UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

WESLLEY DO NASCIMENTO ALMEIDA

INTELIGÊNCIA ARTIFICIAL EM SISTEMAS EMBARCADOS UTILIZANDO ABORDAGEM TINYML

PATO BRANCO

2022

WESLLEY DO NASCIMENTO ALMEIDA

INTELIGÊNCIA ARTIFICIAL EM SISTEMAS EMBARCADOS UTILIZANDO ABORDAGEM TINYML

ARTIFICIAL INTELLIGENCE IN EMBEDDED SYSTEMS USING TINYML APPROACH

Trabalho de Conclusão de Curso apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Gustavo Weber Denardin
Coorientador: Prof. Dr. Dalcimar Casanova

PATO BRANCO

2022

WESLLEY DO NASCIMENTO ALMEIDA ✉

INTELIGÊNCIA ARTIFICIAL EM SISTEMAS EMBARCADOS UTILIZANDO ABORDAGEM TINYML

Trabalho de Conclusão de Curso apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de Aprovação: 12 de dezembro de 2022.

Prof. Dr. Gustavo Weber Denardin
Universidade Tecnológica Federal do Paraná

Prof. Dr. Dalcimar Casanova
Universidade Tecnológica Federal do Paraná

Prof. Dr. Jeferson José de Lima
Universidade Tecnológica Federal do Paraná

Prof. Dr. Marcelo Teixeira
Universidade Tecnológica Federal do Paraná

PATO BRANCO

2022

Com todo o meu amor, dedico este trabalho as duas mulheres da minha vida, minha mãe e minha amada Lhais, sem elas eu não teria chegado até esse momento.

AGRADECIMENTOS

Primeiramente, agradeço a minha namorada Lhais, que percorreu essa longa jornada ao meu lado e com muito carinho me motivou, apoiou e muito me ensinou, meu muito obrigado. Eu amo muito você.

Aos meus pais, Edmilson e Madalena, que não mediram esforços para me auxiliar nessa conquista. Eles foram fundamentais durante esse processo, me deram todo o suporte necessário para que eu acreditasse e tornasse possível esse nosso sonho. Ao meu irmão William, por ser meu parceiro e grande amigo. Ao meu irmão David Junio, o qual espero poder motivar e inspirar durante sua construção enquanto sujeito.

Deixo meu reconhecimento a aqueles que vieram antes de mim, em particular a minha querida vó Dona Geralda, mulher batalhadora que deixa seu legado gravado em minha história. Vó, saiba que não cabe em mim a satisfação de poder compartilhar contigo a felicidade dessa conquista. Se hoje tenho a possibilidade de escolher meu futuro, reconheço o esforço vindo de toda a nossa família.

Agradeço aos esforços da minha segunda família, Celso, Matheus e especialmente a minha sogra Terezinha, por ser muito amorosa e fazer sempre pra mim aquela broa de milho, maravilhosa.

Agradeço ao meu orientadores professor Dr. Gustavo Denardin e Prof. Dr. Dalcimar Casanova, pela oportunidade e por compartilharem seu conhecimento. Aos professores da Universidade Tecnológica Federal do Paraná, Campus de Pato Branco que fizeram parte da minha formação e que dedicam-se para levar um ensino de qualidade dentro de uma instituição pública.

Agradeço a todos as pessoas que convivi durante esse tempo na cidade de Pato Branco-PR, principalmente aos meus amigos Rafael Dalmolin, Fernando Paes, Welliton Leal e Giancarlo Abreu, os quais seguem comigo. Amigos, fiquem com a certeza de que vivenciar com vocês essa experiencia fez toda a diferença, compartilhar e conviver durante esse tempo, foi valioso.

“Irmão, você não percebeu que você é o único representante do seu sonho na face da terra? Se isso não fizer você correr chapa, eu não sei o que vai.” (Emicida)

RESUMO

Atualmente, existem mais de 250 bilhões de microcontroladores no mundo, logo, viabilizar a possibilidade de executar modelos de aprendizado de máquina em sistemas embarcados é motivado porque muitos dos dados captados por sensores são descartados devido ao custo, necessidade de conexão com internet ou restrições de energia. Diante desse cenário, surge uma nova abordagem de soluções de aprendizado de máquina, que integram sistemas com recursos limitados e inteligência artificial, o Tiny Machine Learning (TinyML), essa abordagem tem como objetivo viabilizar a implantação de modelos de inteligência artificial em sistemas embarcados de baixo custo e pouco poder de processamento. Entre as aplicações que poderiam se beneficiar da integração de aprendizado de máquina e hardware com recursos limitados, são os dispositivos de reconhecimento automático. Portanto, o presente trabalho busca realizar a implantação de uma rede neural artificial que realize o reconhecimento de placas veiculares em um hardware de baixo custo. Dessa forma, treinou-se um modelo MobileNetV2 SSD FPN-Lite para detecção de placas automotivas, efetuando a quantização fazendo uso do *framework* do TensorFlow Lite e implantando a solução em um sistema embarcado Raspberry Pi Zero 2W. Foram realizados experimentos em quatro formas de quantização e entre duas linguagens distintas, Python e C++. O melhor resultado apresentado considerando o tamanho de armazenamento, índice de confiabilidade e tempo de latência, foi o da quantização dinâmica em C++, pois, comparado ao modelo não quantizado, obteve uma redução em armazenamento de 75%, apresentando um score de 72,38% contra 72,28% do modelo não quantizado, e, uma eficiência no tempo de execução de 20%. Tornando assim, o tinyML uma alternativa viável para aplicações em sistemas com limitações de recursos.

Palavras-chave: aprendizado de máquina; sistemas embarcados; TensorFlow Lite; TinyML.

ABSTRACT

Currently, there are more than 250 billion microcontrollers in the world, therefore, enabling the possibility of running machine learning models in embedded systems is motivated because much of the data captured by sensors is discarded due to cost, need for internet connection or restrictions of energy. In this scenario, a new approach to machine learning solutions emerges, which integrate systems with limited resources and artificial intelligence, Tiny Machine Learning (TinyML), this approach aims to enable the implementation of artificial intelligence models in embedded systems of low cost and little processing power. Among the applications that could benefit from the integration of machine learning and resource-limited hardware are automatic recognition devices. Therefore, the present work seeks to implement an artificial neural network that recognizes license plates in low-cost hardware. In this way, a MobileNetV2 SSD FPN-Lite model was trained to detect automotive license plates, performing the quantization using the TensorFlow Lite framework and deploying the solution in a Raspberry Pi Zero 2W embedded system. Experiments were carried out in four forms of quantization and between two different languages, Python and C++. The best result presented considering the storage size, reliability index and latency time, was the dynamic quantization in C++, because, compared to the non-quantized model, it obtained a storage reduction of 75%, presented a score of 72, 38% against 72.28% of the non-quantized model, and a runtime efficiency of 20%. Thus making tinyML a viable alternative for applications in systems with limited resources.

Keywords: machine learning; embedded systems; TensorFlow Lite; TinyML.

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

SIGLAS

AI	Inteligência Artificial, do Inglês *Artificial intelligence*
AIoT	Inteligência Artificial das Coisas, do Inglês *Artificial Intelligence of Things*
API	Interface de Programação de Aplicação, do Inglês *Application Programming Interface*
FPN	Rede Pirâmide de Recursos, do Inglês Feature Pyramid Network
FPS	Frames por segundo, do Inglês *Frames Per Second*
IoT	Internet das coisas, do Inglês *Internet of Things*
MCU	Unidade de Microcontrolador, do Inglês *Microcontroller Unit*
ML	Aprendizado de Máquina, do Inglês *Machine Learning*
OCR	Reconhecimento Óptico de Caracteres, do Inglês *Optical Character Recognition*
RAM	Memória de Acesso Aleatório, do Inglês *Random Access Memory*
RNA	Redes Neurais Artificiais, do Inglês Recurrent Neural Network
SSD	Detector de Disparo Único, do Inglês Single Shot Detector
tflite	Arquivo TensorFlow Lite
TinyML	Aprendizado de máquina minúsculo, do Inglês *Tiny Machine Learning*
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO

Atualmente, existem mais de 250 bilhões de microcontroladores no mundo e apenas no ano de 2021 cerca de 32,2 bilhões de unidades foram vendidas (ALSOP, 2022), e a *IC Insights* prevê que o volume anual de remessas cresçam para 35,2 bilhões até 2023. Logo, viabilizar a possibilidade de executar modelos de aprendizado de máquina em sistemas embarcados é motivado porque muitos dos dados captados por sensores são descartados devido ao custo, necessidade de conexão com internet ou restrições de energia (SANCHEZ-IBORRA; SKARMETA, 2020, p. 5) – ou às vezes uma combinação dos três.

Diante desse cenário, surge uma nova abordagem de soluções de aprendizado de máquina, que integram sistemas com recursos limitados e inteligência artifical (WARDEN, 2018). O *Tiny Machine Learning* (TinyML) é definido como um conjunto de tecnologias em *Machine Learning* (ML) e sistemas embarcados, para fazer uso de aplicações de aprendizado de máquina em dispositivos de consumo de energia extremamente baixo (IODICE, 2022, p. 2). Essa abordagem tem como objetivo viabilizar a implantação de modelos de inteligência artificial em sistemas embarcados de baixo custo e pouco poder de processamento (DAVID *et al.*, 2021).

O valor econômico de mercado de TinyML é de US$ 70 bilhões, o que atrai empresas a desenvolverem soluções para o mercado de TinyML (ARTIBA, 2022). Entre as empresas que já disponibilizam *frameworks*, destacam-se a Google com o TensorFlow Lite, a Microsoft com Embedded Learning Library (ELL), a ARM com ARM-NN e a STM32 com a STM32Cube.AI. Essas soluções pretendem viabilizar a integração e disponibilização de modelos de aprendizado de máquina e microcontroladores (SANCHEZ-IBORRA; SKARMETA, 2020, p. 9-10).

Aplicações que podem se beneficiar da integração de aprendizado de máquina e hardware com recursos limitados, são os dispositivos de reconhecimento automático. Na literatura, há sistemas desse perfil para o reconhecimento de nível de maturação de frutas (NICOLAS; NAILA; AMAR, 2022), monitoramento das cores sinalizadoras de semáforos (ROSHAN *et al.*, 2021) e a identificação de gestos realizados com a mão (DAI; ZHOU, L., 2022). Porém, um domínio de aplicação que pode usufruir do tinyML, mas que ainda não conta com muitos trabalhos relacionados, é a área de reconhecimento de placas automotivas.

Portanto, neste trabalho desenvolveu-se uma solução de tinyML para reconhecimento de placas automotivas. Buscando reduzir o tamanho do modelo e acelerar o tempo de inferência, utilizou-se dos métodos de quantização do *framework TensorFlow Lite*.

1.1 OBJETIVOS

O objetivo geral deste trabalho é realizar a implantação de um modelo de aprendizado de máquina que possua capacidade de detectar e reconhecer placas automotivas em um sistema embarcado Raspberry Pi Zero 2W, utilizando-se da abordagem tinyML e o *framework* do TensorFlow Lite para adequação do modelo MobileNetV2 SSD FPN-Lite.

1.1.1 Objetivos Específicos

- Analisar o tamanho dos arquivos quantizados para o problema proposto, utilizando diferentes técnicas de quantização;
- Mensurar o tempo de inferência (latência) entre diferentes modelos MobileNetV2 SSD FPN-Lite quantizados em *Python* e *C++*;
- Calcular o índice de confiabilidade de detecção entre os diferentes modelos MobileNetV2 SSD FPN-Lite quantizados;
- Mensurar o consumo médio de memória do modelo MobileNetV2 SSD FPN-Lite de detecção quantizado e de reconhecimento no Raspberry Pi Zero 2W.

2 REVISÃO DA LITERATURA

Este capítulo está dividido conforme a cronologia dos métodos, onde são apresentados os conceitos básicos de aprendizado de máquina e as principais características de uma rede neural. Discorrendo sobre o campo de visão computacional, detecção de objeto e reconhecimento de caracteres ópticos. Após, o TinyML é contextualizado, apresentando a relação entre aprendizado de máquina e sistemas embarcados. Por fim, será detalhado sobre a implantação de modelos tinyML e como a biblioteca do TensorFlow Lite contribui para a disponibilização desse tipo de modelo.

Ressalta-se que a abordagem de embasamento teórico usada neste trabalho objetiva a aplicação utilizando a abordagem tinyML. Para um aprofundamento, recomenda-se a leitura das referências indicadas. Como a proposta do trabalho não é detalhar o desenvolvimento do modelo de aprendizado de máquina, faz-se necessária uma breve noção dos fundamentos para compreender as etapas de quantização.

2.1 APRENDIZADO DE MÁQUINA

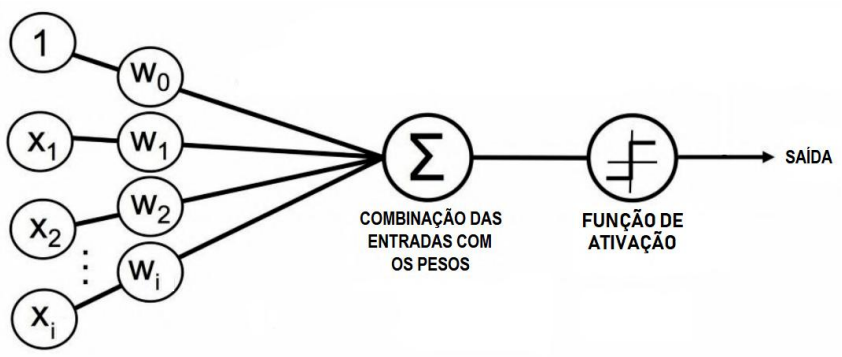
Aprendizado de máquina é definido como um campo de estudo que possibilita aos computadores ter a capacidade de aprenderem sem serem explicitamente programados (SAMUEL, 1959). O aprendizado de máquina apresenta diversas ramificações de diferentes tipos de aprendizado, porém, quatro categorias destacam-se, sendo: aprendizado supervisionado, aprendizado não supervisionado, aprendizado semi-supervisionado e aprendizado por reforço (SARKER, 2021).

As redes neurais fornecem uma gama de novas técnicas para solucionar problemas de reconhecimento de padrões (BISHOP, 1994) e no âmbito de problemas de visão computacional métodos baseados em redes neurais, obtiveram êxito em solucionar tarefas de reconhecimento (ALEXEY; WANG; MARK LIAO, 2020). A família de redes neurais artificiais é de interesse particular, pois sua estrutura flexível permite que sejam modificadas para uma ampla variedade de contextos em todos os quatros tipos de ramificações (SARKER, 2021).

2.1.1 Redes Neurais

As redes neurais artificiais (RNAs) são um conjunto de algoritmos de inteligência artificial, que apresentam um modelo inspirado na estrutura neural de organismos inteligentes, que adquirem conhecimento através da experiência (NEGNEVITSKY, 2005). Os primeiros trabalhos que relatam o funcionamento de um neurônio, foram desenvolvidos por McCulloch e Pitts em 1943 e foram inspirados em um neurônio biológico (MCCULLOCH; PITTS, 1943).

Figura 1 – Modelo de neurônio de McCulloch e Pitts



Fonte: (6 NICOLAU, 2020, p. 26 apud RASCHKA, 2015).

A Figura 1 apresenta as características de um neurônio composto por um conjunto de entradas (x) que ao serem multiplicadas por pesos (w) e posteriormente somados, realizarão a inserção do resultado em uma função de ativação, que tem o objetivo de decidir se o neurônio irá ou não realizar uma sinapse (RASCHKA, 2015, p. 21). A Equação (1) apresenta matematicamente a operação apresentada pelo neurônio.

$$z = x_0 w_0 + x_1 w_1 + \ldots + x_n w_n = \sum_{i=0}^{n} x_i w_i = \boldsymbol{w}^T \boldsymbol{x} \tag{1}$$

Com a entrada $x_0 = 1$, podemos representar $x_0 w_0 = \theta$. A definição de θ representa o viés (bias), esse parâmetro é responsável por deslocar a reta de decisão, tornando o neurônio capaz de dividir o espaço de entrada em duas regiões. Dessa forma a Equação (1) é definida na Equação (2). Para classificação, a Equação (2) é passada na função de ativação binária Equação (3) que representa a sinapse do neurônio (HAGAN; DEMUTH; BEALE, 1997).

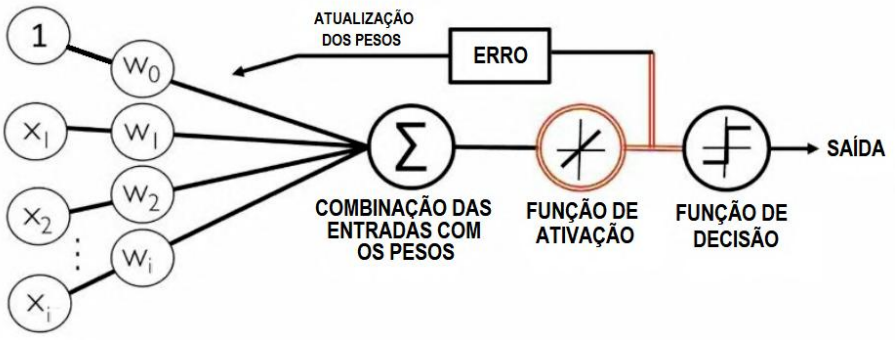
$$z = \theta + \sum_{i=0}^{n} x_i w_i \tag{2}$$

$$\phi(z) = \begin{cases} 1, & \text{se } z \geq 0 \\ 0, & \text{caso contrário} \end{cases} \tag{3}$$

Uma das propriedades mais importantes de uma rede neural artificial é a capacidade de aprender por intermédio de exemplos e fazer inferências sobre o que aprendeu, melhorando gradativamente o seu desempenho (FERNEDA, 2006, p. 27 apud BRAGA; LUDERMIR; CARVALHO, 2000).

O Perceptron representado na Figura 2, apresenta o modelo desenvolvido por Rosenblatt (1958) que trouxe a capacidade de tomada de decisão para as redes neurais ao incluir uma função de decisão (ROSENBLATT, 1958). Por ter essa finalidade é considerado uma importante arquitetura dos modelos de redes neurais. Sendo um classificador binário que recebe sinais externos na camada de entrada e os transmite para a camada de saída (ZHOU, Z.-H., 2021). O algoritmo aprende o coeficiente de peso ótimo para a multiplicação das características de entrada e realiza a decisão se o neurônio será ou não ativado e funciona como um classificador (RASCHKA, 2015).

Figura 2 – Modelo de Perceptron



Fonte: (6 NICOLAU, 2020, p. 37 apud RASCHKA, 2015).

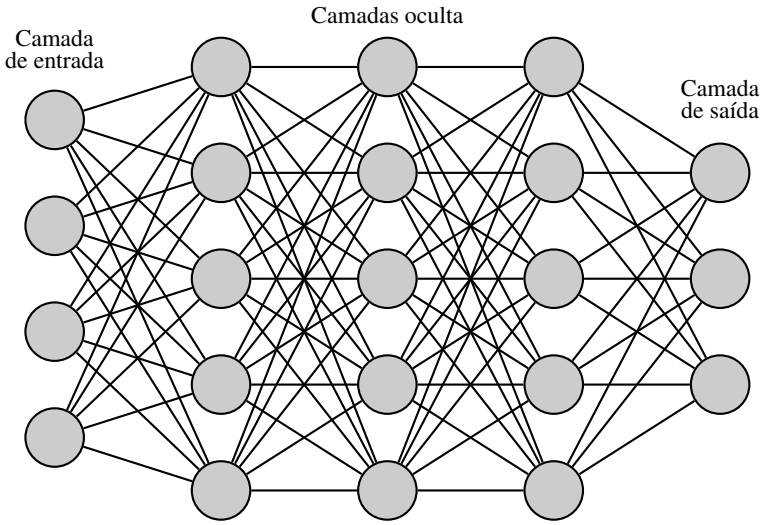
2.1.2 Aprendizado Profundo

Aprendizado profundo é apresentado por Goodfellow, Bengio e Courville (2016) como um tipo específico de aprendizado de máquina. Além de ser um subcampo de aprendizado de máquina, foi inspirado nas redes neurais (BROWNLEE, 2016).

Diante das definições LeCun, Bengio e Hinton (2015) o aprendizado profundo melhorou o estado da arte em reconhecimento de fala, reconhecimento visual de objetos, detecção e diversos outros domínios. Os avanços do aprendizado profundo têm a ver com novos conceitos, como

arquiteturas inovadoras e as regras de treinamento, porém, as diversas arquiteturas compartilham componentes fundamentais, sendo os números de camadas, pesos, funções de ativações (XU *et al.*, 2021). Ressalta-se que a profundidade está relacionada a quantidade de camadas ocultas, onde modelos que apresentam duas ou mais camadas, se enquadram em aprendizado profundo (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 189).

Figura 3 – Redes Neurais feedforward multicamadas



Fonte: Adaptado de (ZHOU, Z.-H., 2021, p. 108)

Observa-se na Figura 3, que cada neurônio apresenta as características fundamentarias de redes neurais. Entretanto, uma grande quantidade de pesos consomem um armazenamento considerável da memória (largura de banda de memória), o que para sistemas com limitações de recursos pode ser considerado uma desvantagem. Por exemplo, o *Faster RCNN* [1] possui mais de 200MB, o que dificulta a implantação de redes neurais profundas em dispositivos de borda (e.g, microcontroladores, microprocessadores, etc) (HAN; MAO; DALLY, 2015).

2.2 VISÃO COMPUTACIONAL

Visão computacional é uma subárea do aprendizado profundo, responsável por extrair informações do mundo a partir de imagens (FORSYTH; PONCE, 2011). O campo de visão computacional tem como ramificações a detecção de objetos, reconhecimento e restauração de imagem (SZELISKI, 2022). Portanto, o presente trabalho especificará informações a respeito de detecção e reconhecimento de objetos, pois, esses métodos descritos, foram utilizados para

[1] https://tfhub.dev/tensorflow/faster_rcnn/inception_resnet_v2_640x640/1.

realizar a detecção de placas automotivas e posteriormente o reconhecimento dos caracteres presentes na placa.

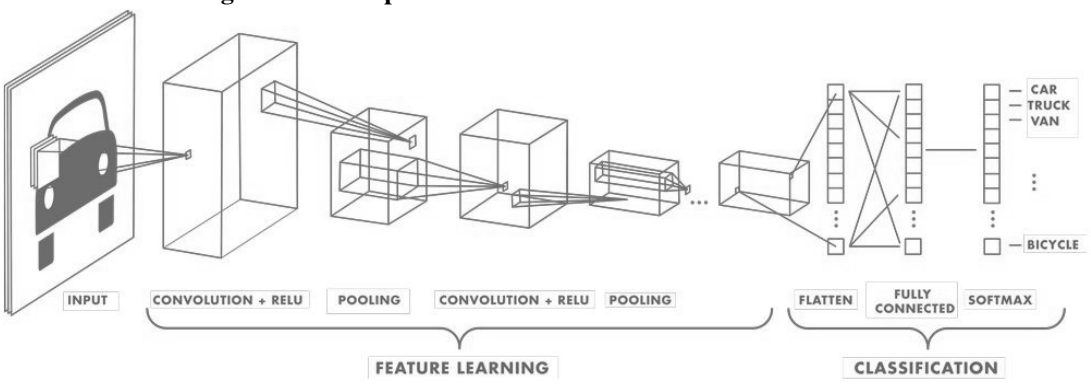
2.2.1 Detecção de Objetos

Detecção de objetos é um processo de varredura de objetos em uma imagem, visando determinar a sua localização e seu reconhecimento em classes definidas (SZELISKI, 2022). Ou seja, é basicamente a habilidade de determinar uma classe (e.g, humanos, animais ou carros) em imagens digitais (ZOU *et al.*, 2019).

Segundo LeCun, Bengio e Hinton (2015) criadores da abordagem de redes neurais convolucionais (ConvNets ou CNNs), os métodos clássicos de detecção de imagens eram manuais, o que tornava complexo o desenvolvimento de modelos eficientes. Atentando para isso, buscou-se diminuir essa complexidade de desenvolvimento de modelos capazes de detectar objetos de maneira sofisticada, desenvolveu-se a rede neural convolucional, um método que foi disruptivo na forma como os modelos modernos seriam desenvolvidos. Essa contribuição foi tão significativa que praticamente todos os métodos de classificação de imagens modernos são uma variação do método tradicional das CNNs (ALZUBAIDI *et al.*, 2021).

A Figura 4 ilustra os três tipos principais de camadas das CNNs: camadas convolucionais (convolutional), camadas de subamostragem (*pooling layer*) e camadas totalmente conectadas (*fully connect*) (LECUN; BENGIO; HINTON, 2015). A rede apresentada na Figura 4 está enquadrada na categoria de classificação, porém, é possível remover as camadas totalmente conectadas (*fully connect*) e softmax e substituí-las por redes de detecção, como SSD, Faster R-CNN e outras para realizar a detecção de objetos (IMPULSE, 2022).

Figura 4 – Exemplo de uma rede com camadas convolucionais



Fonte: Adaptado (MATHWORKS, 2020).

2.2.2 Reconhecimento de Caracteres Ópticos (OCR)

Reconhecimento de caracteres ópticos (do inglês *Optical Character Recognition* - OCR) é uma técnica de tradução de caracteres manuscritos, datilografados ou impressos em um texto codificado por máquina, amplamente utilizado para reconhecimento de placas veiculares, leitura de passaportes e etc (MORI; NISHIDA; YAMADA, 1999).

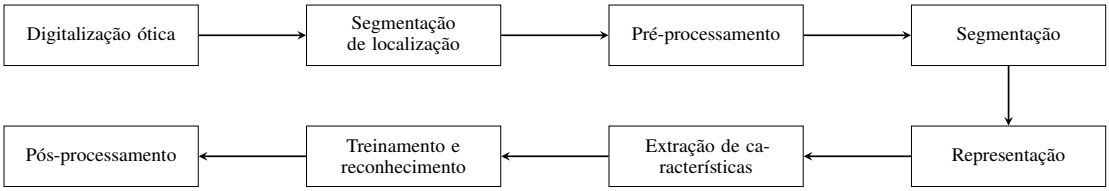
Chaudhuri *et al.* (2017) apresenta as etapas para desenvolvimento de um modelo de OCR conforme representado na Figura 5, encontram-se digitalização ótica, segmentação de localização, pré-processamento, segmentação, representação, extração de características, treinamento e pós-processamento.

- **Digitalização ótica:** Consiste em capturar a imagem e disponibilizar ao sistema para que seja possível realizar o processamento das imagens (CHAUDHURI *et al.*, 2017).
- **Segmentação de localização:** Consiste em localizar as regiões de interesse, descartando elementos que não fazem parte do objeto a ser classificado. Quando em elementos de texto, consiste em selecionar carácter por carácter, para classificação individual (CHAUDHURI *et al.*, 2017).
- **Pré-processamento:** Etapa que visa corrigir detalhes como inclinação e ruídos presentes na imagem através de filtros e normalizações (CHAUDHURI *et al.*, 2017).
- **Segmentação:** A etapa de segmentação consiste em subdividir o texto em pequenas áreas ou caracteres, para que possa ser realizado a inferência em nível fragmentado, o que coopera para interpretação do elemento (CHAUDHURI *et al.*, 2017).
- **Representação:** Após a segmentação, a representação irá estruturar a imagem de uma forma simplificada, preservando as propriedades da imagem. Esse método é aplicado visando reduzir a complexidade da etapa de extração de características (CHAUDHURI *et al.*, 2017).
- **Extração de características:** O objetivo da extração de recursos é capturar as principais características dos símbolos para que seja possível realizar a classificação do caractere (CHAUDHURI *et al.*, 2017).
- **Treinamento e reconhecimento:** Etapa que consiste treinar o modelo para que seja possível realizar a classificação. Existem quatro abordagens gerais que são: correspondência de modelo (*template matching*), técnicas estatísticas, técnicas estruturais e redes neurais artificiais. Ressalta-se que não necessariamente as abordagens precisam ser excludentes em um modelo final de classificação e normalmente são utilizados a

união dos métodos para uma melhor tomada de decisão (CHAUDHURI *et al.*, 2017).

- **Pós-processamento:** Nesta etapa é realizado o agrupamento dos caracteres e se possível o ajuste de pequenos erros, para disponibilizar a inferência final (CHAUDHURI *et al.*, 2017).

Figura 5 – Componentes de um sistema OCR



Fonte: Adaptado de (CHAUDHURI *et al.*, 2017) .

As soluções estado da arte para problemas de reconhecimento de caracteres são de modelos baseados em aprendizado profundo (MITTAL; GARG, 2020). Atualmente, entre as variedades de soluções de aprendizado profundo para reconhecimento de caracteres, estão Keras-OCR e EasyOCR, utilizados para reconhecimento de placas automotivas no trabalho de Bhardwaj *et al.* (2022) e Tesseract Mittal e Garg (2020).

2.3 SISTEMAS EMBARCADOS

Um sistema embarcado possui apenas uma única tarefa e faz interação direta com o ambiente, utilizando-se de sensores e atuadores (CHASE; ALMEIDA, 2007). Alguns exemplos de sistemas embarcados em diferentes aplicações são apresentados por (DENARDIN; BARRIQUELLO, 2019):

- Automotivos: controle de injeção eletrônica, controle de tração, controle de sistemas de frenagem antibloqueio (ABS) etc.;
- Domésticos: micro-ondas, lavadouras de louça, lavadouras de roupa etc.;
- Robótica: robôs industriais, humanoides, drones etc.;
- Controle de processos: processamento de alimentos, controle de plantas químicas e controle de manufaturas em geral.

Um sistema embarcado está dividido em duas partes: Uma com os microprocessadores que executam instruções do tipo aritméticas e são responsáveis pelo controle de comunicação entre os dispositivos e periféricos. E outra parte com os microcontroladores, que são chips que integram em sua estrutura interna memória, processador e funções de entrada e saída (PEREIRA *et al.*, 2011). As limitações de recursos disponíveis em sistemas embarcados tornam

sua utilização em sistemas complexos (e.g., inteligência artificial), um desafio de implementação (MARWEDEL, 2021).

2.3.1 Computação de borda

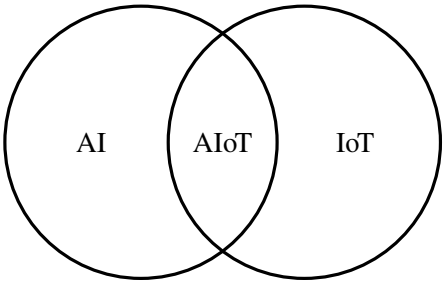
Computação de borda (edge) consiste em dispositivos individuais (e.g., tablets, celulares, sensores, etc.) que realizam o processamento de informações em ambiente local, evitando a centralização de processamento na nuvem (GARCIA LOPEZ *et al.*, 2015). Soluções leves que realizem tarefas de aprendizado de máquina ou processamento de dados tem viabilizado o processamento em ambiente local, evitando o envio de informações para ambiente da nuvem e gerando a tomada de decisão na borda (VARGHESE *et al.*, 2016).

2.4 TINYML

Segundo Warden e Situnayake (2019) o aprendizado de máquina minúsculo (tinyML) tem o objetivo de vincular o aprendizado de máquina que ajusta os modelos para que sejam pequenos o suficiente para serem implantados em dispositivos de borda, com baixa capacidade de armazenamento e com baixo poder computacional.

O tinyML surgiu para suprir a necessidade de realizar a interpretação dos dados em tempo real em dispositivos de baixo custo sem a necessidade de envio para nuvem (IODICE, 2022). Com esse intuito, Dong *et al.* (2021) estudou o comportamento da Inteligência Artificial incorporada ao IoT e esclarece que a combinação criou uma nova geração de dispositivos inteligentes (Artificial Intelligence of Things, AIoT, Figura 6), que une técnicas de coleta de dados através de dispositivos IoT e realiza a tomada de decisão em tempo real e localmente por um aprendizado tinyML.

Figura 6 – Paradigma AIoT



Fonte: Autoria própria (2022).

Dentro desse cenário, Banbury *et al.* (2021) buscou quantificar os modelos tinyML, abordando em sua pesquisa características sobre as limitações em etapas de implementação, sendo elas:

- **Baixo consumo de energia:** O consumo de energia é um dos recursos fundamentais em sistemas tinyML, necessitando ser prioritariamente uma combinação eficiente no contexto de energia, para que seja viável a implantação (BANBURY *et al.*, 2021).
- **Limitação de memória:** Os sistemas tradicionais de aprendizado de máquina lidam com restrições na casa de *gigabyte*, enquanto os sistemas tinyML precisam fornecer total funcionamento em sistemas com limitações na casa de *megabyte* (BANBURY *et al.*, 2021).

2.5 IMPLANTAÇÃO TINYML EM DIPOSITIVOS EMBARCADOS

Os dispositivos TinyML podem ser disponibilizados em sistemas centralizados ou distribuídos (IODICE, 2022), porém, os sistemas embarcados não possuem uma estrutura TinyML unificada (DAVID *et al.*, 2021).

No momento da implantação de redes neurais nesses sistemas, é necessário construir estruturas únicas que exigem uma otimização manual para cada plataforma de hardware. Essas estruturas personalizadas tendem a ter um foco restrito, sem recursos para suportar vários aplicativos e sem portabilidade em uma ampla variedade de hardware. Esse fator, tem tornado a experiência do desenvolvedor trabalhosa, exigindo otimização manual de modelos para execução em um dispositivo específico (DAVID *et al.*, 2021). Diante desse cenário, a utilização do TensorFlow Lite busca viabilizar a implantação de soluções TinyML em dispositivos de borda.

2.5.1 TensorFlow

O TensorFlow é uma plataforma de código aberto para desenvolvimento de modelos de aprendizado de máquina (MARTÍN ABADI *et al.*, 2015). Oferecendo suporte a uma variedade de aplicativos, com foco em treinamento e inferência em redes neurais profundas, foi desenvolvido baseado em grafos computacionais e realiza operações de alta escalabilidade, utilizando as características dos tensores para representar e consolidar as informações dos modelos (ABADI *et al.*, 2016).

Matematicamente, os tensores podem ser entendidos como uma generalização de

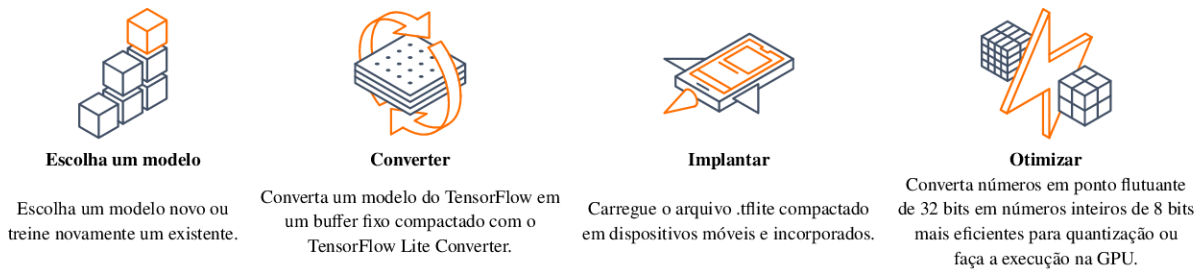
escalares, vetores, matrizes e assim por diante, já no contexto de operações do Tensorflow, o tensor é utilizado como um identificador simbólico para se referir à entradas e saídas das operações (RASCHKA, 2015), dessa forma, na API do Tensorflow todos os tensores são representados como matrizes n-dimensionais para que os cálculos possam ser executados de maneira eficaz (ABADI *et al.*, 2016).

2.5.2 TensorFlow Lite

O TensorFlow Lite é uma ramificação do Tensorflow, desenvolvido para executar modelos de aprendizado de máquina em sistemas com recursos limitados de processamento e memória (DAVID *et al.*, 2021). Por esse motivo, para que seja possível executar um modelo em um sistema embarcado, torna-se necessário a utilização de um interpretador diferente, com padrões que respeitem as características limitantes do sistema de destino (WARDEN; SITUNAYAKE, 2019).

A documentação do TensorFlow Lite apresenta um roteiro para disponibilização de modelos em dispositivos de borda. A Figura 7 ilustra as etapas a serem consideradas para implantação. A seguir são apresentados os detalhes de cada uma dessas etapas.

Figura 7 – Pipeline TensorFlow Lite



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

2.5.2.1 Escolha do modelo de aprendizado de máquina

No momento de decidir uma arquitetura, é necessário analisar o tipo de problema que está tentando resolver, os tipos de dados que se tem acesso e as maneiras de manipular esses dados antes de alimentá-los em um modelo (WARDEN; SITUNAYAKE, 2019). Os dados e a arquitetura do modelo são profundamente entrelaçados e é fundamental considerar as características do dispositivo de destino (IODICE, 2022).

2.5.2.2 Conversão do modelo para Tensorflow Lite

A conversão do modelo para Tensorflow Lite consiste em transformar o modelo de aprendizado de máquina em um formato flatbuffers[2] que é representado como uma extensão TensorFlow Lite (tflite). A conversão irá fornecer diversas vantagens sobre o modelo buffer de protocolo do TensorFlow, entre elas um tamanho reduzido e uma inferência mais rápida, o que o torna ideal para dispositivos com limitação de memória (MARTÍN ABADI *et al.*, 2015).

2.5.2.3 Implantação do modelo Tensorflow Lite

A implantação é o processo de execução de um modelo TensorFlow Lite no dispositivo, buscando fazer previsões com base nos dados de entrada. Nesta etapa é necessário escrever códigos que realizem a captura dos dados e adequá-los para que a inferência seja realizada no dispositivo. Esse processo em si, irá resultar numa saída contendo as previsões (MARTÍN ABADI *et al.*, 2015). Considerando a Tabela 1, os índices de confiabilidade validam o limiar para considerar a identificação da placa veicular como válida ou não, quanto maior a pontuação, maior será a credibilidade do resultado de detecção.

Tabela 1 – Exemplo de índice de confiabilidade

Score	Explicação
0,90	Alta confiança da detecção da placa
0,70	Confiança intermediária da detecção da placa
0,50	Resultado inconclusivo
0.20	Detecção de placa não confiável

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

2.5.2.4 Otimização de modelos de aprendizado de máquina

Na sequência, após o modelo implantado, são realizados os testes de inferência em ambiente real, fazendo a verificação de desempenho no dispositivo de implantação, pois, os dispositivos de borda geralmente tem uma memória limitada, assim, várias otimizações podem ser aplicadas aos modelos, para que possam ser executados dentro dessas restrições. Portanto, os modelos podem ser otimizados conforme a necessidade, visando buscar a redução do tamanho, a

[2] 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://google.github.io/flatbuffers/`.

redução de latência e também para compatibilidade com o dispositivo de destino (WARDEN; SITUNAYAKE, 2019, cap. 15, cap. 16, cap. 17).

Ressalta-se, que as quantizações podem apresentar uma perda de precisão nos modelos. Existe uma relação entre a otimização do tamanho e/ou tempo de latência do modelo com a sua precisão. Sendo assim, é responsabilidade do time de desenvolvimento decidir o limiar de alteração dos otimização (YANG *et al.*, 2019). A seguir serão detalhados os métodos de quantização, poda e agrupamento, essas técnicas são utilizas com finalidade de otimizar a compressão e latência dos modelos.

- **Quantização:** 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 (MARTÍN ABADI *et al.*, 2015). Entre os tipos disponibilizados na documentação do Tensorflow Lite, irei definir as quantizações enquadradas no contexto de pós-treinamento, onde um modelo já previamente treinado tem seus pesos quantizados.
  - **Quantização de faixa dinâmica:** Método responsável por efetuar ajustes dos pesos dos neurônios do modelo, que por padrão são pontos flutuantes representados em 32 bits e convertê-los para 8 bits, dessa forma, a representação em ponto flutuante é convertida para inteiro (MARTÍN ABADI *et al.*, 2015).
  - **Quantização float16:** Para a quantização em float16, somente os pesos dos neurônios do modelo são convertidos para números de ponto flutuante de 16 bits, e as operações (e.g. cálculo de função de ativação) continuam com suporte de operações de 32 bits (MARTÍN ABADI *et al.*, 2015).
  - **Quantização inteira completa:** A quantização inteira completa possui a função de ajustar todas as operações e parâmetros internos do modelo para serem executadas em inteiro, porém, mantendo a entrada e saída em ponto flutuante (JACOB *et al.*, 2018).
  - **Quantização somente inteiro:** A quantização somente inteiro, converte todas as operações dos modelos, inclusive, entrada e saída para a representação de 8 bits (JACOB *et al.*, 2018).

  A Tabela 2 detalha fatores a serem considerados para escolha da técnica de quantização, disponibilizadas no Tensorflow Lite e apresenta os hardwares compatíveis de acordo com as quantizações, afirmando que cada forma de quantização apresenta alterações na precisão do modelo quando comparado ao modelo original:

Tabela 2 – Características dos diferentes tipos de quantização

Técnica	Requisitos de dados	Redução de tamanho	Precisão	Hardware Suportado
Quantização pós-treinamento float16	Sem dados	Até 50%	Perda de precisão insignificante	CPU, GPU
Quantização pós-treinamento faixa dinâmica	Sem dados	Até 75%	Menor perda de precisão	CPU, GPU (Android)
Quantização pós-treinamento inteira	Amostra representativa sem etiqueta	Até 75%	Pequena perda de precisão	CPU, GPU (Android), EdgeTPU, Hexagon DSP
Quantização pós-treinamento inteira completa	Amostra representativa sem etiqueta	Até 75%	Pequena perda de precisão	CPU, Edge TPU, Microcontrollers
Treinamento ciente de quantização	Dados de treinamento rotulados	Até 75%	Menor perda de precisão	CPU, GPU (Android), EdgeTPU, Hexagon DSP

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

- **Poda:** É uma técnica que tem como objetivo remover os parâmetros de pouco impacto em um modelo. Normalmente os modelos podados tem o mesmo tamanho no disco e também tem a mesma latência, porém, podem ser compactados com mais eficácia (ZHU; GUPTA, 2017).
- **Agrupamento:** Nessa etapa, são agrupados os pesos de cada camada de um modelo em uma quantidade de número pré-definido de aglomerados. Em seguida, os valores utilizam apenas os pesos do centroide do grupo pertencente. Isso reduz o número de valores de peso exclusivos em um modelo, reduzindo assim, a complexidade do modelo (MARTÍN ABADI *et al.*, 2015). Como resultado, os modelos em agrupamentos podem ser compactados com mais eficácia, fornecendo benefícios de implantação semelhantes ao de poda (HAN; MAO; DALLY, 2015).

2.6 TRABALHOS CORRELATOS

Nesta seção são abordados trabalhos que assemelham-se e serviram de base para o desenvolvimento deste trabalho, pois, fizeram uso de métodos de detecção de objetos em dispositivos com limitações de recursos.

2.6.1 TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems

Em (DAVID *et al.*, 2021) é apresentado um mecanismo de conversão TensorFlow Lite Micro e para demonstração do funcionamento foi realizado testes em modelos de aprendizado profundo. Entre os modelos desenvolvidos e apresentados, destaca-se o modelo de detecção de pessoas que foi treinado em um modelo MobileNetV1, e posteriormente, implantado em Sparkfun Edge, que possui um processador Arm Cortex-M4, que opera com clock de 96MHz e possui 1MB de memória flash. Pelo fato do trabalho apresentar somente um modelo de detecção de pessoas, ele não é integralmente compatível com o objetivo deste trabalho, mas, é uma base fundamental para o desenvolvimento de funções como o método de quantização.

2.6.2 Detecting Driver Drowsiness in Real Time Through Deep Learning Based Object Detection

Em (SHAKEEL *et al.*, 2019) apresenta-se o desenvolvimento de uma rede neural que realiza a detecção de sonolência em motoristas. O modelo utilizado para detecção foi MobileNet-SSD e foi quantizado fazendo uso do Tensorflow Lite. O modelo foi disponibilizado no *smartphone* Sony Xperia Z, e, possui um processador Snapdragon S4 Pro Qualcomm com 1.5GHz Quad Core e 2GB de memória RAM. Segundo os autores o modelo gerado é compatível com as limitações do Raspberry Pi 3. Esse trabalho auxiliou para compreensão do funcionamento de um modelo de detecção baseado em MobileNet-SSD.

2.6.3 Automated Yard Processes using TinyML

Em (KHINE, 2020) realizou-se o desenvolvimento e implantação de uma solução de reconhecimento de placas em um dispositivo Arduino Portenta H7 + Vision Shield processador dual core STM32H747 com um Cortex M7 com clock de 480MHz e um Cortex M4 com clock de 240MHz, esse dispositivo apresenta uma capacidade de armazenamento em memória 16MB flash e 8MB SDRAM. Para a detecção da placa do automóvel utilizou-se uma função do Micropython e para o treinamento de um modelo personalizado de caracteres ópticos utilizou-se a plataforma da *Edge Impulse* e posteriormente, a etapa de quantização foi realizada no Tensorflow Lite. Apesar do trabalho obter êxito na disponibilização da solução, utilizou-se a plataforma *Edge Impulse* responsável por abstrair a camada de disponibilização no dispositivo final.

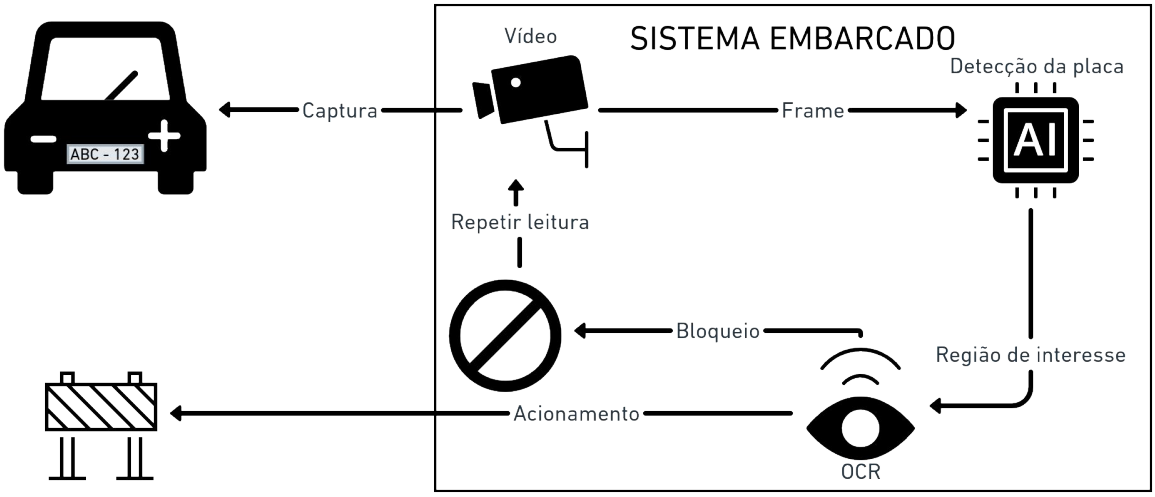
2.6.4 Low-Power License Plate Detection and Recognition on a RISC-V Multi-Core MCU-Based Vision System

Em (LAMBERTI *et al.*, 2021) desenvolve-se um pipeline de processamento visual, que utiliza uma abordagem de inferência multi-modelo baseada em SSDlite-MobilenetV2 para detecção de placas de veículos e a topologia LPRNet para reconhecimento óptico de caracteres, alcançando uma pontuação mAP de 38,9% para a primeira tarefa e uma taxa de reconhecimento de 99,13% para a última em conjuntos de dados públicos. Devido às estratégias de compressão e otimização aplicadas, a inferência multi-modelo atinge uma taxa de transferência de 1.09 FPS a um custo de energia de 117 mW ao executar em um MCU GAP8 de 512 kB de RAM e um processador de 8 núcleos a 175 MHz de clock. A solução é o primeiro dispositivo de classe MCU que incorpora tal nível de complexidade de rede, resultando em um sistema ALPR de classe móvel, 73 vezes mais eficiente em termos de energia com um Raspberry Pi3. O trabalho apresentado nessa seção, atinge todos os objetivos propostos no desenvolvimento deste trabalho, com ressalva de avanços significativos, obtendo êxito em realizar o reconhecimento em dispositivos microcontroladores, servindo assim, de modelo na proposta.

3 DESENVOLVIMENTO

O sistema de reconhecimento de placas automotivas proposto neste trabalho, pode ser visualizado na Figura 8, no qual uma câmera realiza a captura em vídeo do automóvel e faz o envio da imagem (frame) para o modelo de detecção de placa automotiva, posteriormente a realização da inferência sobre a imagem, o modelo apresenta um índice de confiabilidade de detecção (score), caso o score respeite o limiar de 70% de confiabilidade, a região de interesse selecionada pelo modelo de detecção será enviada para o modelo de reconhecimento dos caracteres, que por sua vez, retornará os caracteres da imagem capturada e irá permitir a tomada de decisão (e.g., abertura de uma cancela).

Figura 8 – Modelo proposto



Fonte: Autoria própria (2022).

Desta forma, o capítulo apresenta o desenvolvimento de um sistema de reconhecimento de placas automotivas e por se tratar de uma aplicação que é executada em um sistema com recursos limitados, foi escolhido a utilização do MobileNet V2, pois a sua arquitetura é compatível com esse formato e os detalhes são apresentados na Seção 3.2.2. Para o reconhecimento de caracteres ópticos (OCR), foi utilizada a API do Tesseract. O sistema desenvolvido posteriormente é executado em um Raspberry Pi 3B+ limitado (*underclock*) para simular um Raspberry Pi Zero 2W, utilizando a abordagem TinyML.

A seguir, serão apresentados os materiais e os procedimentos para o treinamento da rede neural, utilizando o método de aprendizado por transferência e as técnicas aplicadas para implantação no Raspberry Pi 3B+. Todas as etapas relatadas foram desenvolvidas nos passos representandos na Figura 7, disponibilizados pela equipe do TensorFlow.

3.1 MATERIAIS

Os materiais utilizados para desenvolvimento desse trabalhado foram categorizados para facilitar a identificação em cada etapa.

3.1.1 Transferência de aprendizado do modelo para detecção de placas

- Python 3.7.15
- Google Colab
- Tensorflow 2.9.2
- CuPy-cuda11x 11.0.0

3.1.2 Inferência em Python

- Notebook Dell, Intel® Core™ i5-3337U CPU @ 1.80GHz × 4 com 8GB RAM
- Python 3.8.13
- Tensorflow 2.7.0
- Tesseract 4.1.1
- Leptonica 1.79.0
- PyTesseract 0.3.10
- OpenCV-Python 4.6.0.66

3.1.3 Inferência em C++

- Raspberry Pi 4B+, ARM-Cortex A72 64-bit @1.5GHz e 2GB LPDDR4 RAM
- Linux raspberrypi 5.15.61-v8+
- GCC/G++ versão 10.2.1
- Tesseract 4.1.1
- Leptonica 1.79.0
- Bazel 4.2.2
- Tensorflow 2.6.0
- OpenCV2 4.6.0-dev

3.1.4 Execução em tempo real

- Raspberry Pi 3B+, Cortex-A53 64-bit @1.4GHz e 1GB LPDDR2 SDRAM
- Linux raspberrypi 5.15.61-v8+
- Webcam USB Full HD 1080P WB
- GCC/G++ versão 10.2.1
- Tesseract 4.1.1
- Leptonica 1.79.0
- Bazel 4.2.2
- Tensorflow 2.6.0
- OpenCV2 4.6.0-dev

3.2 TRANSFERÊNCIA DE APRENDIZADO DO MODELO PARA DETECÇÃO DE PLACAS

O desenvolvimento dessa etapa consiste em realizar o treinamento de detecção de placa em ambiente *Google Colab*, fazendo uso do método de transferência de aprendizado (*transfer learning*), possibilitando reaproveitar as camadas iniciais do treinamento e especializar a rede neural em reconhecimento de placas automotivas.

3.2.1 SELEÇÃO DAS BASES DE DADOS

Para especializar o modelo de detecção de placas veiculares, foi selecionada a base *Car License Plate Detection* (LARXEL, 2020), que contém 455 imagens de veículos, que já apresentam as anotações de caixa delimitadora das placas, não sendo necessário a anotação manual. Essas anotações são fornecidas no formato PASCAL VOC[1].

Na etapa de teste do modelo, foram selecionadas 50 imagens aleatórias das bases *Dataset for Vehicle License Plate Location*, disponibilizada publicamente pela Universidade Federal de Ouro Preto (UFOP) (MENDES JÚNIOR *et al.*, 2011) e (ANAGNOSTOPOULOS *et al.*, 2008).

[1] Pascal VOC, arquivo XML contendo as características da imagem.

3.2.2 Escolha do modelo de detecção de objeto

O desafio do treinamento de redes neurais capazes de reconhecer objetos, está em captar dados suficientes para o treinamento do modelo e o poder computacional necessário para o processamento dos dados (ERHAN *et al.*, 2009). Portanto, o método utilizado para a especialização do modelo foi o de transferência de aprendizado, para que haja proveito das camadas intermediárias e especializando somente as últimas camadas da rede neural, tornando um modelo anteriormente pré-treinado generalista, especialista em detecção de placas automotivas.

Por se tratar de uma solução onde a arquitetura é limitada, é necessário escolher modelos que funcionem de acordo com as características do sistema. Em um comparativo entre modelos de detecção de objetos o modelo MobileNetV2 SSD supera outros detectores em tempo real de última geração (e.g. YOLOv2, MobileNet V1) (SANDLER *et al.*, 2018b).

O modelo MobileNetV2 SSD FPN-Lite 320x320 (CHIU *et al.*, 2020) foi treinado utilizando como base o modelo Mobilenet V2 desenvolvido pela equipe do Google (SANDLER *et al.*, 2018a). Sua arquitetura é composta por uma rede de detecção de disparo único (SSD) e acrescenta um extrator de características de redes de pirâmide de recursos (FPN) que tem a finalidade acelerar o processo de detecção (IMPULSE, 2022). Desse modo, a combinação de características apresentada pelo MobileNetV2 SSD FPN-Lite justifica a seleção do modelo para implantação em dispositivos que apresentam limitações de recursos. O modelo é disponibilizado no *Hub* do *Tensorflow*[2].

3.2.3 Especialização do modelo

Para especializar o modelo utilizou-se o material disponibilizado na documentação do Tensorflow[3], utilizando também o tutorial apresentando por Nicholas Renotte, que detalha o desenvolvimento de modelos de reconhecimento de placas automotivas[4].

No treinamento das últimas camadas do modelo, foram utilizadas 433 imagens, disponíveis na base (LARXEL, 2020) representando assim 95% das amostras. Os arquivos foram serializados para o formato TFRecord[5], possibilitando o agrupamento das imagens com suas

[2] https://tfhub.dev/tensorflow/ssd_mobilenet_v2/fpnlite_320x320/1.
[3] https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html.
[4] https://youtu.be/0-4p_QgrdbE.
[5] O formato TFRecord é um formato simples para armazenar uma sequência de registros binários.

caixas delimitadoras. Após a finalização do treinamento, o modelo foi congelado no formato *savedmodel*[6], tornando-se útil para compartilhamento ou implantação em Tensorflow Lite. Todos os parâmetros utilizados na etapa de treinamento pode ser encontrado no repositório [7].

3.3 EXPORTAÇÃO DO MODELO PARA TENSORFLOW LITE

A técnica de quantização utilizada para a conversão dos modelos, foi a de pós-processamento, conforme apresentando na Tabela 2, pois para utilizar a técnica de pré-processamento requer-se conhecimento aprofundado da arquitetura do modelo utilizado, o que não faz parte do escopo desse trabalho. Para adequação do modelo final para o Raspberry Pi 3B+, foi utilizado a API do TensorFlow Lite Converter.

Conforme definido na Seção 2.5.2.4, a principal característica do método de quantização de pós treinamento é ajustar os parâmetros representados em 32 bits, para formatos que busquem reduzir o tamanho e/ou a latência dos modelos, ajustando os pesos e as funções de ativações.

A seguir, serão detalhadas as formas de quantizações de faixa dinâmica, float16, inteira completa e somente inteiro, que foram utilizadas para conversão do modelo de detecção de placas automotivas.

3.3.1 Quantização de faixa dinâmica

Sabe-se que as quantizações tem como objetivo de atuar em dois propósitos: na redução do tamanho do modelo e na aceleração da inferência durante a execução. O Tensorflow Lite, anteriormente disponibilizava ambos os métodos separadamente, porém, uniu os métodos como padrão, portanto, a linha de código 6 de otimização padrão está atualmente presente em todos os métodos de quantizações apresentados.

[6] https://www.tensorflow.org/guide/saved_model.
[7] https://github.com/weslleyalmeid/plate_detection_w_tinyml/blob/main/train_model/train_model.ipynb.

Listagem 1 – Método para aplicar melhoria ao tamanho do arquivo e latência na inferência do modelo

```
1 # OPTIMIZE_FOR_SIZE e  OPTIMIZE_FOR_LATENCY estão obsoletos
2 converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
3 converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_LATENCY]
4
5 # Método atual que engloba os dois anteriores
6 converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

Fonte: Autoria própria (2022).

A quantização de faixa dinâmica apresentada Listagem 2, é a função responsável por efetuar ajustes dos pesos dos neurônios do modelo, que por padrão são pontos flutuantes representados em 32 bits e convertê-los para 8 bits, dessa forma, a representação em ponto flutuante é convertida para inteiro.

Listagem 2 – Quantização de faixa dinâmica

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

Fonte: Autoria própria (2022).

3.3.2 Quantização float16

Para a quantização em float16, somente os pesos dos neurônios do modelo são convertidos para números de ponto flutuante de 16 bits, e as operações (e.g. cálculo de função de ativação) continuam com suporte de operações de 32 bits. O script utilizado para efetuar a quantização de float16 é apresentado na Listagem 3. Se comparado o *script* com a quantização dinâmica, a solução tem apenas a inclusão do método responsável por habilitar o tipo de quantização a ser aplicada no modelo *supported_types*, apresentado na linha 6. Nesse caso, torna-se fundamental a compatibilidade do dispositivo de destino com as operações aplicadas nesse exemplo, sendo necessário ter a capacidade de operação com ponto flutuante.

Listagem 3 – Quantização float16

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

# habilitar conversão para float16
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model = converter.convert()
```

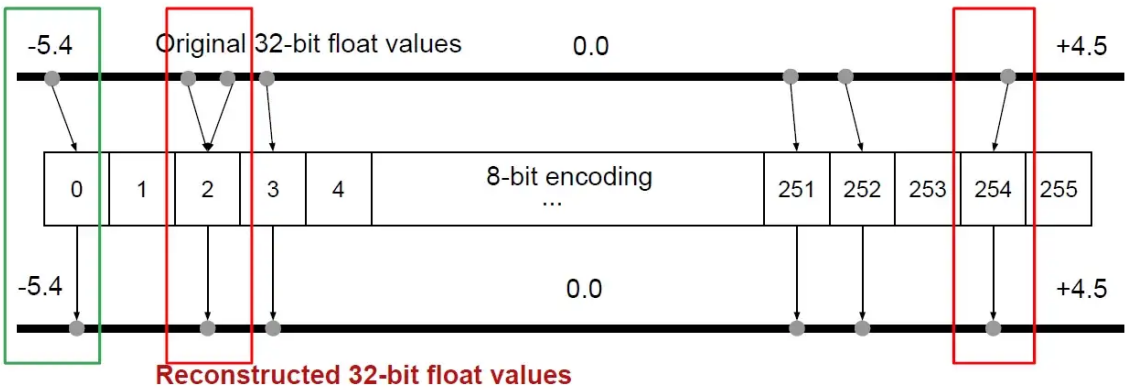
Fonte: Autoria própria (2022).

3.3.3 Quantização inteira completa

A quantização inteira completa possui a função de ajustar todas as operações e parâmetros internos do modelo para serem executadas em inteiro, porém, mantendo a entrada e saída em ponto flutuante. Nesse tipo de quantização é importante considerar as características especificas do sistema embarcado, pois dispositivos que operam em sua totalidade apenas com operações de inteiro, não serão aptos para esse formato de disponibilização.

Para a quantização inteira, a Figura 9 ilustra como são realizadas as calibragens dos tensores, os valores máximos e mínimos representados como ponto flutuante, são enquadrados na dimensão de 8 bits. Todas as operações nas camadas intermediárias são realizadas somente em ponto fixo, ou seja, não necessitam de ajuste aos dados de *input* para efetuar a inferência.

Figura 9 – Representação da quantização para inteiro



Fonte: (BRAUN, 2020).

A implementação é apresentada na Listagem 4 e diferente dos modelos anteriores, é necessário realizar a calibragem dos tensores, dessa forma, repassamos uma amostra de 400 imagens para que os tensores sejam calibradas, as imagens precisam ser representativas conforme

a base utilizada para treinamento, apresentar mesma dimensão da imagem, características de pré-processamento e ajuste de dimensão de *input*. Para calibração, esse trabalho optou-se por utilizar amostras de imagens utilizadas no treinando do modelo.

Listagem 4 – Quantização inteira completa

```
from tensorflow.keras.applications.mobilenet import preprocess_input
import tensorflow as tf
import numpy as np

def representative_dataset():
  dataset_list = tf.data.Dataset.list_files(
    os.path.join(paths['IMAGE_PATH_TFLITE'], 'plate/*')
  )

  for i in range(len(dataset_list)):
    image = next(iter(dataset_list))
    image = tf.io.read_file(image)
    image = tf.io.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, [320, 320])
    image = tf.cast(image / 255., tf.float32)
    image = tf.expand_dims(image, 0)
    yield [image]

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.input_shapes = 1,320,320,3
converter.representative_dataset = representative_dataset
tflite_quant_model = converter.convert()
```

Fonte: Autoria própria (2022).

3.3.4 Quantização somente inteiro

A quantização somente inteiro, converte todas as operações dos modelos, inclusive, entrada e saída para a representação de 8 bits. Esse formato de quantização do modelo é adequado para operações em microcontroladores que não realizam operações com ponto flutuante.

Assim como o modelo inteiro completo 9, também são necessárias as imagens de entrada para calibrar os tensores do modelo, porém, conforme detalhado no Listagem 5, é realizada a inclusão da especificação do tipo de representação das camadas e entrada e saída *inference_input_type* e *inference_output_type* a qual foi convertida para uma representação de 8 bits.

Listagem 5 – Quantização somente inteiro

```
from tensorflow.keras.applications.mobilenet import preprocess_input
import tensorflow as tf
import numpy as np


def representative_dataset():
  dataset_list = tf.data.Dataset.list_files(
    os.path.join(paths['IMAGE_PATH_TFLITE'], 'plate/*')
  )
  for i in range(len(dataset_list)):
    image = next(iter(dataset_list))
    image = tf.io.read_file(image)
    image = tf.io.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, [320, 320])
    image = tf.cast(image / 255., tf.float32)
    image = tf.expand_dims(image, 0)
    yield [image]

  for ind in range(len(test_generator.filenames)):
    img_with_label = test_generator.next()
    yield [np.array(img_with_label[0], dtype=np.float32, ndmin=2)]


converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.input_shapes = 1,320,320,3
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8  # or tf.uint8
converter.inference_output_type = tf.int8  # or tf.uint8
tflite_quant_model = converter.convert()
```

Fonte: Autoria própria (2022).

O modelo de quantização completa necessita que todos os dados passados como entrada para o modelo, sejam convertidos para o formato inteiro, para que seja possível realizar a inferência.

3.4 EXECUTAR MODELO TFLITE NO PYTHON

Após o modelo quantizado, utilizou-se o Listagem 6 desenvolvido na linguagem Python para compreensão do funcionamento da estrutura de inferência de modelos quantizados. Conforme a Seção 2.5.2 o modelo *tflite* requer vínculo de um interpretador diferente, que seja capaz de

operar com esse formato de inferência, dessa forma foi utilizado o interpretador Tensorflow Lite para que assim, seja possível obter as características dos tensores da camada de entrada e saída dos modelos.

Listagem 6 – Carregar modelo Tensorflow Lite no Python

```
import tensorflow as tf
import numpy as np
import os

interpreter = tf.lite.Interpreter(model_path='model.tflite')
interpreter.allocate_tensors()

# Obter características da camada de entrada do modelo
input_details = interpreter_ocr.get_input_details()
print(input_details)

# Obter características da camada de saída do modelo
output_details = interpreter_ocr.get_output_details()
print(output_details)
```

Fonte: Autoria própria (2022).

Após ser verificado as características do modelo, através dos detalhes de entrada e saída, foi consolidado a Tabela 3 com os índices da camada de saída dos tensores, que apresentam as informações resultantes de uma operação de detecção.

Tabela 3 – Características da saída do modelo

Índice	Nome	Descrição
0	Scores	Matriz de N valores de ponto flutuante entre 0 e 1 representando a probabilidade de que uma classe foi detectada
1	Localização	Matriz multidimensional de [N][4] valores de ponto flutuante entre 0 e 1, as matrizes internas representando caixas delimitadoras na forma [superior, esquerda, inferior, direita]
2	Número de detecção	Número de detecções Valor inteiro de N
3	Classes	Matriz de N inteiros (saída como valores de ponto flutuante), cada um indicando o índice de um rótulo de classe do arquivo de rótulos

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

Na sequência, após a compreensão de todas as características do modelo, foi desenvolvido o Algoritmo 1, que apresenta a forma de realizar a inferência dos modelos nas imagens. Observa-se que o fluxo entre todos os modelos tem um comportamento similar, os modelos basicamente necessitam que as imagens sejam repassadas para a camada de entrada, através do *set_input_tensor* ou *interpreter.set_tensor*. Após definidos os tensores da camada de entrada com as imagens, o modelo irá realizar a inferência utilizando o método *invoke()* do objeto *interpreter*, com

a inferência executada, precisamos acessar as camadas de saídas do modelo para obter as informações do resultado.

O tensor em um modelo de rede neural tem seu índice numerado no contexto global e local. Sendo que, global significa em qual posição o tensor se encontra quando comparado a todos os tensores presentes no modelo. Enquanto o modelo local, refere-se a posição do tensor apenas na sua camada. A Tabela 3 apresenta os índices dos tensores para obter os resultados da predição.

Ressalta-se a personalização do modelo *integer_only*, por executar todas as operações somente com inteiros, é preciso ajustar a camada de entrada para o formato inteiro e também ajustar a imagem para os limites da calibragem já ajustada do modelo.

Algoritmo 1 – Modelo de inferência no Python

```
import tensorflow as tf
import numpy as np

def set_input_tensor(interpreter, image):
    """Inserir imagem na camada de entrada do tensor"""
    tensor_index = interpreter.get_input_details()[0]['index']
    input_tensor = interpreter.tensor(tensor_index)()[0]
    input_tensor[:, :] = np.expand_dims((image-255)/255, axis=0)

def get_output_tensor(interpreter, index):
    """Obter resultado de cada tensor"""

    # obter índice global do modelo, baseado em seu índice na camada de saída
    output_details = interpreter.get_output_details()[index]

    # obter resultado da infererência, baseado no índice global do modelo
    tensor = np.squeeze(interpreter.get_tensor(output_details['index']))
    return tensor

def detect_objects(interpreter, image, threshold, modelo):
    if  modelo == 'integer_only':
        input_details = interpreter.get_input_details()[0]
        input_scale, input_zero_point = input_details["quantization"]
        image = (image / input_scale) + input_zero_point
        image = np.expand_dims(image, axis=0).astype(input_details["dtype"])
        interpreter.set_tensor(input_details["index"], image)
    else:
        set_input_tensor(interpreter, image)

    # tempo de inferência
    start = time.process_time()
    interpreter.invoke()
    end = time.process_time() - start

    boxes = get_output_tensor(interpreter, 1)
    classes = get_output_tensor(interpreter, 3)
    scores = get_output_tensor(interpreter, 0)

    # obter o valor do maior score
    score = scores[scores.argmax()]
    count = int(get_output_tensor(interpreter, 2))

    # Converter resultado inteiro para float
    if 'integer_only' in modelo:
        output_scale, output_zero_point =
        ↪  interpreter.get_output_details()[0]['quantization']
        score = (score - output_zero_point) * output_scale

    return score, end

detection_threshold = 0.7
detect_objects(interpreter, img, detection_threshold, my_func=True)
```

3.5 PREPARAR AMBIENTE NO RASPBERRY PI

Após os modelos quantizados, faz-se necessário a escolha do dispositivo a ser implantado a solução. Ressalta-se, que inicialmente esperava-se que os modelos de detecção pudessem ser disponibilizados no microcontrolador da família *STM32F746 Discovery*, porém, após o menor modelo apresentar 3MB de armazenamento, inviabilizou-se a disponibilização no STM32F746, pois, o microcontrolador apresenta a capacidade de armazenamento de somente 1MB, enquanto o Raspberry Pi Zero 2W apresenta 512MB de armazenamento, assim este foi escolhido.

Dessa forma, buscando implantar o modelo treinado no Raspberry Pi Zero 2W, deve-se preparar o ambiente de desenvolvimento para a disponibilização da solução. Essa etapa de preparação foi realizada no Raspberry Pi 3B+ e no Raspberry Pi 4B.

3.5.1 Underclock no Raspberry Pi 3B+

O Raspberry Pi 3B+ possui características similares as do Raspberry Pi Zero 2W, ambos apresentam processador Quad-core de 64 bits ARM Cortex-A53 Quad-Core, porém, diferem no quesito taxa de clock de processamento. O clock do Raspberry Pi 3B+ no entanto é de 1.4GHz e o do Raspberry Pi Zero 2W é de 1GHz. Dessa forma, para simular a solução de um ambiente que possa representar de forma aproximada o sistema de destino final, foi necessário ajustar o clock de processamento do Raspberry Pi 3B+ para o limite de 1GHz. Abaixo, será apresentado o arquivo para ajustar a alteração do arm_freq para 1 GHz.

3.5.2 Tensorflow Lite

Com a finalidade de utilizar o Tensorflow Lite, é necessário realizar o *download* dos arquivos necessários diretamente da API do Tensorflow Lite, para que os modelos possam executar em compatibilidade com o dispositivo. A instalação do tensorflow lite foi baseada na documentação[8] do Tensorflow, e ela é disponibilizada em duas formas de instalação, utilizando Cmake ou Bazel. No desenvolvimento do presente trabalho optou-se por utilizar o instalador Bazel, pois, o material apresentado por Koen Vervloesem[9] demonstra como realizar a construção do

[8] https://www.tensorflow.org/lite/guide/build_arm.
[9] https://github.com/koenvervloesem/bazel-on-arm.

modelo Tensorflow Lite utilizando esse instalador. Optou-se por salvar o conjunto de bibliotecas no diretório *./usr/local/lib/* para facilitar o vínculo ao compilar o programa. Outras formas de instalação podem ser encontradas no repositório do github do prepkg[10] e Q-engineering[11].

3.5.3 OpenCV

Com o objetivo de trabalhar com imagens em C++, escolheu-se a biblioteca OpenCV, que conta com um conjunto de ferramentas para manuseio de imagens (eg.: ajuste de tamanho de imagem, alteração de cores, seleção de região de interesse). Para instalação foi utilizado o material do Q-engineering[12].

3.5.4 Tesseract

Com a finalidade de trabalhar com o reconhecimento dos caracteres ópticos (OCR) das placas automotivas, primeiramente foi planejado utilizar o modelo Keras-OCR, que seria quantizado conforme o modelo de detecção e feita a disponibilização no dispositivo, porém, ao utilizar o modelo em C++, foi identificou-se inconsistências de operação[13], pois, nem todas as operações do Tensorflow são disponibilizadas pelo Tensorflow Lite. A implementação de uma solução OCR não seria viável, levando em consideração que a mesma a necessidade de amostras significativas e demandaria um tempo não planejado.

Dessa forma, para não descartar a etapa de OCR e viabilizar o reconhecimento das placas automotivas, optou-se por realizar o reconhecimento do caracteres ópticos das placa automotivas, utilizando-se da API do Tesseract, ferramenta Open Source e disponível para as linguagens C++ e Python.

O Tesseract apresenta um conjunto de ferramentas para o manuseio e reconhecimento de caracteres e o site oficial[14] detalha diferentes formas de instalações, mas, por a execução estar sendo feita em um Raspberry Pi com um sistema operacional baseado em unix, optou-se por instalar via gerenciador de pacote padrão do sistema.

[10] https://github.com/prepkg/tensorflow-lite-raspberrypi.
[11] https://qengineering.eu/install-tensorflow-2-lite-on-raspberry-64-os.html.
[12] https://qengineering.eu/install-opencv-lite-on-raspberry-pi.html.
[13] https://www.tensorflow.org/lite/guide/ops_compatibility.
[14] https://tesseract-ocr.github.io/tessdoc/Compiling.html.

```
sudo apt install tesseract-ocr
```

3.6 EXECUTAR MODELO TFLITE EM C++

Ao capturar o vídeo em tempo real no Raspberry, a rede neural pretende buscar e efetuar a detecção de uma placa automotiva, e, ao alcançar um índice satisfatório de 70% de confiabilidade, o algoritmo avançará para a etapa de extração de caracteres da imagem. Fundamentado nas instruções que a documentação do Tensorflow Lite disponibiliza, será especificado abaixo as fases necessárias para executar um modelo Tensorflow Lite em C++.

3.6.1 Carregar o modelo em C++

Buscando viabilizar a utilização do modelo, foi necessário realizar a carga do modelo tflite em memória. O modelo Tensorflow Lite realiza alocação dinâmica, dessa forma, basta especificar o diretório e nome do modelo.

Listagem 7 – Carregar o modelo

```
// carregar modelo tflite
std::unique_ptr<tflite::FlatBufferModel> model =
↪  tflite::FlatBufferModel::BuildFromFile(modelFileName);
```

Fonte: Autoria própria (2022).

3.6.2 Construir o interpretador em C++

O interpretador é fundamental para a execução do modelo, pois ele é responsável por possibilitar a conexão entre a imagem e o modelo de detecção. Dessa forma, para construir o interpretador faz-se necessário instanciar o objeto da classe *resolver*. Conforme apresentado no Listagem 8, o *resolver* responsabiliza-se pela intermediar as operações disponibilizadas pelo Tensorflow Lite ao modelo de detecção de placas automotivas, para assim, tornar-se possível realizar a inferência.

Listagem 8 – Construir interpretador do modelo

```
// conectar interpretador com operações suportadas pelo Tensorflow Lite
tflite::ops::builtin::BuiltinOpResolver resolver;
std::unique_ptr<tflite::Interpreter> interpreter;
tflite::InterpreterBuilder(*model.get(), resolver)(&interpreter);
if (interpreter == nullptr)
{
    fprintf(stderr, "Failed to initiate the interpreter\n");
    exit(-1);
}

// Atualiza as alocações de memória para todos os tensores
if (interpreter->AllocateTensors() != kTfLiteOk)
{
    fprintf(stderr, "Failed to allocate tensor\n");
    exit(-1);
}
```

Fonte: Autoria própria (2022).

3.6.3 Realizar inferência em C++

Nessa fase, para realizar a inferência, faz-se necessário a obtenção dos detalhes das características de entrada do modelo. Conforme especificado na Seção 3.2.2, o modelo foi treinado baseado nas características do MobileNetV2, assim, necessita-se ajustar a dimensão da imagem de captura, para que esteja de acordo com a camada de entrada do modelo de detecção. O Listagem 9 detalha a etapa de ajuste de dimensão da imagem e detecção do modelo.

Listagem 9 – Realizar inferência

```
// ajustar a dimensão da imagem para 320x320
cv::resize(image, image, cv::Size(width, height), cv::INTER_NEAREST);

// obter ponteiro para início da imagem
float* inputImg_ptr = image.ptr<float>(0);

// obter ponteiro para endereço inicial do tensor de entrada
float* inputLayer = interpreter->typed_input_tensor<float>(0);

// setar tensores com a imagem
memcpy(inputLayer, inputImg_ptr, width * height * channels * sizeof(float));

// executar inferência
interpreter->Invoke();
```

Fonte: Autoria própria (2022).

3.6.4 Obter resultados do modelo em C++

A identificação do valor de representação do tensor da camada de saída do modelo é necessária para obter os resultados da detecção, e, para efetuar a identificação utilizou-se o Listagem 6, que apresenta as características do modelo. Para validação dos índices apresentados naTabela 3, recorreu-se a ferramenta de visualização de arquiteturas de modelos de redes neurais *Netron*[15].

O Listagem 10 apresenta as formas de de acesso aos tensores da camada de saída, e descreve as formas de acesso das características de interesse de detecção. Nesse contexto, os *output_scores* representam os índices de confiabilidade na identificação da placa e os *output_boxes* apresentam as regiões de interesse capturadas. Na *output_classes* é onde a detecção está categorizada e a *output_counts* faz o armazenamento das quantidades de capturas que foram realizadas.

[15] https://netron.app/.

Listagem 10 – Obtendo as características da detecção

```
// Obter posição de identificação do tensor
auto scores_tensor = interpreter->outputs()[0];

// Obter posição de identificação do tensor
TfLiteTensor *output_score = interpreter->tensor(scores_tensor);

// Índice de confiabilidade da detecção
auto output_scores = output_score->data.f;

// Região de interesse em torno da placa identificada
// Obter posição de identificação do tensor
auto box_tensor = interpreter->outputs()[1];
TfLiteTensor *output_box = interpreter->tensor(box_tensor);
auto output_boxes = output_box->data.f;
std::vector<float> boxes;

// Obter posição de identificação do tensor
auto classes_tensor = interpreter->outputs()[3];
TfLiteTensor *output_class = interpreter->tensor(classes_tensor);
// Pertencete a categoria placa ou não
auto output_classes = output_class->data.f;
std::vector<float> classes;


// Obter posição de identificação do tensor
auto count_tensor = interpreter->outputs()[2];
TfLiteTensor *output_count = interpreter->tensor(count_tensor);
auto output_counts = output_count->data.f;
```

Fonte: Autoria própria (2022).

3.6.5 Compilação e Execução

Com relação a compilação do modelo, utilizou-se o compilador *G++* para efetuar a especificação de todas as bibliotecas em seu comando, para assim, tornar-se possível a criação do executável. Abaixo apresenta-se o comando de compilação.

```
g++ main.cc -o ./build/platerun `pkg-config --cflags --libs opencv4`
↪ -I/usr/local/include -ltesseract -ltensorflow-lite
```

3.7 DETECÇÃO DE PLACA EM TEMPO REAL

Objetivando consolidar todas as etapas apresentadas anteriormente, o Algoritmo 2 apresenta um pseudo-código de execução do modelo de detecção e reconhecimento de placas em tempo real. A linha 1 apresenta o carregamento dos modelos quantizados conforme detalhados na Seção 3.3, todos os tratamentos de imagem apresentados na linha 2, 4, 5 são executados com o OpenCV2, para o reconhecimento de caracteres utilizou-se o Tesseract, detalhado na Seção 2.2.2 . Os *scripts* consolidados para a execução em tempo real nas linguagens Python e C++, podem ser acessados no repositório do *github* [16] e todos os detalhes da implementação em C++ são apresentado no vídeo [17].

Algoritmo 2 – Pseudo código em tempo real

```
1  interpreter = modelo.tflite
2  cam = conectarWebcam
3  enquanto True faz
4      frame = obterFrame(cam)
5      frame = normalizarImagem(frame)
6      resultado = detectarPlaca(interpreter, frame)
7      se resultado['score'] ≥ 0.7 então
8          roi = result['box']
9          numeroPlaca = ocr(roi)
10         mostrar(frame, numeroPlaca)
11     senão
12         mostrar(frame)
13     finaliza se
14 finaliza enquanto
```

Fonte: Autoria própria (2022).

[16] https://github.com/weslleyalmeid/plate_detection_w_tinyml.
[17] https://youtu.be/fLbp8kD0ng4.

4 RESULTADOS EXPERIMENTAIS

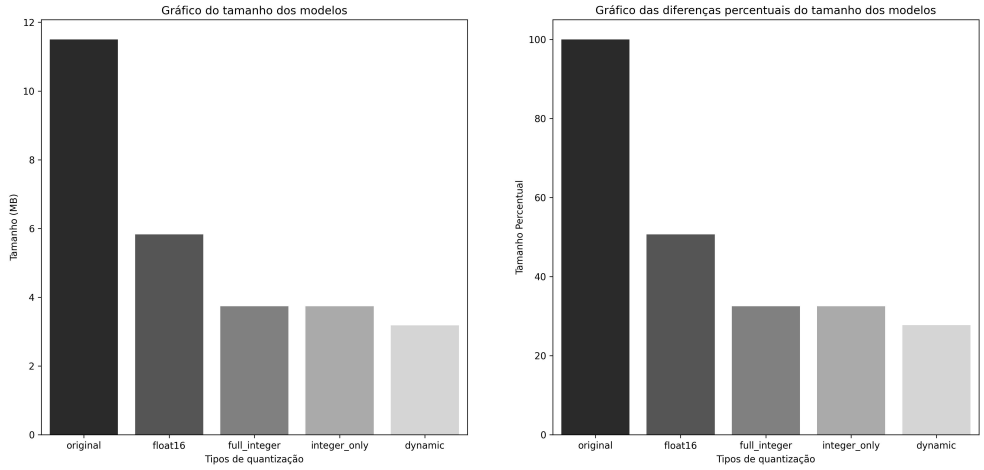
Neste capítulo serão apresentados e discutidos os resultados experimentais, testando a quantização dos modelos descritos no Seção 3.3, contendo detalhes comparativos do tempo de latência para inferência dos modelos executados nas linguagens Python e C++. Ressalta-se que os experimentos comparativos entre a linguagens Python e C++ foram executados em um dispositivo Raspberry Pi 4B, evitando a interferência de processos externos e criando um ambiente minimamente controlado.

4.1 TAMANHO DOS MODELOS QUANTIZADOS

Após a conversão dos modelos detalhados na Seção 3.3 para o formato Tensorflow Lite e quantizados na categoria de pós-treinamento, a Figura 10 ilustra os tamanhos finalizados do modelo. Identifica-se que o modelo original apresentou 11MB, na sequência, o modelo float16 reduziu para 5,6MB e no caso dos modelos de quantização inteira e somente inteira, esses obtiveram a mesma redução que ficou 3,6MB, enquanto o modelo de quantização dinâmica se destacou obtendo a melhor redução de tamanho, indo para 3MB.

A Tabela 2 mostra que os resultados dos modelos quantizados foram satisfatórios perante o que a documentação do Tensorflow Lite expõe. Dessa forma, é possível concluir que os modelos se aproximam da conversão retratada por eles.

Figura 10 – Representação em megabites e percentual do modelo original e quantizados



Fonte: Autoria própria (2022).

4.2 LATÊNCIA DOS MODELOS PYTHON E C++

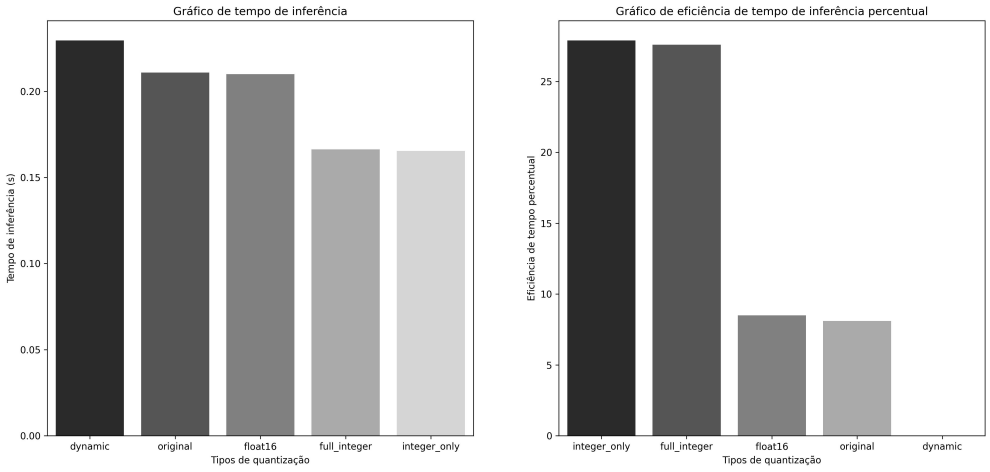
As linguagens C++ e Python possuem métodos de disponibilização diferentes, a linguagem C++ utiliza o método de compilação, enquanto o Python utiliza o método interpretador. Através do trabalho de Zehra *et al.* (2020) realiza-se comparativos entre o tempo de execução das linguagens e conclui-se que a linguagem C++ é superior a linguagem Python.

Dessa forma, buscando apresentar um comparativo do tempo de execução entre Python e C++ para execução dos modelo, efetuou-se a simulação dos modelos no dispositivo Raspberry Pi 4B. Para coletar os resultados, considera-se apenas o tempo de inferência do modelo, portanto, desconsidera-se fatores de ajustes de imagens e padronizações para disponibilização do resultado.

Os testes foram realizados efetuando a detecção em 50 imagens retiradas aleatoriamente da base de dados da (MENDES JÚNIOR *et al.*, 2011) e (ANAGNOSTOPOULOS *et al.*, 2008) já especificadas na Seção 3.2.1 e efetuando a inferência do modelo não quantizado (original) e os quantizados, utilizando a API do Tensorflow Lite.

Conforme apresentado na Figura 11, o modelo que apresentou o melhor tempo de execução foi o de quantização somente inteiro, esse resultado é condizente com o esperado, pelo fato do modelo trabalhar apenas com operações matemáticas sem a presença de ponto flutuante. Esse fato, corrobora para que a execução seja mais rápida. Em contraponto, o modelo de quantização dinâmica apresentou o pior tempo de execução, ficando atrás do modelo original.

Figura 11 – Representação do tempo e eficiência percentual do modelo original e quantizados executados no Python

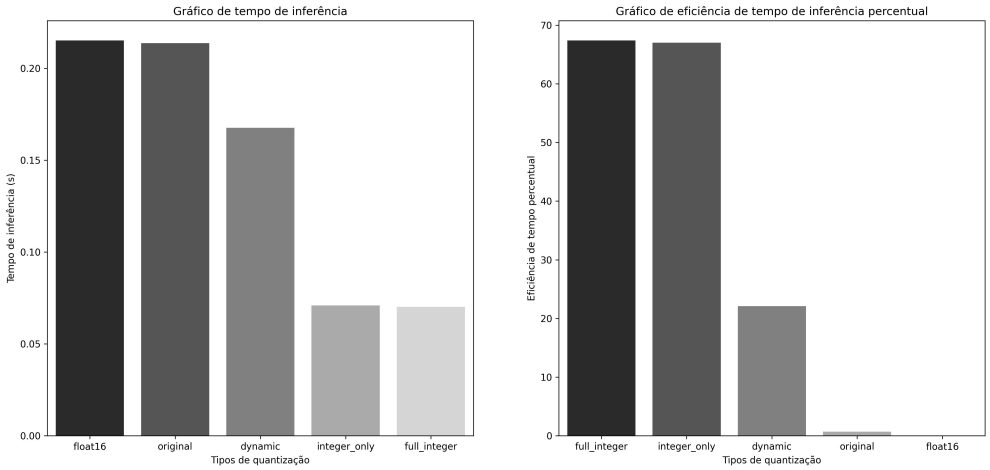


Fonte: Autoria própria (2022).

A Figura 12 ilustra o tempo de execução entre os modelos para a linguagem C++. Nesse

contexto, o modelo com menor latência é o modelo de quantização inteira completa (full_integer), e o modelo com pior tempo de execução foi o float16. Ressalta-se que apesar de ilustrado na imagem o modelo quantizado somente inteiro (only_integer) não obteve o desempenho real mensurado, pois, o mesmo apresentou erro na tentativa de inferência no modelo C++.

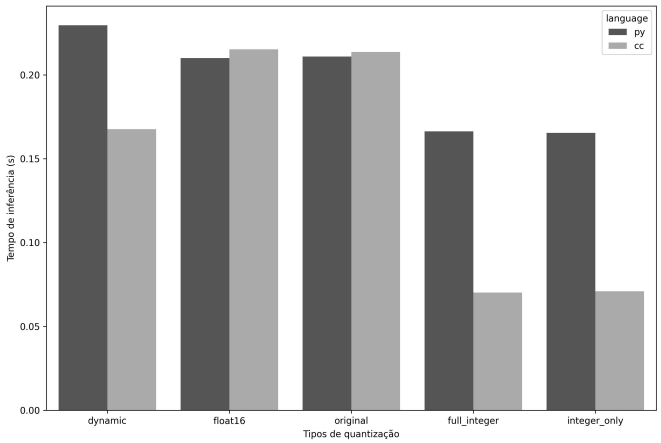
Figura 12 – Representação do tempo e eficiência percentual dos modelos quantizados executados no C++



Fonte: Autoria própria (2022).

Observa-se na Figura 13 que o modelo de quantização inteira completa (full_integer) e o de conversão dinâmica (dynamic) apresentaram diferenças significativas ao executar em C++. Constata-se que o tempo de inferência dos modelos executados na linguagem C++, não foram predominantemente superiores as execuções dos modelos na linguagem Python. A hipótese para não ocorrer uma diferença predominante do C++ é motivado ao fato de que as operações matemáticas utilizadas pela linguagem Python, costumam ser realizadas por meio de bibliotecas, que por sua vez, tendem a efetuar as operações matemáticas na linguagem C.

Figura 13 – Gráfico comparativo de latência entre Python e C++



Fonte: Autoria própria (2022).

4.3 PRECISÃO DOS MODELOS PYTHON E C++

Os experimentos simulados e executados no Raspberry Pi 4B, consideraram alguns fatores para a consolidação das métricas de desempenho, como as diferentes formas de quantização e o modelo original.

A Tabela 4 apresenta o comportamento dos modelos e a partir desse resultado, é possível observar que o modelo de conversão de faixa dinâmica detalhado na Seção 3.3.1 apresentou melhor score se comparado aos demais modelos. Já o modelo quantizado completamente para inteiro, apresentou uma diferença percentual de 20% se comparado aos demais modelos. Durante a inferência dos modelos quantizados inteiro completo (full_integer) e somente inteiro (integer_only) no C++ o modelo somente inteiro (integer_only) apresentou falha de segmentação para alocação da imagem para o tensor da camada de entrada do modelo de detecção. Já o modelo inteiro completo (full_integer) apresentou um *score* abaixo do esperado, pois todos os modelos deveriam ter score próximos, pois só diferem quanto a linguagem de execução.

Tabela 4 – Precisão média dos modelos

Modelos	Score C++	Score Python
dynamic	**72,38%**	**72,46%**
float16	72,24%	72,31%
original	72,28%	72,28%
full_integer	6,64%	72,27%
integer_only	*error*	52,08%

Fonte: Autoria própria.

4.4 EXEMPLO DE DETECÇÃO ENTRE PYTHON E C++

Conforme ilustrado na Figura 14, utilizando o modelo quantizado em float16, nota-se que visualmente a detecção da placa automotiva ocorreu com sucesso. Considerando que a região de interesse foi identificada e ambas apresentam todos os caracteres de interesse, foi possível realizar o reconhecimento pelo OCR. Para assistir a demonstração da solução executada em tempo real acesse link-youtube.

Figura 14 – Exemplo de detecção entre linguagem Python e C++

(a) Detecção Python **(b) Detecção C++**

(c) Região de interesse Python **(d) Região de interesse C++**

Fonte: Autoria própria (2022).

4.5 OCUPAÇÃO DE MEMÓRIA E CPU DA SOLUÇÃO C++

Essa seção tem a finalidade de mensurar o consumo médio da solução completa de detecção e reconhecimento de placa automotiva em C++, disponibilizada no Raspberry Pi 3B+ com underclock para 1GHz, simulando assim, o Raspberry Pi Zero 2W.

Pretendendo calcular o consumo médio de memória em tempo execução, realizaram-se 60 experimentos, desses, 30 experimentos utilizavam somente o OpenCV e o restante dos experimentos foram executados o OpenCV, Tensorflow Lite e o Tesseract. O consumo médio exigido pelo OpenCV foi de 23MB, dessa forma, foi possível calcular que a solução utilizando o modelo quantizado float16 5.3MB, apresentou o consumo médio de 45MB. Destaca-se, que ao efetuar a inferência, a memória não apresentava nenhum pico de consumo.

5 CONCLUSÃO

É perceptível o crescimento e avanço das aplicações que envolvem o aprendizado de máquina e sistemas embarcados. Considerando esse fato, resultados de estudos como o deste trabalho, motivam e estimulam a continuação da exploração desses campos. Outro fator importante que torna-o relevante, é por se tratar de uma área com fundamentações multidisciplinares que abrange diferentes setores e ramos, pois, isso possibilita uma perspectiva positiva considerável de crescimento.

A seção dos resultados experimentais comprovaram que os modelos de quantizações desempenharam de forma adequada a redução do tamanho do arquivo conforme proposto pelo Tensorflow Lite, e as características do dispositivo de destino são importantes para mensurar o tempo de latência na etapa de inferência. Observa-se que foi possível utilizar todos os modelos quantizados na linguagem Python. No entanto, na linguagem C++, não foi possível executar a inferência no modelo de somente inteiro, dessa forma, os valores de latência e score devem ser ignorados. O score de confiabilidade de detecção do modelo não foi afetado de forma significativa, apenas os modelos quantizados para inteiro tiveram uma diferença expressiva na execução. Conforme a Seção 4.4, a comparação entre as detecções C++ e Python em uma mesma figura, selecionam a região de interesse de forma similar.

Portanto, a partir do trabalho desenvolvido, foi possível realizar a detecção de placas automotivas em um sistema embarcado Raspberry Pi Zero 2W, utilizando-se da abordagem tinyML e o framework do TensorFlow Lite para quantização do modelo MobileNetV2-SSDLite.

6 TRABALHOS FUTUROS

Por fim, espera-se que este trabalho sirva de suporte para que outras pessoas usem tanto da abordagem, quanto do modelo, para alavancar o desenvolvimento de soluções de aprendizado de máquina em dispositivos com limitações de recursos. O método TinyML e o campo de aprendizado de máquina, encontram-se em um processo atual de desenvolvimento, visto que tratam-se de ferramentas e abordagens recentes. Por esse motivo, trabalhos de otimização estão acontecendo e outros estudos podem ser desenvolvidos visando as novas metodologias de classificação cada vez mais sofisticadas, com novos e mais complexos algoritmos, que podem, possivelmente, retornar resultados mais acurados, com implementações de custo ainda mais reduzidos. Dessa forma, abaixo será listado opções de trabalhos futuros.

- Implantar uma solução tinyML em um microcontrolador (e.g. Detecção de anomalia).
- Treinar modelo de aprendizado de máquina, ciente de quantização.
- Combinar métodos de otimização de poda e quantização, para redução do tamanho modelo de aprendizado de máquina.
- Pesquisa sobre os diferentes métodos de quantizações e aplicabilidade mediante o problema proposto.

REFERÊNCIAS

ABADI, M. *et al.* TensorFlow: A System for Large-Scale Machine Learning. In: 12TH USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association, nov. 2016. P. 265–283. ISBN 978-1-931971-33-1. Disponível em: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

ALEXEY, B.; WANG, C.; MARK LIAO, H. Optimal speed and accuracy of object detection. **arXiv preprint arXiv:2004.10934**, 2020.

ALSOP, T. **TinyML Sees Big Hopes for Small AI**. [*S. l.: s. n.*], 2022. Disponível em: https://www.statista.com/statistics/935382/worldwide-microcontroller-unit-shipments/. Acesso em: 6 nov. 2022.

ALZUBAIDI, L. *et al.* Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. **Journal of big Data**, Springer, v. 8, n. 1, p. 1–74, 2021.

ANAGNOSTOPOULOS, C.-N. E. *et al.* License Plate Recognition From Still Images and Video Sequences: A Survey. **IEEE Transactions on Intelligent Transportation Systems**, v. 9, n. 3, p. 377–391, 2008. DOI: 10.1109/TITS.2008.922938.

ARTIBA. **TinyML: The Future of Machine Learning**. [*S. l.: s. n.*], 2022. Disponível em: https://www.artiba.org/blog/tinyml-the-future-of-machine-learning. Acesso em: 6 nov. 2022.

BANBURY, C. *et al.* MLPerf Tiny Benchmark. **arXiv preprint arXiv:2106.07597**, 2021.

BHARDWAJ, A. *et al.* A Novel Approach to Recognize Optical Characters of Number Plate Using Object Detection. In: SINGH, P. K. *et al.* (Ed.). **Futuristic Trends in Networks and Computing Technologies**. Singapore: Springer Nature Singapore, 2022. P. 851–863. ISBN 978-981-19-5037-7.

BISHOP, C. M. Neural networks and their applications. **Review of scientific instruments**, American Institute of Physics, v. 65, n. 6, p. 1803–1832, 1994.

BRAGA, A. d. P.; LUDERMIR, T. B.; CARVALHO, A. C. P. d. L. F. **Redes neurais artificiais: teoria e aplicações**. [*S. l.*]: LTC, 2000.

BRAUN, A. **Quantization Tutorial in TensorFlow to optimize an ML model like a pro**. [*S. l.: s. n.*], 2020. Disponível em: https://medium.com/codex/quantization-tutorial-in-tensorflow-to-optimize-a-ml-model-like-a-pro-cadf811482d9. Acesso em: 1 dez. 2022.

BROWNLEE, J. **Deep learning with Python: develop deep learning models on Theano and TensorFlow using Keras**. [*S. l.*]: Machine Learning Mastery, 2016.

CHASE, O.; ALMEIDA, F. Sistemas embarcados. **Mídia Eletrônica. Página na internet:< www. sbajovem. org/chase>, capturado em**, v. 10, n. 11, p. 13, 2007.

CHAUDHURI, A. *et al.* Optical character recognition systems. In: OPTICAL Character Recognition Systems for Different Languages with Soft Computing. [*S. l.*]: Springer, 2017. P. 9–22.

CHIU, Y.-C. *et al.* Mobilenet-SSDv2: An Improved Object Detection Model for Embedded Systems. In: 2020 International Conference on System Science and Engineering (ICSSE). [*S. l.: s. n.*], 2020. P. 1–5. DOI: 10.1109/ICSSE50014.2020.9219319.

DAI, W.; ZHOU, L. TinyML-Enabled Static Hand Gesture Recognition System Based on an Ultra-Low Resolution Infrared Array Sensor and a Low-Cost AI Chip. In: 2022 5th International Conference on Pattern Recognition and Artificial Intelligence (PRAI). [*S. l.: s. n.*], 2022. P. 804–809. DOI: 10.1109/PRAI55851.2022.9904109.

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. l.: s. n.*], 2021. v. 3, p. 800–811. Disponível em: https://proceedings.mlsys.org/paper/2021/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

DENARDIN, G. W.; BARRIQUELLO, C. H. **Sistemas operacionais de tempo real e sua aplicaçao em sistemas embarcados**. [*S. l.*]: Editora Blucher, 2019.

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

ERHAN, D. *et al.* The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. In: DYK, D. van; WELLING, M. (Ed.). **Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics**. Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA: PMLR, abr. 2009. v. 5. (Proceedings of Machine Learning Research), p. 153–160. Disponível em: https://proceedings.mlr.press/v5/erhan09a.html.

FERNEDA, E. Redes neurais e sua aplicação em sistemas de recuperação de informação. **Ciência da Informação**, SciELO Brasil, v. 35, p. 25–30, 2006.

FORSYTH, D.; PONCE, J. **Computer vision: A modern approach.** [*S. l.*]: Prentice hall, 2011.

GARCIA LOPEZ, P. *et al.* **Edge-centric computing: Vision and challenges**. v. 45. [*S. l.*]: ACM New York, NY, USA, 2015. P. 37–42.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [*S. l.*]: MIT press, 2016.

HAGAN, M. T.; DEMUTH, H. B.; BEALE, M. **Neural network design**. [*S. l.*]: PWS Publishing Co., 1997.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv preprint arXiv:1510.00149**, 2015.

IMPULSE, E. **MobileNetV2 SSD FPN**. [*S. l.: s. n.*], 2022. Disponível em: https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/mobilenetv2-ssd-fpn. Acesso em: 6 nov. 2022.

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

JACOB, B. *et al.* Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: PROCEEDINGS of the IEEE conference on computer vision and pattern recognition. [*S. l.: s. n.*], 2018. P. 2704–2713.

KHINE, H. M. **Automated Yard Processes using TinyML**. [*S. l.: s. n.*], 2020. Disponível em: https://blogs.sap.com/2022/09/29/automated-yard-check-in-processes-using-tinyml-technical-dive-on-ml-model-part-2-of-2/. Acesso em: 29 set. 2022.

LAMBERTI, L. *et al.* Low-Power License Plate Detection and Recognition on a RISC-V Multi-Core MCU-Based Vision System. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). [*S. l.: s. n.*], 2021. P. 1–5. DOI: 10.1109/ISCAS51556.2021.9401730.

LARXEL. **Car License Plate Detection**. [*S. l.: s. n.*], 2020. Disponível em: https://www.kaggle.com/datasets/andrewmvd/car-plate-detection.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **nature**, Nature Publishing Group, v. 521, n. 7553, p. 436–444, 2015.

MARTÍN ABADI *et al.* **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. [*S. l.: s. n.*], 2015. Software available from tensorflow.org. Disponível em: https://www.tensorflow.org/.

MARWEDEL, P. **Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things**. [*S. l.*]: Springer Nature, 2021.

MATHWORKS. **How CNNs Work**. [*S. l.: s. n.*], 2020. Disponível em: %5Curl%7Bhttps://www.mathworks.com/discovery/convolutional-neural-network-matlab.html%7D. Acesso em: 29 set. 2022.

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

MENDES JÚNIOR, P. R. *et al.* Towards an automatic vehicle access control system: License plate location. In: IEEE International Conference on Systems, Man, and Cybernetics. Anchorage, AK, USA: [*s. n.*], out. 2011. P. 2916–2921. DOI: 10.1109/ICSMC.2011.6084108. Disponível em: https://pedrormjunior.github.io/dataset-VLPL.html.

MITTAL, R.; GARG, A. Text extraction using OCR: A Systematic Review. In: 2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA). [*S. l.: s. n.*], 2020. P. 357–362. DOI: 10.1109/ICIRCA48905.2020.9183326.

MORI, S.; NISHIDA, H.; YAMADA, H. **Optical character recognition**. [*S. l.*]: John Wiley & Sons, Inc., 1999.

NEGNEVITSKY, M. **Artificial intelligence: a guide to intelligent systems**. [*S. l.*]: Pearson education, 2005.

NICOLAS, C.; NAILA, B.; AMAR, R.-C. TinyML Smart Sensor for Energy Saving in Internet of Things Precision Agriculture platform. In: 2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN). [*S. l.: s. n.*], 2022. P. 256–259. DOI: 10.1109/ICUFN55119.2022.9829675.

NICOLAU, V. N. Mapeamento de cobertura e uso de solo a partir de dados de sensoriamento remoto utilizando redes neurais. Repositório institucional da UTFPR, 2020.

PEREIRA, L. A. M. *et al.* Software embarcado, o crescimento e as novas tendências deste mercado. **Revista de Ciências Exatas e Tecnologia**, v. 6, n. 6, p. 85–94, 2011.

RASCHKA, S. **Python machine learning**. [*S. l.*]: Packt publishing ltd, 2015.

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

ROSHAN, A. N. *et al.* Adaptive Traffic Control With TinyML. In: 2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET). [*S. l.: s. n.*], 2021. P. 451–455. DOI: 10.1109/WiSPNET51692.2021.9419472.

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. DOI: 10.1147/rd.33.0210.

SANCHEZ-IBORRA, R.; SKARMETA, A. F. Tinyml-enabled frugal smart objects: Challenges and opportunities. **IEEE Circuits and Systems Magazine**, IEEE, v. 20, n. 3, p. 4–18, 2020.

SANDLER, M. *et al.* MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv, 2018. DOI: 10.48550/ARXIV.1801.04381. Disponível em: https://arxiv.org/abs/1801.04381.

SANDLER, M. *et al.* Mobilenetv2: Inverted residuals and linear bottlenecks. In: PROCEEDINGS of the IEEE conference on computer vision and pattern recognition. [*S. l.: s. n.*], 2018. P. 4510–4520.

SARKER, I. H. Machine learning: Algorithms, real-world applications and research directions. **SN Computer Science**, Springer, v. 2, n. 3, p. 1–21, 2021.

SHAKEEL, M. F. *et al.* Detecting driver drowsiness in real time through deep learning based object detection. In: SPRINGER. INTERNATIONAL Work-Conference on Artificial Neural Networks. [*S. l.: s. n.*], 2019. P. 283–296.

SZELISKI, R. **Computer vision: algorithms and applications**. [*S. l.*]: Springer Nature, 2022.

VARGHESE, B. *et al.* Challenges and Opportunities in Edge Computing. In: 2016 IEEE International Conference on Smart Cloud (SmartCloud). [*S. l.: s. n.*], 2016. P. 20–26. DOI: 10.1109/SmartCloud.2016.18.

WARDEN, P. **Why the Future of Machine Learning is Tiny**. [*S. l.: s. n.*], 2018. Disponível em: https://petewarden.com/2018/06/11/why-the-future-of-machine-learning-is-tiny/. Acesso em: 6 nov. 2022.

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

XU, Y. *et al.* Machine learning in construction: From shallow to deep learning. **Developments in the built environment**, Elsevier, v. 6, p. 100045, 2021.

YANG, J. *et al.* Quantization Networks. In: PROCEEDINGS of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). [*S. l.: s. n.*], jun. 2019.

ZEHRA, F. *et al.* Comparative analysis of C++ and Python in terms of memory and time. Preprints, 2020.

ZHOU, Z.-H. **Machine learning**. [*S. l.*]: Springer Nature, 2021.

ZHU, M.; GUPTA, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. **arXiv preprint arXiv:1710.01878**, 2017.

ZOU, Z. *et al.* Object detection in 20 years: A survey. **arXiv preprint arXiv:1905.05055**, 2019.

ANEXO A — LEI N.º 9.610, DE 19 DE FEVEREIRO DE 1998: DIREITOS AUTORAIS / DISPOSIÇÕES PRELIMINARES

Presidência da República
Casa Civil
Subchefia para Assuntos Jurídicos

LEI Nº 9.610, DE 19 DE FEVEREIRO DE 1998.

Mensagem de veto

Regulamento

Altera, atualiza e consolida a legislação sobre direitos autorais e dá outras providências.

O PRESIDENTE DA REPÚBLICA Faço saber que o Congresso Nacional decreta e eu sanciono a seguinte Lei:

Título I

Disposições Preliminares

Art. 1º Esta Lei regula os direitos autorais, entendendo-se sob esta denominação os direitos de autor e os que lhes são conexos.

Art. 2º Os estrangeiros domiciliados no exterior gozarão da proteção assegurada nos acordos, convenções e tratados em vigor no Brasil.

Parágrafo único. Aplica-se o disposto nesta Lei aos nacionais ou pessoas domiciliadas em país que assegure aos brasileiros ou pessoas domiciliadas no Brasil a reciprocidade na proteção aos direitos autorais ou equivalentes.

Art. 3º Os direitos autorais reputam-se, para os efeitos legais, bens móveis.

Art. 4º Interpretam-se restritivamente os negócios jurídicos sobre os direitos autorais.

Art. 5º Para os efeitos desta Lei, considera-se:

I - publicação - o oferecimento de obra literária, artística ou científica ao conhecimento do público, com o consentimento do autor, ou de qualquer outro titular de direito de autor, por qualquer forma ou processo;

II - transmissão ou emissão - a difusão de sons ou de sons e imagens, por meio de ondas radioelétricas; sinais de satélite; fio, cabo ou outro condutor; meios óticos ou qualquer outro processo eletromagnético;

III - retransmissão - a emissão simultânea da transmissão de uma empresa por outra;

IV - distribuição - a colocação à disposição do público do original ou cópia de obras literárias, artísticas ou científicas, interpretações ou execuções fixadas e fonogramas, mediante a venda, locação ou qualquer outra forma de transferência de propriedade ou posse;

V - comunicação ao público - ato mediante o qual a obra é colocada ao alcance do público, por qualquer meio ou procedimento e que não consista na distribuição de exemplares;

VI - reprodução - a cópia de um ou vários exemplares de uma obra literária, artística ou científica ou de um fonograma, de qualquer forma tangível, incluindo qualquer armazenamento permanente ou temporário por meios eletrônicos ou qualquer outro meio de fixação que venha a ser desenvolvido;

VII - contrafação - a reprodução não autorizada;

VIII - obra:

a) em co-autoria - quando é criada em comum, por dois ou mais autores;

b) anônima - quando não se indica o nome do autor, por sua vontade ou por ser desconhecido;

c) pseudônima - quando o autor se oculta sob nome suposto;

d) inédita - a que não haja sido objeto de publicação;

e) póstuma - a que se publique após a morte do autor;

f) originária - a criação primígena;

g) derivada - a que, constituindo criação intelectual nova, resulta da transformação de obra originária;

h) coletiva - a criada por iniciativa, organização e responsabilidade de uma pessoa física ou jurídica, que a publica sob seu nome ou marca e que é constituída pela participação de diferentes autores, cujas contribuições se fundem numa criação autônoma;

i) audiovisual - a que resulta da fixação de imagens com ou sem som, que tenha a finalidade de criar, por meio de sua reprodução, a impressão de movimento, independentemente dos processos de sua captação, do suporte usado inicial ou posteriormente para fixá-lo, bem como dos meios utilizados para sua veiculação;

IX - fonograma - toda fixação de sons de uma execução ou interpretação ou de outros sons, ou de uma representação de sons que não seja uma fixação incluída em uma obra audiovisual;

X - editor - a pessoa física ou jurídica à qual se atribui o direito exclusivo de reprodução da obra e o dever de divulgá-la, nos limites previstos no contrato de edição;

XI - produtor - a pessoa física ou jurídica que toma a iniciativa e tem a responsabilidade econômica da primeira fixação do fonograma ou da obra audiovisual, qualquer que seja a natureza do suporte utilizado*;*

XII - radiodifusão - a transmissão sem fio, inclusive por satélites, de sons ou imagens e sons ou das representações desses, para recepção ao público e a transmissão de sinais codificados, quando os meios de decodificação sejam oferecidos ao público pelo organismo de radiodifusão ou com seu consentimento;

XIII - artistas intérpretes ou executantes - todos os atores, cantores, músicos, bailarinos ou outras pessoas que representem um papel, cantem, recitem, declamem, interpretem ou executem em qualquer forma obras literárias ou artísticas ou expressões do folclore.

XIV - titular originário - o autor de obra intelectual, o intérprete, o executante, o produtor fonográfico e as empresas de radiodifusão. (Incluído pela Lei nº 12.853, de 2013)

Art. 6º Não serão de domínio da União, dos Estados, do Distrito Federal ou dos Municípios as obras por eles simplesmente subvencionadas.

...

Texto completo da lei:

ÍNDICE REMISSIVO