron* se deu por conta das características que a base de dados descrita na seção 3.2 apresenta. Estas são conhecidas pela capacidade de aproximação universal em que uma rede com uma única camada intermediária é suficiente para aproximação uniforme, tendo um conjunto de treinamento significativo para que qualquer função contínua seja representada (ZHANG et al., 2021).

Para o desenvolvimento e especialização do modelo de aprendizado, foram adotadas as diretrizes presentes na documentação oficial do *TensorFlow*[4]. As principais características adotadas no modelo para solucionar o problema proposto foram as seguintes:

[3]Disponível em <https://github.com/Tayco110/tcc/tree/main/python_model/database>
[4]Disponível em <https://www.tensorflow.org/guide/core/mlp_core>

1. **Estrutura da Rede Neural**

   (a) **Camadas Ocultas**: O modelo foi projetado com uma camada oculta contendo sete neurônios.

   (b) **Funções de Ativação**: Foi utilizada a função ReLU (do termo em inglês *Rectified Linear Unit*)[5] para a camada oculta, por seu bom desempenho no treinamento (CHOLLET et al., 2015). Para a camada de saída, foi empregada uma função linear[6], ideal para problemas de regressão, como a previsão de valores contínuos da concentração de cloro.

2. **Treinamento do Modelo**

   (a) **Tamanho do Lote**: O tamanho do lote (*batch size*) foi definido como 10, equilibrando a precisão da atualização dos gradientes e a velocidade do treinamento.

   (b) **Número de Épocas**: O modelo foi treinado por 100 épocas, garantindo tempo suficiente para aprender os padrões dos dados sem aumentar excessivamente o risco de *overfitting*.

   (c) **Taxa de Aprendizagem**: A taxa de aprendizagem foi fixada em 0.001, valor padrão do otimizador *Adam* (ABADI et al., 2015), escolhido pela sua eficiência em lidar com grandes volumes de dados e otimizar o treinamento de redes neurais.

Os hiperparâmetros foram ajustados por meio de experimentação e otimização sistemática, garantindo que o modelo fosse eficiente em termos de previsão e execução na borda.

3.4.1 Construção do Modelo

Para construir o modelo, foi feito uso dos pacotes `Sequential` e `Dense`, presentes no `tensorflow.keras`. O método `Sequential` foi escolhido por permitir adicionar camadas de forma linear, ideal para o *multilayer perceptron* (MLP). O método `Dense` foi utilizado para criar camadas totalmente conectadas.

O otimizador `Adam`[7] foi selecionado por ajustar automaticamente a taxa de aprendizagem, garantindo um treinamento mais eficiente. O código completo para a construção do modelo está no Algoritmo 3.1.

[5]Disponível em <https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU>
[6]Disponível em <https://www.tensorflow.org/lattice/api_docs/python/tfl/layers/Linear>
[7]Disponível em <https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam>

Algoritmo 3.1: Construção do Modelo

```
def build_and_evaluate(hidden_neurons):
    inputs = tf.keras.layers.Input(shape=(X_train.shape[1],), name='Input_Layer')
    hidden = tf.keras.layers.Dense(hidden_neurons,
                                   activation='relu',
                                   name='Hidden_Layer')(inputs)
    outputs = tf.keras.layers.Dense(1, name='Output_Layer')(hidden)

    model = tf.keras.Model(inputs=inputs,
                           outputs=outputs,
                           name=f"MLP_Model_with_{hidden_neurons}_Neurons")
```

Após a definição da arquitetura do modelo e de sua criação é feito seu treinamento e compilação. O trecho de código presente em Algoritmo 3.2 é responsável por esta tarefa.

Algoritmo 3.2: Compilação e treinamento do modelo

```
model.compile(optimizer='adam', loss='mse')
history = model.fit(X_train, y_train, epochs=100, batch_size=10, validation_split=0.2)
```

A função de perda escolhida, `mean_squared_error`, é amplamente utilizada em problemas de regressão, pois mede a precisão das previsões do modelo ao calcular a média dos erros quadráticos entre os valores reais e previstos. Utilizando o método `summary()` do `Keras` no *TensorFlow*, é possível visualizar as informações detalhadas do modelo criado. A Tabela 3.5 resume essas informações, confirmando que o modelo atende às especificações definidas no início Seção 3.4.

Tabela 3.5: Resumo do modelo.

Layer	**Output Shape**	**Param #**
Input_Layer	(None, 2)	0
Hidden_Layer	(None, 7)	21
Output_Layer	(None, 1)	8
Total params:		29
Trainable params:		29
Non-trainable params:		0

Fonte: Autor.

A coluna *Layer* lista as camadas da rede neural e seus respectivos tipos. No modelo em questão, todas as camadas, com exceção da camada de entrada, são *Dense*, ou seja, totalmente conectadas; A coluna *Output Shape* exibe o formato da saída de cada camada. A forma é representada como uma tupla, onde `None` indica o tamanho flexível do lote (*batch size*), e os demais números representam a dimensão da saída da camada. Por exemplo, (None, 7) significa que a camada gera um vetor de sete dimensões para cada elemento do lote; A coluna *Param*

mostra o número de parâmetros (pesos e vieses) que a camada aprende durante o treinamento. Por exemplo, a camada `Hidden_Layer` tem 21 parâmetros.

Cada linha da tabela corresponde a uma camada, detalhando como os dados são transformados em cada etapa e quantos parâmetros são usados para essas transformações. Algumas informações foram suprimidas para facilitar a leitura.

3.4.2 Especialização do Modelo

Para realizar a especialização do modelo, foi feito uso do método `fit`, presente no objeto `model` criado conforme descrito na Seção 3.4.1. A base de dados utilizada para o treinamento é detalhada na Seção 3.2, e as transformações aplicadas estão descritas na Seção 3.3.

Após o término do treinamento, o modelo foi salvo no formato `.h5`[8], permitindo seu compartilhamento ou implantação no *TensorFlow Lite*[9].

O *pipeline* completo de criação e treinamento do modelo pode ser encontrado no arquivo `mpl_model.ipynb`[10] presente no repositório que agrega os resultados deste trabalho.

3.5 Exportação do modelo para TensorFlow Lite

Para o realizar o processo de exportação do modelo para o ESP32-S3, foi necessário convertê-lo para o formato *TFLite*. As técnicas de quantização utilizadas para otimizar o modelo foram do tipo pós-treinamento, conforme mostrado na Tabela 2.1, repetida aqui por conveniência de leitura.

Tabela 3.6: Técnicas de Quantização

Técnica	Requisitos de Dados	Redução de Tamanho	Precisão	Hardware Compatível
Quantização float16 pós-treinamento	Não há dados	Até 50%	Perda de acurácia insignificante	CPU, GPU
Quantização de intervalo dinâmico pós-treinamento	Não há dados	Até 75%	Menor perda de precisão	CPU, GPU (Android)
Quantização de números inteiros pós-treinamento	Amostra representativa sem rótulo	Até 75%	Pequena perda de precisão	CPU, GPU (Android), EdgeTPU
Treinamento com reconhecimento de quantização	Dados de treinamento rotulados	Até 75%	Menor perda de precisão	CPU, GPU (Android), EdgeTPU

Fonte: Adaptado(ABADI et al., 2015)

[8]Disponível em <https://www.tensorflow.org/js/guide/conversion?hl=pt-br>

[9]Disponível em <https://www.tensorflow.org/lite/guide?hl=pt-br>

[10]Disponível em <https://github.com/Tayco110/tcc/blob/main/python_model/src/mlp_model.ipynb>

A conversão do modelo para sistemas embarcados, foi realizada por meio da *API TensorFlow Lite Converter*. Como descrito na Seção 2.4.2, a quantização pós-treinamento ajusta os parâmetros, originalmente representados em 32 *bits*, para formatos mais eficientes, visando reduzir o tamanho e/ou a latência do modelo, ajustando pesos e funções de ativação.

As formas de quantização testadas incluem faixa dinâmica, *float16*, quantização inteira completa e somente inteira. Caso o modelo seja convertido sem quantização, ele manterá os dados em ponto flutuante de 32 *bits* (*float32*) por padrão. Essas técnicas permitem que o modelo seja implementado de forma eficiente em sistemas embarcados, garantindo um desempenho adequado em termos de consumo de recursos.

O *pipeline* completo para as quantizações do modelo pode ser encontrado no arquivo `micro_model.ipynb`[11] presente no repositório que reúne os resultados deste trabalho.

3.5.1 Quantização de faixa dinâmica

A quantização por faixa dinâmica é a técnica mais simples e rápida, pois converte apenas os pesos do modelo de ponto flutuante para inteiros de 8 *bits*, sem a necessidade de um conjunto de dados representativo para calibração (ABADI et al., 2015). O trecho de código abaixo mostra como esta quantização é realizada.

Algoritmo 3.3: Quantização de faixa dinâmica.

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quant_model = converter.convert()
```

Essa técnica é ideal como ponto de partida, já que reduz o uso de memória e acelera a computação sem exigir dados adicionais para calibração. É especialmente útil quando se busca uma implementação rápida e ganhos de desempenho com mínimas alterações no modelo.

3.5.2 Quantização *Float16*

Na quantização em *float16*, apenas os pesos dos neurônios são convertidos para números de ponto flutuante de 16 *bits*, enquanto as operações, como funções de ativação, continuam utilizando ponto flutuante de 32 *bits*. O *script* utilizado para realizar essa quantização é apresentado no Algoritmo 3.4. A principal diferença em relação à quantização por faixa dinâmica é a inclusão do método `supported_types`, que habilita a quantização em *float16*.

[11] Disponível em <https://github.com/Tayco110/tcc/blob/main/python_model/src/micro_model.ipynb>

Algoritmo 3.4: Quantização Float16.

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model = converter.convert()
```

Para esta técnica, é preciso garantir que o dispositivo de destino seja compatível com operações de ponto flutuante, pois a capacidade de operar com este formato é essencial para o bom desempenho do modelo.

3.5.3 Quantização inteira completa

A quantização inteira completa, quantiza tanto os pesos quanto as ativações para inteiros de 8 *bits*. Esse método exige um conjunto de dados representativo para calibração. Diferente das técnicas anteriores, é necessário calibrar os tensores, para este fim, se utiliza a base de dados original empregada no treinamento do modelo. O código responsável por essa quantização é apresentado abaixo.

Algoritmo 3.5: Quantização inteira completa.

```
def representative_dataset_generator():
    for value in x_train.values:
        yield [np.array(value, dtype=np.float32, ndmin=2)]

converter = tf.lite.TFLiteConverter.from_keras_model(model_py)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_generator
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_model = converter.convert()
```

No Algoritmo 3.5, é definido um gerador de conjunto de dados representativo para calibrar o modelo durante o processo de quantização, seguido pela conversão do mesmo. Esta técnica é ideal para maximizar a eficiência computacional e reduzir a latência, especialmente em dispositivos com suporte limitado para operações de ponto flutuante. O uso de um conjunto de dados representativo garante que a quantização seja ajustada para minimizar a perda de precisão.

3.5.4 Quantização somente inteiro

A quantização somente de inteiros (*weight-only quantization*) converte todas as operações do modelo, incluindo entradas e saídas, para a representação em 8 *bits*. Este formato é ideal para

microcontroladores que não suportam operações de ponto flutuante. O código responsável por essa quantização é apresentado abaixo.

Algoritmo 3.6: Quantização somente inteiro.

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_keras_model(model_py)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]
tflite_model_weights_only = converter.convert()
```

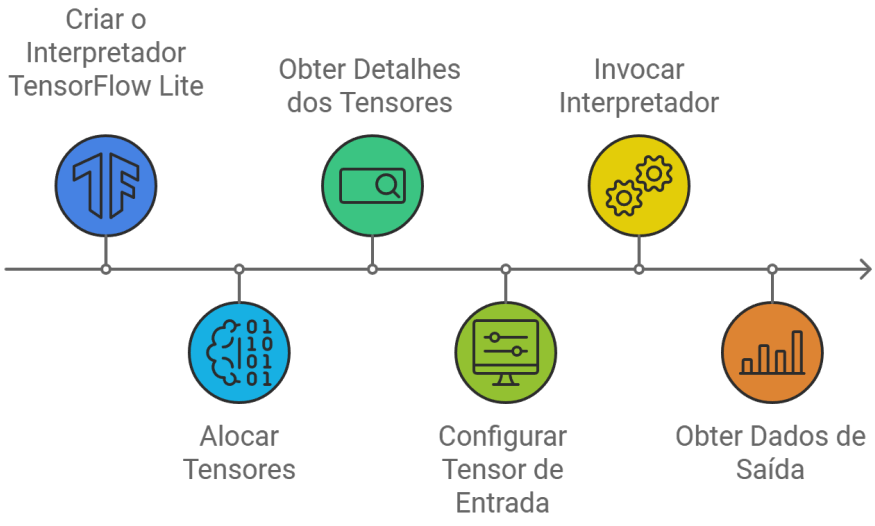
Esta abordagem é uma solução intermediária, que oferece uma redução significativa no tamanho do modelo e ganhos de desempenho, sem a necessidade de calibrar todas as operações. É especialmente útil em cenários onde a redução de tamanho é importante, mas manter a precisão das ativações também é.

3.6 Execução dos modelos *TFLite* no *Python*

Após a quantização do modelo nas diferentes formas descritas na seção 3.5, foi realizada a exploração do processo de inferência dos modelos quantizados. O formato *TFLite* exige um interpretador especializado, capaz de operar nesse formato. Para isso, foi empregado o interpretador *TensorFlow Lite*, o que permitiu acessar as características e os valores dos tensores nas camadas de entrada e saída dos modelos.

O processo de carregamento e execução dos modelos quantizados foi desenvolvido em *Python*. O diagrama presente na Figura 3.3 detalha as etapas dessa execução, desde a criação do interpretador até a obtenção dos dados de saída.

Figura 3.3: Pipeline de execução para o moelo TFLite no Python.



Fonte: Autor.

Para cada execução de cada um dos modelos otimizados, foram aplicados *benchmarks* para gerar métricas de desempenho que servem como linha base para comparação, estas métricas estão descritas na Tabela 3.7.

Tabela 3.7: Métricas de desempenho utilizadas para avaliar as características do modelo, como MSE, RMSE e MAE.

Nome	Descrição
Volumetria dos Arquivos	Volume em disco dos modelos.
Métricas de Desempenho	Inclui MSE, RMSE, MAE, R2, MEDAE, MAPE e tempo de inferência, medindo a precisão das previsões em relação aos valores reais.
Gráficos de Dispersão	Gráficos usados para ilustrar a correlação e a concordância entre valores reais e preditos.

Fonte: Autor.

As métricas em questão, serão discutidas com mais detalhes no Capítulo 4. No entanto, os resultados preliminares indicam que **o modelo convertido em *float 32*** foi o mais adequado para embarque no ESP32-S3, oferecendo um bom equilíbrio entre desempenho, consumo de recursos e facilidade de implementação.

3.7 Exportação do modelo para o microcontrolador

Após o treinamento do modelo para predizer o valor de cloro a partir dos valores de potássio e sódio, e a realização das quantizações, testes e avaliações, a etapa final consiste em embarcar o arquivo binário no ESP32-S3 para realizar inferências em borda.

O primeiro passo foi configurar o ambiente de desenvolvimento para dispositivos Espressif, seguindo o tutorial oficial[12]. Esse processo assegurou que o ESP32-S3 estivesse preparado para receber o modelo otimizado.

3.7.1 Incorporação de Modelo *TensorFlow Lite* em Código C/C++

Os arquivos `model.h`[13] e `model.cc`[14], que contêm os dados e metadados do modelo otimizado, são gerados a partir do arquivo binário do modelo seguindo os passos descritos abaixo. Esses arquivos garantem que o modelo seja incorporado diretamente no código-fonte a ser executado no dispositivo embarcado.

[12]Disponível em <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html>
[13]Disponível em <https://github.com/Tayco110/tcc/blob/main/micro_model/main/model.h>
[14]Disponível em <https://github.com/Tayco110/tcc/blob/main/micro_model/main/model.cc>

Arquivo model.h

A primeira parte do processo envolve a criação de um arquivo de cabeçalho em linguagem C/C++, denominado `model.h`, que declara duas variáveis para armazenar o modelo TFLite no código. A variável `extern const unsigned char model[];` declara um *array* de caracteres sem sinal, que será usado para armazenar os dados binários do modelo. A palavra-chave `extern` indica que esse *array* será definido em outro arquivo, permitindo que o cabeçalho seja incluído em múltiplos arquivos de origem sem conflitos.

Adicionalmente, a variável `extern const int model_len;` é usada para armazenar o tamanho do *array* `model`, ou seja, o tamanho do modelo em *bytes*. As diretivas como `#ifndef`, `#define` e `#endif` são utilizadas para evitar problemas de inclusão múltipla. O conteúdo gerado é então escrito no arquivo `model.h`.

Arquivo model.cc

O segundo passo envolve a leitura do arquivo de modelo *TensorFlow Lite* convertido sem quantização e a criação de um arquivo de implementação, denominado `model.cc`. Este arquivo converte os dados do modelo em um formato que pode ser utilizado como um *array* de caracteres em C/C++. Cada byte do modelo é formatado como um valor hexadecimal no formato "0xXX", que é a representação padrão de *bytes* em *arrays* no C/C++.

Para garantir a legibilidade do código, o *array* é organizado de forma que cada linha contenha até 12 *bytes*. Além disso, o comprimento total do modelo, é gravado no mesmo arquivo, para que o programa possa utilizar essa informação ao carregar o modelo no dispositivo embarcado.

3.7.2 Desenvolvimento do código-fonte em C/C++

Após a configuração do ambiente de desenvolvimento descrita na Seção 3.7.1, o próximo passo é instalar as dependências necessárias para a execução do *TFLite* Micro[15]. Caso ocorra algum problema com as dependências, estes serão identificados no momento da compilação do projeto, gerando uma mensagem de erro correspondente.

Com base nas instruções fornecidas pela documentação do *TensorFlow Lite*, as etapas necessárias para executar um modelo em C/C++ no dispositivo embarcado serão descritas a seguir. Essas etapas garantem que o modelo esteja corretamente integrado ao ambiente embarcado e funcione conforme esperado no hardware de destino. As etapas detalhadas do código podem ser encontradas no repositório do projeto[16]

[15]Disponível em <https://github.com/espressif/esp-tflite-micro>
[16]Disponível em <https://github.com/Tayco110/tcc/blob/main/micro_model/main/main_functions.cc>

Configuração das variáveis

A primeira parte do processo envolve a configuração das variáveis necessárias para inicializar o ambiente de execução do modelo em um dispositivo embarcado. O arquivo `main_functions.cc` define as variáveis que permitem o carregamento e a execução do modelo, em um *namespace* anônimo, são declaradas as variáveis que o referenciam; assim como o interpretador; E os tensores de entrada e saída, além de alocar um espaço de memória (40 kB) para a execução dos tensores. O uso de um *namespace* anônimo evita conflitos de escopo no projeto.

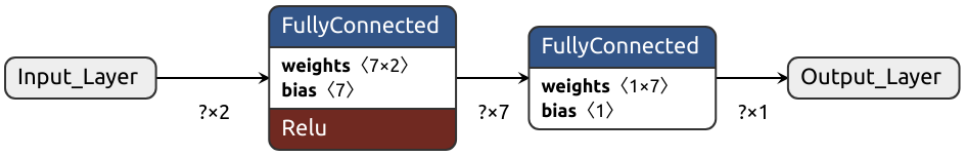
A implementação detalhada desta etapa pode ser encontrada no arquivo em questão presente no repositório do projeto no intervalo de linhas 11-19.

Construção do Interpretador

O interpretador é a peça chave para a execução do modelo. Ele é responsável por processar as entradas (valores de sódio e potássio) e gerar a predição do valor de saída (valor de cloro). Para isso, o código inicializa o interpretador e adiciona as operações necessárias ao *resolver*, como as funções *fully connected* e *ReLU*, que foram identificadas na análise do modelo otimizado com a ferramenta Netron[17]. A arquitetura do modelo pode ser observada na imagem 3.4.

O interpretador é configurado para utilizar a arena de tensores e associar o modelo às operações que ele irá executar. A implementação detalhada desta etapa pode ser encontrada no repositório do projeto no intervalo de linhas 35-41.

Figura 3.4: Arquitetura do modelo otimizado.



Fonte: Autor.

Alocação dos Tensores

Após a configuração do interpretador, o código tenta alocar os tensores na memória. Essa etapa se faz necessária, pois estes incluem os dados de entrada, pesos e a saída predita pelo modelo. Caso a alocação falhe, uma mensagem de erro é exibida. Em caso de sucesso, os ponteiros para os tensores de entrada e saída são recuperados, indicando que o modelo está pronto para executar as inferências.

A implementação detalhada desta etapa pode ser encontrada no repositório do projeto no intervalo de linhas 43-51.

[17] Disponível em <https://netron.app/>

Realização de inferência e coleta dos resultados

Nesta fase, os valores de entrada são normalizados com base nas transformações descritas na Seção 3.3, que inclui a aplicação da transformação logarítmica e a padronização via *Standard Scaler*. Os dados normalizados são então copiados para o tensor de entrada do modelo, que realiza a inferência através do método `interpreter->Invoke()`. O valor predito de cloro é recuperado do tensor de saída e convertido de volta para a escala original.

O código completo para a normalização dos dados de entrada e a inferência pode ser encontrada no repositório do projeto no intervalo de linhas 54-79.

Compilação e Exportação para o ESP32-S3

Para compilar e exportar o modelo para o ESP32-S3, foi utilizado o compilador G++ no ambiente de desenvolvimento configurado previamente, conforme descrito na Seção 3.7. Os comandos de compilação e exportação do modelo podem ser encontrados em Algoritmo 3.7.

Algoritmo 3.7: Compilação e exportação do modelo para a ESP32.

```
cd "workspace"
. $HOME/esp/esp-idf/export.sh
idf.py build flash
```

Inferência realizada no dispositivo embarcado

Por fim, todas as etapas descritas foram integradas para permitir a execução contínua do modelo no dispositivo embarcado, visando a detecção em tempo real do valor predito. O processo começa com a coleta dos valores de entrada, que são então submetidos a uma normalização baseada na transformação logarítmica e na padronização, garantindo compatibilidade com o modelo treinado. Estes dados normalizados são transferidos para o tensor de entrada do modelo. A inferência é realizada pelo interpretador que processa os dados de entrada através do modelo. O valor de saída predito é gerado no tensor de saída e posteriormente convertido de volta à escala original. Finalmente, o resultado predito é exibido ou armazenado. O pseudocódigo a seguir resume o fluxo de execução contínua, destacando as principais etapas envolvidas no processo:

Algoritmo 3.8: Pseudocódigo para o fluxo principal.

```
interpreter = modelo.tflite
enquanto True faz:
    Na, K = Valores presentes no ambiente
    Na_norm, K_norm = normalize([Na, K], [Na_mean, K_mean], [Na_scale, K_scale])
    input_buffer[0] = Na_norm
    input_buffer[1] = k_norm
    inference(interpreter, input_buffer*)
    Cl_norm = output->data.f[0]
    resultado = reverse(Cl_norm, Cl_mean, Cl_scale);
    exibir(resultado)
fim enquanto
```

Os *scripts* completos para execução em tempo real podem ser acessados no repositório do projeto[18].

3.8 Conclusão do Capítulo

Neste capítulo, foram apresentadas todas as etapas para a implementação do modelo de inferência embarcado, desde a configuração do ambiente de desenvolvimento até a execução do modelo otimizado no ESP32-S3. Procedimentos importantes, como a normalização dos dados de entrada e o uso do interpretador *TensorFlow Lite*, garantiram a correta operação do modelo, permitindo uma predição eficiente do valor de saída. O ciclo de inferência foi implementado de forma contínua, possibilitando o monitoramento autônomo das variáveis. No próximo capítulo, serão discutidos os resultados dos diferentes métodos de quantização, com análise do impacto na precisão, além de outras métricas supracitadas.

[18]Disponível em <https://github.com/Tayco110/tcc/tree/main/micro_model>

4

Resultados

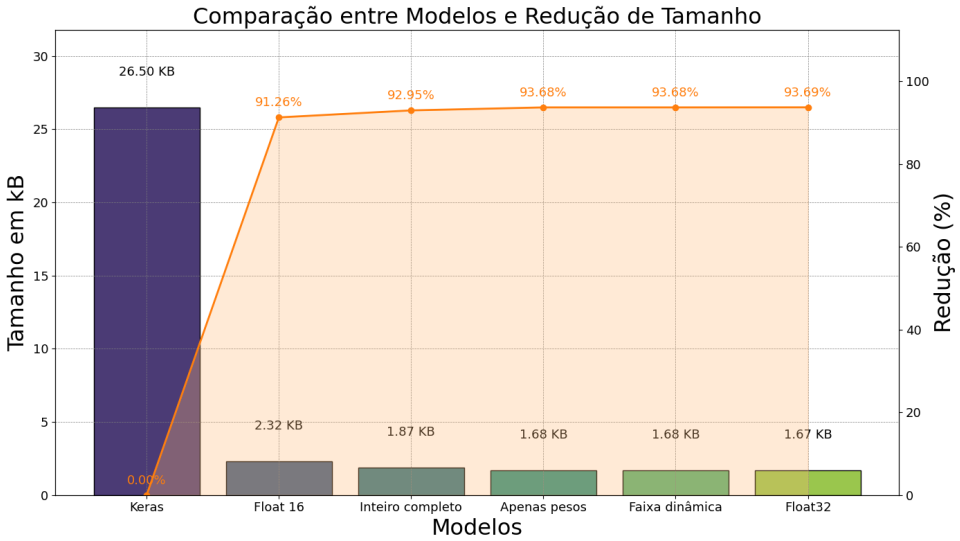
Neste capítulo, serão apresentados e discutidos os resultados experimentais obtidos com a quantização dos modelos descritos na Seção 3.5, além das diferentes comparações entre os modelos, destacando suas volumetrias, o impacto da quantização no desempenho e sua viabilidade para execução em dispositivos embarcados. Além disso, as análises incluem métricas de erro que avaliam a precisão preditiva dos modelos após a quantização, visando verificar a manutenção de seu desempenho em relação ao modelo original. Por fim, um comparativo entre as abordagens tradicionais de monitoramento da qualidade do ar e a solução proposta utilizando *TinyML* é apresentado, discutindo as vantagens e limitações de cada metodologia.

4.1 Volume dos modelos quantizados

Após a conversão e otimização dos modelos para o formato *TensorFlow Lite*, a volumetria final de cada um deles é ilustrado na Figura 4.1 no formato de gráfico de barras. Os resultados apresentados indicam uma redução significativa no tamanho dos modelos otimizados em comparação ao modelo original. As barras verticais representam o tamanho de cada modelo em kilobytes (kB), enquanto a curva laranja indica a porcentagem de redução em relação ao modelo original.

Observa-se que o modelo original possui aproximadamente 26,50kB, enquanto o modelo otimizado em *Float* 32 foi reduzido para 1,67kB, resultando em uma redução de 93,69%. Entre os modelos de menor volumetria, o "Apenas pesos" e o "Faixa dinâmica" alcançaram tamanhos similares, com aproximadamente 1,68kB, respectivamente, atingindo uma redução superior a 93%. O modelo quantizado em *Float* 16 reduziu seu tamanho para cerca de 2,32kB, enquanto o modelo "Inteiro completo" apresentou um tamanho de 1,87kB.

Figura 4.1: Tamanho dos modelos quantizados e sua porcentagem de redução.



Fonte: Autor.

A análise dos tamanhos dos modelos otimizados confirma que a quantização proporcionou uma diminuição eficiente no volume dos modelos, facilitando sua execução em dispositivos com recursos limitados, como microcontroladores. Esses resultados estão conforme as expectativas descritas na documentação do *TensorFlow Lite* para modelos de borda.

Contudo, é importante destacar que, embora a quantização tenha sido eficaz na redução do tamanho dos modelos, pode haver um possível *trade-off* em termos de precisão preditiva. Essa relação entre a volumetria dos modelos e sua precisão será explorada no próximo tópico.

4.2 Precisão dos Modelos em *Python*

Esta seção apresenta as métricas de precisão dos vários modelos quantizados, descritos anteriormente, em comparação com o modelo original. As métricas de erro utilizadas para avaliar a precisão dos modelos são sumarizadas na Tabela 4.1, repetida aqui por conveniência de leitura.

Tabela 4.1: Métricas de desempenho utilizadas para avaliar as características do modelo, como MSE, RMSE e MAE.

Nome	**Descrição**
Volumetria dos Arquivos	Volume em disco dos modelos.
Métricas de Desempenho	Inclui MSE, RMSE, MAE, R2, MEDAE , MAPE e tempo de inferência, medindo a precisão das previsões em relação aos valores reais.
Gráficos de Dispersão	Gráficos usados para ilustrar a correlação e a concordância entre valores reais e preditos.

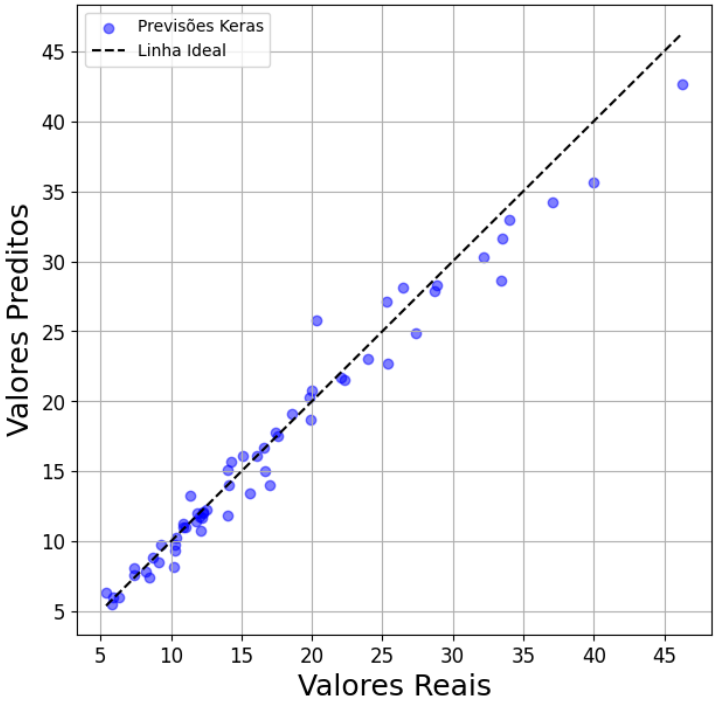
Fonte: Autor.

Além disso, a análise visual dos dados de predição é apresentada por meio de gráficos de dispersão, que comparam as predições dos modelos com os valores reais.

Os gráficos apresentados nas figuras a seguir, mostram a relação entre os valores preditos pelos modelos e os valores reais. A linha tracejada representa a correlação ideal entre as previsões e os dados reais, onde uma melhor correspondência entre os pontos e a linha ideal indica uma maior precisão.

4.2.1 Modelo Original

Figura 4.2: Gráfico de dispersão - Modelo Keras.



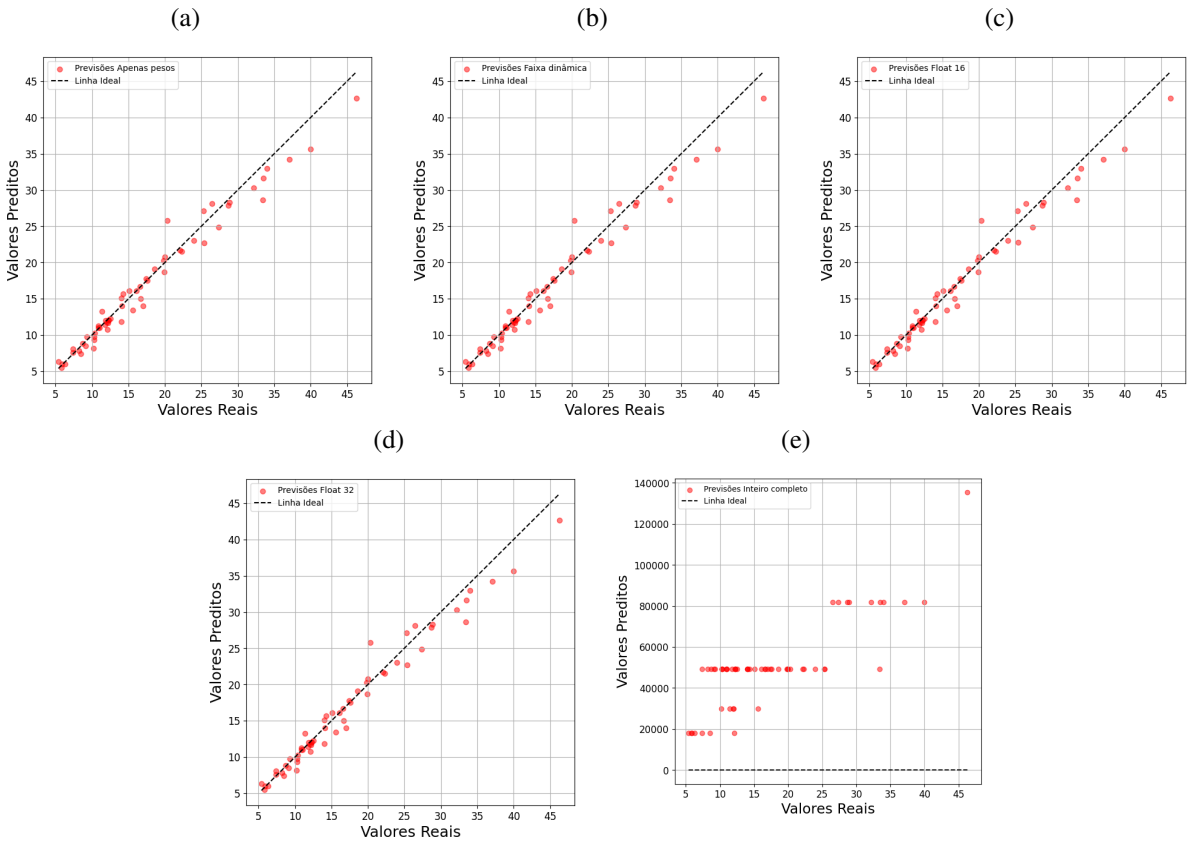
Fonte: Autor.

O gráfico de dispersão para o modelo Keras, apresentado na Figura 4.2, mostra uma boa correspondência entre os valores previstos e os valores reais. A maioria dos pontos está concentrada ao longo da linha ideal, refletindo uma alta precisão nas predições. Isso indica que o modelo original oferece uma previsão confiável, sendo um bom ponto de referência para comparações com os modelos otimizados.

4.2.2 Modelos Quantizados

Os gráficos de dispersão dos modelos quantizados estão apresentados na Figura 4.3, incluindo as versões em *Float* 32, Inteiro Completo, *Float* 16, Faixa Dinâmica, e Apenas Pesos. Cada gráfico ilustra a capacidade preditiva de cada modelo em comparação com o modelo Keras.

Figura 4.3: Gráfico de dispersão - Modelos Quantizados: (a) Apenas pesos, (b) Faixa dinâmica, (c) *Float* 16, (d) *Float* 32, (e) Inteiro completo.



Fonte: Autor.

Modelo Float32

O gráfico do modelo "*Float* 32" mostra uma boa correspondência com a linha ideal, indicando que as previsões estão bastante próximas dos valores reais. A distribuição uniforme dos pontos reflete uma precisão semelhante à do modelo Keras, sugerindo que a quantização em *Float* 32 mantém a relação entre os dados de maneira eficaz, sem comprometer significativamente a precisão.

Modelo Inteiro Completo

O gráfico para o modelo "Inteiro Completo" apresenta um comportamento completamente diferente dos demais modelos. Os pontos estão agrupados e distantes da linha ideal, indicando uma grande perda de precisão. Isso sugere que a quantização completa para inteiros comprometeu severamente a capacidade preditiva do modelo, possivelmente devido à perda excessiva de informação durante a quantização.

Modelo Float 16

O gráfico de dispersão do modelo "*Float* 16" exibe uma boa correspondência com a linha ideal, com apenas uma leve dispersão dos pontos. Isso indica que a quantização em *Float* 16 oferece uma boa precisão, mantendo um equilíbrio entre a redução de tamanho e a manutenção da capacidade preditiva.

Modelo Faixa Dinâmica

O modelo "Faixa Dinâmica" também apresenta um forte alinhamento com a linha ideal. A distribuição dos pontos reflete uma precisão satisfatória, sugerindo que a quantização de faixa dinâmica foi bem-sucedida em manter a precisão das previsões.

Modelo Apenas Pesos

Por fim, o gráfico do modelo "Apenas Pesos" mostra um excelente ajuste à linha ideal, indicando que a quantização exclusiva dos pesos do modelo não comprometeu sua capacidade de previsão. Este resultado demonstra que é possível reduzir o tamanho do modelo de maneira eficiente sem sacrificar a precisão.

4.2.3 Métricas de Desempenho

A Tabela 4.2 consolida as métricas de desempenho dos diferentes modelos quantizados em comparação com o modelo original Keras, utilizando indicadores como *MSE*, *RMSE*, *MAE*, R^2, *MEDAE*, e *MAPE*.

Tabela 4.2: Comparativo de Métricas de Desempenho para Modelos Quantizados e Keras

Modelo	**MSE**	**RMSE**	**MAE**	**R^2**	**MEDAE**	**MAPE**
Keras	2,78087	1,66759	1,14554	0,96778	0,70469	0,06412
Float32	2,78087	1,66759	1,14554	0,96778	0,70469	0,06412
Float16	2,77471	1,66575	1,14435	0,96786	0,70452	0,06409
Apenas Pesos	2,78087	1,66759	1,14554	0,96778	0,70469	0,06412
Faixa Dinâmica	2,78087	1,66759	1,14554	0,96778	0,70469	0,06412
Inteiro Completo	$2,96 \times 10^9$	$5,44 \times 10^4$	$5,04 \times 10^4$	$-3,43 \times 10^7$	$4,94 \times 10^4$	$3,22 \times 10^3$
Microcontrolador	3,96343	1,99084	1,14136	0,95493	0,68500	0,06978

Fonte: Autor.

As métricas indicam que os modelos quantizados, com exceção do modelo "Inteiro Completo", mantêm um desempenho praticamente equivalente ao modelo original. Modelos como *Float* 32, *Float* 16, "Apenas Pesos" e "Faixa Dinâmica" apresentam valores de *MSE*, *RMSE*, *MAE* e R^2 muito próximos aos do modelo Keras, sugerindo uma perda de precisão mínima após

a otimização. Essa consistência nos resultados demonstra que a quantização pós-treinamento foi eficaz em reduzir o tamanho dos modelos sem comprometer significativamente a precisão preditiva.

Mais especificamente, o coeficiente de determinação (R^2) para esses modelos permanece em torno de 0.968, indicando uma alta correlação entre as predições e os valores reais. Os valores de *MSE* e *RMSE* também são baixos, evidenciando que o desvio entre os valores previstos e os reais é pequeno. O *MAE* e o *MEDAE* corroboram essa avaliação, mostrando que a quantização desses modelos não introduziu grandes distorções nas predições.

Por outro lado, o modelo "Inteiro Completo" apresentou métricas de desempenho extremamente inferiores, com um *MSE* e *RMSE* elevados, indicando um erro médio substancialmente maior. O valor de R^2 negativo ($-3,43 \times 10^7$) sugere que as predições desse modelo são menos informativas do que simplesmente usar a média dos valores reais, evidenciando uma significativa perda de precisão. Além disso, os altos valores de *MAPE* e *MEDAE* reforçam que o modelo em questão não é adequado para previsões precisas, mostrando-se inadequado para aplicações que exigem precisão.

4.3 Resultados Obtidos do Microcontrolador

Nesta seção, serão apresentados os resultados coletados diretamente do microcontrolador durante os testes de inferência do modelo, um exemplo da saída do código pode ser observada em Algoritmo 4.1. Os testes foram realizados utilizando uma base de dados gravada na memória do microcontrolador, desta forma os resultados obtidos foram experimentais.

Algoritmo 4.1: Exemplo de Saída do sensor virtual.

```
Iniciando casos de testes!

Na:6.20 K:4.80 Cl:13.22 Inf.Time:197µs

Na:14.20 K:17.20 Cl:35.19 Inf.Time:56µs

...

Na:13.20 K:3.20 Cl:23.28 Inf.Time:54µs

Na:14.50 K:3.30 Cl:25.18 Inf.Time:43µs

Encerrando casos de teste!
```

Os resultados obtidos[19] no terminal seguem o seguinte padrão: **Na (Sódio):** Representa o valor medido de entrada para o parâmetro sódio, **K (Potássio):** Representa o valor medido de entrada

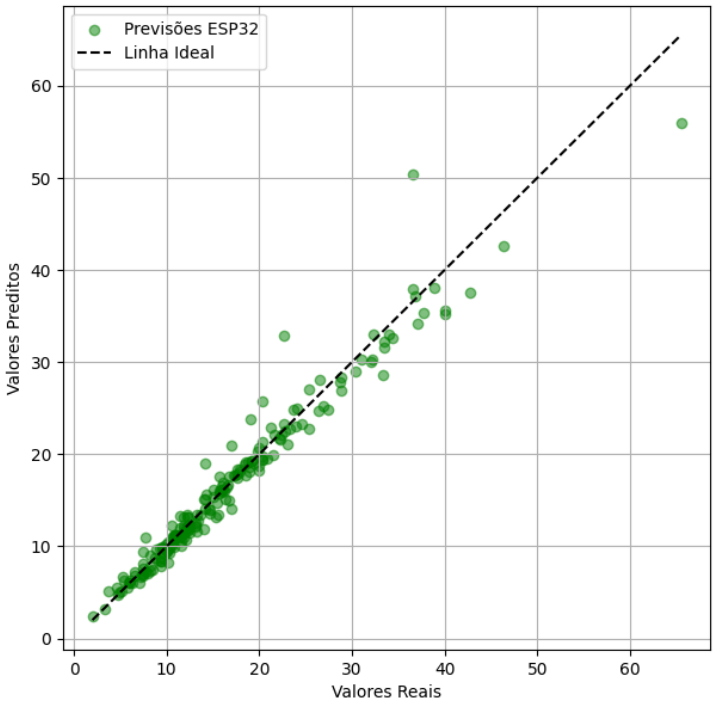
[19] Disponível em <https://github.com/Tayco110/tcc/blob/main/utils/valores_saida.csv>

para o parâmetro potássio, **Cl (Cloro):** Representa o valor predito pelo modelo baseado nas entradas anteriores **Inf.Time:** Representa o tempo de inferência em μs.

4.3.1 Análise dos valores obtidos

Os resultados obtidos demonstram a capacidade do modelo embarcado de prever os valores de cloro (Cl) com base nas entradas de sódio (Na) e potássio (K). A precisão do modelo foi avaliada por meio de uma comparação direta entre os valores preditos pelo sistema e os valores reais, conforme ilustrado na Figura 4.4.

Figura 4.4: Gráfico de dispersão - Sensor virtual.



Fonte: Autor.

A dispersão dos valores preditos mostrados na figura em relação ao os valores reais, evidenciando uma forte correlação entre as previsões do modelo e os dados reais. A linha pontilhada indica a linha ideal de correlação perfeita, onde os valores preditos seriam idênticos aos valores reais. A proximidade dos pontos ao longo dessa linha sugere que o modelo embarcado está gerando boas previsões, com desvios mínimos em algumas instâncias.

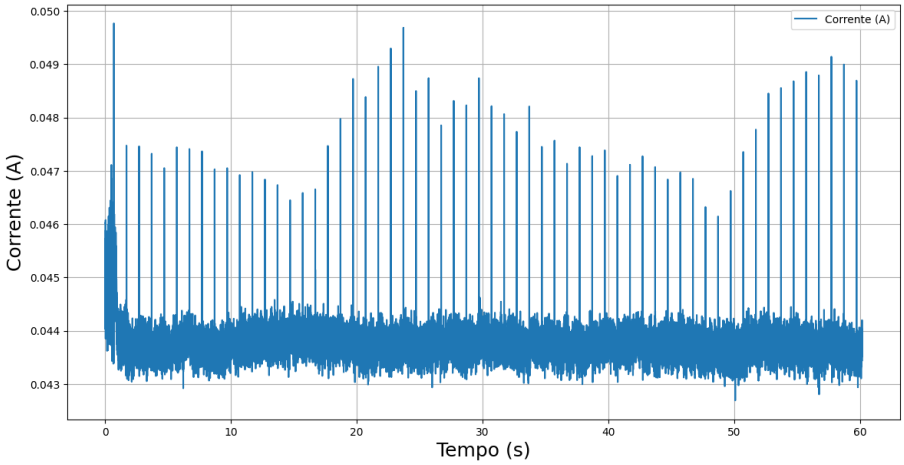
4.3.2 Desempenho do Modelo no Microcontrolador

O desempenho do modelo embarcado foi analisado considerando o consumo de energia, as métricas de precisão do modelo e o tempo de inferência, aspectos importantes para a avaliação de sua viabilidade em ambientes industriais de tempo real.

Consumo de Energia

A análise do consumo de corrente foi realizada durante a execução do modelo no microcontrolador, utilizando o software JLink[20]. Conforme ilustrado na Figura 4.5, a variação da corrente (em amperes) ao longo do tempo (em segundos) apresenta picos em intervalos uniformes, possivelmente associados a operação de inferência do modelo.

Figura 4.5: Consumo de Corrente ao Longo do Tempo.



Fonte: Autor.

Os valores de consumo de corrente variam entre um mínimo de 0,04269A e um máximo de 0,04976A, com uma média de 0,04377A. Esses resultados indicam que, embora haja picos de consumo, o microcontrolador opera majoritariamente em uma faixa de consumo energeticamente eficiente durante o tempo de execução contínua.

Para uma análise mais detalhada, os valores de corrente (A) foram convertido em energia (Joules), para este fim é necessário levar em conta a tensão de operação (V) do microcontrolador e o tempo de operação (t). A energia consumida (E) é calculada pela fórmula:

$$E = I \cdot V \cdot t \tag{4.1}$$

Onde:

- I é a corrente média em amperes (A),
- V é a tensão de operação do dispositivo em volts (V),
- t é o tempo de operação em segundos (s).

Assumindo que o microcontrolador opera a uma tensão de 3,3V (valor típico para dispositivos embarcados), podemos calcular o consumo de energia para diferentes durações de operação.

[20]Disponível em <https://www.segger.com/downloads/jlink/>

1. Corrente Média (I): $0,04377$
2. Tensão (V): $3,3$ V
3. Tempo (t): Para um exemplo de operação contínua por 1 hora ($t = 3600\,\text{s}$).

Logo:

$$E = 0,04377 \cdot 3,3 \cdot 3600 \tag{4.2}$$

$$E \approx 518,3\,\text{J} \tag{4.3}$$

Portanto, em 1 hora de operação contínua, o microcontrolador consome aproximadamente 518,3 Joules, se o dispositivo for alimentado por uma bateria de 1000 mAh (capacidade típica), operando a $3,3\,\text{V}$, a energia disponível seria:

$$E_{\text{bateria}} = 1\,\text{Ah} \cdot 3,3\,\text{V} \cdot 3600\,\text{s} = 11.880\,\text{J}$$

Assim, o microcontrolador poderia operar por cerca de:

$$\frac{11.880}{518,3} \approx 22,9\,\text{horas}$$

Este tipo de cálculo ajuda a compreender o impacto do consumo de corrente em termos de energia e a estimar a autonomia de dispositivos alimentados por bateria.

Precisão do Modelo

As métricas de erro fornecem uma visão da precisão das predições realizadas pelo modelo embarcado. O desempenho foi avaliado com as seguintes métricas: *MSE* = 3.96343, *RMSE* = 1.99084, *MAE* = 1.14136, *MedAE* = 0.68500, *MAPE* = 6.978% e R^2 = 0.95493.

Estes resultados indicam que o modelo apresentou alta precisão na predição de valores de Cl a partir dos dados de Na e K. O coeficiente de determinação (R^2) revela que aproximadamente 95,5% da variância dos dados foi explicada pelo modelo, o que confirma sua confiabilidade para a aplicação pretendida. A baixa taxa de erro, conforme indicado pelas métricas *MSE*, *RMSE*, e *MAE*, reforça a capacidade do modelo de realizar previsões consistentes, tornando-o adequado para o monitoramento preciso em dispositivos embarcados.

Tempo de Inferência

Outro ponto que deve ser destacado é o tempo de inferência do modelo, que foi medido em microssegundos (μs). Os valores demonstram que o modelo consegue realizar predições com uma latência extremamente baixa, com o tempo médio de inferência sendo de 49,01μs. Isso permite a operação do sistema com respostas rápidas a partir dos dados de entrada.

4.4 Comparação Entre Metodologia usual e proposta

A comparação entre a metodologia usual de medição de cloro e a proposta utilizando *TinyML* e sensores virtuais é importante para demonstrar as melhorias em termos de eficiência, custo e escalabilidade.

Conforme discutido nos capítulos 1 e 2, a abordagem tradicional, baseada em técnicas laboratoriais, apresenta limitações significativas, como alto custo operacional e demora na obtenção de resultados. Em contraste, a metodologia proposta, que utiliza modelos de aprendizado de máquina embarcados e sensores, oferece monitoramento em tempo real, maior portabilidade e menores custos a longo prazo.

Tabela 4.3: Equipamentos Utilizados

Aspecto	**Metodologia Usual**	**Metodologia Proposta**
Coleta de Amostras	Amostrador de Volume Grande (AVG) para coleta de massa de ar	Sensores em tempo real sem necessidade de amostrador de volume grande
Medição de Potássio e Sódio	Fotômetro de chama – equipamento de laboratório que excita átomos por chama	Sensores eletroquímicos ou óticos em tempo real
Medição de Cloro	Titulação argentimétrica, realizada em laboratório	Modelo preditivo TinyML que estima a concentração de cloro a partir dos sensores de Na e K
Portabilidade	Equipamentos de laboratório, não portáteis	Dispositivos portáteis ou fixos, com possibilidade de instalação em locais críticos

Fonte: Fonte: (KOTZ et al., 2018) e (BENVENUTO, 2022).

Conforme a Tabela 4.3, os métodos tradicionais dependem de equipamentos laboratoriais sofisticados, como o fotômetro de chama para medir potássio e sódio e a titulação argentimétrica para medir cloro, o que limita sua portabilidade e praticidade para uso em campo. Já a proposta com *TinyML* utiliza sensores portáteis, o que permite monitoramento contínuo e em tempo real.

Os sensores eletroquímicos e ópticos oferecem soluções modernas e precisas para a medição de íons como sódio (Na^+), potássio (K^+) e cloro (Cl^-) em diferentes ambientes. Para sódio e potássio, sensores eletroquímicos, como eletrodos iônicos seletivos, permitem detectar esses íons com alta sensibilidade, enquanto sensores ópticos baseados em fluorescência ou absorção óptica fornecem medições rápidas e diretas, mesmo em condições ambientais adversas (KOTZ et al., 2018) e (BENVENUTO, 2022).

Tabela 4.4: Custo

Aspecto	**Metodologia Usual**	**Metodologia Proposta**
Investimento Inicial	Alto (fotômetro de chama, reagentes para titulação)	Médio a alto (sensores de Na/K, desenvolvimento do modelo preditivo)
Custo por Análise	Alto – envolve preparo de amostras e reagentes	Baixo – após implementação, o custo por análise é praticamente zero
Manutenção	Reposição de reagentes, calibração regular	Manutenção dos sensores e calibração periódica

Fonte: (KOTZ et al., 2018) e (BENVENUTO, 2022).

Em termos de custo, como descrito na Tabela 4.4, os métodos tradicionais exigem um investimento inicial elevado em equipamentos laboratoriais e reagentes, com altos custos operacionais por análise. Por outro lado, a proposta com *TinyML* apresenta custos operacionais muito menores a longo prazo, já que o custo por análise praticamente desaparece após a implementação.

Tabela 4.5: Tempo de Análise

Aspecto	**Metodologia Usual**	**Metodologia Proposta**
Tempo de Coleta	Demorado – envolve a coleta de grandes volumes de ar	Monitoramento em tempo real
Tempo de Análise	Demorado – preparo e execução da titulação podem levar horas	Instantâneo – os sensores medem e o modelo preditivo estima os níveis de cloro em segundos
Tempo de Resposta	Lento – os dados só estão disponíveis após as análises laboratoriais	Rápido – dados em tempo real, permitindo decisões imediatas

Fonte: (KOTZ et al., 2018) e (BENVENUTO, 2022).

A Tabela 4.5 destaca que o tempo de análise e resposta na abordagem tradicional pode ser bastante prolongado, enquanto a solução *TinyML* oferece resultados em tempo real, possibilitando tomadas de decisão imediatas.

Tabela 4.6: Complexidade Operacional

Aspecto	**Metodologia Usual**	**Metodologia Proposta**
Treinamento Necessário	Técnicos especializados para operar fotômetro e realizar titulações	Operadores de campo com treinamento básico para usar sensores
Número de Passos	Múltiplas etapas: coleta, transporte, titulação, análise de dados	Simplificado – sensores capturam e o modelo preditivo processa automaticamente
Erros Humanos	Potencial para erros em cada etapa	Menor – automação reduz a intervenção humana e minimiza erros

Fonte: (KOTZ et al., 2018) e (BENVENUTO, 2022).

Conforme a Tabela 4.6 destaca, a complexidade operacional é significativamente reduzida com a proposta baseada em *TinyML*. Enquanto o método tradicional envolve várias etapas e exige operadores treinados, a automação com *TinyML* simplifica o processo e minimiza os erros humanos.

Tabela 4.7: Precisão e Sensibilidade

Aspecto	**Metodologia Usual**	**Metodologia Proposta**
Precisão	Alta – Titulação e fotometria são métodos precisos	Alta – desde que o modelo TinyML seja bem treinado com dados de qualidade
Interferências	Possíveis interferências de outros íons durante titulação e fotometria	Sensores podem ser calibrados para minimizar interferências

Fonte: (KOTZ et al., 2018) e (BENVENUTO, 2022).

Por fim, conforme a Tabela 4.7, a precisão dos métodos tradicionais é alta, embora possa ser afetada por interferências. A abordagem *TinyML* também oferece alta precisão, desde que os sensores sejam devidamente calibrados e o modelo seja bem treinado.

A comparação entre a metodologia tradicional e a proposta com *TinyML* e sensores virtuais destaca a superioridade da abordagem moderna. Equipamentos volumosos da metodologia tradicional limitam sua flexibilidade, enquanto sensores eletroquímicos permitem monitoramento contínuo em campo. Embora o investimento inicial em *TinyML* seja alto, os custos operacionais caem significativamente a longo prazo. Além disso, a solução com sensores oferece dados em tempo real e reduz a complexidade operacional, mantendo a precisão comparável à dos métodos tradicionais (KOTZ et al., 2018; BENVENUTO, 2022).

4.5 Conclusão do Capítulo

Neste capítulo, foram discutidos os resultados experimentais da implementação do modelo *TinyML* no microcontrolador, abrangendo desde a volumetria dos modelos quantizados até a análise de desempenho. Foram avaliadas as métricas de precisão, o consumo de energia e o tempo de inferência, demonstrando que as otimizações aplicadas utilizando o *TensorFlow Lite* tornaram o modelo eficiente para execução em tempo real. Além disso, a comparação entre a metodologia usual e a proposta reforçou as vantagens em termos de custo, automação e escalabilidade.

No próximo capítulo, será feita uma síntese dos principais achados do trabalho, destacando as contribuições e sugerindo possíveis caminhos para pesquisas futuras e aprimoramentos do sistema.

5

Conclusão

Neste trabalho, foi implementado e testado um modelo de *TinyML* embarcado no microcontrolador ESP32-S3 para predição da concentração de cloro (Cl) a partir de dados de sódio (Na) e potássio (K). Todos os resultados foram obtidos experimentalmente utilizando uma base de dados de teste que simulou os valores esperados no uso real. Os resultados demonstraram que o modelo foi capaz de atingir os objetivos estabelecidos, apresentando boas previsões e com baixo tempo de inferência, o que é essencial para aplicações de monitoramento em tempo real.

Embora os sensores físicos para monitoramento ainda não tenham sido implementados, os experimentos realizados forneceram uma base sólida para futuros desenvolvimentos. Esse desempenho eficiente, tanto em termos de consumo de energia quanto de processamento rápido, sugere que a solução será viável quando aplicada em dispositivos embarcados no campo.

O desenvolvimento do projeto enfrentou desafios técnicos relacionados à otimização do modelo, pois, foi necessário reduzir o consumo de recursos, enquanto se mantinham níveis satisfatórios de precisão nas predições. Adicionalmente, para lidar com possíveis variações nos dados da base de teste e melhorar a precisão das predições, foram utilizadas técnicas de pré-processamento, como normalização.

Com base nos resultados e discussões anteriores, há várias áreas de desenvolvimento que podem ser exploradas. A principal evolução proposta é a integração de sensores físicos para coleta de dados em tempo real, o que permitirá a validação completa do modelo em ambientes reais. Além disso, o desenvolvimento de um *dashboard* de monitoramento será uma extensão importante, possibilitando a visualização eficaz dos dados coletados e das predições realizadas pelo modelo.

Finalmente, futuros trabalhos podem focar na integração de novos tipos de sensores, ampliando as variáveis monitoradas, bem como em otimizações adicionais para lidar com maiores volumes de dados ou condições ambientais adversas. Outra direção promissora é a conexão do sistema com redes IoT, ampliando as possibilidades de monitoramento distribuído e remoto em larga escala.

Referências Bibliográficas

ABADI, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>.

BANBURY, C. R. et al. Mlperf tiny benchmark. *ArXiv*, abs/2106.07597, 2021. Disponível em: <https://api.semanticscholar.org/CorpusID:235421921>.

BENVENUTO, M. A. *Materials Chemistry*. Berlin, Boston: De Gruyter, 2022. ISBN 9783110656770. Disponível em: <https://doi.org/10.1515/9783110656770>.

BISHOP, C. M. Neural networks and their applications. *Review of Scientific Instruments*, v. 65, n. 6, p. 1803–1832, 06 1994. ISSN 0034-6748. Disponível em: <https://doi.org/10.1063/1.1144830>.

CARVALHO, F. de O. et al. Redes neurais artificiais e estatística multivariada no projeto de sensores virtuais para monitoramento da qualidade do ar. In: UNIVERSIDADE FEDERAL DE ALAGOAS (UFAL). Maceió, Brazil, 2007. Departamento de Engenharia Química, Universidade Federal de Alagoas (UFAL). Disponível em: <http://www.ufal.br>.

CESAR, J. et al. Partículas suspendidas (pst) y partículas respirables (pm10) en el valle de aburrá, colombia. *Revista Facultad de Ingeniería*, 01 2004.

CHOLLET, F. et al. *Keras*. GitHub, 2015. Disponível em: <https://github.com/fchollet/keras>.

DAVID, R. et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. In: SMOLA, A.; DIMAKIS, A.; STOICA, I. (Ed.). *Proceedings of Machine Learning and Systems*. [s.n.], 2021. v. 3, p. 800–811. Disponível em: <https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf>.

DIAS, J. W. C. *AGV manual de opreção*. 8rd. ed. Rua Gravataí, 99 – Rocha, Rio de Janeiro, 2016. Available at <https://www.energetica.ind.br/wp-content/uploads/2016/01/env1_manual-pts_rev_08.pdf>.

DONG, B. et al. Technology evolution from self-powered sensors to aiot enabled smart homes. *Nano Energy*, v. 79, p. 105414, 01 2021.

HAN, J.; PEI, J.; TONG, H. *Data mining: concepts and techniques*. [S.l.]: Morgan kaufmann, 2022.

HOLLER, F. et al. *Princípios de análise instrumental*. Bookman, 2009. ISBN 9788577804603. Disponível em: <https://books.google.com.br/books?id=Pl1OPwAACAAJ>.

IODICE, G. M. *TinyML Cookbook: Combine artificial intelligence and ultra-low-power embedded devices to make the world smarter*. Packt, 2022. ISBN 9781801814973. Disponível em: <https://www.packtpub.com/product/tinyml-cookbook/9781801814973>.

JUNIOR, R. A. *Classificação do Uso do Solo Utilizando Sensoriamento Remoto e Redes Neurais Artificiais*. Dissertação (Trabalho de Conclusão de Curso (TCC)) — Universidade Federal de São Paulo – UNIFESP, São José dos Campos, SP, 10 2020. Instituto de Ciência e Tecnologia, Engenharia de Computação.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research*, v. 4, p. 237–285, 1996.

KOTZ, J. et al. *Chemistry & Chemical Reactivity*. Cengage Learning, 2014. ISBN 9781305176461. Disponível em: <https://books.google.com.br/books?id=i1g8AwAAQBAJ>.

KOTZ, J. et al. *Chemistry & Chemical Reactivity*. Cengage Learning, 2018. ISBN 9781337399074. Disponível em: <https://books.google.com.br/books?id=ZH0mtAEACAAJ>.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, v. 5, n. 4, p. 115–133, 1943. ISSN 1522-9602. Disponível em: <https://doi.org/10.1007/BF02478259>.

MOHAMMED, M.; KHAN, M. B.; BASHIER, E. B. M. *Machine learning: algorithms and applications*. [S.l.]: Crc Press, 2016.

NEGNEVITSKY, M. *Artificial Intelligence: A Guide to Intelligent Systems (3rd Edition)*. 3. ed. [S.l.]: Addison Wesley, 2011. ISBN 1408225743,9781408225745.

RASCHKA, S. *Python Machine Learning*. [S.l.]: Packt Publishing, 2015.

RASCHKA, S.; MIRJALILI, V. *Python Machine Learning, 2nd Ed.* 2. ed. Birmingham, UK: Packt Publishing, 2017. 17–21 p. ISBN 978-1787125933.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, v. 3, n. 3, p. 210–229, 1959.

SARKER, I. H. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, v. 2, n. 3, p. 160, 2021. ISSN 2661-8907. Disponível em: <https://doi.org/10.1007/s42979-021-00592-x>.

SARKER, I. H. et al. Cybersecurity data science: an overview from machine learning perspective. *Journal of Big data*, Springer, v. 7, p. 1–29, 2020.

WARDEN, P.; SITUNAYAKE, D. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly, 2020. ISBN 9781492052043. Disponível em: <https://books.google.com.br/books?id=sB3mxQEACAAJ>.

YANG, J. et al. Quantization networks. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 7300–7308, 2019. Disponível em: <https://api.semanticscholar.org/CorpusID:198903430>.

ZANNETTI, D. P. *Air Pollution Modeling: Theories, Computational Methods and Available Software*. 1. ed. [S.l.]: Springer US, 1990. ISBN 978-1-4757-4467-5,978-1-4757-4465-1.

ZHANG, W. et al. Deep learning for click-through rate estimation. *CoRR*, abs/2104.10584, 2021. Disponível em: <https://arxiv.org/abs/2104.10584>.