FEDERAL UNIVERSITY OF SERGIPE

EXACT SCIENCES AND TECHNOLOGY CENTER

COMPUTER SCIENCE POSTGRADUATE PROGRAM

A Resource Constrained Pipeline Approach to Embed Convolutional Neural Models (CNNs)

Master's Dissertation

Rafael Andrade da Silva

São Cristóvão – Sergipe

2022

FEDERAL UNIVERSITY OF SERGIPE

EXACT SCIENCES AND TECHNOLOGY CENTER

COMPUTER SCIENCE POSTGRADUATE PROGRAM

Rafael Andrade da Silva

A Resource Constrained Pipeline Approach to Embed Convolutional Neural Models (CNNs)

Master's dissertation proposal presented to the Post-graduate program in Computer Science at the Federal University of Sergipe as partial requirement for the Master degree in Computer Science.

Advisor: Prof. Dr. Bruno Otávio Piedade Prado
Co-advisor: Prof. Dr. Leonardo Nogueira Matos

São Cristóvão – Sergipe

2022

Dedico este trabalho a minha família, amigos, professores e a todos que de alguma forma deram suporte para que eu chegasse até aqui.

O difícil nós fazemos agora,
o impossível leva um pouco mais de tempo.
(David Ben Gurion)

Abstract

The objective of autonomous driving edge computer systems is to ensure the safety of Autonomous Vehicles (AV). However, this is extremely difficult. Advanced Driver Assistance Systems (ADAS) are of great importance in AV systems, as they increase the level of safety in vehicles. As vehicles become more connected, some ADAS features can be improved with the cooperation of the surrounding vehicles. For example, cooperative adaptive cruise control or a lane departure warning for all vehicles in the vicinity. Traffic Signal Detection and Recognition (TSDR) is a recent technology applied to intelligent driving responsible for identifying and recognizing traffic signs in the images captured by the vehicle's sensors. TSDR systems have a wide range of applications. However, many of the proposed techniques use solutions based on expensive devices and are unsuitable for large-scale and low-cost edge computing solutions. Implementing these systems on OEM embedded platforms will provide the opportunity to create genuinely cost-effective and low-energy systems.

In order to contribute to this research area, our study proposes not only the development of a convolutional neural network capable of performing the classification of vertical traffic signals but also the creation of a neural model compression pipeline. Based on the literature and experiments located through a systematic review, we chose to use the GTSRB dataset to evaluate the work. The pipeline has three stages: knowledge distillation, pruning, and quantization of neural models. The goal is to reduce the complexity of the final neural network, thus allowing the model to be embedded in a device with limited computational resources. The final models are evaluated considering performance metrics such as accuracy, precision, recall, F_1-Score, inference time, and model size in bytes.

Using the proposed methodology, our compressed CNN model achieved an accuracy of 85.91% and an F1-Score of 85.80%. The final model size was only 59 KB and the inference of a color image with a resolution of 32x32 pixels took only 80 ms to run in ESP32 and 83 ms to run in ESP32-S2, demonstrating the capability of this resource-constrained device to detect an image with a reasonable accuracy rate.

Keywords: CNN. Edge devices. Quantization. Traffic signs. Knowledge distillation. Pruning. TSDR. GTSRB.

List of Figures

List of Tables

List of abbreviations and acronyms

ABS - Anti-lock Braking System

ACC - Adaptative Cruise Control

ACM - Association for Computing Machinery

ADAS - Advanced Driver Assistance Systems

AO - Average box Overlap

API - Application Programming Interface

AUC - Area Under precision-recall Curve

auto-HUD - Automotive Head-Up Display

BDD100K - Berkeley DeepDrive Dataset

BNN - Binarized Neural Network

BTSD - Belgian Traffic Sign Dataset

BTSR - Belgium Traffic Sign Recognition

BTSD - Belgian Traffic Sign Dataset

CAS - Collision Avoidance System

CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior

CIFAR-10 - Canadian Institute For Advanced Research (10 classes)

CNN - Convolutional Neural Network

CPU - Central Process Unit

CUDA - Compute Unified Device Architecture

DMA - Direct Memory Access

DNN - Deep Neural Network

DRAM - Dynamic Random Access Memory

DSP - Digital Signal Processor

ESC - Electronic Stability Control

FLIX - Flexible Length Instruction eXtensions

FN - False Negative

FP - False Positive

FPGA - Field-Programmable Gate Array

FPN - Feature Pyramid Network

FPS - frames per second

GFLOPS - Giga Floating-point Operations Per Secong

GMAC - Billion (Giga) Multiply And Accumulate

GPU - Graphics Processing Unit

GTSDB - German Traffic Sign Detection Benchmark

GTSRB - German Traffic Sign Recognition Benchmark

HSV - Hue Saturation Value (space color)

IEEE - Institute of Electrical and Electronics Engineers

KITTI - Karlsruhe Institute of Technology and Toyota Technological Institute

IoT - Internet of Things

IoU - Intersection over Union

IPAS - Intelligent Parking Assist System

ISA - Instruction Set Architecture

LIDAR - Light Detection And Ranging

LKA - Lane Keep Assist

LSTM - Long Short-TermMemory

MAC - Multiply-Accumulator

mAP - mean Average Precision

MB - Megabyte

MCU - Microcontroller Unit

MHz - MegaHertz

ML - Machine Learning

MLP - Multilayer Perceptron

NHTSA - National Highway Traffic Safety Administration

NMS - Non-Maximum Suppression

OCR - Optical Character Recognition

OEM - Original Equipment Manufacturer

OMS - Organização Mundial da Saúde

Pascal VOC - Pascal Visual Object Classes

PIO - PlatformIO

PPM - Portable PixMap

PSRAM - Psuedostatic DRAM

QNN - Quantized Neural Networks

RAM - Random Access Memory

ReLU - Rectified Linear Unit

ResCoNN - Resource-Efficient CNN

RGB - Red Green Blue

RMSProp - Root Mean Square Propagation

ROI - Region Of Interest

ROM - Read-Only Memory

RPN - Region ProposalNetwork

RS - Revisão Sistemática

SAE - Society of Automotive Engineers

SDTS - Spatial Data Transfer Standard

SIMD - Single Instruction, Multiple Data

SoC - System on Chip

SRAM - Static Random-Access Memory

SSD - Single-Shot Multibox Detector

TCS - Traction Control System

TFL - TensorFlow Lite

TFLOPS - Tera Floating-point Operations Per Secong

TN - True Negative

TP - True Positive

TSDR - Traffic Sign Detection and Recogntion System

TSR - Traffic Sign Recognition

VGGNet - Visual Geometry Group Network

VLIW - Very Long Instruction Word

VM - Virtual Machine

VSCode - Visual Studio Code

WHO - World Health Organization

Contents

1 Introduction

In recent years, the world population has experienced significant benefits due to technological advances in different areas. This advance has brought greater convenience and quality of life to citizens, enabling tasks that previously required hours of execution or a great deal of human effort to be performed in minutes or seconds. In most modern industries, these innovations were enabled by electronic systems, which allowed the development of control and information processing devices to become increasingly cost-effective and power-efficient while reducing the size of their components.

The automotive industry is a notable example of this development of semiconductor components, which, as one of the most dynamic industries in the world, is considered to have a significant influence on other sectors of the economy. Driven by groundbreaking innovations, such as the introduction of the mass production systems by Henry Ford in 1914 and Toyotism in 1970 (TURI, 2015), which provided several improvements in vehicle production. Due to its competitive market (SENHORAS, 2005), the automobile industry must continually reformulate its strategies, deciding where to deploy factories or which technologies are required or should be developed. In order to achieve a large production volume which is mandatory to reduce unitary costs, the basic raw materials, assembly machinery, and labor must be carefully selected to optimize all processes (RETORNO, 2019). This economy of scale is a core concept in this industry and platform sharing lately employed by car manufacturers is another excellent example of one among various strategies used to reduce production costs (CARTHROTTLE, 2016).

According to data from IBGE (2018), the number of vehicles in Brazil has reached more than 100 million units (54.7 million are automobiles), a growth of 3.7 percent over the previous year of 2017. Thus, it is a straightforward deduction that this increasing number of vehicles increasingly jams Brazilian city traffic. This is also happening in other cities around the world because cars are still a predominant mode of transport, especially where public transportation has limited coverage, accessibility, and safety.

With a large number of human-controlled vehicles moving around the world, it is also expected that the number of traffic accidents is also high. According to the World Health Organization (WHO), road accidents are the leading cause of death of people in the world, and 90 percent of these deaths occur in middle and low-income countries, despite having approximately 54 percent of the world's vehicles (BRASIL, 2018). The reasons for vehicle accidents can be classified into human, vehicle, and road factors. In Brazil, 90% of the accidents are caused by the human factor. In other words, some of these deaths can be avoidable (OBSERVADOR, 2018). Some conditions contributing to human error in driving include speeding, drunk driving, mobile device or passenger distractions, and driver fatigue. All these factors reduce the driver's time needed to observe, process, and react promptly if dangerous situations occur on the road. In order to help to solve these issues, the employment of the advancement of electronic systems and digital cameras into ever more robust and efficient devices allows for the prevention of several vehicle accidents. These systems are known as Advanced Driver Assistance Systems (ADAS) and they are designed to assist drivers in areas around the vehicle that the driver cannot directly observe, such as traffic lane changes (LKA - Lane Keep Assist), vehicle stabilization (ESC - Electronic Stability Control), and traction control (TCS - Traction Control System). Estimated at US$24 billion in 2018, the ADAS market may rise to US$91 billion by 2030. The increasing demand for a safe, efficient, and convenient driving experience, added to the growing demand for luxury vehicles worldwide, is the primary growth factor of the ADAS industry (GLOBALLOGIC, 2018). There are some other systems included in ADAS: Adaptive Cruise Control (ACC), Intelligent Parking Assist System (IPAS), Collision Avoidance System (CAS), Anti-lock Braking System (ABS), Automotive Head-Up Display (auto-HUD), and Traffic Sign Detection and Recognition System (TSDR), the latter of which will be the focus of our work.

TSDR solutions operate by continuously acquiring images of the vehicle's surroundings through one or more cameras, typically positioned in zones near the vehicle's sides and/or roof. After image acquisition, methods and techniques for information extraction, detection, and object recognition are applied to retrieve information about the surrounding environment. Once a target is identified, the driver is alerted about the traffic sign presence and what action should be performed.

1.1 Motivation

Developing embedded systems that can assist drivers in the tasks involved in driving is highly relevant in constructing a safer road ecosystem. Both society and industry have great expectations regarding how automated vehicles can assist in decision-making and reduce the number of traffic accidents. However, this promising vehicle solution for traffic issues have currently few adopters due to high automotive sensitivity to production costs. These devices that allow vehicle automation, including associated actuators/sensors and how they can support drivers, still represent a challenge for success in this niche (RICARDO, 2019).

Recently, in the North American society, the National Highway Traffic Safety Administration[1] (NHTSA) has started using the vehicle automation level classification system defined by the Society of Automotive Engineers (SAE) published in 2014 via the J3016 standard and used for the first time in 2016 (INTERNATIONAL, 2019).

Table 1 – SAE-defined levels of vehicle automation

Level	**Name**	**Definition**
0	No Automation	Systems limited to issuing momentary alerts and assists, such as emergency brake control, blind spot warning, or lane departure alerts.
1	Driver Support	Steering or Braking Control, such as lane centering or cruise control, both with driver supervision.
2	Partial steering automation	Both steering and braking controls are performed by the system with driver supervision.
3	Conditional automation of driveability	all steering controls operated by the system. Human control will be required when the system cannot handle some tasks.
4	Highly automated steering	All steering operations are performed by the system, which remains in charge even in emergencies, without waiting for human intervention. At this point, the steering wheel and pedals are completely removed from the vehicle.
5	Fully automated steering	Steering operations are completely performed by the system without human intervention.

Source: INTERNATIONAL (2019)

Considering the information in Table 1 and the recent advances of some automakers in the development of autonomous vehicles, we will detail the case of Tesla Incorporation[2]. Even with the various automation and drivability features delivered by Tesla's products, the brand's vehicles are officially in automation category three as defined by SAE INTERNATIONAL (2019). Even though its powerful on-board computers in vehicles (VERGE, 2019), Tesla's cars provided only in August 2020 the automatic speed adjustments upon traffic sign detection regarding speed limits (INDEPENDENT, 2020; PCMAG, 2020). Many researchers are developing ADAS solutions that act in a fully automated way. However, implementing these systems requires robust hardware (DSPs or GPUs) to optimize the solution's run-time. This specialized hardware adds more cost to solution cost, starting at US$59 [3] (price per module, low volume purchase), as in Han e Oruklu

[1] https://www.nhtsa.gov/
[2] https://www.tesla.com/
[3] https://www. newegg.com/nvidia-945-13541-0000-000/p/N82E16813190013

(2017), Chougule et al. (2018), Baicu et al. (2019). For the automobile industry that relies on scalability and low production cost, these high input values may be prohibitive to creating a competitive market.

Resource-constrained devices, i.e., with low storage capacity and computational power, are increasingly used globally in the Internet of Things (IoT) solutions, such as those developed by Espressif[4] and ARM[5]. Despite the limited availability of computational resources, approaches such as Yamada et al. (2020) and Petrova (2017) showed that these significantly reduced cost devices could be successfully used for computer vision applications. However, given the storage and processing characteristics, it is not always possible to fully embed complex algorithms in these devices. Considering the heterogeneity and number of neural model parameters commonly used in computer vision applications, the required memory size to embed this solution can be prohibitive. To illustrate this limitation, the popular low-cost IoT platform Espressif ESP32 has only 320KB of RAM and 4MB of FLASH memory.

1.2 Main Objective

This work aims to develop a Convolutional Neural Network (CNN) compression pipeline capable of reducing the model size. This achievement would allow a resource-constrained device, such as Espressif ESP32 or ESP32-S2, to perform vertical traffic signs image classification. In order to reach this goal, deep learning methods are optimized for memory and processing limited hardware deployment. The neural network compression is based on knowledge distillation, pruning, quantization, and optimization of network hyperparameters methods to fit neural model size to embedded device capabilities. As the automotive industry is quite sensitive to production costs, a proof-of-concept of a TSDR low-cost platform able to perform at acceptable accuracy and latency could be a game changer. Some steps are listed in order to meet these work objectives:

1. Create a pipeline flow capable of reducing the size of a neural model to the point where it can be embedded in a resource-constrained device;

2. Develop a model with acceptable inference run-time speed latency to be used in image detection;

3. Demonstrate that experiments provide accuracy levels similar to traditional implementations.

[4] https://www.espressif.com/
[5] https://www.arm.com/

1.3 Methodology

In many related works described in Chapter 3, we can observe the stages required for a fully functioning TSDR system. Usually, the steps involved are image acquisition, preprocessing, segmentation, description, and classification. Some of these papers also describe compression techniques or methods to speed up the inference of their neural models. However, this work will focus only on constructing mechanisms for use in the classification stage.

The prototype will be based on the ESP32 microcontroller with restricted resources, which will be better described in Section 2.3. The idea behind this choice is that the purchase price is meager, which would reduce the production cost for TSDR systems.

To this end, the methodology for developing the prototype will stick to the following steps:

1. Identify, through a Systematic Review (SR), the related works that use embedded devices for traffic sign recognition;
2. Determine which data sets are more popular in the classification of traffic signs or daily situations referring to the road environment;
3. Verify which metrics are most used to check the efficiency of a given solution;
4. Ensure that a neural model has acceptable accuracy in solving an image classification problem involving traffic signs;
5. Evaluate the final proposed model to be embedded in a device with memory and processing constraints if it can still perform inference in real-time[6].

1.4 Document Structure

For ease of navigation and a better understanding of this work, we have structured this document into the following chapters, namely:

- Chapter 2 - Theoretical Foundation: This chapter will address issues and concepts that will support the understanding of the work in question;
- Chapter 3 - Related Works: Here, the selected papers through a Systematic Review are presented in order to base the content addressed in this proposal;
- Chapter 4 - Proposed Detection Model: On this chapter is described the proposed methodology and architecture for building the compression pipeline and neural model;

[6] Real-time control systems are closed loop control systems where one has a tight time window to gather data, process that data, and update the system (CHON, 2008)

- Chapter 5 - Results: The results obtained at each stage of pipeline implementation and testing are demonstrated;
- Chapter 6 - Conclusion: The final considerations about the work are presented, including its contributions, the limitations found, and suggestions for future work.

2 Basic Concepts

This chapter will address concepts, device architectures and characteristics regarding the theories and platforms used in this proposed work. Topics such as artificial neural networks, the quantization of neural models, the framework TensorFlow and the resource-constrained devices will be detailed using information acquired from related works and various references.

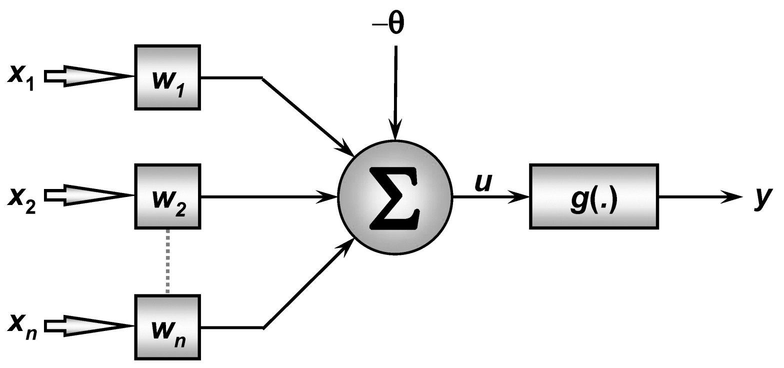
2.1 Artificial Neural Networks - ANN

An Artificial Neural Network (ANN) is a complex structure interconnected by simple processing elements called neurons, which can perform operations such as parallel calculations and data processing. They have the ability to store and deal with knowledge representations. The ANN is similar to the human brain because of its capacity to acquire knowledge from the environment using a learning process and using it to force the connection between neurons to store acquired knowledge. These characteristics, added to the ability of generalization, adaptability, and fault tolerance, allow ANN to solve complex problems more agilely than traditional approaches. In next subsections we will detail some key points ANN conception.

2.1.1 The Artificial Neuron

A neural network is a massively parallel distributed processor consisting of simple processing units, which has a natural propensity to store experimental knowledge and make it available for use (HAYKIN, 2009). Also, according to Haykin (2009), a neuron is an information processing unit that is fundamental to the operations of an ANN. In 1943, a mathematical neuron model proposed by Mcculloch e Pitts (1943) was presented and it would become the most well-accepted standard. Its structure mimics the operation of a biological neuron in a very simplified way. In Figure 1 it is shown this artificial neuron representation and its further detailed description.

Figure 1 – Mathematical model of a neuron.



- x_1 to x_n are inputs and represent the pulses received by the network;
- Each input signal is multiplied by a weight, which indicates its influence on the network. These weights are defined by w_1 to w_n;
- The sum of the product of the inputs by their respective weights, represented in Figure 1 by the $\sum$ block, depicts the activity level of the neuron;
- Θ represents the firing threshold of the neuron;
- u represents the output pulse of the neuron;
- g corresponds to the activation function. The activation function corresponds to the neuron's decision making, regarding the firing or not of the pulse in its output (SILVA; SPATTI; FLAUZINO, 2010);
- The output y equals the electrical pulses emitted by biological neurons.

This neuron model and its parameters can be mathematically defined by the equations 2.1 and 2.2.

$$u = \sum_{i=1}^{n} w_i * x_i - \theta \tag{2.1}$$

$$y = g(u) \tag{2.2}$$

Although this model has a straightforward representation, the combination of several neurons (ANN) is capable of performing complex functions. Through values assigned to **w**, Θ and setting the activation function (**g**), the ANN may be able to adapt to different problems. These parameters adjustments are called ANN training.

As defined in Silva, Spatti e Flauzino (2010), the activation function limits the output of a neuron over a finite range of values. The main activation functions are the linear function, the ramp function, the step function, and the sigmoid function. These definitions are described in Table 2.

Table 2 – Search strings defined for each repository after initial effectiveness evaluation.

Function	**Definition**
Linear Function	$f(u) = a * u$
Ramp Function	$f(u) = \begin{cases} +\alpha, & \text{if } u \geq +\alpha \\ u, & \text{if } \lvert u \rvert < +\alpha \\ -\alpha, & \text{if } u \leq -\alpha \end{cases}$
Step Function	$f(u) = \begin{cases} +\alpha, & \text{if } u \geq 0 \\ -\alpha, & \text{if } u < 0 \end{cases}$
Sigmoid Function	$f(u) = \frac{1}{1+e^{-u/T}}$

Its ability to learn from a data set is the most appreciated characteristic of an ANN. As described by Silva, Spatti e Flauzino (2010), training an ANN consists of applying well-defined steps and adjusting parameters, intending to generalize the results given by its outputs. A well-designed set of these steps is called the learning algorithm, which aims to train the ANN. The ANN training is an iterative process, performing parameters adjustments based on input data and network's desired goal. Ongsulee (2017) classifies learning methods as follows:

- **Supervised Learning:** this method is executed by providing a set of previously labeled data whose goal is to find a function that can predict the unknown labeling. The algorithm should take care observing the examples of the input and associate a label. defining the probability that an input x belongs to a class y;

- **Unsupervised Learning:** in this model, the algorithm should predict similarities between objects without the aid of labels. This way, the algorithm should explore all the input data and find recognizable structures. Some techniques that help in this task are K-Means Clustering and Singular Value Decomposition;

- **Semi-supervised Learning:** : is used for the same applications as supervised learning. However, it uses both classified and unclassified data for training and is useful when the cost associated with classification is high;

- **Reinforcement learning:** this is where the algorithm learns to achieve a goal in an uncertain environment. The program uses trial and error to maximize a solution to a given problem. The code receives rewards or penalties for the actions it performs.

2.1.2 Deep Learning

Goodfellow, Bengio e Courville (2016) report that in recent years the interest in Machine Learning (ML) has grown a lot. In his article, Dietterich (2003) defines ML as the study of methods for developing systems in which the computer no longer follows a set of instructions imperatively defined by the programmer but can learn through past knowing how to perform a designated task.

Deep Learning can be understood as a method set that allows a machine receives data and automatically extract necessary representations for classification or detection tasks LeCun, Bengio e Hinton (2015). A traditional ANN has only 2 or 3 hidden layers, whereas a deep ANN can have hundreds. A deep ANN is much more difficult to train than a simple ANN, but for a more complex set of input data, a deep ANN has a higher efficiency (NIELSEN, 2018). In Deep Learning, learning is end-to-end learning from the raw input data. Its main advantage is that the learning of a deep network improves as the input data grows, while in other algorithms, it is stabilized (MATLAB, 2021). Deep Learning techniques are responsible for significant advances in the computer vision area. Among these techniques, the most relevant and studied are the Convolutional Neural Networks (CNNs).

2.1.3 Convolutional Neural Networks

A Convolutional Neural Network, also known by the abbreviations ConvNet or CNN, are neural networks that employ a mathematical operation known as a convolution in order to extract information. The CNN is a feedforward class of Artificial Neural Networks widely used in pattern recognition area (LI et al., 2018). In addition, CNN uses the backpropagation algorithm during training. This algorithm is formed by two stages. In the first stage, called forward pass) the inputs was passed through the network and the output is the calculated probabilities in relation to the expected classes. The second step is the backward pass, were the gradient of error function is calculated with respect to the neural network's weights. ConvNets were developed based on ocular biology, starting with the pioneering study by Hubel e Wiesel (1962) that evaluated cat visual cortex structures.

2.1.3.1 Convolution

Recently, much has been heard about convolution operations in Machine Learning (SINDAGI; PATEL, 2018) (ZHANG et al., 2019). Many possibilities have emerged with this use, such as in image classification or feature extraction tasks.

In its most general form, convolution is a mathematical operation, like summation, that operates between two signals to obtain the third signal as response. Goodfellow, Bengio e Courville (2016) define the discrete convolution operation calculus for any one-dimensional input x and a filter w as:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.3}$$

Thus, considering Equation 2.3, for each time portion t, the signal s receives the result of the summation of a signal x multiplied by a weighting factor w, where the parameter a refers to the intervals of the input signal dimension. Goodfellow, Bengio e Courville (2016) define based on Equation 2.3 and 2.4, where they use multidimensional input signals such as images.

$$s(i,j) = (I * K)(i,j) = \sum_{M}\sum_{N} I(m,n)K(i-m, j-n) \tag{2.4}$$

I is the input image, K is the filter, and MxN is the input signal matrix. One last adjustment is still made in Equation 2.4 to make it more convenient for implementation in software since the neural network libraries use the cross-correlation operation, which is the same as convolution but keeps the kernel. Therefore, Goodfellow, Bengio e Courville (2016) rewrite the equation as follows:

$$s(i,j) = (K * I)(i,j) = \sum_{M}\sum_{N} I(i+m, j+n)K(m,n) \tag{2.5}$$

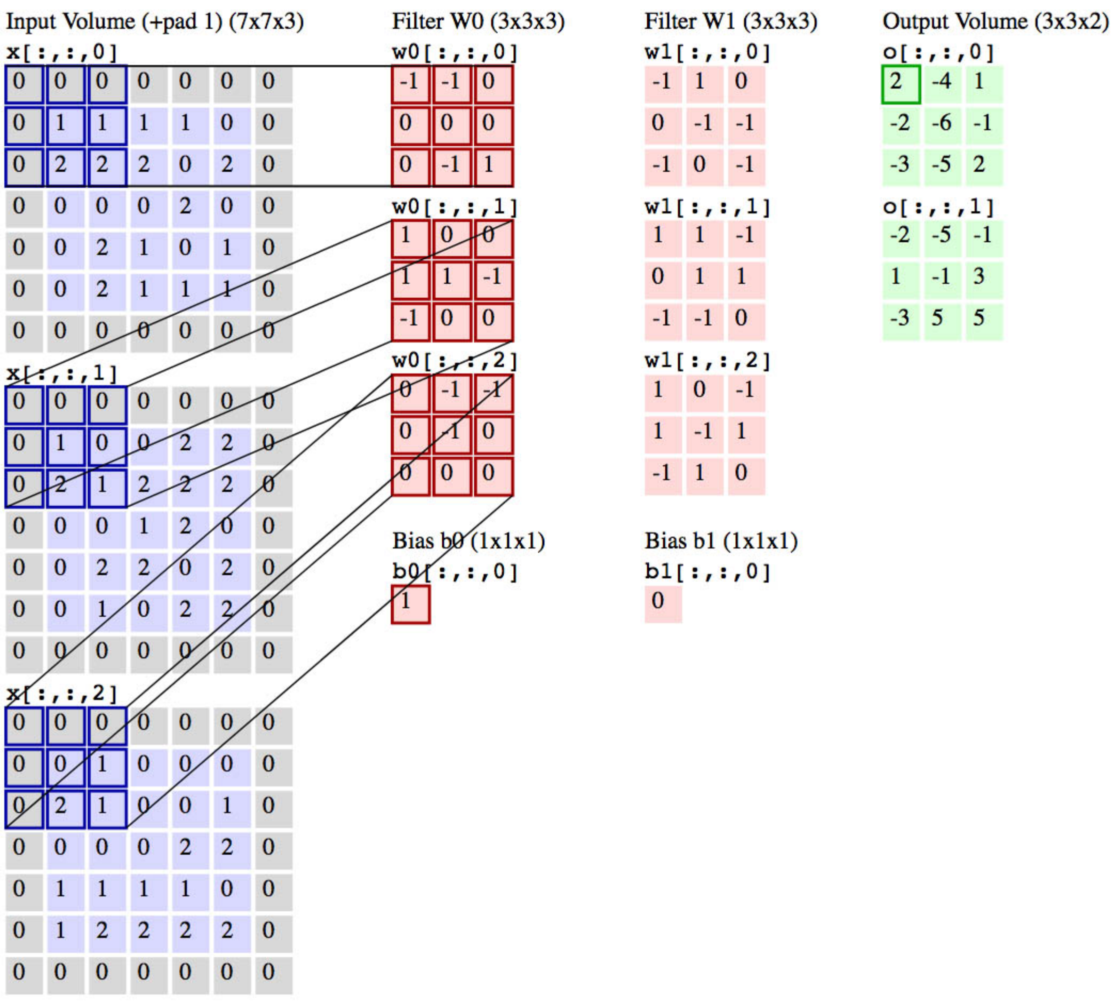
A CNN is a Deep Learning model capable of capturing an input image, assigning importance, such as weights and features, to various aspects and objects in the image, and differentiating between them (ACADEMY, 2019a). The convolutional layer is responsible for extracting the so-called features from the input. This extraction takes place by applying convolutional filters and traversing the input data in width, height, and depth. The convolutional layer has a set of feature maps generated from convolutions on the input data or from another map. The local receptive area is the input region where the filters are applied (ACADEMY, 2019a). A Convolutional Network employs three primary operations: local receptive fields, shared weights, and pooling.

An example of the convolution operation on an RGB image x can be seen in Figure 2. In this image, it can be verified the application of the convolutional filter $W0$ on all image channels x. For the filter, also known as the kernel, to move across the entire image, a parameter called stride (step) is introduced. The stride is the number of steps used by moving the filter through the input image. The results obtained by filter application on the image are summed to the bias parameter. In the end, a value is assigned in the resulting output.

2.1.3.2 Pooling

The objective of the pooling layer is to progressively reduce the input data volume, reducing the number of parameters and the need for additional operations performed on the network (KARPATHY, 2016). According to Goodfellow, Bengio e Courville (2016), the pooling

Figure 2 – Example of a convolutional operation.



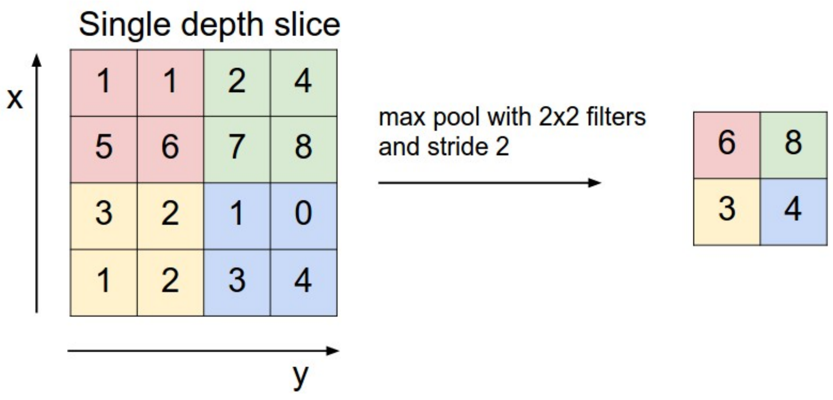
Source: KARPATHY (2016)

function replaces the network output at a given location with a summary statistic of the nearest outputs, i.e., in general, it serves to simplify the information provided by the immediately preceding layer. The most widely used method for pooling is Max Pooling, but there are other methods known as Average Pooling, and L2-norm Pooling (KARPATHY, 2016). In the Max Pooling technique, the most significant value read by a filter is passed to the output. As with the convolution layer, the pooling operation also has the parameters stride and size of the kernel. In Figure 3 we can observe the pooling operation being performed on an image, using a kernel of dimension 2x2 and the step size (stride) set to 2.

2.1.3.3 Fully Connected Layer

The fully connected layer is responsible for performing the object classification. Also known as a dense layer or fully connected, it works as a multilayer neural network of architecture feedforward, like MLP, which is responsible for interpreting the features extracted by the initial layers (EBERMAM; KROHLING, 2018). Usually, the previous layer is a flatten layer, responsible

Figure 3 – Example of a Max Pooling operation.



Source: KARPATHY (2016)

for flattening all the feature maps from the last pooling layer that have more than one dimension, relocating them into a one-dimensional vector to connect them to the final part of the (EBERMAM; KROHLING, 2018) network. The output of this layer is N neurons, where N is the number of classes in the data set that the model needs to classify (ALVES, 2018).

2.1.3.4 Performance Metrics

In evaluating classification models, the basic concept is the notion of failure. If applying the classification model to a selected case leads to the prediction of a different class than the actual examples, then there is a classification error. If any error is equally important, then the number of errors in the observed set can be a working indicator of a classifier (NOVAKOVIĆ et al., 2017). This approach relies on accuracy as a metric for evaluating the quality of a classification model.

Kohavi, R. and Provost (1998) defined a confusion matrix as a way to evaluate the system's performance, displaying the predicted classifications and the classifications obtained by the model. The confusion matrix will show the following frequencies:

- **True Positive (TP):** occurs when in the actual set data, the object we are looking to predict is correctly classified;
- **False Positive - FP):** occurs when in the actual set data, the object we are looking to predict is classified incorrectly;
- **True Negative (TN):** occurs when in the actual set data, the object we are not seeking to classify is correctly predicted;
- **False Negative (FN):** occurs when in the actual set data, the object we are not seeking to classify is incorrectly predicted.

This is the starting point of information gathering for defining the following metrics:

- **Accuracy:** is a way of measuring how often an algorithm classifies a set of data correctly. The number of true positives (TP) and true negatives (TN) divided by the number of true positives, true negatives, false positives (FP), and false negatives (FN) defines the accuracy calculation.

$$A_c = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.6}$$

- **Recall:** the recall defines the proportion of real positives that have been correctly identified. Furthermore, recall is considered primary in traffic sign recognition since the goal is to identify all real positives. The mathematically form to recall is established as follows:

$$R_e = \frac{TP}{TP + FN} \tag{2.7}$$

- **Precision:** the precision is defined as the fraction of relevant instances among all retrieved instances. In some cases, accuracy is not a good indicator of model performance. One such scenario is when the dataset is unbalanced. At this point, even if the model predicts all instances as the most frequent class, the model would still get a high accuracy rate. To combat this, we use the precision metric. Precision is defined as follows:

$$P_r = \frac{TP}{TP + FP} \tag{2.8}$$

- **F$_1$-Score:** the F$_1$-Score measure combines precision and recovery into a single measure. It is a harmonic mean of $_{\text{recall}}$ and precision. The advantage of this harmonic measure is that to increase the value of F$_1$-Score, it is necessary that both (recall and precision) increase, unlike the arithmetic mean:

$$F_1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.9}$$

- **Root Mean Squared Error**: is the measure that calculates the root mean square of the errors between observed (real) values and predictions (hypotheses). RMSE is defined as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{j=1}^{n}\left(y_j - \hat{y}_j\right)^2} \tag{2.10}$$

2.1.3.5 Training and Network Parameters

For training, we used the Tensorflow framework (ABADI et al., 2016) and Google Colab (GOOGLE, 2020) because this combination allows us to build and train machine learning models using intuitive high-level APIs (such as Keras (KERAS, 2020)) and free powerful cloud services.

Batch Size: This parameter defines the total number of instances used for a training epoch (an epoch is one such pass of the entire dataset). This parameter can be set in three different ways: (i) batch gradient descent, where the batch size is set to the total number of instances in the training dataset; (ii) Stochastic Gradient Descent (SGD), where the batch size is an equal one, therefore the gradient and the parameters of the neural network are refreshed after each epoch; and (iii) mini-batch gradient descent, where the batch size is more significant than one, but less than which the total instances of the training dataset.

Learning rate and loss function: Deep learning neural networks are trained using the stochastic gradient descent algorithm. This algorithm is an optimization code that estimates the error gradient for the current model state, using instances from the training and validation dataset. That allows the update of the weights of the model using backpropagation of errors calculated by the algorithm. We seek to minimize errors between actual and predicted values when training a neural network. We use a loss function to help a neural network improve its weight. That is applied through a loss function. Mathematically, if the loss function is $E(X;W,b)$, our goal is to minimize E. Considering the vanilla SGD, X is input to the neural network, W is the model weight parameter, and b is the bias value (Konar; Khandelwal; Tripathi, 2020a). The $i-th$ model weight is updated using Equation 2.11.

$$W_{i+1} = W_i - \eta \left(\frac{\partial E}{\partial W_i} \right) \tag{2.11}$$

Here, η is called the learning rate. The learning rate determines how quickly or slowly we want to update the parameters of the neural network. There are different ways to choose the initial learning rate. The approach chosen for this work was the step decay learning rate. We start with a relatively high learning rate in this technique and then gradually decrease it during the training (Konar; Khandelwal; Tripathi, 2020b).

Underfitting occurs when the neural model is not able to reduce training error. That means that our model is too simple to generalize the dataset characteristics. In contrast, when the model is more complex, it may adapt excessively to the training dataset. However, it does not generalize well to the new data. When this occurs, we call it overfitting.

Momentum: The momentum in neural networks is a coefficient applied to Equation 2.11 to increase the training speed. As discussed above, if the learning rate is too high, the model will experience many oscillations and can not converge correctly. To mitigate this problem, we introduce the momentum term here M.

$$M = \left(\lambda * W_i^{t-1}\right) \tag{2.12}$$

In the Equation 2.12, λ is the momentum factor, and W_i^{t-1} is the $i-th$ weight increment at the previous iteration. This term quantifies the importance of weight variance of the previous epoch of training. In the equation 2.13 we describe the vanilla SGD with the addiction of the momentum term.

$$W_{i+1} = W_i - \eta\left(\frac{\partial E}{\partial W_i}\right) + M \tag{2.13}$$

L2 Regularization (Weight decay): The regularization process introduces a new term to prevent overfitting. It helps to avoid the linear models overfitting with the training dataset penalizing the extreme weight values. In this work, we utilize the L2 Regularization. In equation 2.14 we can see the expression for L2 Regularization.

$$L_2 = \alpha\frac{1}{2}||\mathbf{W}||_2^2 \tag{2.14}$$

Batch Normalization: The Batch Normalization (BN) layer reduces Covariate Shift. This shift is the change in network activation distributions due to a change in network parameters during training. We know that the network training converges faster if its inputs are whitened, i.e., linearly transformed to have zero means and unit variances, and decorrelated (IOFFE; SZEGEDY, 2015). Then, to increase neural network stability, BN normalizes the output of a previous activation layer. Usually, inputs to neural networks are normalized to either the range of [0, 1] or [-1, 1] or mean=0 and variance=1, the latter is called Whitening (IOFFE; SZEGEDY, 2015). The practical effects of BN are: (i) reduces the training time because of less Covariate Shift (less exploding/vanishing gradients); (ii) reduces the demand for regularization, e.g., dropout, because the means and variances are calculated over batches, and therefore every normalized value depends on the current batch, e.g., the network can no longer memorize values and their correct answers); and (iii) because of less occurrence of exploding or vanishing gradients, the network allows higher learning rates. Given as input, values of x over a mini-batch $B = \{x_1, ..., m\}$, a batch normalization calculate the mean Equation 2.15 and variance Equation 2.16 of the layers input, as follows.

$$\mu B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.15}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu B)^2 \tag{2.16}$$

According to the following equation, the layer input is normalized in sequence using the previously calculated batch parameters.

$$\hat{x}_i = \frac{x_i - \mu B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{2.17}$$

And finally, scale and shift to obtain the output value y_i of the layer, as follows.

$$y_i = \gamma \hat{x}_i + \beta \tag{2.18}$$

The values γ and β are learned during training along with the original parameters of the network.

Optimizer: The optimizers are algorithms used to change the attributes of a neural network, such as weights or learning rate, which reduce the losses. We utilize the optimizer RMSprop (Root Mean Square Backpropagation) for that work. The RMSprop techniques proposed it (HINTON; SRIVASTAVA; SWERSKY, 2012) is a gradient-based optimization based on the Rprop (Resilient Backpropagation) algorithm. RMSprop deals with the gradient's tendency to either vanish or explodes as the data propagates through the backpropagation function. This occurs because the magnitude of gradients can differ for different weights and change during learning, becoming very difficult to choose a single global learning rate. The RMSprop uses the moving average of squared gradients and adjusts the weight updates by this magnitude. We have the following Equation 2.11 to define these concepts.

$$W_{i+1} = W_i - \frac{\eta}{\sqrt{v_i + \varepsilon}} \cdot \frac{\partial E}{\partial W_i} \tag{2.19}$$

Where v_t is given by:

$$v_i = \beta v_{i-1} + (1 - \beta) \left[\frac{\partial E}{\partial W_i} \right]^2 \tag{2.20}$$

The β corresponds to the exponential moving average weight (decay rate).

2.1.4 Quantized Neural Networks

Quantized Neural Networks (QNN) are net with values of weights and activations expressed with reduced accuracy compared to traditional neural networks. Quantization of weights in networks has become an efficient approach for reducing computational costs with minimal impact on the final accuracy of the model. Complex networks have millions of parameters that are typically unsuitable for edge devices implementation. In addition, quantization allows for a significant reduction in power consumption (Wang et al., 2016). The model quantization process works by mapping the 32-bit floating-point values to numbers with 8-bit precision (IEEE-754, 2019). In addition to reducing the final size of the model, quantization can speed up inference time by up to 3 times (HUBARA et al., 2016).

A requirement for the quantization scheme is that the mapping between real numbers and integers is performed by an affine function (JACOB; CHEN,). Ensuring the mapping occurs correctly in both directions, from fixed-point to floating point and back again. It is also necessary that the real value 0 is faithfully representable. Making it possible to fill with zeros in the input matrix, for example, in convolution or pooling layers. Therefore, the quantization scheme generally allows a shift and scaling from the straight line of real numbers to a straight line of quantized values. As described in (JACOB; CHEN,) and transcribed here in Equation 2.21, we have:

$$r = S \times (q - Z) \tag{2.21}$$

The above equation maps an integer q to a real number r. S and Z are quantization parameters for all values within an activation matrix and all weight matrices. Also, according to the work of (JACOB; CHEN,), the constant S (from Scale - Scale) is an arbitrary positive real number and usually represented as a floating point value, like the real value r. The constant Z (Zero point) is the same type as the quantized values q and is the value corresponding to the real zero value. This allows us to meet the need for a quantized value representing the value 0.

A common approach to quantizing a network is to perform training first, using floating-point values, and then perform quantization of the weights in the resulting model. However, this approach significantly drops in accuracy for small models. To resolve accuracy drops, quantization-aware training has been proposed. The effects of quantization are emulated in the foward propagation step, allowing for simulation inference in a quantized way. As explained in (JACOB; CHEN,), weights are quantized before being convolved with the input, and activations are quantized at points where they would be during inference.

2.1.5 Bayesian Optimization

Hyperparameters are important for machine learning algorithms since they directly control the behaviors of training algorithms and significantly affect the performance of machine learning models (WU et al., 2019). These parameters cannot be learned during neural network training and must be provided by the model's designer in question. However, depending on the dimensionality of the model, this can be a costly task. Bayesian Optimization (BO) is useful in this domain where human expertise could not contribute to better accuracy. Bayesian optimization includes preliminary information about the function to be optimized and updates posterior information, which helps reduce loss and maximize the model's accuracy. (VICTORIA; MARAGATHAM, 2020). BO is an approach that uses the Bayes Theorem to direct the search to find the minimum or maximum of an objective function. This theorem states that the posterior probability of a model M gave evidence E is proportional to the likelihood of E given M multiplied by the prior probability of M (KRAMER; CIAURRI; KOZIEL, 2011), as follows in Equation 2.22.

$$P(M|E) = \frac{P(E|M)P(M)}{P(M)} \tag{2.22}$$

Bayesian Optimization is a strategy to find the extreme of a function that may not have a closed expression but can obtain observations in samples. This method is advantageous when the cost of function evaluation is high, is no access to its derivatives, and (or) when the problem is not convex (BROCHU; CORA; FREITAS, 2010). Thus Bayesian Optimization procedure consists of updating the posterior distribution and maximizing the acquisition function. The acquisition function uses a Gaussian Processes hedge where after each action selection, the algorithm receives a reward for each action and updates the gain vector (FIRDAUS; NUGROHO; SOESANTI, 2021).

2.1.6 Teacher–Student Knowledge Distillation

Knowledge distillation refers to model compression by teaching a smaller network, step by step, exactly what to do using a bigger already trained network (GOU et al., 2021). This is quite useful for devices that have limited computing resources which poses several challenges to implement deep neural networks. For instance, nearly 84.04 million parameters are generated to train a VGG16 model on a 256 × 256 image. If the 8-byte floating point data type is used to define every parameter, the size of this model will be 84.04 M * 8B = 640 MB, and its computational complexity up to approximately 10 GFLOPs, which is far beyond the capabilities of these resource-constrained devices (SEPAHVAND; ABDALI-MOHAMMADI; TAHERKORDI, 2022). The existing mainstream knowledge distillation methods can be classified into two categories. In the first category, the student network is optimized using the classified soft labels and genuine labels generated by the teacher network (HINTON et al., 2015). The second method uses the middle layer characteristics of the teacher network to guide the student network (ROMERO et al., 2014). Hinton et al. (2015) also introduces a new parameter called the smoothing coefficient or temperature. This coefficient is incorporated in the Softmax function, and its purpose is to control the entropy of the probabilistic distribution used by the student model, thus preserving not only the information of a given class but of all the others, briefly described as follows:

- A **low temperature** (below 1) makes the model more confident;
- A **high temperature** (above 1) makes the model less confident.

Given the logits z from a network, the class probability p_i of an image is calculated as:

$$p_i = \frac{\exp \frac{z_i}{p}}{\sum_j \exp \frac{z_i}{p}} \tag{2.23}$$

As in equation 2.23, p is the temperature parameter. When $p = 1$, we get the standard softmax function. As p increases, the probability distribution produced by the softmax function becomes softer, providing more information as to which classes the teacher found more similar to the predicted class (WANG; YOON, 2020).

2.1.7 Kullback–Leibler Divergence

The Kullback–Leibler divergence (KLD) shown in Equation 2.24 is widely used to measure the similarity between two distributions and plays an essential role in many applications (JI et al., 2022).

$$D_{KL}(P||Q) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{2.24}$$

2.1.8 Pruning

As discussed in 2.1.6, the high predictive performance of CNNs comes at the expense of high computational storage and processing values. Reducing the computational requirements of a neural model becomes critical for wider applicability (YEOM et al., 2021). The central idea of the pruning algorithm in neural networks is to eliminate unimportant or redundant weights or neurons that could be eliminated, thus reducing the network's size and keeping its original performance as much as possible. Pruning also helps in the process of avoiding overfitting a neural network (SRIVASTAVA et al., 2014). Two categories divide network pruning algorithms: unstructured and structured. Unstructured pruning is fine-grained, and its purpose is to cut off the unimportant weight connections in the pre-trained neural network. This result in sparse CNNs with irregularities, which usually require special software and hardware accelerators to speed up the inference speed (LIU et al., 2021). In contrast, structured pruning is coarse-grained and can altogether remove unimportant filters (LIU et al., 2021).

2.2 TensorFlow

The open source deep learning framework TensorFlow[1] is developed by Google. Its front-end supports multiple development languages such as Python, C++ and Java, while its back-end is written in C++ and CUDA (Zeng; Gong; Zhang, 2019). TensorFlow is a compelling and widely used platform. It provides various tools that facilitate the implementation of a model for Machine Learning applications in different areas. This framework is cross-platform and can be used on many different operating systems such as Windows, macOS and Linux. All computation involved in TensorFlow is expressed as a directed graph, where all computations are converted into nodes in this directed graph and the edges represent the flow of data between nodes in the

[1] TensorFlow: https://www.tensorflow.org/

graph (Zeng; Gong; Zhang, 2019). This tool is one of the most widely used machine learning systems operating at a large scale on heterogeneous systems.

Also available in this framework is the TensorFlow Lite package, which allows the implementation of ML models for resource-constrained devices such as Raspberry Pi, mobile devices and MCUs. TensorFlow Lite allows a pre-trained neural model to be converted and embedded in an edge device or a smartphone also. The basic implementation flow follows these steps:

1. Select and train a neural model for the desired task in the usual way. At this point, you can set the quantization-aware training, thus improving the optimization procedure;
2. Using the converter of the Tensor Flow platform, convert the trained model into a TFL readable version. The model generated after conversion has higher efficiency in terms of space occupied and, depending on the desired target device, in terms of accuracy;
3. As the model is finalized, it is possible to use optimization techniques such as quantization or weight pruning, reducing the computation requirement and making the neural model more agile during inference.
4. At the end, a tflite file containing the model ready to be implemented in the final device will be generated.

2.3 Espressif ESP32

Due to semiconductor industry increasingly competitive and complexity challenges, processor manufacturers are massively investing in a more compact, multi-purpose devices. One of these companies is Espressif. It is aims to occupy this vital slice of this cost-effective market. For that, Espressif developed a high performance MCU, entitled Espressif ESP32[2]. With prices starting at US$ 4.08[3], the ESP32 is a dual-core XTensa LX6 processor, operating at 160 or 240 MHz, using Harvard architecture and integrated wireless peripherals (Wi-Fi and Bluetooth).

The ESP32 is used in several IoT platforms, such as the ESP32 KITs[4] and SparkFun Thing [5]. Its Instruction Set Architecture (ISA) defines the operations that a processor or other peripherals support. It also provides a list of mnemonics that represent the machine codes used by the architecture. The XTensa implements a 24-bit ISA that, in turn, performs 32-bit operations (or 64-bit, provided FLIX mode[6]. It allows the XTensa processor to execute instructions in multiple forms, such as Very Long Instruction Width (VLIW) operations, the width of the

[2] Espressif: https://www.espressif.com/en/prod cts/socs/esp32
[3] Prices searched in https://www.digikey.com/
[4] Espressif Kits: https://www.espressif.com/en/products/devkits
[5] Sparkfun: https://www.sparkfun.com/products/13907
[6] Flexible Length Instruction eXtensions

instructions was chosen by the manufacturer with space-saving storage in mind. The XTensa ISA is organized as a basic instruction set with several additional packages that extend optional functionality, allowing the developer to maximize the solution's efficiency by including only the functionality needed for the project. An example of these additional packages is the MAC16 and Floating-Point Coprocessor options. MAC16 adds 40-bit accumulators and four 32-bit registers for use in multiplication functions. The Floating-Point Coprocessor option adds logic and structural components needed for floating-point operations following the standard defined by IEEE 754. The ISA default of XTensa is divided into ten categories: load, store, memory ordering, jump/call, conditional branch, move, arithmetic, bitwise logical, shift and processor control.

2.3.1 ESP32-S2

ESP32-S2 family is a highly-integrated, low-power, 2.4 GHz Wi-Fi System-on-Chip (SoC) solution. With its state-of-the-art, this SoC is an good choice for a wide variety of application scenarios relating to Internet of Things (IoT), wearable electronics and smart home (ESPRESSIF, 2020). At the core of this chip is an Xtensa® 32-bit LX7 CPU that operates at up to 240 MHz. The chip supports application development, without the need for a host MCU. The on-chip memory includes 320 KB SRAM and 128 KB ROM (ESPRESSIF, 2020).

2.4 GTSRB

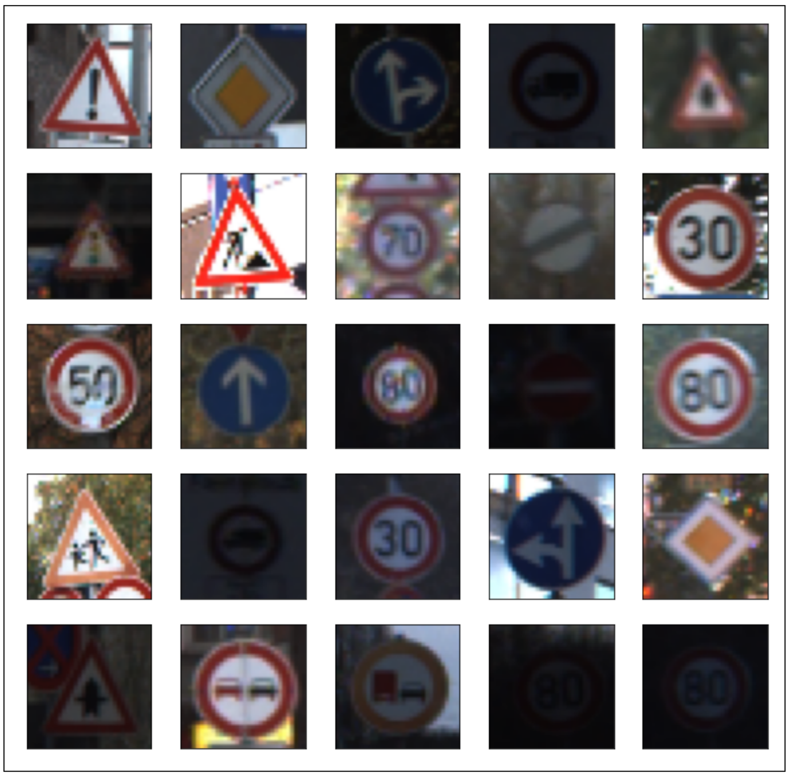
The German Traffic Sign Recognition Benchmark (GTSRB) (HOUBEN et al., 2013a) is a multi-class, single-image classification challenge held at the 2011 International Joint Conference on Neural Networks (IJCNN). This dataset has more than 50 thousand images, distributed in 43 classes (traffic signs). The GTSRB consists of colored images with resolutions from 15x15 to 250x250 pixels in PPM format (Portable PixMap).

Table 3 – Dataset images distribution.

Dataset	**No. of images**
Training	35209
Validation	4000
Testing	12630
Total	51839

The images of benchmark dataset have been selected from sequences recorded near Bochum, Germany, on several tours in spring and autumn 2010, capturing different scenarios during daytime and dusk featuring (HOUBEN et al., 2013a). All images in the dataset are converted to RGB color space. The dataset is divided into six subsets: speed limit, other prohibitory, derestriction, mandatory, danger, and unique signs. Figure 4 shows some images that make up this dataset.

Figure 4 – Some instances from the GTSRB training dataset.



The dataset authors used a Prosilica GC 1380CH camera with automatic exposure control and captured all images with a resolution of 1360x1024 pixels. These initial images received a preliminary manual annotation with data about the bounding boxes delimiting the area where the traffic signs appear.

2.5 CoreMark

It is a simple CPU benchmark developed in 2009 at EEMBC, and designed to test the core functionality of a processor. The objective of EEMBC is make it CoreMark an industry standard, replacing the Dhrystone benchmark.

To test the MCUs, Coremark uses implementations of the following algorithms: list processing (find and sort), common matrix manipulation, state machine (determine if an input stream contains valid numbers), and CRC (Cyclic Redundancy Check) (EEMBC, 2020). To ensure compilers cannot pre-compute the results at compile time, every operation in the benchmark derives a value that is not available at compile time. Furthermore, all code used within the timed portion of the benchmark is part of the benchmark itself (GALON; LEVY, 2008).

As a result of the evaluation, CoreMark produces a single score that allows for between platform comparisons (EMBC, 2021). The CoreMark/MHz metric indicates the performance of a single thread by the processor's clock frequency. To obtain the metric, we divide the CoreMark (single-core) benchmark score by the clock frequency used for the benchmark test. In Table 4 we can see the CoreMark benchmark results[7] for some microcontrollers used in surveyed papers.

Table 4 – This table lists some of the scores for registered MCUs at EEMBC database.

Processor	**MHz**	**Cores**	**CoreMark**	**CoreMark/MHz**
NVIDIA Tegra X1	1900	4	30638.54	16.13
Espressif ESP32	160	2	660.70	4.13
Xilinx XC7Z020 ARM Cortex-A9	800	1	2930.40	3.66
Espressif ESP32 S2	240	1	472.81	1.97
Atmel ATmega2560	8	1	4.25	0.53

[7] EEMBC CoreMark: https://www.eembc.org/coremark/scores.php

3 Related Works

In order to retrieve related works contributions and results to this proposed work, we will use experimental research method to conduct tests and investigate the acquired information. The purpose of this methodology is to establish a model that can be used as an example and applied in future studies.

3.1 Systematic Review

As a first step, a Systematic Review was conducted, inspired by the method presented by Kitchenham (2012), whose objective is to identify, analyze and interpret all available evidence on a research question in an unbiased and replicable manner. The following will discuss drafting the research questions, defining the search strategy, and selecting inclusion and exclusion criteria.

3.1.1 Methodology

Defining the problem, objective and hypothesis was the initial step to elaborating the systematic mapping for TSDR studies search. The next step in constructing methodology is preparing the research questions that will guide the study. It is defined as the keywords that will help in the characterization of the investigated theme and later will compose the search strings in the sequence the search bases will be selected and finally select the located studies by the defined selection criteria.

3.1.1.1 Research Questions

The purpose of the research questions is to obtain relevant data regarding the topic selected for study in the Systematic Review after selecting relevant studies. Preliminary, considering the objective of the work, we defined the research questions described in Table 5.

Table 5 – State of the Art Research Questions.

Question	**Expected data**
Q1) What hardware types are TSDR systems used?	It is hoped to collect information regarding the devices used in capturing, detecting, and recognizing a road sign.
Q2) What machine learning techniques do the TSDR/ADAS studies use?	The expected of this question is to gather data regarding the machine learning techniques used in TSDR/ADAS applications.
Q3) What are the metrics considered in evaluating the performance of the systems used in the localized work?	The objective is to collect information about the performance evaluation metrics used in the localized papers.
Q4) Which researchers use hardware acceleration (GPU, TSU, or FPGA)?	The objective is to gather information about which studies have used machine learning models with some hardware-based acceleration solution
Q5) What data sets do the studies use?	It is hoped to obtain information about the datasets used by the studies located.
Q6) What research uses compression techniques to reduce the size of neural networks?	The objective is to identify which studies use some compression technique for neural networks to enable data reduction and performance increment.

Source: The Author

3.1.1.2 Sources Used

The following factors defined the selection of the repositories used to fulfill the research questions:

1. The availability of studies written in English;
2. Bases registered in the CAPES platform;
3. Sources that have greater involvement with the research theme;
4. Possibility of querying for the search terms in specific areas.

In the list of selection criteria for search sources, Item 4 was applied to ensure that search engines always scour the same point of interest in the works, avoiding a cluttered search. Using the criteria defined above, the following repositories were selected for the present work: IEEE Xplore[1], ACM Digital Library[2] and Elsevier Scopus[3].

[1] IEEE Xplore: https://ieeexplore.ieee.org/Xplore/home.jsp
[2] ACM Digital Library: https://dl.acm.org/
[3] Elsevier Scopus: https://www.scopus.com/

3.1.1.3 Search Strings

After formulating the research questions and choosing the search sources, it is necessary to define studies selection strategy. The systematic review publication's final list is populated based on this strategy. Knowing that the process of defining the search string is iterative and involves much experimentation and verification, the following step-by-step was defined:

1. Perform preliminary searches in order to identify potentially relevant studies.
2. Evaluate the main keywords of these studies.
3. Define an initial search string based on the verified keywords, evaluate possible derivations, and gauge the effectiveness of each one.
4. Verify that the relevant primary studies, previously verified, were returned in the searches.

Based on the research questions raised and also considering preliminary research carried out as a way of initial marking, the following keywords were defined: traffic sign recognition, traffic sign detection, traffic sign tracking, traffic sign detection recognition, TSDR, traffic sign detection, and recognition, ADAS, advanced driver assistance systems and road sign detection.

After the initial evaluation of the search strings defined using the keywords chosen above and selecting those whose results eliminated as many unwanted results as possible, we arrived at the strings set out in Table 6.

3.1.1.4 Selection Criteria

We apply the selection criteria (inclusion and exclusion) in each publication from chosen research sources to qualify for the final primary studies list. As follows, we define the inclusion criteria (IC) and the exclusion criteria (EC) :

IC1) Only publications developed from 2016 on-wards will be selected;

IC2) Studies that implement TSDR/ADAS solutions on embedded systems or in embedded systems using acceleration or compression methods will be included;

IC3) The datasets used in the implementations, training, validations, or testing of the proposed solutions must be open source;

IC4) Works should have results of quantitative evaluations of the proposed solution;

EC1) Publications that do not meet the inclusion criteria will not be used;

EC2) Duplicate publications will not be included;

EC3) Works that do not provide sufficient information for comparison purposes will not be considered;

Table 6 – Strings of search defined for each repository after initial effectiveness evaluation.

Source	Search Strings
IEEE Xplore	("Abstract": "traffic sign recognition" OR "Abstract": "traffic sign detection" OR "Abstract":"traffic sign detection recognition" OR "Abstract":"TSDR" OR "Abstract":"traffic sign tracking" OR "Abstract":"adas" OR "Abstract":"advanced drive assistance systems" OR "Abstract": "traffic sign detection and recognition")
ACM Digital Library	[Abstract: "traffic sign recognition"] OR [Abstract: "traffic sign detection"] OR [Abstract: "traffic sign detection recognition"] OR [Abstract: "tsdr"] OR [Abstract: "traffic sign tracking"] OR [Abstract: "adas"] OR [Abstract: "advanced drive assistance systems"]
Scopus	ABS("traffic sign recognition" OR "traffic sign detection" OR "traffic sign detection recognition" OR "TSDR" OR "traffic sign tracking" OR "adas" OR "advanced drive assistance systems" OR "traffic sign detection and recognition")

Source: The Author

EC4) Publications with purely theoretical content will not be selected;

EC5) Works are written in languages other than English or Portuguese will not be considered;

EC6) Will not be considered publications that do not apply machine learning-based solutions.

3.1.2 Selecting Studies

With the selection of the primary studies finalized, we define the information that will be extracted from each publication as follows:

- Type of hardware used;
- Techniques or algorithms implemented for the detection and recognition of traffic signs;
- Metrics considered to evaluate each proposed solution;
- Type of solution used for data acceleration or compression, if any;
- Dataset chosen for the tests.

3.1.3 Results

We will now discuss the results obtained during the Systematic Review. Later, we will also discuss the answers to the research questions. In Table 7 are number of publications located by repository applying the search strings defined in the sub-section 3.1.1.3. In the original Scopus search string, we embed four additional filters. This change aims to remove results referring to unrelated areas and book chapters. All searches occurred in the time range starting in the year 2015 through the year 2020. The search in repositories occurred on January 6, 2020.

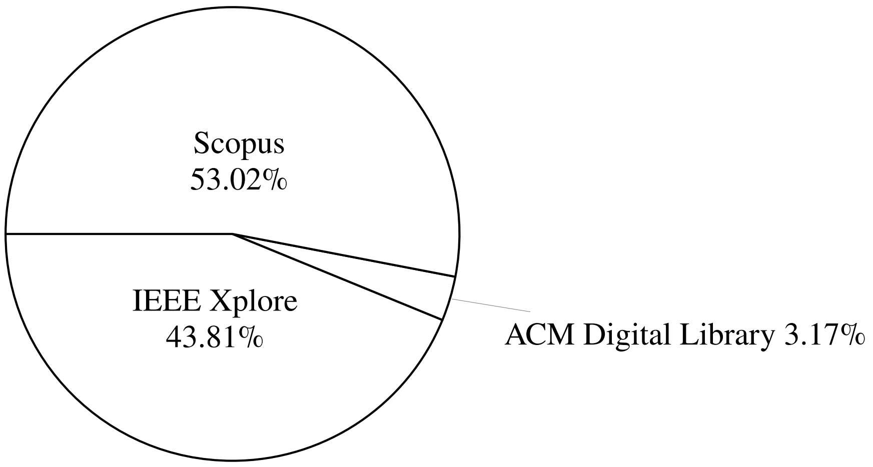
Table 7 – The quantitative result obtained from applying the search strings.

Source	**Results**
IEEE Xplore	1923
ACM Digital Library	113
Scopus	4465
Total	6501

Source: The Author

The next step after the initial survey is the application of the selection criteria defined in the Section 3.1.1.4. In Figure 5 we can verify the number of results obtained after applying the search strings in each repository. In this step, we established 2016 as the initial year for search in repositories. As can be seen, the ACM Digital Library source was the one that returned the minor results, providing only 3.17% of the publications evaluated. After selecting the primary studies, we apply the inclusion and exclusion criteria. For this, we analyzed the paper titles, abstracts, and the entire content.

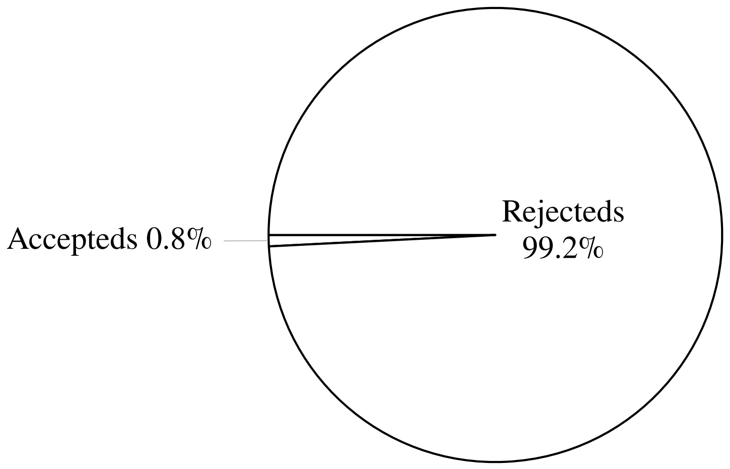
Figure 5 – Results distribution after applying the search strings for each repository.



Source: The Author.

Figure 6 illustrates the distribution of accepted and rejected studies after selection. At this stage, the studies considered rejected correspond to duplicate studies or studies that at some point met some exclusion criteria. The studies considered accepted those that passed the selection based on the inclusion criteria. In the end, only 19 studies remained, corresponding to less than 1% of the total number of primary studies.

Figure 6 – Distribution of the results obtained after applying the inclusion and exclusion criteria.



Source: The Author.

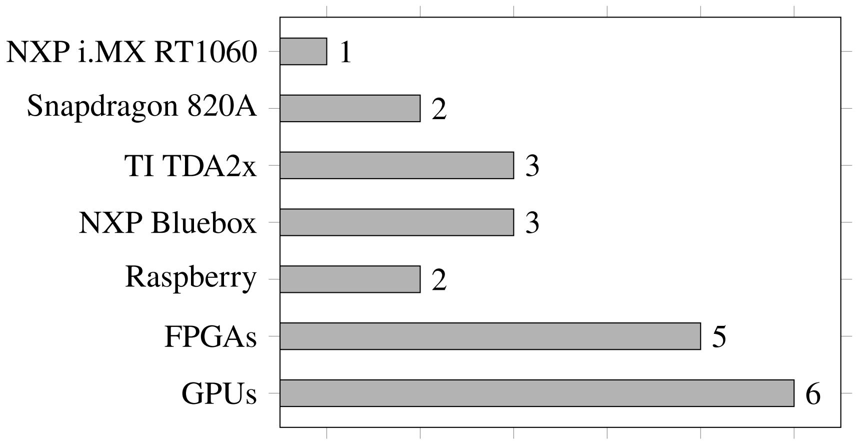
3.1.4 Analysis

After the selection of the studies is finalized, applying the data extraction protocol begins so that the answers to the found research questions. Figure 7 displays the hardware distribution used to embed the proposed solution in each study. Overall, the number of citations of a given platform is higher than the number of publications found. That occurs because, in some works, the described proposal uses more than one platform to implement the final solution or makes comparisons between different hardware. The extensive use of GPU-based solutions, 28.6%, was already expected since it is possible to implement complex algorithms with reduced inference time.

The metrics for evaluating the results serve as a reference for comparison between selected works and also help define the present work's evaluation method. During evaluation of the SR studies, the metrics used were: execution time in milliseconds (ms), evaluation capacity of frame rate (FR), accuracy, precision-recall curve (Area Under the Curve - AUC), latency, GMAC per second (Billion (Giga) Multiply-And-Accumulate), Joules consumed per processed frame, mAP (mean Average Precision), Giga FLOPS and model size in bytes.

At Figure 8 is seen that 73.7% of the used classifiers are CNN's type. The result is unsurprising since ConvNets have been applied with great success to the detection, segmentation, and recognition of objects and regions in images (LOPEZ-MONTIEL et al., 2020). Companies

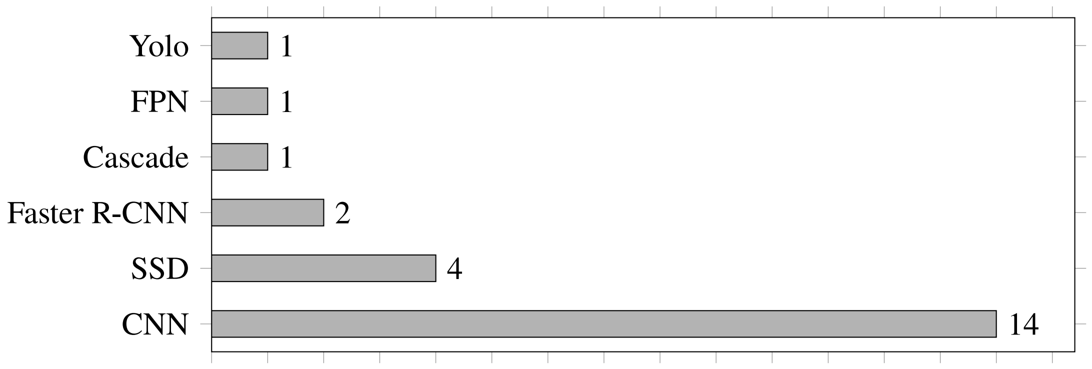
Figure 7 – Frequency of devices in traffic sign recognition systems.



Source: The Author

such as Mobileye[4] and NVIDIA are using such CNN-based methods in their upcoming vision systems cars (LECUN; BENGIO; HINTON, 2015).

Figure 8 – Frequency of classifiers in selected works.



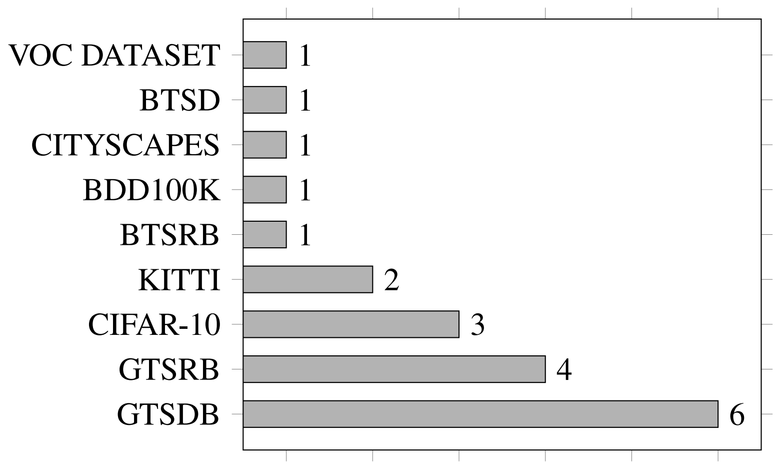
Source: The Author

Figure 9 presents the distribution of the datasets used by the selected publications. As can be observed, the most used dataset is the GTSDB (HOUBEN et al., 2013b) with six occurrences, then we have the GTSRB (STALLKAMP et al., 2012) base with four occurrences. Although the CIFAR-10 (KRIZHEVSKY; NAIR; HINTON, a) and the road environment have no connection, three related papers use this dataset. KITTI is a dataset assembled from the Annieway platform (FRITSCH; KUEHNL; GEIGER, 2013). The primary aim is to serve as a computer vision benchmark, and two located studies use them. The BTSR (Timofte; Zimmermann; Van Gool, 2009), BDD100K[5] (YU et al., 2020), Cityscapes (CORDTS et al., 2016), BTSD (TIMOFTE; ZIMMERMANN; GOOL, 2009) and Pascal VOC (EVERINGHAM et al., 2015) had only 1 occurrence each. From the point of view of hardware acceleration, only one study used it. Three

[4] Mobileye: https://www.mobileye.com/
[5] A Diverse Driving Dataset for Heterogeneous Multitask Learning: https://www.bdd100k.com/

works presented solutions for the compression of neural networks for allocation in embedded devices.

Figure 9 – Frequency of datasets in selected works.



Source: The Author

3.1.5 Analysis of Selected Publications

In this subsection, we will analyze each paper related to traffic sign detection techniques using machine learning algorithms and ported to embedded systems. We will present a brief introduction to the content and a general description of each proposed work. These papers met the Inclusion Criteria defined in subsection 3.1.1.4.

3.1.5.1 Optimization for object detector using the deep residual network on embedded board

Lee et al. (2016) proposed optimizing a neural model for road object detection in terms of processing time and memory consumption. They implemented a Single-Shot Multibox Detector (SSD) object detector, a VGGNet-based neural network, and a multi-box classifier, using feature maps at various scales.

The authors worked on reducing the complexity of the VGGNet network so that they obtained a neural model with five times fewer parameters and three times less memory consumption compared to the original model. They got this using a smaller residual network that performs fewer computations per interaction, achieving a higher inference speed than the standard VGGNet network. Lee et al. (2016) embedded the proposed model in a NVIDIA dev board Jetson TX1 which has a 1024 GFLOPs Maxwell GPU integrated with an ARM Cortex-A57. On this testbed, the proposed model achieved a mark of 0.094 seconds for each inference with an average accuracy of up to 85%, ten times faster than the original VGGNet model. The dataset chosen by the authors was KITTI.

3.1.5.2 Traffic sign recognition based on the NVIDIA Jetson TX1 embedded system using convolutional neural networks

Han e Oruklu (2017) used the NVIDIA Jetson TX1 with a USB camera to implement a TSR system based on the OpenCV4 Tegra and Theano libraries. The authors perform traffic sign detection in this work by applying shape, and color-based segmentation since traffic signs have well-defined colors and shapes.

Given any RGB image, applies normalization in the picture, and using morphological filters, the pixels sets that may contain occurrences of traffic signs are selected. In sequence, color detection removes speckles and any image noise. After delimiting the Region of Interest (ROI), the CNN receives this slice containing the candidate blobs for the classification step. The neural model proposed in this paper is based on the LeNet network, using 40-by-40 pixel color images as input. Han e Oruklu (2017) use a set of three convolutional layers intercalated with three max-pooling layers. The authors used three fully connected layers, the first two applying the activation function tanh and the output layer using the function softmax. The GTSRB dataset has been used to train the proposed neural model,

The proposed system used accuracy as the standard metric. In tests using the GTSRB and the neural model defined in the paper, the authors achieved an accuracy of 96.2%. The embedded system took an average of 0.67 seconds to process a single image with 1360x800 resolution.

3.1.5.3 An FPGA-Based Hardware Accelerator for Traffic Sign Detection

The system proposed by Shi et al. (2017) used a Xilinx ZC706 FPGA as a hardware acceleration medium for traffic sign detection. The architecture defined by the authors was implemented in four main blocks: an accelerator for the detection, the CPU, a DMA bus, and external DRAM memory. The accelerator, CPU, and DMA are all on the Xilinx chip. The DRAM exists to store the video frames and transfer them to the accelerator for detection. The authors use three-stage cascaded classifiers for the classification step.

Shi et al. (2017) design aims to facilitate and accelerate the feature extraction and object detection steps. To this end, the proposed approach takes advantage of the critical property that several image blocks processed by a classifier often overlap. Therefore, several identical pixels can be used by feature extraction for these image blocks. Between the second and third stages, the authors inserted data buffers. The objective is to avoid unnecessary workload in the last stage of the cascade classifier. To reduce power consumption, they implemented an adaptive workload distribution. The implementation uses a priority comparator, which determines the "priority" of different processing units; a multiplexer, responsible for managing the control signals; and two buffer trackers, which monitor the read and write signals of a buffer, thus tracking the available memory space.

In the experiment, cascaded classifiers use three datasets to train the STOP traffic sign to recognize: BTSD, GTSRB, and GTSDB. Once trained, the classifiers achieved a true detection rate of 99.80% and a false detection rate of 0.013%. Compared to CPU and GPU implementations, the authors' model achieves 17.25 times and 5.61 times more processing speed, respectively. Comparing the energy efficiency of these three implementations, the proposed hardware accelerator reduces power consumption by 252.44 times compared to the CPU implementation and 42.68 times compared to the implementation on a GPU. The accelerator achieves throughput of 126 FPS with energy efficiency of 0.041 Joules/frame.

3.1.5.4 FPGA-based convolution neural network for traffic sign recognition

The authors Yao et al. (2017) propose a CNN-based topology suitable for implementation in FPGA. In addition, optimization strategies were applied to improve the utilization of the neural network parameters, thus reducing the complexity of the network. The four strategies were: compression of the convolutional layer parameters, replacement of the fully connected layer, use of the MLP (Multilayer Perceptron) model for the convolutional layer, and carry gate.

Adjustments in AlexNet original kernel size allowed to reduce the number of parameters by 58.7%. Regarding the fully connected layer, the authors replaced it with a layer called "Global Parameterization," located after the last convolution layer, and the number of feature maps is equal to the number of categories (classes). A feature layer value equals the sum of products of all features and the weight in the corresponding feature map. Unlike the fully connected layer, the element in this layer is connected to one instead of all feature maps, causing the number of parameters to decrease substantially. The MLP convolution layer combines a standard convolution layer and two convolution layers whose size of kernel is 1×1. MLP convolution layer can introduce a more nonlinear relationship into the network than the traditional convolution layer. With this, the flexibility and accuracy of the model can be increased. In addition, DSP in FPGA may quickly generate the 1 x 1 convolution results, making implementing MLP convolution layers more efficient. The carry gate can give the network a better descriptive capability, increasing the module accuracy. They work as shortcut connections between the input and output of a given layer.

In work the GTSRB dataset and the framework Caffe[6] were used. The accuracy rate on training and testing was 99.7 percent and 98.1 percent, respectively. Compared to the original AlexNet model, the workload was 61.4% lower, and the final model has only 4.7 million parameters, 12.5% less than the original model.

3.1.5.5 Simultaneous Traffic Sign Detection and Boundary Estimation Using Convolutional Neural Network

Lee e Kim (2018) proposed a novel and efficient method for traffic sign detection. Using convolutional neural networks, this work not only predicts the class but also accurately estimates

[6] Caffe - Deep learning framework: https://caffe.berkeleyvision.org/

the boundaries of the detected object, making the proposed model quite robust on images with occlusion and objects of small size.

Based on the SSD framework, the proposed method performs the predictions on multi-level feature maps. The evolution over the other methods is that instead of predicting the bounding box (bounding box that indicates the location of the detected object in the image), the article's network performs pose estimation. An input image is passed and processed on an initial CNN (e.g., VGG16 or Inception V2), extracting feature maps using a series of convolutions, nonlinear activations, and clustering operations. Then, from the obtained feature maps, a class's 2D poses and shape probabilities are estimated by two separate convolutional layers, i.e., a pose regression layer and a shape classification layer, combined with successive operations that convert the convolution outputs to 2D pose values and class probabilities, respectively. The proposed network was trained on an NVIDIA Xavier and implemented on a Qualcomm Snapdragon 820A (an automotive processor with Adreno GPU). The dataset used was the GTSDB used framework Caffe.

Because it needs information regarding the positioning of a bounding polygon around the object for training, the method cannot have its accuracy compared to "ordinary" datasets. Using the VGG16 network for the basic block of the proposed method, the authors used the Average box Overlap (AO) and the Area Under precision-recall Curve (AUC) as metrics to evaluate the efficiency of the model on the GTSDB. Considering the AO metric, the model obtained the best results compared to the methods, wgyHIT501 (WANG et al., 2013), visics(MATHIAS et al., 2013), LITS1(LIANG et al., 2013), and BolognaCVLab(SALTI et al., 2013). The authors used the dataset SDTS with 6324 annotated traffic signs for accuracy evaluation. The mAP is calculated for each IoU (Intersection over Union) value. IoU is the intersection value between the calculated and standard bounding boxes. To be considered a positive prediction, the IoU value must be greater than 0.5. Using VGG16 e Inception V2 nets in this dataset with 0.5 and 0.7 IoU values, the model obtained the mAP above 0.8.

3.1.5.6 Traffic Sign Recognition with Light Convolutional Networks

In this paper, Wu et al. (2018) designed a light convolutional network (SAFENet) from a modified VGG network that can be run on an embedded system in real-time using RGB images converted to grayscale as input. The authors reduced the size of the filters applied in the convolutional layers to compress the neural network and speed up the inference. The proposed model has three convolutional blocks, with 32, 64 and 128 filters. On the first and second pooling layers Wu et al. (2018) added feature maps. Before the fully connected layer, all maps are concatenated. With these changes, the SAFENet obtained an accuracy of 99.34% and an inference time of 4.58 ms per image, running on the NVIDIA hardware embedded Jetson TX1. The dataset used was GTSRB.

3.1.5.7 Autonomous Embedded System Enabled 3-D Object Detector: (with Point Cloud and Camera)

Katare e El-Sharkawy (2019) propose an architecture for 3-D object detection from the LIDAR (Light Detection And Ranging) sensors and cameras installed in a vehicle, connected to a vehicular embedded system, in this case, NXP's Bluebox 2.0 platform.

The architecture in question uses Frustum-PointNet (QI et al., 2018) as the base model for acquiring data from the LIDAR sensor. In addition, the model also contains a Segmentation-PointNet and a Regressed PointNet. The proposed model uses features extracted from the image to estimate the position of the 3D bounding boxes, adding a block to explore the global features of the LIDAR. Thus, the architecture merges a block based on Frustum-Pointnet and a second block based on the ResNet architecture to evaluate the bound box referring to objects captured by the sensors. The KITTI dataset used in the work has an instances distribution as follows: 70% for the training set and 30% for the evaluation set. The final architecture achieved an accuracy of 79.80% on images considered easy, 65.83% on images of medium difficulty, and 62.71% on complex images. The data obtained is slightly lower than that of the standard Frustum-PointNet model.

3.1.5.8 ResCoNN: Resource-Efficient FPGA-Accelerated CNN for Traffic Sign Classification

Lechner, Jantsch e Dinakarrao (2019) show an efficient CNN (ResCoNN) architecture using few resources and with a small number of weights (only 60,000 compared to a few million state-of-the-art CNNs), and employ it in traffic sign detection and classification. For efficiency, the network takes advantage of binary weights and integer activation rather than employing complex computations such as batch normalization and exponential linear units. The ResCoNN model can achieve a classification accuracy of over 96% on real-world images at a frame rate of 36 FPS on a Zynq SoC (xc7z020clg484-1) with 90% of reduction in the number of weights compared to state-of-the-art CNNs.

To binarize the weights, the authors followed the strategy presented by Courbariaux et al. (2016). By using the binary representation [-1,1], the network structure does not need to be changed, storing the weights still as full precision variables to enable gradient-based weight optimization, but for the forward and backward propagation steps, the weights are binarized as follows:

$$w^b = sign(w) = \begin{cases} +1 \text{ if } w \geq 0 \\ -1 \text{ else} \end{cases} \tag{2}$$

The dataset used was a modified version of GTSDB, adding some images of the Belgium Traffic Sign for testing.

3.1.5.9 Real-Time Object Detection On Low Power Embedded Platforms

Jose et al. (2019) used the TDA2PX System-on-Chip (SoC) and the TIDL (TI Deep Learning) library in their work. The first strategy of the paper was to explore existing lightweight sensing networks such as MobiliNets, SqueezeDets, and ShuffleNets. However, these networks proved slow on the platform, varying latencies from 103.23 ms to 559.18 ms. Therefore, since much of the complexity of the network is in the initial part due to the large filter sizes, the authors propose: (i) balancing the spatial dimensions and the depth of the channels through all layers of the network; (ii) using sparse convolutions, setting the weights to zero below a threshold that is dynamically determined based on the range of weights; and (iii) using fixed 8-bit representation quantization.

Comparing the proposed model (HX-LPNet) with the JDetNet model, the proponents observe a six-fold reduction in required computations. The Dense JDetNet model has parameters of 3.15 M, while the dense HX-LPNet has only 0.42 M. In the sparse version, HX-LPNet achieved 22.47 FPS on TDA2PX with a latency of only 0.2 ms. The dataset used was the Berkeley DeepDrive Dataset (BDD100K).

3.1.5.10 Traffic Signs Detection and Recognition System using Deep Learning

William et al. (2019) implements a TSDR system on a Raspberry Pi 3 Model B+ device. The dataset defined for use by the authors was GTSDB. The images employed are loaded in RGB mode and then converted to HSV color space. Only then is it then submitted to the neural network for ROI stipulation. The network proceeds to traffic signal classification with the area of interest detected. To accomplish this script the authors used the F-RCNN Inception V2 and Tiny-YOLO v2 models. The F-RCNN consists of a fast R-CNN detector and a Region Proposal Network (RPN), and then the Non-Maximum Suppression (NMS) is applied to choose the best region. NMS is a computer vision method that selects a single entity from multiple overlapping bound boxes using the discard criterion for any entity whose IoU is below 0.5. With this model, the accuracy obtained was 96%. The second model used was Tiny-Yolo V2, a real-time object detection model based on the Yolo V2 model, designed to run on handheld devices without an available GPU. The reduced neural network size goes to the FPS rises from 40 to 244, but its mAP drops from 48.1 to 23.7. Running on the platform proposed in the article, the authors obtained with Tiny-Yolo V2 an average accuracy of 73

3.1.5.11 Traffic Sign Identification Using Deep Learning

The focus of this paper aims at selecting a Deep Neural Network (DNN) approach for traffic sign detection and classification to achieve a balance between accuracy, measured based on mAP, and execution time, measured in FPS, for a real-time application with a maximum vehicle speed of 5 miles per hour, which is a requirement of the Intelligent Ground Vehicle Competition[7]

[7] The Annual Intelligent Ground Vehicle Competition: http://www.igvc.org

(IGVC) competition.

The model proposed by Ravindran et al. (2019) implements the Faster R-CNN Inception V2 algorithm to detect and classify traffic signs for real-time testing. The paper uses the German Traffic Sign Detection Benchmark (GTSDB) dataset. The authors use the Data Augmentation technique to resolve the unbalanced classes problem. In order to raise the accuracy in detecting sure traffic signs, the authors have implemented cross-referencing of data from an Optical Character Recognition (OCR) called Tesseract (an artificial recurrent neural network of Long Short-Term Memory - LSTM), with the data obtained by the neural network classification. By embedding the Faster R-CNN network with the suggested changes into an NVIDIA Drive PX2 embedded system the authors obtained a mAP of 90.62.

3.1.5.12 Image Classification on NXP i.MX RT1060 using Ultra-thin MobileNet DNN

In Desai, Sinha e El-Sharkawy (2020), the authors used the Design Space Exploration technique to modify the basic MobileNet V1 model and develop a more space-efficient and faster inference speed version. The paper proposes seven modifications to the MobileNet V1 baseline. These are: (i) separable convolution layers; (ii) erasing areas in the images and using random filling, increasing the generalization ability of the model; (iii) eliminating the layers with the same output shape in order to reduce the size of the model; (iv) reducing the depth of the last convolutional block, thereby reducing the number of parameters in this block by 60%; (v) use of the Swish activation function instead of ReLu, this increases the accuracy concerning the baseline model; (vi) definition of a width multiplication factor, this factor (value between 0 and 1) works as a reducer of the number of input and output channels making the model smaller as it approaches zero; (vii) choice of the Nadam optimizer, which combines the effects of the RMSProp optimizer, the Adam optimizer and Nesterov's momentum.

The proposed model remained small enough to be embedded in the NXP i.MX RT1060 board and efficient, with the accuracy at 84.32% and only 0.3 million parameters. The size of the neural model is 3.9 MB. The dataset used in the experiments was CIFAR-10.

3.1.5.13 Real-time Implementation of RMNv2 Classifier in NXP Bluebox 2.0 and NXP i.MX RT1060

Using the Reduced Mobilinet V2 (RMNv2) model trained to the CIFAR-10 dataset and only 4.3 MB in size, the authors Ayi e El-Sharkawy (2020) successfully implemented an image classifier at the NXP i.MX RT1060 embedded board and the NXP Bluebox 2.0. Implementing the RMNv2 classifier on the NXP i.MX RT1060 involved two steps: first, converting the RMVv2 model to the TensorFlow Lite model and deploying that TensorFlow Lite model on the board.

3.1.5.14 Convolutional Neural Network based Traffic-Sign Classifier Optimized for Edge Inference

Shabarinath e Muralidhar (2020) propose a VGGNet-based CNN architecture for traffic signal image classification using the GTSDB dataset on an FPGA. The base model consists of 3 convolutional blocks, each consisting of two convolutional layers with the ReLu activation function, followed by a pooling layer. In the end, three fully connected layers perform the classification work. In this dataset, it was possible to obtain an accuracy of 98% in the tests. The paper addresses some image preprocessing problems related to traffic sign images. The trained model is optimized using combined 50% pruning of the weights with post-training quantization techniques. The pruned model reached an accuracy loss of less than 1% after retraining. This optimization reduces the memory consumption of the model so that it can be embedded on the FPGA platform. For the base model, the test accuracy was 98%, the base model with pruning had an accuracy of 97.5%, and the model with pruning and quantization had an accuracy of 96.5% in the tests performed. Using the techniques mentioned above, the number of weights needed in the forward pass of the optimized model was reduced by approximately 69%, to 40805 weights.

3.1.5.15 A Light Weight Multi-Head SSD Model For ADAS Applications

In Lai et al. (2020) a lightweight Single Shot Detector (SSD) type detector is proposed for the task of recognizing moving objects, such as vehicles, pedestrians, and bicycles). The standard SSD method has two components, a backbone and a head. The backbone is a pre-trained image classification network as a feature extractor, with the fully connected layer removed. The head is one or a convolutional layer set added to this backbone. Its outputs are interpreted to be the bounding boxes and object classes.

In this method, more heads are added to the SSD model to fill the gaps between adjacent anchors, populating the corners of the grid cells with more anchors. To implement this, the authors added 3x3 convolutional layers in the feature detector head to generate mixed features. This change combines the information from adjacent cells to obtain the mixed features at the corner positions. It is correct to say that adding multiple heads increases the computational cost, but the gains are also significant. One change also made was to modify the original cross-entropy loss function to a focal loss function. Due to a slight class disproportion in the training dataset, the authors implemented that, inducing poor detection.

The two embedded platforms used to evaluate the concept developed in the Lai et al. (2020) paper were: the iCatch Inc. V37 and the TDA2X. Both platforms have accelerators. Evaluating images with 512x256 resolution, the iCatch V37 reached a speed of 30 FPS. TDA2X, on the other hand, was able to analyze images at 27 FPS. At this exact resolution, the mAP reached 74.35%, and the total network parameters were 4.62 million. The datasets used were: Cityscapes, CreDa, and VOC 2007.

3.1.5.16 Traffic Signs Recognition System with Convolution Neural Networks

Using a convolutional neural network embedded in a Raspberry Pi 3 and the dataset BTSD, Sisido et al. (2018) obtained an accuracy of 96% on the task of detecting traffic signs. A webcam was attached to the Raspberry Pi 3 and assembled on a robotic platform that would traverse a field with the posted traffic signs. The approximate processing and inference time was 4.9 seconds.

3.1.5.17 Real-Time Traffic Sign Recognition using Deep Network for Embedded Platforms

In Nagpal et al. (2019), the authors implemented a two-stage neural network for traffic sign detection. The first stage is responsible for detecting the location of the traffic sign in a given image. The detector was pre-trained on the Pascal VOC dataset. Then it underwent training on BTSDB and GTSRB, ensuring the model has good generalization. To train the classifier Nagpal et al. (2019) use the GTSRB dataset with the data augmentation technique. The advantages of this approach are: (i) since the first stage was trained with segmentation and classification data from multiple datasets, it becomes a robust segmenter and avoids the need for retraining for other environments; (ii) it offers a computational advantage, since it is not necessary to apply, for example, the use of all the bound boxes present in the SSD (more than 8 thousand possibilities), since, as the localizer received classification data in training, its output will be more refined; (iii) by separating the classifier from the detector complexity, it is possible to apply binarization more easily.

Nagpal et al. (2019) selected the local distillation method to reduce the network size without affecting accuracy. After this step, they prune the model. Finally, fixed-point quantization is applied to the model, making it possible to embed the network in a constrained device. The authors evaluated the model on the NVIDIA Tegra X2, Snapdragon 820A, and the Texas TDA2X. Implemented on the NVIDIA Tegra X2, the pruned-only model achieves an execution time of 42 ms per inference. On the TDA2X, the pruned and quantized model was shipped, achieving 49.5 ms in inference execution. On the Snapdragon 820A, also using the pruned and quantized model, it reached 48.5 ms in execution time.

3.1.5.18 Squeeze-and-Excitation SqueezeNext: An Efficient DNN for Hardware Deployment

Convolutional neural networks are being used in autonomous driving vehicles or ADAS systems and have succeeded. Chappa e El-Sharkawy (2020) have proposed an architecture based on SqueezeNet, Mobilenet, and SqueezeNext. The architecture contains basic blocks organized in a 4-stage configuration called bottleneck modules, a compression and excitation block (Squeeze-and-Excitation - SE), a medium pooling layer, a fully connected layer, and a spatial resolution layer. This paper implement Nesterov momentum and learning rate decay with the SGD optimizer. To reduce the computational cost of the network, two multiplication factors, Resolution Multiplier (RM) and Width Multiplier (WM) were defined to reduce the internal representation of each layer. RM is applied to the input image and WM to the layers to reduce the number of total parameters. Evaluating the architecture on the CIFAR-10 dataset, we observed an accuracy of 92.60% using WM = 2 (model with 6.59MB) and 86.71% with WM = 0.5 (model with 0.595MB). The inference time starts from 21 seconds (WM=2) and drops to 10 seconds (WM=0.5).

3.1.5.19 Mobilenet-SSDv2: An Improved Object Detection Model for Embedded Systems

Chiu et al. (2020) propose a lightweight object detection model developed from the MobileNet V2 network. To improve the accuracy of the object detection network model, the inverse residual module technology in Mobilenet-v2 and FPN architecture was employed to improve the performance of the proposed object detector, named Mobilenet-SSDv2. Experimental results show that the proposed Mobilenet-SSDv2 detector achieves an accuracy rate of 75.9% mAP on the Pascal VOC test suite and a processing speed of 19 FPS running on the NVIDIA Jetson AGX Xavier platform. Moreover, the memory usage of the proposed detector is only 32 MB, which is helpful in implementing ADAS on embedded devices.

3.1.6 Comparative Analysis

The works assessed during Systematic Review were summarized and can be seen in Table 8. Unlike the publicationsresco verified during the SR, the present work intends to use low-cost and resource-constrained hardware and apply compression and quantization techniques to improve efficiency during neural network inference.

Table 8 – Comparison of the selected papers in Systematic Review.

Author	Hardware	Technique	Metric	Dataset	Compression/Quantization
(Lee et al., 2016)	NVIDIA Jetson TX1	CNN	Execution Time/FPS	GTSRB	-
(Han; Oruklu, 2017)	NVIDIA Jetson TX1	CNN	Accuracy	KITTI	-
(Shi et al., 2017)	FPGA Xilinx ZC706	Cascade Classifier	FPS/Joule-Frame/Accuracy	BTSRB/GTSR-B/GTSDB	-
(Yao et al., 2017)	FPGA Xilinx Zynq-7000	CNN	Accuracy	GTSRB	-
(Lee; Kim, 2018)	SnapDragon 820A	CNN	Precision-Recall Curve (AUC)	GTSDB	-
(Wu et al., 2018)	NVIDIA Jetson TX1	Light-CNN	Accuracy	GTSRB	-
(Katare; El-Sharkawy, 2019)	NXP BlueBox 2.0	CNN	Accuracy	KITTI	-
(Lechner; Jantsch; Dinakarrao, 2019)	FPGA Zynq Platform	CNN (ResConNN)	Accuracy	GTSDB	-
(Jose et al., 2019)	TI TDA2PX	SSD	FPS/Latency/GMACs/mAP	BDD100K	Sparse Convolutions and Quantization
(William et al., 2019)	Raspberry PI Model B+	F-RCNN/SSD/Yolo v2	Accuracy	GTSDB	-
(Ravindran et al., 2019)	NVIDIA Drive PX2	F-RCNN	mAP/FPS/GigaFLOPS	GTSDB	-
(Desai; Sinha; El-Sharkawy, 2020)	NXP i.MX RT1060	Ultra Thin MobileNet	Accuracy	CIFAR-10	-
(Shabarinath; Muralidhar, 2020)	FPGA Zynq Platform	CNN	Accuracy	GTSDB	Pruning and Post-Training Quantization
(Lai et al., 2020)	iCatch Inc. V37/TDA2x	Light Weight Multi-Head SSD	mAP/FPS	Cityscapes/VOC 2007/CreDa	-
(Sisido et al., 2018)	Raspberry Pi 3	CNN	Accuracy	BTSD	-
(NAGPAL et al., 2019)	TDA2x/SnapDragon 820A/NVIDIA Tegra X2	CNN/SSD	Accuracy	GTSDB	-

(Chappa; El-Sharkawy, 2020)	NXP Bluebox 2.0	CNN	Accuracy	CIFAR-10	Channel Reduction/Separable convolutions
(Chiu et al., 2020)	NVIDIA Jetson AGX Xavier	Mobilinet-v2 combined w/FPN	mAP/FPS/Size	VOC Dataset	-
(Lee et al., 2016)	NVIDIA Jetson TX1	CNN	Execution Time/FPS	GTSRB	-
(Han; Oruklu, 2017)	NVIDIA Jetson TX1	CNN	Accuracy	KITTI	-
(Shi et al., 2017)	FPGA Xilinx ZC706	Cascade Classifier	FPS/Joule-Frame/Accuracy	BTSRB/GTSR-B/GTSDB	-
(Yao et al., 2017)	FPGA Xilinx Zynq-7000	CNN	Accuracy	GTSRB	-
(Lee; Kim, 2018)	SnapDragon 820A	CNN	Precision-Recall Curve (AUC)	GTSDB	-
(Wu et al., 2018)	NVIDIA Jetson TX1	Light-CNN	Accuracy	GTSRB	-
(Katare; El-Sharkawy, 2019)	NXP BlueBox 2.0	CNN	Accuracy	KITTI	-
(Lechner; Jantsch; Dinakarrao, 2019)	FPGA Zynq Platform	CNN (ResConNN)	Accuracy	GTSDB	-
(Jose et al., 2019)	TI TDA2PX	SSD	FPS/Latency/GMACs/mAP	BDD100K	-
(William et al., 2019)	Raspberry PI Model B+	F-RCNN/SSD/Yolo v2	Accuracy	GTSDB	-
(Ravindran et al., 2019)	NVIDIA Drive PX2	F-RCNN	mAP/FPS/GigaFLOPS	GTSDB	-
(Desai; Sinha; El-Sharkawy, 2020)	NXP i.MX RT1060	Ultra Thin MobileNet	Accuracy	CIFAR-10	-
(Shabarinath; Muralidhar, 2020)	FPGA Zynq Platform	CNN	Accuracy	GTSDB	-
(Lai et al., 2020)	iCatch Inc. V37/TDA2x	Light Weight Multi-Head SSD	mAP/FPS	Cityscapes/VOC 2007/CreDa	-
(Sisido et al., 2018)	Raspberry Pi 3	CNN	Accuracy	BTSD	-
(NAGPAL et al., 2019)	TDA2x/SnapDragon 820A/NVIDIA Tegra X2	CNN/SSD	Accuracy	GTSDB	-

(Chappa; El-Sharkawy, 2020)	NXP Bluebox 2.0	CNN	Accuracy	CIFAR-10	-
(Chiu et al., 2020)	NVIDIA Jetson AGX Xavier	Mobilinet-v2 combined w/FPN	mAP/FPS/Size	VOC Dataset	-

Source: The Author

4 Network Compression Pipeline

According to the information provided in Chapter 3, several works evaluate the use of neural networks for traffic sign recognition and the subsequent embedding of these neural models into embedded devices with considerable results, such as Lee et al. (2016) and Chiu et al. (2020). However, none of these selected works use a resource-constrained device for this task. Aware of this, this present work proposes the implementation of an optimized neural network for use in low-end devices.

4.1 Methodology and Proposed Pipeline

Regarding development and experimentation environment, we chose the TensorFlow library as the official library for the development of our work because it is open source and has extensive documentation regarding its use. Also due to be open source and supported on Google Colab[1] which is an environment that allows anyone to write and execute arbitrary Python code from the browser and is especially suited for machine learning. This choice was made because of the possibility of using state-of-the-art hardware for training our neural networks. The programming language used in the pipeline was Python[2] version 3.7. The available hardware for running the model in the Google Colab is a Tesla P100 GPU with the respective driver version 460.32.03 and 16 GB of RAM available, running TensorFlow framework version 2.8.2. In the environment for programming and installing the model on embedded device, we use the IDE Visual Studio Code (VSCode) in version 1.69.2. As an extension of VSCode, the PlatformIO plugin provide us an integrated, user-friendly, and extensible development environment for embedded devices. This extension creates an Arduino-like environment that will facilitate the programming of our functions. Within PlatformIO, we use version 4.0.0 of the Espressif ESP32

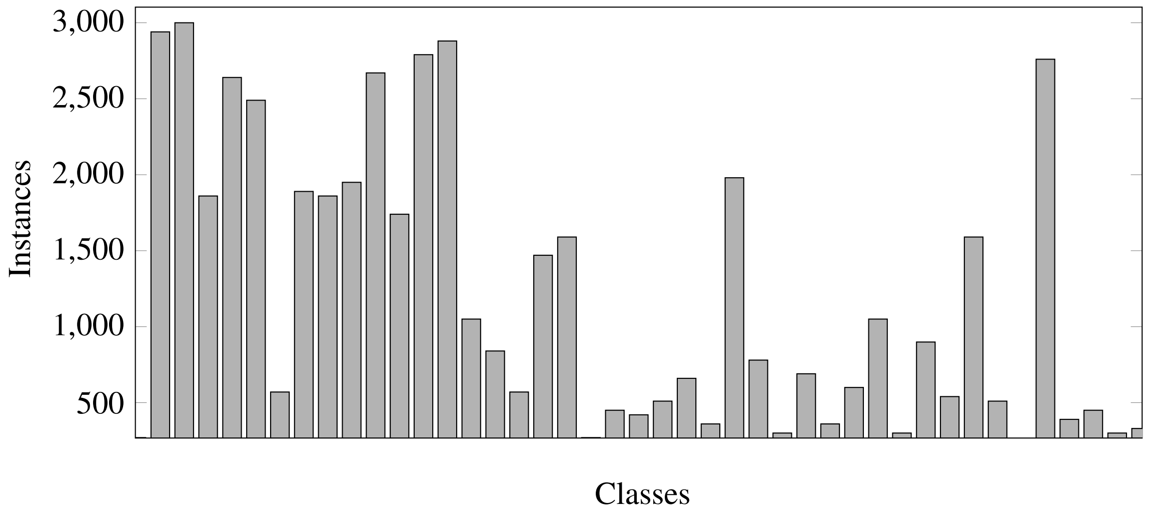
[1] Google Colab: https://colab.research.google.com/
[2] Python: https://www.python.org/

platform and 5.1.0 version for Espressif ESP32-S2. All experiments run on our architecture used the GTSRB dataset.

Due to the number of operations, part of the CNN models inference execution time is in the convolutional layers, which is closely linked to the complexity of the neural model. The ESP32 device has too low RAM and FLASH storage capacity to support a robust neural model such as VGG or AlexNet. Such models have millions of parameters, making them impossible to implement on a resource-constrained edge device. Aware of this, our approach involves developing a neural model capable of performing image recognition tasks with sufficient accuracy. The model must have a small parameter set and allows embedded in resource-constrained edge devices.

The proposed pipeline consists of eight stages. In the first stage of the pipeline, a neural model is developed to effectively detect the images from the GTSRB dataset. A convolutional network model, even with few layers, performs thousands or millions of multiplication and accumulation operations, which can be time consuming depending on the hardware employed. As already mentioned, the operational complexity is on the convolutional layers. To speed up the inference process, it is desirable to reduce the complexity of the neural network if it does not affect the performance metrics. The success of a machine learning model depends closely on the choice of desired hyperparameters (hparams). Generally speaking, some hparams of the neural network have a direct bearing on the final volume of the model. To increase the assertiveness in choosing the hyperparameters of our neural model, we use Bayesian Optimization. This way, we reduce the time spent in search and choosing these parameters. The use of Bayesian optimization for an optimized search of hparams is the second stage in our pipeline. In this phase, we will search for the best hyperparameters used by the student and teacher models.

Figure 10 – Distribution of instances over classes number in the GTSRB dataset.



After selecting the parameters that will make up the student and teacher models, we subject the GTSRB dataset to a Data Augmentation (DA) procedure. Figure 10 displays the distribution of the number of instances over classes in the GTSRB dataset. In the image, we can observe the unbalanced data. This leads to lower generality for the network due to a few samples for some classes and may result in possible overfitting during neural network training. The Data Augmentations technique uses simple transformations such as horizontal flipping, color space augmentations, and random cropping. These transformations encode many invariances in images used in recognition tasks. In our work, we used the TensorFlow Image Data Generator[3] (IDG) class. It provides various augmentation techniques like standardization, rotation, shifts, flips, and brightness. The Image Data Generator class ensures that the model receives new variations of the images at each epoch. In Figure 11, we can see some samples of the new images generated by the data augmentation technique. The configuration chosen for the DA consists of rotating the images up to thirty degrees to the left or right, shifting the source image to 20% on the x or y axis, using blur filters, and random adjustments to the brightness. At each training epoch, the DA generates 128 new images, corresponding to 11.53% of the images used.

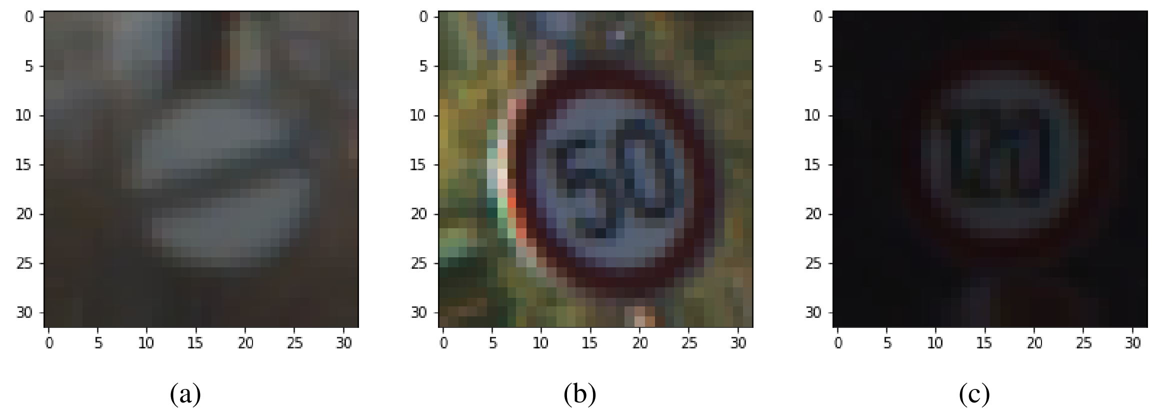


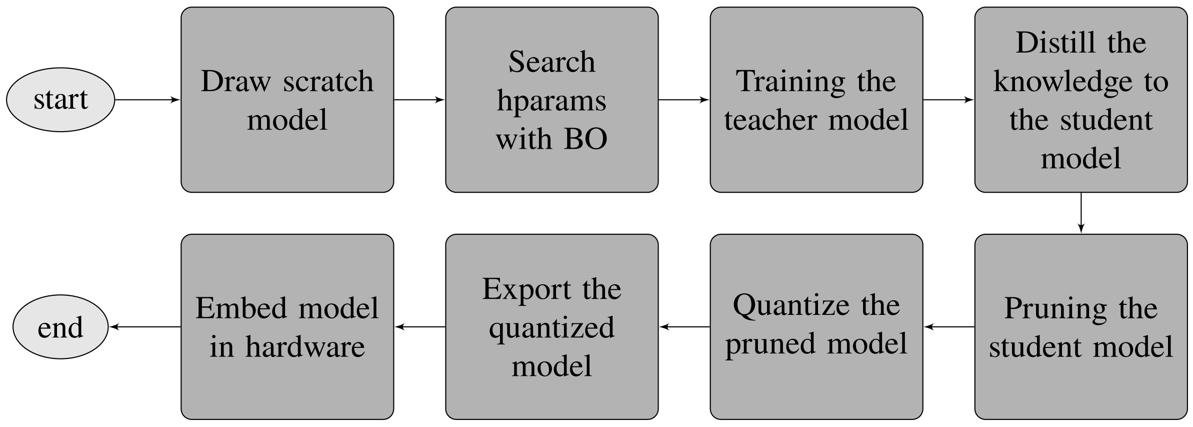
Figure 11 – **(a)** Image with blur applied by DA, **(b)** Image with rotation applied by DA, and **(c)** Image with brightness adjustment applied by DA

After dataset is balanced, we will move on to the teacher model training stage (stage three of the pipeline). The teacher model should be robust enough and have good accuracy and precision metrics since its weights will be distilled to the student model. For this purpose, we use the Knowledge Distillation technique, also known as the Teacher/Student Model. The main objective is to construct a small neural model (student) from knowledge of one or several models with more considerable capabilities (teacher). At the end of the Knowledge Distillation procedure, a pruning operation is applied to the output model from the previous step to reduce the final model's size further. From pruning, the quantization operation is applied to the model. As discussed in Shabarinath e Muralidhar (2020), quantization of a neural network allows for a reduction in computational cost and neural network volume, coupled with minimal impact on final accuracy. Figure 12 shows the complete proposed pipeline.

After training the baseline model (teacher model) and model compression steps (knowledge distillation, pruning, and quantization), we embed the neural network on the desired

[3] IDG: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

Figure 12 – Proposed pipeline for compressing a neural network for a resource-constrained device.


start
Draw scratch model
Search hparams with BO
Training the teacher model
Distill the knowledge to the student model
Pruning the student model
Quantize the pruned model
Export the quantized model
Embed model in hardware
end

Source: The Author.

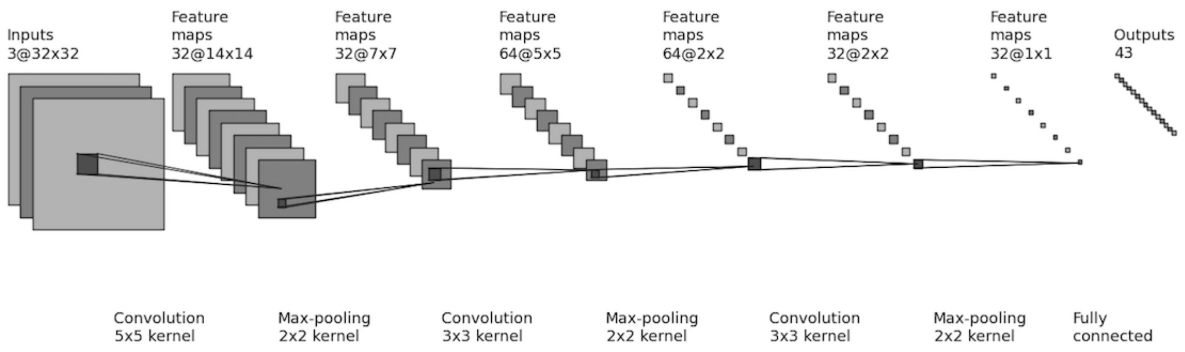
hardware, in our case, the ESP32. As a result of the compression pipeline, two files representing the final neural model are generated: a tflite file, a Flatbuffer[4] containing the model run data; and a "cc" file, the model information in vector form (c-hex array) obtained from the hex dump of the tflite file. Ultimately, we run the model by providing the necessary test images through the hardware's serial port. Then we collect the pertinent information to compose the evaluation metrics. To validate the results of the experiments, we will use the measures of accuracy, F1-Score, loss, recall, and precision as metrics for validating the final neural network. Furthermore, to evaluate the inference speed, we will use the frame rate metric (FPS) and the execution time of each convolutional layer.

4.2 Proposed CNN Architecture

This work uses the architecture based on the LeNet version presented by Lecun et al. (1998). The CNN was developed using three convolutional sets: one convolutional layer, each with a batch normalization layer, an activation layer and a max pooling layer. After convolutional block, we have flatten layers and one fully connected layer does the classification work of all dataset classes. The input layer of the convolutional set has an input resolution of 32 x 32 pixels. This format was chosen because it is close to some selected works Hosseini et al. (2017), Jang et al. (2016), Wang, Wang e Zhou (2019) that showed a good trade-off between number of parameters and accuracy. Moreover, the choice of architecture was also guided by the preliminary results obtained in the work of Silva et al. (2021) which reached a maximum accuracy of 99.31% in the training set. Also, the accuracy achieved was 93.73% in the validation set and the test traffic signs were predicted with 91.41% of accuracy, indicating that model was properly trained. The best loss values obtained were 0.3737 during training and 0.5574 for the validation set. Figure 13 shows a diagram with the convolutional neural network proposed as the basis for the experiments.

[4] FlatBuffers: https://google.github.io/flatbuffers/

Figure 13 – Diagram of the baseline neural network proposed by this work.



Source: The Author.

In this proposed work it is used the Softmax activation function in the last layer and the ReLU function in the convolutional layers, as in Kumar Reddy, Srinivasa Rao e Raju (2018). ReLU, short for Rectified Linear Unit, produces results in the range $[0, \infty)$. The Softmax function is usually used in the output layer because it produces probabilistic values for each attribute. The optimization algorithm chosen for the work was Root Mean Square Propagation (RMSProp), proposed by Hinton, Srivastava e Swersky (2012). RMSProp is a method used for optimization with an adaptive learning pace. This algorithm divides the learning rate by an exponentially decreasing mean of the square of the gradients, addressing the vanishing gradient problem (RUDER, 2016). In Tables 9 and 10, rho and momentum are parameters of optimizer RMSProp. The regularization process introduces a new way to prevent overfitting. It helps to avoid the linear models overfitting with the training dataset penalizing the extreme weight values. In this work, we chose L2 Regularization at the convolutional and dense layers. We use a normal distribution of tensors to reduce variability during kernel initialization resulting in more constant model evaluations. The same is applied to the bias initializers. This setting is valid for all convolutional layers of the teacher and student models. As mentioned in section 2.4, the GTSRB dataset contains images of varying resolutions containing three color channels (RGB). Thus, the 32x32 dimension was chosen for the input layer of the convolutional network since our initial work took place in the CIFAR-10[5] dataset.

The hyperparameters, such as learning rate, convolutional filters, rho, or regularization, are selected by the BO step. As the library chosen for this work was TensorFlow, we used the Keras Tuner to apply the BO. The choice of the search space is a subjective task. Bayesian optimization can optimize any number and type of hyperparameters, but observations are costly, so we limit the dimensionality and size of the search space. For that, we defined a range of hyperparameters and let the optimization algorithm choose the best set for our two neural models: Student Model and Teacher Model. In the initial model search, we defined the baseline structure

[5] The CIFAR-10 dataset: https://www.cs.toronto.edu/ kriz/cifar.html

Table 9 – Hyperparameters (h-params) values choose by the Bayesian Optimization for the Student Model.

H-Params	**Min. Value**	**Max. Value**	**Best Value**
Conv0 Filters	12	14	12
Conv0 L2 Regularization	1e-5	1e-3	4e-05
Conv1 Filters	12	16	16
Conv1 L2 Regularization	1e-5	1e-3	1e-05
Conv2 Filters	4	8	8
Conv2 L2 Regularization	1e-5	1e-3	1e-05
L2 Regularizarion Dense	1e-4	1e-3	1e-4
Rho	0.7	0.9	0.8
Momentum	0.7	0.9	0.7
Learning Rate	1e-5	1e-2	1.04e-3

as in Tables 9 and 10, which was information from previous training of the neural model based on the work of Joshi et al. (2019).

The metric chosen for selecting the best model using BO was validation accuracy. The best set of BO hyperparameters built a student model with a 92.96% validation accuracy. To the teacher net, the best metric was 98.69%. The search for the best BO hyperparameters took 100 trials. Each trial runs a given neural model with a set of hyperparameters from a space of 15 epochs.

Table 10 – Hyperparameters (h-params) values choose by the Bayesian Optimization for the Teacher Model.

H-Params	**Min. Value**	**Max. Value**	**Best Value**
Conv0 Filters	16	32	32
Conv0 L2 Regularization	1e-5	1e-3	1e-5
Conv1 Filters	32	64	64
Conv1 L2 Regularization	1e-5	1e-3	1e-05
Conv2 Filters	16	32	24
Conv2 L2 Regularization	1e-5	1e-3	1e-3
L2 Regularizarion Dense	1e-5	1e-3	1e-3
Rho	0.8	0.95	0.95
Momentum	0.7	0.9	0.0
Learning Rate	1e-5	1e-2	8.27e-05

4.3 Pipeline Experiment

As already discussed, the GTSRB dataset was chosen for already being a widely used dataset and having several experiments with quantitative and qualitative results that would serve as a basis for comparisons. The dataset has more than 50,000 images, distributed in 43 possible classes (traffic signs). The GTSRB is composed of color images with resolutions from 15x15 to 250x250 pixels in PPM format.

Table 11 – Distribution of the images from the experiment dataset.

Dataset	**No. of images**
Training	35209
Validation	4000
Tests	12630
Total	**51839**

Source: The Author

The images in the GTSRB dataset were resized to images with a resolution of 32 x 32 pixels, like the CIFAR-10/100 dataset (KRIZHEVSKY; NAIR; HINTON, b). In models that seek data generalization, the use of a dataset is required for training, validation, and testing. Therefore, after loading the data, the instances were divided into sets according to Table 11. The image normalization is applied to all images by dividing each position of the image array by 255. This step adapts the input data to the dynamic range of the activation functions of a neural network.

4.3.1 Knowledge Distillation

In this section, we describe the step-by-step process involved in defining the models mechanisms, training, testing, validation, and analysis of the teacher and student nets. The hyperparameters for the proposed models in this pipeline stage were selected as in Section 4.2. The declared layers structure for the teacher and student models can be seen in Appendices B and C.

Teacher Model: the proposed teacher net was trained with the learning rate (lr) set dynamically through a callback function passed to the fit function. The learning rate update scheme of the learning rate updates can be seen in Equation 4.1. The training lasted 250 epochs (E) as the batch size was set to 32. With this, in each epoch, 1100 images were used for training. Using the environment described at the beginning of the chapter and the parameters mentioned above, the time required for training was 1 hour and 31 minutes. Each training epoch took approximately 22 seconds to run.

$$lr = \begin{cases} \text{lr}, & \text{if } E <= 120 \\ \text{lr}*0.1, & \text{if } 120 < E <= 200 \\ \text{lr}*0.01, & \text{if } 200 < E < 220 \\ \text{lr}*0.001, & \text{if } E > 220 \end{cases} \tag{4.1}$$

As an evaluation model metric, we monitor the accuracy of the model on the validation set data. At each training epoch, a check function evaluates the current validation accuracy metric and compares them to the best previous value. If the net performs better in this epoch, we save this model in a ckpt file (Training Checkpoint[6]) for future use. The accuracy, loss, F_1-Score, precision, and recall metrics obtained after training, on validation, and the test set can be seen in the following table. In terms of RMSE, the baseline model achieves the lowest value.

Table 12 – Results obtained using the Teacher baseline neural model (without quantization) run in the Google Colab environment.

Metric	**Train**	**Validation**	**Test**
Accuracy	99.12%	99.62%	95.56%
Loss	0.1244	0.1058	0.2707
F1-Score	98.86%	99.54%	97.06%
Precision	99.46%	99.72%	94.46%
Recall	98.70%	99.40%	93.39%
RMSE	0.0330	0.0255	0.0408

Source: The Author

Considering the test accuracy of the model and compared to the results of the International Joint Conference on Neural Networks (IJCNN) 2011[7], the benchmark event where the GTSRB was used, the performance of the professor model would be placed within the TOP100 registered networks in the first phase. When the teacher model training stage was completed, we begin the knowledge distillation phase for the student model. The objective of this phase is to create a reduced model in terms o parameters and maintain the teacher model accuracy.

Student Model: Using a custom class called Distiller (see in Appendix A), we override the train_step, test_step, and compile() methods inherited from the Model class. In the train_step method, we perform a forward pass of both the teacher and student. Afterward, calculate the loss with the weighting of the student_loss and distillation_loss by $alpha$ and $1 - alpha$, respectively. In the end, the code executes a backward pass. Distillation loss is the calculation of the loss function *versus* the teacher's soft targets, using the same value of T to calculate the Softmax in the student logits. On the other hand, student loss is the standard loss between the class probabilities predicted by the student model and the teacher's hard targets (ground truth) and soft targets. In this stage, we introduce the weight parameter $alpha$. In the distillation process, we have two models and two different objective functions, Loss1 and Loss2. Loss1 is the cross entropy loss (CE) of the two Softmax temperatures for both teacher q and student p with temperature $T > 1$ multiplied by the weight parameter $alpha$.

$$q_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)} \tag{4.2}$$

$$p_i = \frac{\exp\left(\frac{v_i}{T}\right)}{\sum_j \exp\left(\frac{v_j}{T}\right)} \tag{4.3}$$

[6] Checkpoints files capture the value of all parameters used by a model at a point in time (TENSORFLOW, 2022c).

[7] Results for IJCNN 2011 competition (1st stage): https://benchmark.ini.rub.de/gtsrb_results_ijcnn.html

$$Loss1 = \alpha * CE(q_i, p_i) \tag{4.4}$$

Loss2 is the CE loss of the correct labels and the student hard targets with $T = 1$. Loss2 pays little attention $1 - alpha$ to the hard targets (student_pred) made by the student model to match the easy targets q of the teacher model. So we have the following equations:

$$student\_pred = argmax \left(\frac{\exp\left(\frac{v_i}{T}\right)}{\sum_j \exp\left(\frac{v_j}{T}\right)} \right) \tag{4.5}$$

$$Loss_2 = (1 - \alpha) * CE(student\_pred, y\_true) \tag{4.6}$$

The distillation loss function used by the distillation process, different from the teacher model, was the Kullback-Leibler divergence. It quantifies how much a probability distribution diverges from another distribution (KIM et al., 2021). It is used because of the need to control the smoothness of the soft objectives through the temperature hyperparameter (T). The objective of the student model will be the distillation loss which is the sum of Loss1 and Loss2.

Table 13 – Results obtained using the student neural model in distillation run in the Google Colab environment ($alpha = 0.1$ and $temperature = 10$).

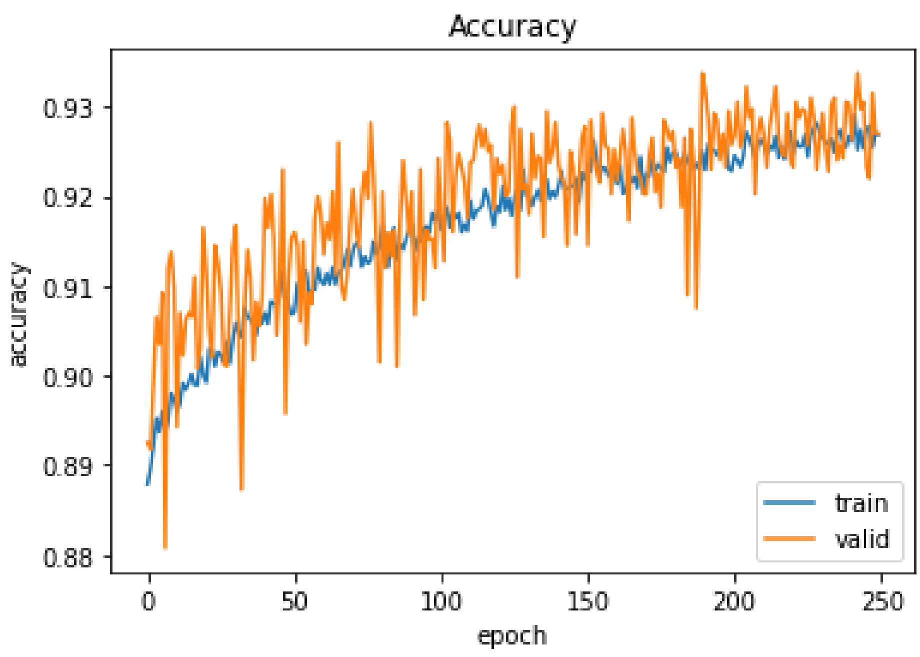
Metric	Train	Validation	Test
Accuracy	95.83%	93.37%	84.60%
Loss	0.1276	0.0901	0.9646
F1-Score	94.34%	91.97%	77.35%
Precision	97.28%	94.96%	87.78%
Recall	83.04%	83.04%	83.04%
RMSE	0.0602	0.0629	0.0791

Source: The Author

The student model has 3,323 parameters, 13.3% of the total parameters of the teacher model (24,939 parameters). During training, the proposed model reached the optimum value for validation accuracy of 93.37%. The values of the other metrics can be seen in Table 13. These metrics were obtained using the alpha values at 0.1 and the temperature set to 10. Due to the small size of the student model, the options to avoid overfitting the model was insufficient. In this way, we can verify this behavior in the plot of Figure 14. This can be validated by checking the metrics on the test set in Table 13.

To solve this problem of overfitting in student model, we raised the temperature to 20 while maintaining the same value for the alpha coefficient. Hinton, Vinyals e Dean (2015) used in their work temperature values ranging from 1 to 20 and empirically observed that when the student model is much smaller relative to the teacher model, the lower temperatures work better. This makes sense since as the Softmax temperature increases, the resulting label distribution is smoother, becoming more information-rich.

Figure 14 – Accuracy curve of the student model after training at 250 epochs.



Source: The author

Thus the student model, which, because it is very small, cannot fully generalize the data, is favored with the "heated" labels. This way, the temperature elevation improved the network performance, allowing a concise generalization, and increasing the gain in the test metrics and the validation precision, as seen in Table 14. Temperature values above 20 brought inconsistencies in the metrics of the proposed student model. In Figure 15 we visualize the plot for validation accuracy metrics after temperature adjustment in the distillation procedure.

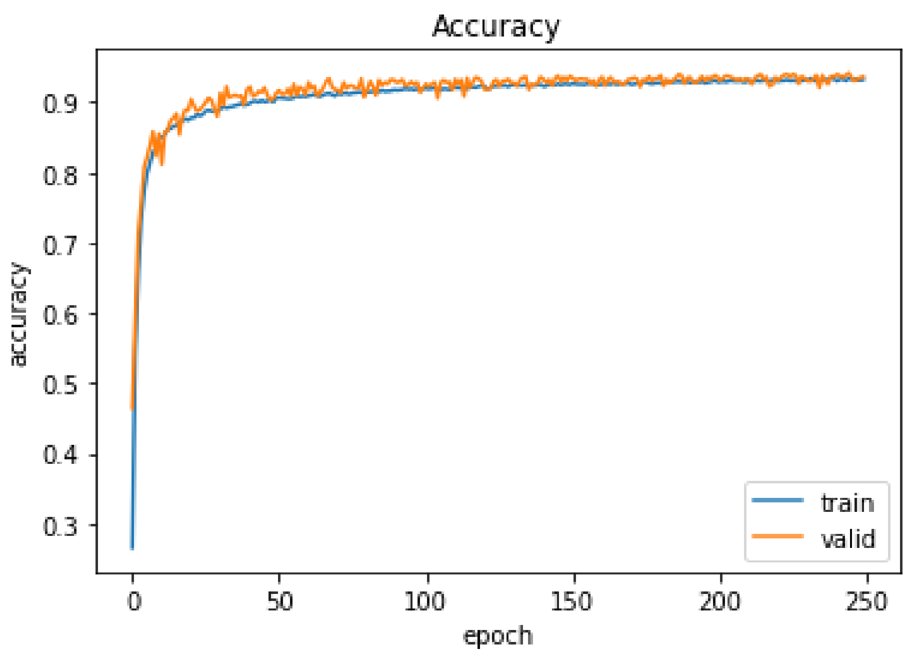
Table 14 – Results obtained using the student neural model in distillation run in the Google Colab environment ($alpha$ = 0.1 and $temperature$ = 20).

Metric	**Train**	**Validation**	**Test**
Accuracy	96.04%	93.35%	86.15%
Loss	0.1378	0.2535	0.6890
F1-Score	94.59%	90.98%	78.26%
Precision	97.37%	94.83%	88.88%
Recall	94.74%	92.20%	84.88%
RMSE	0.0456	0.0536	0.0762

Source: The author

After training a scratch student model, we obtain the metrics that can be seen in Table 15. We observe from the data that the knowledge distillation process improved the performance of the student model compared to the net trained from scratch. The scratch student model has the same structure as the distilled student model, except for the weights. Then its layers and hyperparameters were organized as described in Appendix C.

Figure 15 – Accuracy curve of the student model with the temperature adjusted.



Source: The author

Table 15 – Results obtained using the student baseline model without distillation.

Metric	Train	Validation	Test
Accuracy	93.12%	91.10%	83.16%
Loss	0.4502	0.5464	0.9599
F1-Score	90.73%	88.45%	75.91%
Precision	95.50%	93.80%	87.32%
Recall	91.05%	88.85%	80.89%
RMSE	0.0462	0.0543	0.0753

Source: The author

4.3.2 Pruning

In our work, we use the magnitude-based weight pruning technique. This is one of the most common methods of pruning (ZHU; GUPTA, 2017). For this, TensorFlow/Keras itself provides an API. The input model at this stage will be created at the distillation stage. The chosen pruning technique gradually excludes the weights of the layers during the training process to achieve the network's sparseness. Sparse models are easier to compact, and we can ignore the zeros during inference to improve latency and reduce the model size. In the pruning stage, new training will be necessary. For this, the weights and parameters of the distilled student model will be loaded into the scratch student model. The exported distilled model has layers and functions inherent to the knowledge transfer process, and the TensorFlow API does not recognize this as a readable pattern for the default load act. In the pruning step, we continue to use as optimizers the RMSProp, batch size, and training epochs defined just like the student model. Since pruning will be performed on the already trained student model, we chose to start the procedure at the twentieth epoch of the pruning process training. This is configured through the begin_step (t_0) parameter. The final pruning step called end_step (t_f) is calculated according to Equation 4.7. Where I_s is the number of images in the set for validation, B_s is the batch size, and E is the number of epochs.

$$t_f = \lceil (I_s/B_s) * E \rceil \tag{4.7}$$

The pruning mechanism used in the API consists in evaluating the weights after each iteration of the training instead of abruptly pruning all unused weights. In this way, starting pruning after some epochs have elapsed and stipulated a pruning frequency (Δt), every 20 epochs in our case, allows the model to recover in terms of accuracy. The desired final sparsity (s_f) for this model is 30%. The pruning step was set to occur every 20 epochs with an initial sparsity (s_i) of 10%. The algorithm that defines the desired sparsity for a given pruning step t of pruning defined for this work was Polynomial Decay. Equation 4.8 defines how Polynomial Decay works, where n, defined in Equation 4.9, is the number of pruning steps needed.

$$s_t = s_f - \left(s_i - s_f\right)\left(1 - \frac{t - t_0}{n\Delta t}\right)^p , \{t_0, t_0 + \Delta t, ..., t_0 + n\Delta t\} \tag{4.8}$$

$$n = \lfloor \frac{t_f - t_0}{\Delta t} \rfloor \tag{4.9}$$

Using the logic of the Polynomial Decay equation described above, fewer parameters are removed relative to the previous step as the pruning steps are incremented. From this point onward, the model was pruned in a weighted manner prioritizing validation loss and accuracy.

Table 16 – Results obtained after the model pruning phase.

Metric	Train	Validation	Test
Accuracy	96.06%	93.65%	85.98%
Loss	0.2504	0.2691	0.8762
F1-Score	94.39%	91.35%	77.40%
Precision	97.44%	95.27%	88.59%
Recall	91.25%	91.25%	91.25%
RMSE	0.0519	0.0548	0.0788

Source: The Author

4.3.3 Quantization

In addition to the compression steps mentioned above, quantization is particularly important when embedding neural models in devices with restricted resources. With a leaner model, the time required to perform inference is reduced. Quantization can be performed in two ways: post-training quantization, also known as fall-back quantization; or in Quantization-Aware Training. In post-training quantization, the conversion is done after the model is trained, converting all weights and data relevant to the neural network. In Quantization Aware Training, on the other hand, the model conversion steps are performed in the forward step while the model is undergoing training. In this way, the degradation due to conversion of numerical types is reduced while also gaining the inference speed of the final net.

Table 17 – Metrics obtained by the quantized neural model running in the Google Colab environment.

Metric	Train	Validation	Test
Accuracy	96.24%	93.95%	86.65%
Loss	0.2284	0.3617	0.8549
F1-Score	95.08%	92.03%	79.27%
Precision	98.02%	95.49%	89.95%
Recall	94.59%	92.20%	84.41%
RMSE	0.0491	0.0530	0.0772

Source: The Author

Within the quantization step, we used an optimization strategy based on quantization of the model weights. By default of this framework, the set of biases and activations will also be quantized, seeking to reduce the model's size and latency while minimizing the accuracy loss. During QAT, the best model from the pruning step, considering the validation accuracy, is saved to a CKPT file (the standard used by TensorFlow) via a checkpoint function. This best network will be used in the quantization process. After the quantization-aware training step, the generated model is loaded into the TensorFlow Lite Converter (TLC). The TLC generates a tflite file as the output of the conversion. This file contains the model, optimizations, and parameters needed to run the neural network on MCUs. After exporting the tflite file, we use xxd[8] to create a hex dump of our tflite model into a C source file (c-hex array). This final file contains the entire model in a hexadecimal distribution and a constant with the model's size in bytes. This variable is the arena size and is the space in bytes that will be used to allocate the input tensors, output tensors, and intermediate vectors during model inference within the chosen MCU. Quantization-aware training emulates quantization at inference time, creating a model that TensorFlow's downstream tools will use to produce truly quantized models, unlike post-training quantization, which only reduces the precision of the variables used to store net weights and data. Quantized models use lower precision (e.g., 8 bits instead of 32-bit floating), leading to benefits during deployment. Regarding the pruned model, quantization improved some metrics (such as accuracy and precision), as seen in Table 17.

[8] The xxd command generates a hexadecimal dump of a given file and vice versa (MERTZ, 2022)

5 Results

This chapter describes how the compression pipeline proposed in Chapter 4 can be executed. The following sections describe the final neural network model implementation on the selected resource-constrained devices and experimental analysis. In a similar way, we will discuss the results obtained at each step. The ESP32 models used as test-bed in the experiments can be seen in Table 18.

Table 18 – Hardware used in the experiments.

DevBoard	**Processor**	**Memory**
ESP32	2x LX6 32-bit 240 MHz	320 KB RAM + 4 MB FLASH
ESP32-S2	Single-Core LX7 32-bit 160 MHz	320 KB RAM + 4 MB FLASH

Source: The Author

5.1 Evaluation of the Embedded Model

Visual Studio Code and the PlatformIO[1] development plugin were used to evaluate ESP32 development boards for writing the code and uploading the neural models to the MCU. For PlatformIO plugin version was 4.0.0[2] and for testing with the ESP32-S2 version 5.1.0[3] of SDK. The test environment setup in VSCode was performed as described by Tavares et al. (2018). The dataset images are sent to the MCU using the development board's Universal Asynchronous Receiver/Transmitter (UART) interface. The theoretical maximum transfer speed using the test-beds serial ports can be up to 2,000,000 bits/s using the USB/serial converters CH340[4] or 921,600 bits/s in CP2102[5]. We mean theoretical because the final throughput depends

1 PlatformIO IDE: https://platformio.org/
2 PIO 4.0.0: https://github.com/platformio/platformio-core/blob/v4.3.4/HISTORY.rst
3 PIO 5.1.0: https://github.com/platformio/platformio-core/blob/v5.2.5/HISTORY.rst
4 CH340 Datasheet: https://cdn.sparkfun.com/datasheets/Dev/Arduino/Other/CH340DS1.PDF
5 CP2102 Datasheet: https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf

on elements such as PCB design and quality of electronic components. The ESP32 development board utilized in the experiments is equipped with a CP2102 controller. In contrast, the ESP32-S2 board has a controller model CH340. This difference in USB/serial converters allowed us to test the connection stability for each board. It was identified that using the same set of equipment (notebook, USB port, and cable), the CH340 controller can achieve higher transfer rates in the tests. So the setting for transfer speeds in the experiments was set to 115,200 bits/s for the ESP32 and 460,800 bits/s for the ESP32-S2.

After following the integration steps described in Tavares et al. (2018) and finalizing the environment configuration, we started the programming stage of our project in VSCode with the Arduino[6] framework setup and ESP32 development board selected. In the project's main function (main.cc), we include the libraries needed to compile the application. This encompasses the Arduino library and the dependencies needed to run the TensorFlow Lite model as per the Listing 5.1.

Listing 5.1 – Libraries required for the correct operation of the TensorFlow Lite project.

```
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
#include <Arduino.h>
```

The all_ops_resolver.h library is responsible for executing the operations used by the model interpreter, micro_error_reporter.h is invoked when using debugging options, micro_interpreter.h loads and executes the models, schema_generated.h manages the FlatBuffer file and version.h provides versioning information for the TensorFlowLite schema (TENSORFLOW, 2022a).

For the model environment to be ready to execute, are required some parameters initializations. We starting with the TensorFlow Lite debugger declaration, the neural model, and the interpreter that will load and run the embedded model on the MCU. In addition, the input and output tensors also need to be initialized. This operation can be checked in the Listing 5.2 snippet.

Listing 5.2 – TensorFlow Lite Base Initialization.

```
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* input = nullptr;
TfLiteTensor* output = nullptr;
```

[6] Arduino Framework: https://www.arduino.cc/en/Guide/Introduction

As described in Section 4.1 at the end of each pipeline stage, we export the model in question to a cc file. This file represents the model that will be embedded into the ESP32 for inference execution. The structure of this file can be seen in Listing 5.3.

Listing 5.3 – CC-file with the neural model in hexdump format and the variable responsible for setting the size of this model in the interpreter.

```
unsigned char g_model[] = { 0x1c, 0x00, 0x00, 0x00, 0x54, 0x46,
   0x4c, 0x33, 0x14, 0x00, 0x20, 0x00, ...   ...   ...   ...   ...
   ...   ...   ...   ...   ...   ...   ..., 0x20, 0x00, 0x00, 0x00,
   0x20, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff,
   0xff, 0x04, 0x00, 0x04, 0x00, 0x04, 0x00, 0x00, 0x00};
unsigned int g_model_len = 101556;
```

Inside it we have two variables: an array of characters g_model and a variable of type integer g_model_len. Namely:

- **g_model**: 8-byte aligned hexadecimal vector containing the neural model;
- **g_model_len**: in-memory length of the respective model.

The memory space selected to allocate the tensors of the model is defined through the variable kModelArenaSize (see Listing 5.4). The variable kExtraArenaSize assigns additional space in case the number of tensors changes occur at runtime. Although it sounds counter-intuitive, the size of the tensor arena remains almost the same for two of four models generated in our pipeline (distilled and pruned student models).

Listing 5.4 – Part of the code that defines the area that will be used by the model's tensors.

```
const int kModelArenaSize = 39 * 1024;
const int kExtraArenaSize = 32 + 16 + 64;
const int kTensorArenaSize = kModelArenaSize + kExtraArenaSize;
uint8_t tensor_arena[kTensorArenaSize];
```

The memory consumed by each embedded model in the dev boards and the tensors arena can be seen in Table 19. The column Model on MCU displays the model space loaded into test-beds memory. In the Arena Size column, we see the total size of the tensor arena, that is, the kTensorArenaSize variable. The model's length in the embedded device's memory is obtained by inspecting the ELF[7] file. This inspection can be done via the PlatformIO plugin. In Annex C we see the return of the inspection, displaying the model and the tensor arena space.

At a first contact at the table above, we can assume that only the knowledge distillation stage would be required to ship the neural model to our MCU. Although some constrained devices

[7] ELF is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking (OSDEV, 2022).

Table 19 – Space used by each model file (bytes).

Model	File Size	Model on MCU	Arena Size
Baseline (Teacher)	626.447	101.580	39.936
Student (Distilled)	102.728	16.384	23.552
Student Pruned	102.742	16.384	23.552
Student Quantized	59.289	9.626	17.510

Source: The Author

like the ESP32 and ESP32-S2 have 320 KB of RAM, this space cannot always be fully utilized. According to Espressif (2022), the ESP32 and ESP32-S2 contain multiple types of RAM:

- **DRAM (Data RAM)** is memory used to hold data. This is the most common kind of memory accessed as a heap.;
- **IRAM (Instruction RAM)** usually holds executable data only. If accessed as generic memory, all accesses must be 32-bit aligned;
- **D/IRAM** is RAM that can be used as either Instruction or Data RAM. Which can be used as either Instruction or Data RAM.

Then checking the memory allocation spaces at startup, we can see a log summary of all heap addresses at level info, see Listing 5.5. We can check that the device's memory is segmented, not allowing these areas to overflow. Besides this, the memory is shared with other processes, and if there is not enough area in RAM to allocate the model with the other instructions, it will be moved to FLASH memory, which is slower. In Appendix C, Figure 21, we can see an example of the distilled model embedded in the ESP32. The section in which the model is configured is FLASH, even though it is a smaller model. In Figure 22 of the same Appendix, we can see the quantized model embedded in the ESP32 and allocated in data RAM. The heap sizes allocation is the same in both devices, ESP32 e ESP32-S2.

Listing 5.5 – Code snippet showing spaces used by the heap

```
I (252) heap_init: Initializing. RAM available for dynamic
   allocation:
I (259) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (265) heap_init: At 3FFB2EC8 len 0002D138 (180 KiB): DRAM
I (272) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (278) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (284) heap_init: At 4008944C len 00016BB4 (90 KiB): IRAM
```

About the size of the tensors arena yet, if this parameter is not large enough, the tensors allocation in the model will fail. The API documentation does not mention to any method to calculate the arena size before embedding the model. There is only a method to calculate this value after initialization (as per Listing 5.6). This is insufficient since if the loaded model requires a larger tensor arena, it will fail at MCU startup time. Therefore the size of the tensor arena was chosen using the experimentation method.

Listing 5.6 – An excerpt of the code responsible for calculating the tensor arena of the model, located in the header file micro_interpreter.h of the TensorFLow Lite API.

```
size_t arena_used_bytes() const { return allocator_.used_bytes();}
```

In Espressif development boards, part of the available RAM is allocated for use by the heaps of devices like Bluetooth and WiFi. Our approach uses the Arduino framework with PIO instead of the official ESP-IDF[8] IDE. Therefore some functionality is unavailable, for example, the disabling of hardware resources directly that will not be used. This step becomes necessary to embed the teacher model since the ESP32 and ESP32-S2 does not have enough resources to receive this net in the factory defaults. To get around these gaps, part of the work was hardcoded. This means that at times we use fixed values within the source code to enable or disable some features. Starting with freeing up allocated memory space for functions that will not be used. To do this, we rewrote the file partitions.csv that contains the device's partition table. In a simplified way, we have kept only what is necessary for the correct functioning of the MCU, namely:

- **Non-volatile Storage Library (NVS)**: used to store calibration data for devices. It can also be used for data from other applications;
- **Factory**: default application partition. The bootloader will initialize the factory application execution unless a data/OTA partition exists;
- **Physical (PHY)**: used for storing boot data.

The file responsible for partition table configuration is partitions.csv. It is created at the time the MCU type is defined in the project. Therefore we set up our partition table to contain only necessary items described above as factory default formatting, such a as in 5.7.

Listing 5.7 – ESP-IDF Partition Table used in the ESP32 and ESP32-S2 during the experiments.

```
# Name,    Type, SubType, Offset,  Size, Flags
nvs,       data, nvs,     0x9000,  0x6000,
phy_init,  data, phy,     0xf000,  0x1000,
factory,   app,  factory, 0x10000, 1M,
```

The next step is to deallocate the space used by the heaps from the WiFi and Bluetooth routines. This can be done by simply making a change in the memory.ld file. The location of the file can be seen in Annex D. In the CC code inside the memory.ld file (ESP32 or ESP32-S2), we can check the variable responsible for assigning the size of the dram_0 segment.

[8] ESP-IDF: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/

Listing 5.8 – Memory segment used in the ESP32 during the experiments.

```
# Original Declaration
dram0_0_seg (RW) : org = 0x3FFB0000 + 0xdb5c, len = 0x2c200 - 0xdb5c
# Changed Declaration
dram0_0_seg (RW) : org = 0x3FFB0000 + 0xdb5c, len = 0x30D40
```

As described above, the DRAM segment is responsible for storing non-constant static data, and the remaining space in this region is used as a heap at runtime. Since we have not loaded any libraries or enabled WiFi or Bluetooth usage to the ESP32 DevBoard, we will increment the size of this segment by 0x4B40 bytes (see Listing 5.8). In this way, we could embed the teacher model on the ESP32 for the experiments. For the ESP32-S2, we follow the same logic. We change within the respective ld file the line corresponding to the organization and size of the DRAM segment. We do this by decreasing the size of the originally allocated space by 0x3C00 bytes. As a follows in Listing 5.9. Checking the additional spaces generated in the heaps of both MCUs, we notice a difference of 0xF40 bytes. This was on purpose since the ESP32-S2's LX7 MCU may need more free space in the heap due to new features.

Listing 5.9 – Memory segment used in the ESP32-S2 during the experiments.

```
# Original Declaration
dram0_0_seg (RW) : org = 0x3FFB0000 + 0x4000, len = 0x2C000
# Changed Declaration
dram0_0_seg (RW) : org = 0x3FFB0000 + 0x400, len = 0x2FC00
```

The metrics for evaluating the neural model running on an MCU were the same as those adopted for the network under development within the Google Colab platform. For the experiment on the model embedded in the ESP32 and ESP32-S2 boards, we used the test set consisting of 12630 images of the GTSRB dataset. Before being sent for inference, the images were normalized at the data reading stage. Each position of the received array was normalized and stored in the TF Lite input tensor. On the ESP32 DevBoard, the complete inference of the images using the teacher model was run in 165 minutes and 45.1 seconds. This test does not consider any code optimization. Running the distilled model on the ESP32 DevBoard gives us 89 minutes and 47.9 seconds runtime. For the pruned network, the execution time was 89 minutes and 40.7 seconds. Evaluating the quantized model, we verified an increase in execution time, which was 96 minutes and 46.3 seconds. The time between reading the image via serial, detecting the instance, and inserting the result into the serial for display on the terminal is 380 milliseconds for all neural models. Since the objective of this work is to evaluate images, we choose to use the frame rate metric to verify the image processing capability based on the processing speed of our convolutional neural network. In Table 20, we have the calculated the frame rate for each model in our pipeline.

Table 20 – Frame Rate (FR) and time spent per inference for each pipeline model running in ESP32.

Model	FR	Inference Time
Baseline (Teacher)	2.4	407 ms
Student (Distilled)	21.3	47 ms
Student Pruned	21.7	46 ms
Student Quantized	12.5	80 ms

Source: The Author

For the ESP32-S2 test-bed, the inference times were higher, the results are available in Table 21. Despite being an updated MCU, the LX7 proved slower in performing the inference task. Its lower processing power verified in our tests can be validated with its lower score on the CoreMark benchmark. Aware of that, the complete inference of the images using the teacher model in the ESP32-S2 device was run in 244 minutes and 20.5 seconds. Considering the distilled model, the dev board executed the test in 74 minutes and 8.4 seconds. For the pruned network, the execution time was 73 minutes and 23.2 seconds. The quantized model has taken 90 minutes and 0.1 seconds to execute. The CH340 converter proved to be more stable in preliminary tests, as described at the beginning of the section. So we used a higher transfer rate in the tests of the ESP32-S2. The calculated latency for sending the image via serial and the return of the prediction by the neural model was 113 milliseconds.

Table 21 – Frame Rate (FR) and time spent per inference for each pipeline model running in ESP32-S2.

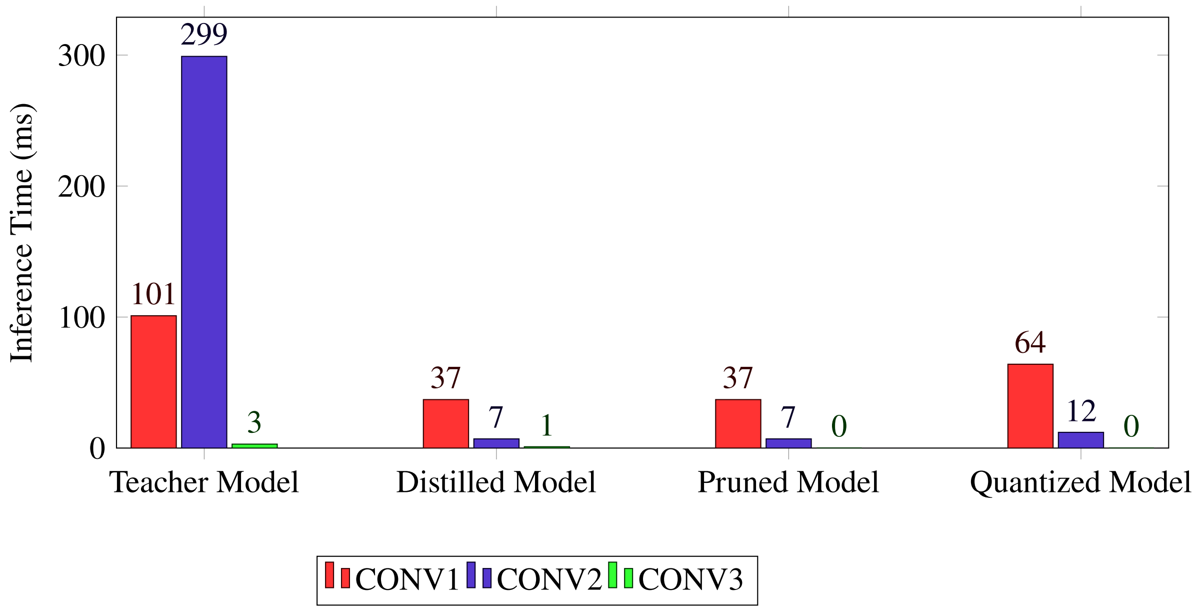
Model	FR	Inference Time
Baseline (Teacher)	0.95	1049 ms
Student (Distilled)	4.2	240 ms
Student Pruned	4.2	234 ms
Student Quantized	12.0	83 ms

Source: The Author

The inference time highlighted in the table is the complete execution time. Comparing the baseline model and the second stage of the pipeline, we have an execution time reduction of 88.46% for the ESP32 and 77.12% for the ESP32-S2. In the pruning step, the model achieved the same FPS as the second stage, decreasing the inference time by only 1ms in ESP32 and 6ms in the ESP32-S2. In the QAT stage, the inference time increased by 73.92% in the ESP32. To the data obtained from the QAT stage tests in the ESP32-S2, there was a 64.52% reduction in inference time. Checking the execution time per convolutional layer of each model (see Figure 16 and 17), we notice that the second layer of the baseline model running in the ESP32 was responsible for 73.5% of the time spent.

As we move down the net layers, the model becomes less computationally expensive. This is due in part to the number of parameters reduced layer by layer and also to the smaller size

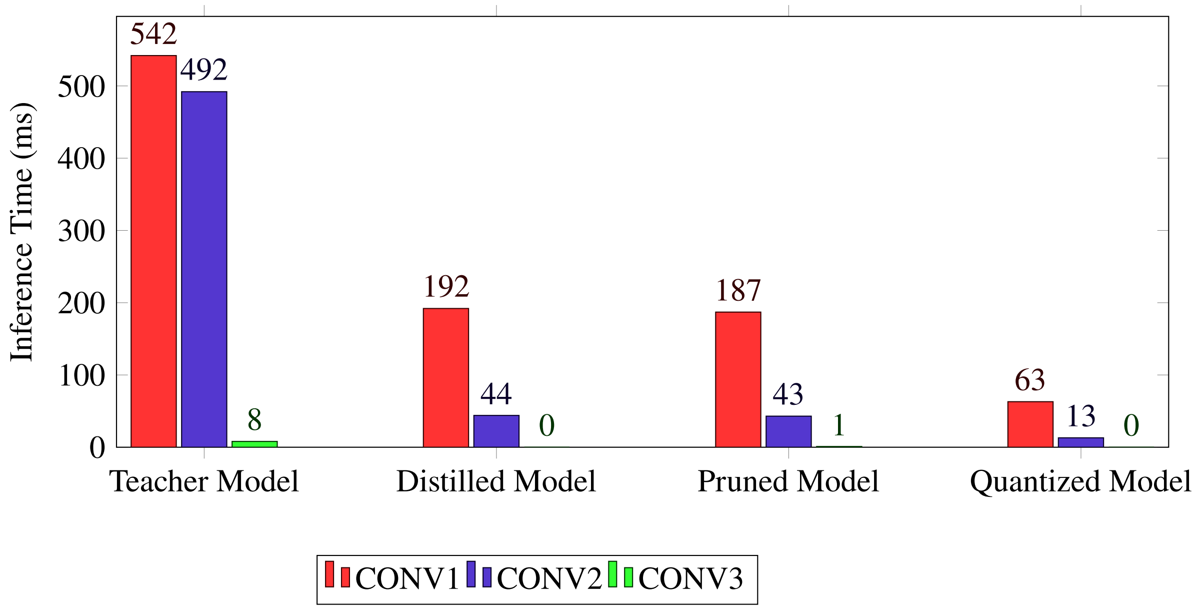
Figure 16 – Execution time per conv layer at ESP32 experiment.



Source: The Author

of feature maps. However, in the teacher model at the second convolutional layer there is an 86% increase in the number of parameters, as shown in Table 22.

Figure 17 – Execution time per conv layer at ESP32-S2 experiment.



Source: The Author

Nevertheless, this same behavior was not verified when running the teacher model on the ESP32-S2. As the neural model is the same for the ESP32 and ESP32-S2 tests, similar behavior was expected for the second convolutional layer of the professor model on the ESP32-S2, which did not occur. It was detected that the version of the PIO plugin used for testing on the ESP32-S2 adds computational complexity to the tests performed on the ESP32, increasing the total inference time of the baseline model by 422ms. At this point, a separate investigation is suggested as future work.

Table 22 – Teacher model layers and parameters.

Layer	Output Shape	Param #
Conv2D	(14, 14, 32)	2432
BatchNormalization	(14, 14, 32)	128
Activation	(14, 14, 32)	0
MaxPooling2D	(7, 7, 32)	0
Conv2D	(5, 5, 64)	18496
BatchNormalization	(5, 5, 64)	256
Activation	(5, 5, 64)	0
MaxPooling2D	(2, 2, 64)	0
Conv2D	(2, 2, 32)	2080
BatchNormalization	(2, 2, 32)	128
Activation	(2, 2, 32)	0
MaxPooling2D	(1, 1, 32)	0
Flatten	(32)	0
Dense	(43)	1419

Source: The Author

After running the inference on all the dataset test images, we obtained the accuracy metrics in Table 23. This table shows the results obtained when running in the Google Colab environment and on both dev boards. The metrics accuracy, recall, and F_1-Score of the models run on the ESP32, and ESP32-S2 consider the weighted average of the results using the support values, which are the numbers of true instances for each class. The accuracy of the baseline (teacher net) model running on the ESP32 DevBoard was the best, with 95.18%. The stage that obtained the best accuracy was pruning, with 85.98%. All classification reports are available in Appendices D to K.

The models created in the distillation pipeline and the pruning pipeline output obtained the best classification in inference time (see Table 20). The student model used in the knowledge distillation step has 3.323 total parameters, 73 of which are untrainable[9]. According to Tables 19 and 20, did not have its performance substantially improved after the pruning step. Analyzing the layers and parameters of the pruned model (Table 24, Output Shape^b and Param^b #), it can be seen that in the pruning step, there was an increase in the number of parameters concerning the output of the previous model, in this case, the distilled model (Table 24 Output Shape^a and Param^a #). The pruned model has 6,441 parameters, of which 3,190 are untrainable. Almost twice as many parameters compared to the distilled net.

[9] Untrainable parameters are parameters that cannot be learned from the training data (ACADEMY, 2019b).

Table 23 – Classification report ESP32 DevBoard.

Model (ESP32)	**Accuracy**	**Recall**	**F1-Score**	**Precision**
Baseline (Teacher)	95.18%	95.18%	95.14%	95.31%
Student (Distilled)	85.48%	85.49%	85.44%	85.73%
Student Pruned	85.98%	85.98%	85.80%	86.19%
Student Quantized	85.91%	85.98%	85.80%	86.19%
Model (ESP32-S2)	**Accuracy**	**Recall**	**F1-Score**	**Precision**
Baseline (Teacher)	93.65%	93.65%	93.64%	93.98%
Student (Distilled)	83.83%	83.94%	83.83%	83.94%
Student Pruned	85.65%	85.65%	85.53%	85.81%
Student Quantized	85.91%	85.91%	85.79%	86.35%
Model (Google Colab)	**Accuracy**	**Recall**	**F1-Score**	**Precision**
Baseline (Teacher)	84.60%	83.04%	77.35%	87.78%
Student (Distilled)	86.15%	84.88%	78.26%	88.88%
Student Pruned	85.98%	91.25%	77.40%	88.59%
Student Quantized	86.65%	84.41%	79.27%	89.95%

Source: The Author

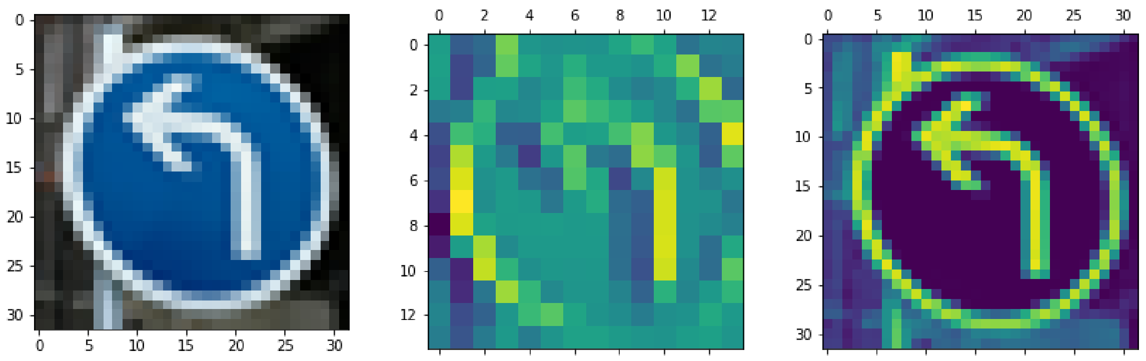
Table 24 – Student and Pruned model layers and parameters.

Layer	**Output Shape**[a]	**Param**[a] **#**	**Output Shape**[b]	**Param**[b] **#**
Conv2D	(14, 14, 12)	912	(14, 14, 12)	1814
BatchNormalization	(14, 14, 12)	48	(14, 14, 12)	49
Activation	(14, 14, 12)	0	(14, 14, 12)	1
MaxPooling2D	(7, 7, 12)	0	(7, 7, 12)	1
Conv2D	(5, 5, 16)	1744	(5, 5, 16)	3474
BatchNormalization	(5, 5, 16)	64	(5, 5, 16)	65
Activation	(5, 5, 16)	0	(5, 5, 16)	1
MaxPooling2D	(2, 2, 16)	0	(2, 2, 16)	1
Conv2D	(2, 2, 8)	136	(2, 2, 8)	266
BatchNormalization	(2, 2, 8)	32	(2, 2, 8)	33
Activation	(2, 2, 8)	0	(2, 2, 8)	1
MaxPooling2D	(1, 1, 8)	0	(1, 1, 8)	1
Flatten	(8)	0	(8)	0
Dense	(43)	387	(43)	733

[a] Student model output shape and parameters
[b] Pruned model output shape and parameters
Source: The Author

By checking TensorFlow pruning library, we will notice that adding parameters to the model to be pruned is part of the control and gauging methods the API uses TensorFlow (2022b). This explains why the parameters of the pruned model are larger than those of the input model. The pruning could have been more aggressive. However, when the model passed 30% sparsity, it suffered from a loss of accuracy.

Now we will investigate the possible reasons for the increase in inference time and the no reduction in space used after the quantization stage running in ESP32, as seen in Tables 19 and 20. Using the Netron[10] tool, we load the exported baseline model and quantized model to visualize the neural network saved in these files graphically. Examining the images in Annexs A and B, we can notice a change at the time before the first convolutional layer of the quantized model is input. Compared to the baseline model, the QAT method adds one more layer in the format 1x32x32x3. This is for the QAT implementation. This additional layer allows the quantized model to receive floating point inputs instead of just uint8. In this way, the quantized model needs to handle an additional layer whose output is also in 1x32x32x3 format. To exemplify this stretch, we generate the activation maps referring to the first layer of each model in question. These maps can be seen in Figure 18. However, for the ESP32-S2 device, the same behavior was not observed, indicating that the addition of this layer did not add computational complexity to the model compiled for this platform.



(a) (b) (c)

Figure 18 – **(a)** Image used to generate the activation map **(b)** Teacher model activation map **(c)** Activation map of quantized model

The first perception in relation to the activation maps of the teacher model and the quantized model is their resolution. In the teacher model the first activation map has the format 14x14, while in the quantized model the first activation map corresponds to the additional layer (Quantization Layer) and has the format 32x32. Even if no convolutional operation is performed in this quantization layer, the network needs to feed this matrix (image) to the next layer. This in itself already increases the number of operations performed when feeding the input to the neural network.

The latency between the acquisition of an image and the result return was 460ms on the quantized set running in the ESP32, and 196 ms for the same set running in the ESP32-S2 (see Table 25). If the neural network is powered by a camera directly connected to the MCU's communication bus, the total latency time will be lower.

The comparison of the works discussed in Chapter 3 that used the same metrics and dataset as ours can be seen in Table 26. The last column of that table concerns the CoreMark/MHz

[10] Netron is a viewer for neural network, deep learning, and machine learning models (ROEDER'S, 2022)

Table 25 – Total spent time in the image recognition in ESP32 and ESP32-S2 for the pipeline final model.

Board	**Inference Time**
ESP32	460 ms
ESP32-S2	196 ms

Source: The Author

Benchmark score (CMM) obtained by each platform used in the considered works. Regarding accuracy, our model shows promise. It is necessary to increase the final model's accuracy by 10% to put it at a level close to state of the art. Looking at inference time, we are second only to work developed by Lechner, Jantsch e Dinakarrao (2019). The small size of the neural network makes it capable of being installed in more restricted devices than the ESP32 dev boards, such as the ARM Cortex-M4 (LAI; SUDA; CHANDRA, 2018).

Table 26 – Results obtained by some of the papers located in the systematic review.

Author(s)	**Accuracy**	**Inference Time**	**Model Size**	**CMM**
Desai, Sinha e El-Sharkawy (2020)	84.32%	≈ 120 ms	3.9 MB	5.03
Lechner, Jantsch e Dinakarrao (2019)	96.53%	≈ 27 ms	-	5.92
Yao et al. (2017)	98.10%	≈ 104 ms	-	5.92
Proposed Approach (ESP32)	85.91%	≈ 80 ms	59.2 KB	4.13
Proposed Approach (ESP32-S2)	86.65%	≈ 83 ms	59.2 KB	1.97

Source: The Author

Considering the cost of each platform concerning the sale of a thousand units and the score obtained using the CoreMark/MHz metric (CMM), we agreed on the US$/CMM metric (US$/CoreMark per MegaHertz), which represents the CoreMark/MHz score for each dollar invested in the platform. After applying this metric, we arrive at the values seen in Table 27. The prices quoted in the table were obtained from Digi-Key[11].

Table 27 – Value in US$ for each CoreMark/MHz point of the Zynq 7000, NXP i.MX RT1060 and ESP32 FPGA platforms.

Platform	**Price (US$)**	**US$/CMM**
NXP i.MX RT1060	19.71	3.92
FPGA XC7Z020-L1CLG484I	184.60	31.18
FPGA XC7Z020-2CLG400I	167.70	28.33
ESP32	4.08	0.98
ESP32-S2	2.40	1.22

Source: The Author, with prices obtained at Digi-Key virtual store

In this chapter, we discussed about a compress pipeline for embedded neural network models in edge devices. The proposed technique creates a model to recognize images using

[11] Digikey: https://www.digikey.com

constrained devices performs satisfactorily on ESP32 and ESP32-S2 based boards. In experimental phase, we detected images with 32 x 32 pixels in less than eighty five milliseconds. The experimental results have presented promising responses of testing classification accuracy with the detected objects. On ESP32 and ESP32-S2 we can process twelve frames per second at the cost of fewer than five dollars. With a reduced cost to this point and with the image processing capacity achieved in this work, the Espressif ESP32 and ESP32-S2 platforms are a strong candidates when a project considers economies of scale, such as automotive products. These results show that the use of constrained edge devices for the implementation of TSDR equipment is possible. Our work demonstrates that real-time traffic sign detection using such devices is feasible. Our pipeline needs further improvement, but despite this scenario, its affordability can be a game changer specially in large scale and ubiquitous systems.

6
Conclusion

This work focused on proving the ability of the proposed pipeline to efficiently reduce a convolutional neural network for use in TSDR solutions. In addition to proving the ability to embed a neural network with good efficiency in a constrained device. The efficiency of the pipeline was evaluated in terms of the accuracy of the models generated, the size of the final neural network, and the speed required to perform inference for a given image. To this end, the experiments sought to assess the metrics generated at each step, from the development of the neural model, its training and compression in the Google Colab environment, including the efficiency measures of the model embedded in the ESP32 and ESP32-S2. The metrics selected to evaluate the efficiency of the work were based on the works discussed in Section 3.

Considering the results obtained and discussed in the previous chapter, we realize that the pipeline did not prove linear in reducing the size and accelerating inference speed at each processing stage. It was expected that at the end of the quantization stage, we would obtain a smaller model than the pruning stage and faster, which has not occurred. However, the pruning stage did not show an increase in efficiency concerning execution time since there was only a 1ms reduction in the time required to perform inference on an image in the model executed on the ESP32.

The objective of compressing a neural network and embedding it on a constrained device has been achieved. The distillation step would be enough to ship a model on an ESP32 or ESP32-S2. However, it is worth noting that even with the compression at this point, the model will not necessarily be placed directly into the fast memory of the device. In our case, this only happened when we reached the quantization step. However, due to the TensorFlow Lite API quantization mechanisms, the model generated in the last stage of the pipeline was slower than the one in the previous step, spending a total of 80 ms in the execution of a complete inference against 46 ms for the model obtained in the pruning step, executing on the ESP32 development board. The inference time considering the pruned and quantized models running on the ESP32-S2 was

234 ms and 83 ms, respectively. Comparing the metrics of the models executed on the test-beds with the Google Colab environment, we observe a decrease in accuracy values. Remember that the metrics shown in Table 23 refer to the test set. The accuracy value between the baseline and the quantized model running in ESP32 had a 9.27% reduction and 8.26% to the ESP32-S2.

6.1 Challenges and Limitations

When the work was deployed on the free version of Google Colab, it turned out to be problematic due to the time limit imposed by the platform. The processes involved in the Bayesian Optimization step by itself overloaded this time, which was approximately 5 hours. This situation led us to sign up for the Google Colab Pro version.

Another situation that delayed the development schedule was testing with the Sipeed Maix Bit devboard. This development board has a K210 CPU based on a 64-bit architecture and dual-core RISC-V running at 400 MHz and also has a KPU (neural network processor) to assist in the inference of the neural networks (TORRES-SáNCHEZ; ALASTRUEY-BENEDé; TORRES-MORENO, 2020). Although promising, the manufacturer does not provide the necessary support for the evolution of the implementations and the platform documentation has lagged behind the constant updates from other manufacturers. Thus, it was not possible to run our neural model on this development board as initially planned.

The complexity of parameter adjustment was also an issue in the ESP-IDF, the official development environment of Espressif, leading us to choose the Arduino IDE environment. The choice of this framework takes away some of the development options of the ESP32, such as more efficient control of memory usage and adjustments in memory section allocation, in addition to not allowing the use of the ESP32-C3, a version of the ESP32 with a RISC-V processor, in a stable way, which could improve the inference time of the developed neural networks.

6.2 Future Work

In order to continue this work, we suggest a number of improvements and new approaches, such as the following ideas as possible new directions:

(a) For the teacher model: develop a new, more robust model that can improve the accuracy of the test dataset while maintaining the same level of accuracy for training and validation.

(b) Evaluate the parameters used in the pruning step. Look for new ways to increase pruning within the student model and to evaluate the efficiency of this stage. Observe whether inefficiencies in pruning was due misconfiguration or to limitations of the model format generated in the previous step.

(c) Analyze the quantization stage and attempt to exclude additional input layer generated by TensorFlow Lite. Reverse-engineer the firmware ELF file to identify the inference operations and verify the convolution steps in order to quantify the changes in the assembly code. The goal here is to understand why QAT increased the inference time when the opposite was expected.

(d) Investigate the possibility of embedding the neural network into a development board with a built-in camera, such as ESP32-CAM. In this way, it would be possible to evaluate the overall latency of the process, taking into account the capture, image pre-processing, inference and display of the result by the device itself.

(e) Convert some arithmetic operations used within the TensorFlow Lite libraries to assembly-based versions using Espressif's DSP library. This is expected to increase the final inference speed.

(f) Investigate the computational complexity increase between the versions of the PlatformIO plugin used in the tests. With this, we hope to find the reason for the increase in total inference time and the divergence in the time to computations performed on convolution layer two of the teacher model.

Bibliography

ABADI, M. et al. *TensorFlow: A system for large-scale machine learning*. 2016. Citado na página 29.

ACADEMY, D. S. misc, *CS231n: Convolutional Neural Networks for Visual Recognition*. 2019. Disponível em: <http://www.deeplearningbook.com.br>. Citado na página 25.

ACADEMY, D. S. *Deep Learning Book*. 2019. Disponível em: <http://www.deeplearningbook.com.br/>. Citado na página 80.

ALVES, G. misc, *Entendendo Redes COnvolucionais (CNNs)*. 2018. Disponível em: <https://medium.com/neuronio-br/entendendo-redes-convolucionais-cnns-d10359f21184>. Citado na página 27.

Ayi, M.; El-Sharkawy, M. Real-time implementation of rmnv2 classifier in nxp bluebox 2.0 and nxp i.mx rt1060. In: *2020 IEEE Midwest Industry Conference (MIC)*. [S.l.: s.n.], 2020. v. 1, p. 1–4. Citado na página 52.

Baicu, D. et al. Real time image recognition based on low cost processing platform. In: *2019 14th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS)*. [S.l.: s.n.], 2019. p. 241–246. Citado na página 18.

BRASIL, O. N. U. do. *Acidentes de trânsito matam 1,25 milhão de pessoas no mundo por ano*. [S.l.], 2018. Disponível em: <https://nacoesunidas.org/acidentes-de-transito-matam-125-milhao-de-pessoas-no-mundo-por-ano/>. Citado na página 16.

BROCHU, E.; CORA, V. M.; FREITAS, N. de. *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*. 2010. Citado na página 33.

CARTHROTTLE. *What Does 'Economies Of Scale' Mean, And Why Is It Important To Cars?* 2016. Disponível em: <https://www.carthrottle.com/post/what-does-economies-of-scale-mean-and-why-is-it-important-to-cars/>. Citado na página 15.

Chappa, R. T. N. V. S.; El-Sharkawy, M. Squeeze-and-excitation squeezenext: An efficient dnn for hardware deployment. In: *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. [S.l.: s.n.], 2020. p. 0691–0697. Citado 3 vezes nas páginas 54, 57, and 58.

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. Citado 4 vezes nas páginas 55, 57, 58, and 59.

CHON, S. *What it Takes to do Effi cient and Cost-Effective Real-Time Control with a Single Microcontroller: The C2000™ Advantage*. [S.l.], 2008. 7 p. Disponível em: <https://www.ti.com/lit/wp/spry157/spry157.pdf?ts=1660011020186>. Citado na página 19.

Chougule, S. et al. An efficient encoder-decoder cnn architecture for reliable multilane detection in real time. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. [S.l.: s.n.], 2018. p. 1444–1451. Citado na página 18.

CORDTS, M. et al. *The Cityscapes Dataset for Semantic Urban Scene Understanding*. 2016. Citado na página 45.

COURBARIAUX, M. et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. Citado na página 50.

Desai, S. R.; Sinha, D.; El-Sharkawy, M. Image classification on nxp i.mx rt1060 using ultra-thin mobilenet dnn. In: *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. [S.l.: s.n.], 2020. p. 0474–0480. Citado 4 vezes nas páginas 52, 56, 57, and 83.

DIETTERICH, T. G. Machine learning. In: ____. *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003. p. 1056–1059. ISBN 0470864125. Citado na página 24.

EBERMAM, E.; KROHLING, R. Uma introdução compreensiva às redes neurais convolucionais: Um estudo de caso para reconhecimento de caracteres alfabéticos. *Revista de Sistemas de Informação da FSMA*, v. 22, p. 49–59, 2018. Disponível em: <http://www.fsma.edu.br/si/edicao21/FSMA_SI_2018_1_Principal_8.html>. Citado 2 vezes nas páginas 26 and 27.

EEMBC. *CoreMark*. 2020. Disponível em: <https://www.eembc.org/coremark/>. Citado na página 37.

EMBC. *CoreMark® An EEMBC Benchmark*. 2021. Disponível em: <https://www.eembc.org/coremark/>. Citado na página 37.

ESPRESSIF. *ESP32-S2 Family Datasheet*. 2020. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf>. Citado na página 36.

ESPRESSIF. *Heap Memory Allocation*. 2022. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/mem_alloc.html>. Citado na página 75.

EVERINGHAM, M. et al. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, v. 111, n. 1, p. 98–136, jan. 2015. Citado na página 45.

FIRDAUS, F. F.; NUGROHO, H. A.; SOESANTI, I. Deep neural network with hyperparameter tuning for detection of heart disease. In: *2021 IEEE Asia Pacific Conference on Wireless and Mobile (APWiMob)*. IEEE, 2021. Disponível em: <https://doi.org/10.1109%2Fapwimob51111.2021.9435250>. Citado na página 33.

FRITSCH, J.; KUEHNL, T.; GEIGER, A. A new performance measure and evaluation benchmark for road detection algorithms. In: *International Conference on Intelligent Transportation Systems (ITSC)*. [S.l.: s.n.], 2013. Citado na página 45.

GALON, S.; LEVY, M. *Exploring CoreMark™ – A Benchmark Maximizing Simplicity and Efficacy*. [S.l.], 2008. 7 p. Disponível em: <https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf>. Citado na página 37.

GLOBALLOGIC. *ADAS Presentation*. 2018. Citado na página 16.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. Citado 2 vezes nas páginas 24 and 25.

GOOGLE. *Google Colab*. 2020. Disponível em: <https://colab.research.google.com/?utm_source=scs-index>. Citado na página 29.

GOU, J. et al. Knowledge distillation: A survey. *International Journal of Computer Vision*, Springer, v. 129, n. 6, p. 1789–1819, 2021. Citado na página 33.

Han, Y.; Oruklu, E. Traffic sign recognition based on the nvidia jetson tx1 embedded system using convolutional neural networks. In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. [S.l.: s.n.], 2017. p. 184–187. Citado 4 vezes nas páginas 18, 47, 56, and 57.

HAYKIN, S. S. *Neural networks and learning machines*. Third. Upper Saddle River, NJ: Pearson Education, 2009. Citado na página 21.

HINTON, G.; SRIVASTAVA, N.; SWERSKY, K. *Neural networks for machine learning Lecture 6a: Overview of mini-baych gradient descent*. 2012. Citado 2 vezes nas páginas 31 and 63.

HINTON, G.; VINYALS, O.; DEAN, J. Distilling the knowledge in a neural network. In: *NIPS Deep Learning and Representation Learning Workshop*. [s.n.], 2015. Disponível em: <http://arxiv.org/abs/1503.02531>. Citado na página 67.

HINTON, G. et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, v. 2, n. 7, 2015. Citado na página 33.

HOSSEINI, H. et al. On the limitation of convolutional neural networks in recognizing negative images. In: *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. [S.l.: s.n.], 2017. p. 352–358. Citado na página 62.

HOUBEN, S. et al. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In: *International Joint Conference on Neural Networks*. [S.l.: s.n.], 2013. Citado na página 36.

HOUBEN, S. et al. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In: *International Joint Conference on Neural Networks*. [S.l.: s.n.], 2013. Citado na página 45.

HUBARA, I. et al. *Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*. 2016. Citado na página 31.

HUBEL, D.; WIESEL, T. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology*, v. 160, p. 106–154, 1962. Citado na página 24.

IBGE. *IBGE | Frota de Veículos*. 2018. Disponível em: <https://cidades.ibge.gov.br/brasil/pesquisa/22/28120?ano=2018>. Citado na página 15.

IEEE-754. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, p. 1–84, 2019. Citado na página 31.

INDEPENDENT. *TESLA CARS CAN NOW 'READ' ROAD SIGNS AFTER MAJOR UPDATE*. 2020. Citado na página 17.

INTERNATIONAL, S. *SAE Standards News: J3016 automated-driving graphic update*. 2019. Citado na página 17.

IOFFE, S.; SZEGEDY, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. Citado na página 30.

JACOB, B.; CHEN, B. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Citado na página 32.

JANG, C. et al. Data debiased traffic sign recognition using msers and cnn. In: *2016 International Conference on Electronics, Information, and Communications (ICEIC)*. [S.l.: s.n.], 2016. p. 1–4. Citado na página 62.

JI, S. et al. Kullback–leibler divergence metric learning. *IEEE Transactions on Cybernetics*, Institute of Electrical and Electronics Engineers (IEEE), v. 52, n. 4, p. 2047–2058, apr 2022. Disponível em: <https://doi.org/10.1109%2Ftcyb.2020.3008248>. Citado na página 34.

Jose, G. et al. Real-time object detection on low power embedded platforms. In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. [S.l.: s.n.], 2019. p. 2485–2492. Citado 3 vezes nas páginas 51, 56, and 57.

Joshi, I. et al. Performance of different optimizers for traffic sign classification. In: *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. [S.l.: s.n.], 2019. p. 1–7. Citado na página 64.

KARPATHY, A. *CS231n Convolutional Neural Networks for Visual Recognition*. 2016. University Lecture. Disponível em: <https://cs231n.github.io/convolutional-networks/>. Citado 3 vezes nas páginas 25, 26, and 27.

Katare, D.; El-Sharkawy, M. Autonomous embedded system enabled 3-d object detector: (with point cloud and camera). In: *2019 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. [S.l.: s.n.], 2019. p. 1–6. Citado 3 vezes nas páginas 50, 56, and 57.

KERAS. *Keras*. 2020. Disponível em: <https://keras.io/>. Citado na página 29.

KIM, T. et al. *Comparing Kullback-Leibler Divergence and Mean Squared Error Loss in Knowledge Distillation*. arXiv, 2021. Disponível em: <https://arxiv.org/abs/2105.08919>. Citado na página 67.

KITCHENHAM, B. A. Systematic review in software engineering: Where we are and where we should be going. In: *Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies*. New York, NY, USA: Association for Computing Machinery, 2012. (EAST '12), p. 1–2. ISBN 9781450315098. Disponível em: <https://doi.org/10.1145/2372233.2372235>. Citado na página 39.

Kohavi, R. and Provost, F. Glossary of Terms. *Kluwer Academic Publishers*, v. 30, n. Editorial for the Special Issue on Applications of Machine Learning and the Knowledge Discovery Process, p. 271–274, 1998. Citado na página 27.

Konar, J.; Khandelwal, P.; Tripathi, R. Comparison of various learning rate scheduling techniques on convolutional neural network. In: *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*. [S.l.: s.n.], 2020. p. 1–5. Citado na página 29.

Konar, J.; Khandelwal, P.; Tripathi, R. Comparison of various learning rate scheduling techniques on convolutional neural network. In: *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*. [S.l.: s.n.], 2020. p. 1–5. Citado na página 29.

KRAMER, O.; CIAURRI, D. E.; KOZIEL, S. Derivative-free optimization. In: *Computational Optimization, Methods and Algorithms*. Springer Berlin Heidelberg, 2011. p. 61–83. Disponível em: <https://doi.org/10.1007%2F978-3-642-20859-1_4>. Citado na página 32.

KRIZHEVSKY, A.; NAIR, V.; HINTON, G. Cifar-10 (canadian institute for advanced research). Disponível em: <http://www.cs.toronto.edu/~kriz/cifar.html>. Citado na página 45.

KRIZHEVSKY, A.; NAIR, V.; HINTON, G. Cifar-10 (canadian institute for advanced research). Disponível em: <http://www.cs.toronto.edu/~kriz/cifar.html>. Citado na página 65.

Kumar Reddy, R. V.; Srinivasa Rao, B.; Raju, K. P. Handwritten hindi digits recognition using convolutional neural network with rmsprop optimization. In: *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*. [S.l.: s.n.], 2018. p. 45–51. Citado na página 63.

Lai, C. Y. et al. A light weight multi-head ssd model for adas applications. In: *2020 International Conference on Pervasive Artificial Intelligence (ICPAI)*. [S.l.: s.n.], 2020. p. 1–6. Citado 3 vezes nas páginas 53, 56, and 57.

LAI, L.; SUDA, N.; CHANDRA, V. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *CoRR*, abs/1801.06601, 2018. Disponível em: <http://arxiv.org/abs/1801.06601>. Citado na página 83.

Lechner, M.; Jantsch, A.; Dinakarrao, S. M. P. Resconn: Resource-efficient fpga-accelerated cnn for traffic sign classification. In: *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*. [S.l.: s.n.], 2019. p. 1–6. Citado 4 vezes nas páginas 50, 56, 57, and 83.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *Nature*, v. 521, n. 7553, p. 436–444, 2015. Disponível em: <https://doi.org/10.1038/nature14539>. Citado 2 vezes nas páginas 24 and 45.

LECUN, Y. et al. Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE*. [S.l.: s.n.], 1998. p. 2278–2324. Citado na página 62.

Lee, H. S.; Kim, K. Simultaneous traffic sign detection and boundary estimation using convolutional neural network. *IEEE Transactions on Intelligent Transportation Systems*, v. 19, n. 5, p. 1652–1663, 2018. Citado 3 vezes nas páginas 48, 56, and 57.

Lee, Y. et al. Optimization for object detector using deep residual network on embedded board. In: *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. [S.l.: s.n.], 2016. p. 1–4. Citado 4 vezes nas páginas 46, 56, 57, and 59.

LI, H. et al. Patternnet: Visual pattern mining with deep neural network. In: *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*. New York, NY, USA: Association for Computing Machinery, 2018. (ICMR '18), p. 291–299. ISBN 9781450350464. Disponível em: <https://doi.org/10.1145/3206025.3206039>. Citado na página 24.

LIANG, M. et al. Traffic sign detection by roi extraction and histogram features-based recognition. In: IEEE. *The 2013 international joint conference on Neural networks (IJCNN)*. [S.l.], 2013. p. 1–8. Citado na página 49.

LIU, Y. et al. SuperPruner: Automatic neural network pruning via super network. *Scientific Programming*, Hindawi Limited, v. 2021, p. 1–11, sep 2021. Disponível em: <https://doi.org/10.1155%2F2021%2F9971669>. Citado na página 34.

LOPEZ-MONTIEL, M. et al. Evaluation of deep learning algorithms for traffic sign detection to implement on embedded systems. In: *Recent Advances of Hybrid Intelligent Systems Based on Soft Computing*. Springer International Publishing, 2020. p. 95–115. Disponível em: <https://doi.org/10.1007/978-3-030-58728-4_5>. Citado na página 44.

MATHIAS, M. et al. Traffic sign recognition—how far are we from the solution? In: IEEE. *The 2013 international joint conference on Neural networks (IJCNN)*. [S.l.], 2013. p. 1–8. Citado na página 49.

MATLAB. misc, *Introducing Deep Learning with MATLAB*. 2021. Disponível em: <https://www.mathworks.com/campaigns/offers/deep-learning-with-matlab.html>. Citado na página 24.

MCCULLOCH, W.; PITTS, W. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, v. 5, p. 127–147, 1943. Citado na página 21.

MERTZ, D. *The GNU Text Utilities*. 2022. Disponível em: <https://gnosis.cx/publish/programming/text_utils.html>. Citado na página 71.

NAGPAL, R. et al. Real-time traffic sign recognition using deep network for embedded platforms. *Electronic Imaging*, v. 2019, n. 15, p. 33–1–33–8, 2019. ISSN 2470-1173. Disponível em: <https://www.ingentaconnect.com/content/ist/ei/2019/00002019/00000015/art00006>. Citado 3 vezes nas páginas 54, 56, and 57.

NIELSEN, M. A. misc, *Neural Networks and Deep Learning*. Determination Press, 2018. Disponível em: <http://neuralnetworksanddeeplearning.com/>. Citado na página 24.

NOVAKOVIĆ, J. D. et al. Evaluation of classification models in machine learning. *Theory and Applications of Mathematics & Computer Science*, v. 7, n. 1, p. 39–46, 2017. Citado na página 27.

OBSERVADOR. *90% DOS ACIDENTES SÃO CAUSADOS POR FALHAS HUMANAS, ALERTA OBSERVATÓRIO*. [S.l.], 2018. Disponível em: <http://www.onsv.org.br/90-dos-acidentes-sao-causados-por-falhas-humanas-alerta-observatorio/>. Citado na página 16.

ONGSULEE, P. Artificial intelligence, machine learning and deep learning. In: IEEE. *2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE)*. [S.l.], 2017. p. 1–6. Citado na página 23.

OSDEV. *ELF*. 2022. Disponível em: <https://wiki.osdev.org/ELF>. Citado na página 74.

PCMAG. *Tesla Software Update Helps Cars ŚeeŚpeed Limit Signs*. 2020. Citado na página 17.

Petrova, K. A. Computer vision system for robotic complex based on arduino and raspberry pi. In: *2017 IEEE II International Conference on Control in Technical Systems (CTS)*. [S.l.: s.n.], 2017. p. 370–372. Citado na página 18.

QI, C. R. et al. Frustum pointnets for 3d object detection from rgb-d data. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2018. p. 918–927. Citado na página 50.

Ravindran, R. et al. Traffic sign identification using deep learning. In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. [S.l.: s.n.], 2019. p. 318–323. Citado 3 vezes nas páginas 52, 56, and 57.

RETORNO, M. *Economia de Escala*. 2019. Disponível em: <https://maisretorno.com/blog/termos/e/economia-de-escala>. Citado na página 15.

RICARDO. *Cost benchmarking of production ADAS systems*. 2019. Disponível em: <https://rsc.ricardo.com/capabilities/mobility-strategy/future-mobility-conference/2019-conference-insights/cost-benchmarking-of-production-adas-systems>. Citado na página 16.

ROEDER'S, L. *Netron*. 2022. Disponível em: <https://github.com/lutzroeder/netron>. Citado na página 82.

ROMERO, A. et al. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014. Citado na página 33.

RUDER, S. An overview of gradient descent optimization. p. 1–14, 2016. Citado na página 63.

SALTI, S. et al. A traffic sign detection pipeline based on interest region extraction. In: IEEE. *The 2013 International Joint Conference on Neural Networks (IJCNN)*. [S.l.], 2013. p. 1–7. Citado na página 49.

SENHORAS, E. *A Indústria Automobilística Sob Enfoque Estático e Dinâmico: Uma Análise Teórica*. 2005. Citado na página 15.

SEPAHVAND, M.; ABDALI-MOHAMMADI, F.; TAHERKORDI, A. Teacher–student knowledge distillation based on decomposed deep feature representation for intelligent mobile applications. *Expert Systems with Applications*, v. 202, p. 117474, 2022. ISSN 0957-4174. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0957417422008053>. Citado na página 33.

Shabarinath, B. B.; Muralidhar, P. Convolutional neural network based traffic-sign classifier optimized for edge inference. In: *2020 IEEE REGION 10 CONFERENCE (TENCON)*. [S.l.: s.n.], 2020. p. 420–425. Citado 4 vezes nas páginas 53, 56, 57, and 61.

Shi, W. et al. An fpga-based hardware accelerator for traffic sign detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 25, n. 4, p. 1362–1372, 2017. Citado 3 vezes nas páginas 47, 56, and 57.

SILVA, I. N. d.; SPATTI, D. H.; FLAUZINO, R. A. *Redes neurais artificiais para engenharia e ciências aplicadas*. [S.l.]: Artliber Editora, 2010. Citado 2 vezes nas páginas 22 and 23.

SILVA, R. et al. Construction of brazilian regulatory traffic sign recognition dataset. In: TAVARES, J. M. R. S.; PAPA, J. P.; HIDALGO, M. G. (Ed.). *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. Cham: Springer International Publishing, 2021. p. 163–172. ISBN 978-3-030-93420-0. Citado na página 62.

SINDAGI, V. A.; PATEL, V. M. A survey of recent advances in cnn-based single image crowd counting and density estimation. *Pattern Recognition Letters*, v. 107, p. 3–16, 2018. ISSN 0167-8655. Video Surveillance-oriented Biometrics. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167865517302398>. Citado na página 24.

Sisido, F. et al. Traffic signs recognition system with convolution neural networks. In: *2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)*. [S.l.: s.n.], 2018. p. 339–344. Citado 3 vezes nas páginas 54, 56, and 57.

SRIVASTAVA, N. et al. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, JMLR. org, v. 15, n. 1, p. 1929–1958, 2014. Citado na página 34.

STALLKAMP, J. et al. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, n. 0, p. –, 2012. ISSN 0893-6080. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0893608012000457>. Citado na página 45.

TAVARES, H. et al. A non-intrusive approach for smart power meter. In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 2018. Disponível em: <https://doi.org/10.1109%2Findin.2018.8471960>. Citado 2 vezes nas páginas 72 and 73.

TENSORFLOW. *Get started with microcontrollers*. 2022. Disponível em: <https://www.tensorflow.org/lite/microcontrollers/get_started_low_level>. Citado na página 73.

TENSORFLOW. *Model Optimization*. 2022. Disponível em: <https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras>. Citado na página 81.

TENSORFLOW. *Training checkpoints*. 2022. Disponível em: <https://www.tensorflow.org/guide/checkpoint>. Citado na página 66.

TIMOFTE, R.; ZIMMERMANN, K.; GOOL, L. van. Multi-view traffic sign detection, recognition, and 3d localisation. In: *Ninth IEEE Computer Society Workshop on Application of Computer Vision*. Snowbird, Utah, USA: [s.n.], 2009. p. 1–8. ISBN 978-1-4244-5496-9. ISSN 1550-5790. Citado na página 45.

Timofte, R.; Zimmermann, K.; Van Gool, L. Multi-view traffic sign detection, recognition, and 3d localisation. In: *2009 Workshop on Applications of Computer Vision (WACV)*. [S.l.: s.n.], 2009. p. 1–8. Citado na página 45.

TORRES-SáNCHEZ, E.; ALASTRUEY-BENEDé, J.; TORRES-MORENO, E. Developing an ai iot application with open software on a risc-v soc. In: *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*. [S.l.: s.n.], 2020. p. 1–6. Citado na página 86.

TURI, A. *Performance analysis in the automotive industry*. [S.l.]: Timişoara: Editura Politehnica, 2015. Citado na página 15.

VERGE, T. *Tesla's new self-driving chip is here, and this is your best look yet*. 2019. Citado na página 17.

VICTORIA, A. H.; MARAGATHAM, G. Automatic tuning of hyperparameters using bayesian optimization. *Evolving Systems*, Springer Science and Business Media LLC, v. 12, n. 1, p. 217–223, may 2020. Disponível em: <https://doi.org/10.1007%2Fs12530-020-09345-2>. Citado na página 32.

WANG, G. et al. A robust, coarse-to-fine traffic sign detection method. In: IEEE. *The 2013 international joint conference on neural networks (IJCNN)*. [S.l.], 2013. p. 1–5. Citado na página 49.

WANG, J.; WANG, W.; ZHOU, A. The faster detection and recognition of traffic signs based on cnn. In: *2019 IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. [S.l.: s.n.], 2019. p. 907–914. Citado na página 62.

WANG, L.; YOON, K. Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks. *CoRR*, abs/2004.05937, 2020. Disponível em: <https://arxiv.org/abs/2004.05937>. Citado na página 34.

Wang, Y. et al. Low power convolutional neural networks on a chip. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. [S.l.: s.n.], 2016. p. 129–132. Citado na página 31.

William, M. M. et al. Traffic signs detection and recognition system using deep learning. In: *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*. [S.l.: s.n.], 2019. p. 160–166. Citado 3 vezes nas páginas 51, 56, and 57.

Wu, B. et al. Traffic sign recognition with light convolutional networks. In: *2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*. [S.l.: s.n.], 2018. p. 1–2. Citado 3 vezes nas páginas 49, 56, and 57.

WU, J. et al. Hyperparameter optimization for machine learning models based on bayesian optimizationb. *Journal of Electronic Science and Technology*, v. 17, n. 1, p. 26–40, 2019. ISSN 1674-862X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1674862X19300047>. Citado na página 32.

Yamada, Y. et al. A 20.5 tops multicore soc with dnn accelerator and image signal processor for automotive applications. *IEEE Journal of Solid-State Circuits*, v. 55, n. 1, p. 120–132, 2020. Citado na página 18.

Yao, Y. et al. Fpga-based convolution neural network for traffic sign recognition. In: *2017 IEEE 12th International Conference on ASIC (ASICON)*. [S.l.: s.n.], 2017. p. 891–894. Citado 4 vezes nas páginas 48, 56, 57, and 83.

YEOM, S.-K. et al. Pruning by explaining: A novel criterion for deep neural network pruning. *Pattern Recognition*, v. 115, p. 107899, 2021. ISSN 0031-3203. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0031320321000868>. Citado na página 34.

YU, F. et al. *BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning*. 2020. Citado na página 45.

Zeng, Z.; Gong, Q.; Zhang, J. Cnn model design of gesture recognition based on tensorflow framework. In: *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. [S.l.: s.n.], 2019. p. 1062–1067. Citado 2 vezes nas páginas 34 and 35.

ZHANG, Q. et al. Recent advances in convolutional neural network acceleration. *Neurocomputing*, v. 323, p. 37–51, 2019. ISSN 0925-2312. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0925231218311007>. Citado na página 24.

ZHU, M.; GUPTA, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. In: . arXiv, 2017. Disponível em: <https://arxiv.org/abs/1710.01878>. Citado na página 69.

Appendix

APPENDIX A – Distiller Class

The first part of the Distiller class concerns the declaration of the constructor. At this point the teacher and student models are entered as parameters.

```
class Distiller(keras.Model):
        def __init__(self, student, teacher):
                super(Distiller, self).__init__()
                self.teacher = teacher
                self.student = student
        .
        .
        .
```

The compile function defines the loss function, the optimizer and the metrics. The variable distillation_loss_fn is the loss function Kullback–Leibler divergence that calculates between soft student predictions and soft teacher predictions. The parameter student_loss_fn is the loss function of difference between student predictions and ground-truth.

```
        def compile(
                self, optimizer,
                metrics,
                student_loss_fn,
                distillation_loss_fn,
                alpha,
                temperature,
                ):
        super(Distiller, self).compile(optimizer=optimizer,
        metrics=metrics)
        self.student_loss_fn = student_loss_fn
        self.distillation_loss_fn = distillation_loss_fn
        self.alpha = alpha
        self.temperature = temperature
```

In the function below we define the training mechanism used for distilling the knowledge from the teacher model to the student model.

```
    def train_step(self, data):
        x, y = data
        teacher_predictions = self.teacher(x, training=False)
        with tf.GradientTape() as tape:
            student_predictions = self.student(x, training=True)
        student_loss = self.student_loss_fn(y, student_predictions)
        distillation_loss = self.distillation_loss_fn(
                tf.nn.softmax(teacher_predictions /
                self.temperature, axis=1),
                tf.nn.softmax(student_predictions /
                self.temperature, axis=1),
        )
        loss = self.alpha * student_loss + (1 - self.alpha) *
        distillation_loss
        trainable_vars = self.student.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))
        self.compiled_metrics.update_state(y, student_predictions)
        results = {m.name: m.result() for m in self.metrics}
        results.update(
                {"student_loss": student_loss,
                "distillation_loss": distillation_loss}
        )
        return results
```

In the definition below we see the function responsible for the test step of the distilled model.

```
    def test_step(self, data):
        x, y = data
        y_prediction = self.student(x, training=False)
        student_loss = self.student_loss_fn(y, y_prediction)
        self.compiled_metrics.update_state(y, y_prediction)
        results = {m.name: m.result() for m in self.metrics}
        results.update({"student_loss": student_loss})
        return results
```

APPENDIX B – Teacher Model Sequential

The following code defines the construction of the teacher model using the TensorFlow API. As described in Section 4.2, the model is nothing more than a convolutional arrangement. Thus we will only display one of the arrangements, since between each set of layers only the hyperparameters are changed.

```
teacher = models.Sequential()
teacher.add(layers.Conv2D(FILTERS_0, (5, 5), (2, 2),
        bias_initializer='random_normal', padding='valid',
        kernel_regularizer=regularizers.l2(WEIGHT_DECAY_0),
        kernel_initializer='random_normal', input_shape=(32, 32, 3)))
teacher.add(layers.BatchNormalization())
teacher.add(layers.Activation('relu'))
teacher.add(layers.MaxPooling2D((2, 2), padding='valid'))
.
.
.
teacher.add(layers.Flatten())
teacher.add(layers.Dense(43, activation='softmax',
kernel_regularizer=regularizers.l2(WEIGHT_DECAY_DENSE)))
```

APPENDIX C – Student Model Sequential

The following code defines the construction of the teacher model using the TensorFlow API. As described in Section 4.2, the model is nothing more than a convolutional arrangement. Thus we will only display one of the arrangements, since between each set of layers only the hyperparameters are changed.

```
student = models.Sequential()
student.add(layers.Conv2D(STUDENT_FILTERS_0, (5, 5), (2, 2),
        bias_initializer='random_normal',
        kernel_regularizer=regularizers.l2(STUDENT_WEIGHT_DECAY_0),
        kernel_initializer='random_normal', padding='valid', input_shape=(32, 32,
student.add(layers.BatchNormalization())
student.add(layers.Activation('relu'))
student.add(layers.MaxPooling2D((2, 2), padding='valid'))
.
.
.
student.add(layers.Flatten())
student.add(layers.Dense(43, activation='softmax',
kernel_regularizer=regularizers.l2(STUDENT_WEIGHT_DECAY_DENSE)))
```

APPENDIX D – Classification Report - Teacher Model embeded in ESP32 DevBoard

```
Classification Report

      precision    recall  f1-score   support
   0     0.9333    0.9333    0.9333        60
   1     0.9658    0.9819    0.9738       720
   2     0.9633    0.9787    0.9709       750
   3     0.9479    0.9711    0.9594       450
   4     0.9772    0.9742    0.9757       660
   5     0.9372    0.9476    0.9424       630
   6     0.9565    0.8800    0.9167       150
   7     0.9608    0.9800    0.9703       450
   8     0.9842    0.9689    0.9765       450
   9     0.9419    0.9792    0.9602       480
  10     0.9954    0.9742    0.9847       660
  11     0.9206    0.9381    0.9292       420
  12     0.9739    0.9203    0.9463       690
  13     0.9930    0.9889    0.9910       720
  14     0.9712    1.0000    0.9854       270
  15     0.9204    0.9905    0.9541       210
  16     1.0000    1.0000    1.0000       150
  17     1.0000    0.9083    0.9520       360
  18     0.9824    0.8564    0.9151       390
  19     1.0000    1.0000    1.0000        60
  20     0.7826    1.0000    0.8780        90
  21     0.7848    0.6889    0.7337        90
  22     0.8016    0.8417    0.8211       120
  23     0.8555    0.9867    0.9164       150
  24     0.9753    0.8778    0.9240        90
  25     0.9607    0.9688    0.9647       480
  26     0.9175    0.9889    0.9519       180
  27     0.5536    0.5167    0.5345        60
```

```
          28     0.9583    0.9200    0.9388       150
          29     0.8208    0.9667    0.8878        90
          30     0.8862    0.7267    0.7985       150
          31     0.9401    0.9889    0.9639       270
          32     0.9375    1.0000    0.9677        60
          33     0.9043    0.9905    0.9455       210
          34     0.9375    1.0000    0.9677       120
          35     0.9866    0.9410    0.9633       390
          36     0.9375    1.0000    0.9677       120
          37     0.9831    0.9667    0.9748        60
          38     0.9654    0.9710    0.9682       690
          39     0.9524    0.6667    0.7843        90
          40     0.7699    0.9667    0.8571        90
          41     0.9231    1.0000    0.9600        60
          42     0.9211    0.7778    0.8434        90
    accuracy                         0.9518     12630
   macro avg     0.9251    0.9285    0.9244     12630
weighted avg     0.9531    0.9518    0.9514     12630
```

APPENDIX E – Classification Report - Distilled Model embeded in ESP32 DevBoard

```
Classification Report

              precision    recall  f1-score   support
           0     0.6800    0.5667    0.6182        60
           1     0.8546    0.8819    0.8681       720
           2     0.8882    0.9107    0.8993       750
           3     0.9020    0.9200    0.9109       450
           4     0.9213    0.8697    0.8948       660
           5     0.8368    0.8222    0.8295       630
           6     0.8571    0.8400    0.8485       150
           7     0.9229    0.8244    0.8709       450
           8     0.8208    0.8756    0.8473       450
           9     0.9343    0.9187    0.9265       480
          10     0.9334    0.9348    0.9341       660
          11     0.8285    0.8167    0.8225       420
          12     0.9398    0.8826    0.9103       690
          13     0.9299    0.9583    0.9439       720
          14     0.9809    0.9519    0.9662       270
          15     0.9143    0.9143    0.9143       210
          16     0.8834    0.9600    0.9201       150
          17     0.9617    0.9056    0.9328       360
          18     0.7357    0.6923    0.7133       390
          19     0.6769    0.7333    0.7040        60
          20     0.5984    0.8111    0.6887        90
          21     0.5775    0.4556    0.5093        90
          22     0.8235    0.7000    0.7568       120
          23     0.7470    0.8267    0.7848       150
          24     0.6170    0.3222    0.4234        90
          25     0.8408    0.8583    0.8495       480
          26     0.8432    0.8667    0.8548       180
          27     0.3429    0.4000    0.3692        60
          28     0.8984    0.7667    0.8273       150
          29     0.5455    0.8667    0.6695        90
          30     0.4961    0.4200    0.4549       150
          31     0.8290    0.9519    0.8862       270
          32     0.6829    0.9333    0.7887        60
          33     0.8251    0.8762    0.8499       210
          34     0.8261    0.9500    0.8837       120
          35     0.9301    0.8872    0.9081       390
          36     0.7069    0.6833    0.6949       120
          37     0.5128    0.6667    0.5797        60
          38     0.9034    0.9217    0.9125       690
          39     0.6044    0.6111    0.6077        90
          40     0.5288    0.6111    0.5670        90
          41     0.5536    0.5167    0.5345        60
          42     0.8571    0.6667    0.7500        90
    accuracy                         0.8549     12630
   macro avg     0.7789    0.7849    0.7774     12630
weighted avg     0.8573    0.8549    0.8544     12630
```

APPENDIX F – Classification Report - Pruned Model embeded in ESP32 DevBoard

```
Classification Report

              precision    recall  f1-score   support
           0     0.8000    0.4667    0.5895        60
           1     0.9015    0.8903    0.8959       720
           2     0.8612    0.9600    0.9079       750
           3     0.8958    0.8978    0.8968       450
           4     0.8914    0.9076    0.8994       660
           5     0.8781    0.8349    0.8560       630
           6     0.9247    0.9000    0.9122       150
           7     0.9338    0.8467    0.8881       450
           8     0.8556    0.8822    0.8687       450
           9     0.9453    0.9354    0.9403       480
          10     0.9495    0.9409    0.9452       660
          11     0.8357    0.8476    0.8416       420
          12     0.9698    0.8841    0.9249       690
          13     0.9293    0.9681    0.9483       720
          14     0.8824    0.9444    0.9123       270
          15     0.8955    0.9381    0.9163       210
          16     0.9241    0.9733    0.9481       150
          17     0.9717    0.8583    0.9115       360
          18     0.7688    0.7077    0.7370       390
          19     0.5091    0.4667    0.4870        60
          20     0.7100    0.7889    0.7474        90
          21     0.5410    0.3667    0.4371        90
          22     0.8585    0.7583    0.8053       120
          23     0.7127    0.8600    0.7795       150
          24     0.7692    0.3333    0.4651        90
          25     0.7947    0.8792    0.8348       480
          26     0.7396    0.7889    0.7634       180
          27     0.3538    0.3833    0.3680        60
          28     0.8077    0.8400    0.8235       150
          29     0.5649    0.8222    0.6697        90
          30     0.6628    0.3800    0.4831       150
          31     0.7917    0.9148    0.8488       270
          32     0.8276    0.8000    0.8136        60
          33     0.8371    0.8810    0.8585       210
          34     0.8273    0.9583    0.8880       120
          35     0.9563    0.8410    0.8950       390
          36     0.8191    0.6417    0.7196       120
          37     0.5000    0.5833    0.5385        60
          38     0.8827    0.9377    0.9093       690
          39     0.6289    0.6778    0.6524        90
          40     0.5000    0.6222    0.5545        90
          41     0.4478    0.5000    0.4724        60
          42     0.7037    0.6333    0.6667        90
    accuracy                         0.8598     12630
   macro avg     0.7851    0.7731    0.7726     12630
weighted avg     0.8619    0.8598    0.8580     12630
```

APPENDIX G – Classification Report - Quantized Model embeded in ESP32 DevBoard

```
Classification Report

              precision    recall  f1-score   support
           0     0.6269    0.7000    0.6614        60
           1     0.8858    0.9264    0.9056       720
           2     0.8856    0.9493    0.9163       750
           3     0.8626    0.9067    0.8841       450
           4     0.9249    0.8773    0.9005       660
           5     0.8390    0.8603    0.8495       630
           6     0.8269    0.8600    0.8431       150
           7     0.8889    0.8178    0.8519       450
           8     0.8267    0.8800    0.8525       450
           9     0.9196    0.9292    0.9244       480
          10     0.9641    0.9348    0.9492       660
          11     0.8605    0.8810    0.8706       420
          12     0.9768    0.7942    0.8761       690
          13     0.9348    0.9750    0.9545       720
          14     0.8258    0.9481    0.8828       270
          15     0.8611    0.8857    0.8732       210
          16     0.9250    0.9867    0.9548       150
          17     0.9627    0.7889    0.8672       360
          18     0.7136    0.7795    0.7451       390
          19     0.4828    0.7000    0.5714        60
          20     0.6522    0.8333    0.7317        90
          21     0.6182    0.3778    0.4690        90
          22     0.9263    0.7333    0.8186       120
          23     0.7175    0.8467    0.7768       150
          24     0.5467    0.4556    0.4970        90
          25     0.9413    0.8354    0.8852       480
          26     0.8872    0.6556    0.7540       180
          27     0.5625    0.4500    0.5000        60
          28     0.7706    0.8733    0.8187       150
          29     0.5753    0.9333    0.7119        90
          30     0.5895    0.3733    0.4571       150
          31     0.7928    0.8926    0.8397       270
          32     0.9400    0.7833    0.8545        60
          33     0.8826    0.8952    0.8889       210
          34     0.8440    0.9917    0.9119       120
          35     0.9529    0.8821    0.9161       390
          36     0.8235    0.7000    0.7568       120
          37     0.7241    0.7000    0.7119        60
          38     0.8491    0.9377    0.8912       690
          39     0.6122    0.6667    0.6383        90
          40     0.5752    0.7222    0.6404        90
          41     0.6383    0.5000    0.5607        60
          42     0.8358    0.6222    0.7134        90

    accuracy                         0.8591     12630
   macro avg     0.7966    0.7917    0.7879     12630
weighted avg     0.8635    0.8591    0.8579     12630
```

APPENDIX H – Classification Report - Teacher Model embeded in ESP32-S2 DevBoard

```
Classification Report

              precision    recall  f1-score   support

           0     0.9423    0.8167    0.8750        60
           1     0.9173    0.9861    0.9505       720
           2     0.9836    0.9613    0.9724       750
           3     0.9196    0.9156    0.9176       450
           4     0.9411    0.9682    0.9544       660
           5     0.9168    0.9619    0.9388       630
           6     0.9710    0.8933    0.9306       150
           7     0.9108    0.9756    0.9421       450
           8     0.9720    0.9267    0.9488       450
           9     0.9720    0.9396    0.9555       480
          10     0.9878    0.9803    0.9840       660
          11     0.9280    0.8595    0.8925       420
          12     0.9889    0.9000    0.9423       690
          13     0.9768    0.9958    0.9862       720
          14     0.9712    1.0000    0.9854       270
          15     0.8991    0.9762    0.9361       210
          16     1.0000    1.0000    1.0000       150
          17     1.0000    0.8972    0.9458       360
          18     0.9563    0.8410    0.8950       390
          19     0.9783    0.7500    0.8491        60
          20     0.8241    0.9889    0.8990        90
          21     0.6774    0.7000    0.6885        90
          22     0.9107    0.8500    0.8793       120
          23     0.7340    0.9933    0.8442       150
          24     0.9167    0.7333    0.8148        90
          25     0.9343    0.9187    0.9265       480
          26     0.8719    0.9833    0.9243       180
          27     0.6531    0.5333    0.5872        60
          28     0.9366    0.8867    0.9110       150
          29     0.7391    0.9444    0.8293        90
          30     0.6686    0.7533    0.7085       150
          31     0.8706    0.9963    0.9292       270
          32     0.9600    0.8000    0.8727        60
          33     0.8601    0.9952    0.9227       210
          34     0.9600    1.0000    0.9796       120
          35     0.9667    0.9667    0.9667       390
          36     0.9597    0.9917    0.9754       120
          37     1.0000    0.9500    0.9744        60
          38     0.9955    0.9652    0.9801       690
          39     1.0000    0.6667    0.8000        90
          40     0.8511    0.8889    0.8696        90
          41     0.9524    1.0000    0.9756        60
          42     0.9189    0.7556    0.8293        90

    accuracy                         0.9365     12630
   macro avg     0.9138    0.9025    0.9044     12630
weighted avg     0.9398    0.9365    0.9364     12630
```

APPENDIX I – Classification Report - Distilled Model embeded in ESP32-S2 DevBoard

```
Classification Report

              precision    recall  f1-score   support

           0     0.6857    0.4000    0.5053        60
           1     0.8323    0.9167    0.8724       720
           2     0.8466    0.8760    0.8611       750
           3     0.8664    0.8356    0.8507       450
           4     0.8802    0.8909    0.8855       660
           5     0.8295    0.7492    0.7873       630
           6     0.7872    0.7400    0.7629       150
           7     0.8411    0.7644    0.8009       450
           8     0.7817    0.8756    0.8260       450
           9     0.9207    0.9187    0.9197       480
          10     0.9273    0.9470    0.9370       660
          11     0.8687    0.8667    0.8677       420
          12     0.9330    0.8884    0.9102       690
          13     0.9047    0.9625    0.9327       720
          14     0.9358    0.9185    0.9271       270
          15     0.9136    0.9571    0.9349       210
          16     0.7796    0.9667    0.8631       150
          17     0.9636    0.8083    0.8792       360
          18     0.8540    0.6897    0.7631       390
          19     0.4032    0.4167    0.4098        60
          20     0.7701    0.7444    0.7571        90
          21     0.3902    0.3556    0.3721        90
          22     0.8776    0.7167    0.7890       120
          23     0.8047    0.6867    0.7410       150
          24     0.4659    0.4556    0.4607        90
          25     0.8245    0.8125    0.8185       480
          26     0.7857    0.7944    0.7901       180
          27     0.5000    0.3500    0.4118        60
          28     0.6450    0.7267    0.6834       150
          29     0.7980    0.8778    0.8360        90
          30     0.5263    0.6667    0.5882       150
          31     0.8084    0.8593    0.8330       270
          32     0.7536    0.8667    0.8062        60
          33     0.8451    0.9095    0.8761       210
          34     0.7803    0.8583    0.8175       120
          35     0.9088    0.8436    0.8750       390
          36     0.7161    0.9250    0.8073       120
          37     0.5000    0.6500    0.5652        60
          38     0.8916    0.9536    0.9216       690
          39     0.7619    0.5333    0.6275        90
          40     0.4444    0.3556    0.3951        90
          41     0.5636    0.5167    0.5391        60
          42     0.6173    0.5556    0.5848        90

    accuracy                         0.8383     12630
   macro avg     0.7613    0.7536    0.7533     12630
weighted avg     0.8394    0.8383    0.8367     12630
```

APPENDIX J – Classification Report - Pruned Model embeded in ESP32-S2 DevBoard

```
Classification Report

              precision    recall  f1-score   support

           0     0.7885    0.6833    0.7321        60
           1     0.8760    0.8833    0.8797       720
           2     0.8705    0.8960    0.8830       750
           3     0.8809    0.8711    0.8760       450
           4     0.9439    0.8924    0.9174       660
           5     0.7991    0.8333    0.8159       630
           6     0.8333    0.8667    0.8497       150
           7     0.9102    0.8333    0.8701       450
           8     0.8445    0.8689    0.8565       450
           9     0.8907    0.9167    0.9035       480
          10     0.9243    0.9621    0.9428       660
          11     0.8765    0.8619    0.8691       420
          12     0.9007    0.9072    0.9040       690
          13     0.8783    0.9625    0.9185       720
          14     0.9663    0.9556    0.9609       270
          15     0.9130    0.9000    0.9065       210
          16     0.8726    0.9133    0.8925       150
          17     0.9570    0.8028    0.8731       360
          18     0.8266    0.6846    0.7489       390
          19     0.6290    0.6500    0.6393        60
          20     0.5702    0.7222    0.6373        90
          21     0.4105    0.4333    0.4216        90
          22     0.8378    0.7750    0.8052       120
          23     0.6989    0.8200    0.7546       150
          24     0.5789    0.3667    0.4490        90
          25     0.8347    0.8729    0.8534       480
          26     0.7206    0.8167    0.7656       180
          27     0.6620    0.7833    0.7176        60
          28     0.8163    0.8000    0.8081       150
          29     0.7209    0.6889    0.7045        90
          30     0.6250    0.5000    0.5556       150
          31     0.8390    0.9074    0.8719       270
          32     0.7703    0.9500    0.8507        60
          33     0.8468    0.9476    0.8944       210
          34     0.7817    0.9250    0.8473       120
          35     0.9119    0.9026    0.9072       390
          36     0.7320    0.5917    0.6544       120
          37     0.6452    0.6667    0.6557        60
          38     0.9664    0.9174    0.9413       690
          39     0.8654    0.5000    0.6338        90
          40     0.5175    0.6556    0.5784        90
          41     0.7333    0.7333    0.7333        60
          42     0.6933    0.5778    0.6303        90

    accuracy                         0.8565     12630
   macro avg     0.7944    0.7907    0.7886     12630
weighted avg     0.8581    0.8565    0.8553     12630
```

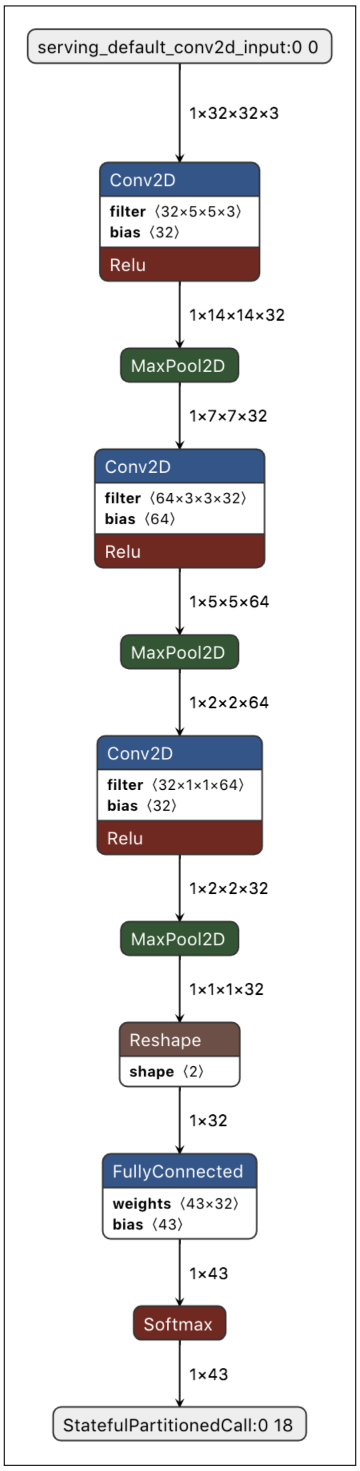
APPENDIX K – Classification Report - Quantized Model embeded in ESP32-S2 DevBoard

```
Classification Report

              precision    recall  f1-score   support

           0     0.6269    0.7000    0.6614        60
           1     0.8858    0.9264    0.9056       720
           2     0.8856    0.9493    0.9163       750
           3     0.8626    0.9067    0.8841       450
           4     0.9249    0.8773    0.9005       660
           5     0.8390    0.8603    0.8495       630
           6     0.8269    0.8600    0.8431       150
           7     0.8889    0.8178    0.8519       450
           8     0.8267    0.8800    0.8525       450
           9     0.9196    0.9292    0.9244       480
          10     0.9641    0.9348    0.9492       660
          11     0.8605    0.8810    0.8706       420
          12     0.9768    0.7942    0.8761       690
          13     0.9348    0.9750    0.9545       720
          14     0.8258    0.9481    0.8828       270
          15     0.8611    0.8857    0.8732       210
          16     0.9250    0.9867    0.9548       150
          17     0.9627    0.7889    0.8672       360
          18     0.7136    0.7795    0.7451       390
          19     0.4828    0.7000    0.5714        60
          20     0.6522    0.8333    0.7317        90
          21     0.6182    0.3778    0.4690        90
          22     0.9263    0.7333    0.8186       120
          23     0.7175    0.8467    0.7768       150
          24     0.5467    0.4556    0.4970        90
          25     0.9413    0.8354    0.8852       480
          26     0.8872    0.6556    0.7540       180
          27     0.5625    0.4500    0.5000        60
          28     0.7706    0.8733    0.8187       150
          29     0.5753    0.9333    0.7119        90
          30     0.5895    0.3733    0.4571       150
          31     0.7928    0.8926    0.8397       270
          32     0.9400    0.7833    0.8545        60
          33     0.8826    0.8952    0.8889       210
          34     0.8440    0.9917    0.9119       120
          35     0.9529    0.8821    0.9161       390
          36     0.8235    0.7000    0.7568       120
          37     0.7241    0.7000    0.7119        60
          38     0.8491    0.9377    0.8912       690
          39     0.6122    0.6667    0.6383        90
          40     0.5752    0.7222    0.6404        90
          41     0.6383    0.5000    0.5607        60
          42     0.8358    0.6222    0.7134        90

    accuracy                         0.8591     12630
   macro avg     0.7966    0.7917    0.7879     12630
weighted avg     0.8635    0.8591    0.8579     12630
```

Annex

ANNEX A – Teacher model graph - NETRON.

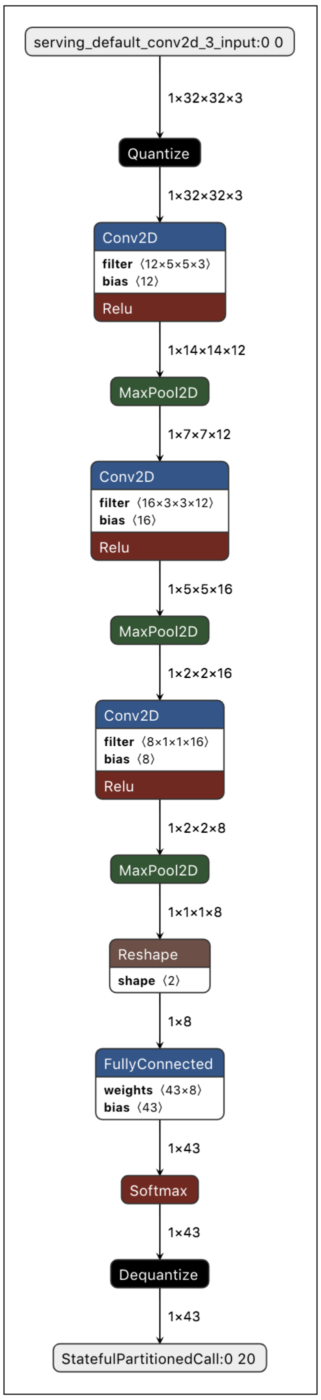
Figure 19 – Teacher model graph.



Source: The author

ANNEX B – Quantized model graph - NETRON.

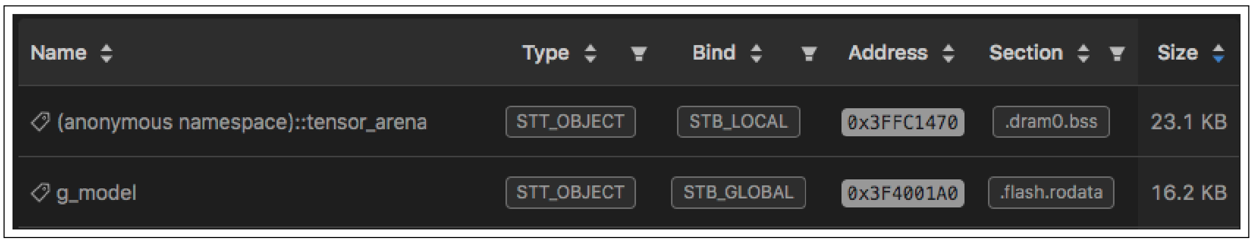
Figure 20 – Quantized model graph.



Source: The author

ANNEX C – ELF Inspection Results - PlatformIO

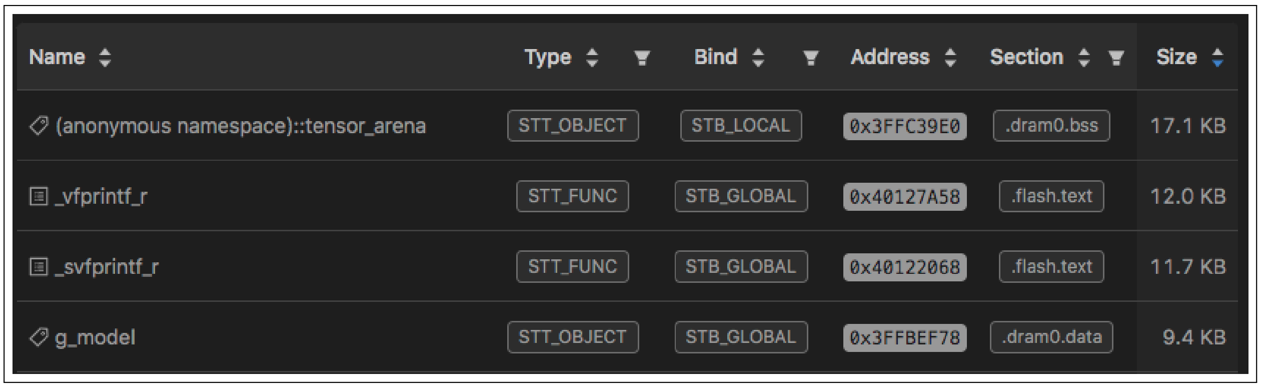
Figure 21 – Analysis of the ELF file generated from the distilled model showing the space occupied by the model and the memory segments in which they are located.



Name	Type	Bind	Address	Section	Size
(anonymous namespace)::tensor_arena	STT_OBJECT	STB_LOCAL	0x3FFC1470	.dram0.bss	23.1 KB
g_model	STT_OBJECT	STB_GLOBAL	0x3F4001A0	.flash.rodata	16.2 KB

Source: The author

Figure 22 – Excerpt from the ELF file inspection performed on PlatformIO.



Name	Type	Bind	Address	Section	Size
(anonymous namespace)::tensor_arena	STT_OBJECT	STB_LOCAL	0x3FFC39E0	.dram0.bss	17.1 KB
_vfprintf_r	STT_FUNC	STB_GLOBAL	0x40127A58	.flash.text	12.0 KB
_svfprintf_r	STT_FUNC	STB_GLOBAL	0x40122068	.flash.text	11.7 KB
g_model	STT_OBJECT	STB_GLOBAL	0x3FFBEF78	.dram0.data	9.4 KB

Source: The author

ANNEX D – Memory.ld File Location.

Figure 23 – Tree view of the location of the memory.ld file.

```
User
└─ Platformio
    └─ packages
        └─ framework-arduinoespressif32
            └─ tools
                └─ sdk
                    └─ esp32/esp32s2
                        └─ ld
                            └─ memory.ld
```