Received May 29, 2021, accepted July 6, 2021, date of publication July 19, 2021, date of current version July 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3097969

Evaluation Method of Deep Learning-Based Embedded Systems for Traffic Sign Detection

MIGUEL LOPEZ-MONTIEL[1], ULISES OROZCO-ROSAS[2], (Member, IEEE), MOISÉS SÁNCHEZ-ADAME[1], KENIA PICOS[2], AND OSCAR HUMBERTO MONTIEL ROSS[1], (Senior Member, IEEE)
[1]Instituto Politécnico Nacional, CITEDI-IPN, Tijuana, Baja California 22435, México
[2]CETYS Universidad, Centro de Innovación y Diseño (CEID), Tijuana, Baja California 22210, México

Corresponding author: Ulises Orozco-Rosas (ulises.orozco@cetys.mx)

This work was supported in part by the Coordinación Institucional de Investigación of Centro de Enseñanza Técnica y Superior (CETYS Universidad), and in part by the Consejo Nacional de Ciencia y Tecnología (CONACYT).

ABSTRACT Traffic Sign Detection (TSD) is a complex and fundamental task for developing autonomous vehicles; it is one of the most critical visual perception problems since failing in this task may cause accidents. This task is fundamental in decision-making and involves different internal conditions such as the internal processing system or external conditions such as weather, illumination, and complex backgrounds. At present, several works are focused on the development of algorithms based on deep learning; however, there is no information on a methodology based on descriptive statistical analysis with results from a solid experimental framework, which helps to make decisions to choose the appropriate algorithms and hardware. This work intends to cover that gap. We have implemented some combinations of deep learning models (MobileNet v1 and ResNet50 v1) in a combination of the Single Shot Multibox Detector (SSD) algorithm and the Feature Pyramid Network (FPN) component for TSD in a standardized dataset (LISA), and we have tested it on different hardware architectures (CPU, GPU, TPU, and Embedded System). We propose a methodology and the evaluation method to measure two types of performance. The results show that the use of TPU allows achieving a processing training time 16.3 times faster than GPU and better results in terms of precision detection for one combination.

INDEX TERMS Traffic sign detection, deep learning, hardware acceleration, computer vision, autonomous vehicles, embedded systems, digital systems.

I. INTRODUCTION

Nowadays, there has been a rapid emergence in the development of Deep Learning (DL) algorithms focused on vision problems for autonomous vehicles. These algorithms have been evolving throughout the years, having different applications such as robotics, object recognition, self-driving cars, among others [1]. Within the applications of autonomous vehicles, different tasks exist to attain vehicle autonomy. One of those is computer vision. One of the main tasks to achieve in computer vision is Traffic Sign Detection (TSD).

The vision problem of traffic sign detection started decades ago with some research that focused on classical computer vision algorithms [2], [3], where image processing was used through geometric characteristics. However, these algorithms were not robust enough to obtain the desired results. This is because they faced different challenges [4], such as the internal and external conditions of the traffic sign's environment. External conditions are variables that cannot be controlled, such as lighting conditions, weather conditions, complex backgrounds, physical degradation of traffic signs, among others. On the other hand, the internal conditions are the variables that can be controlled by the algorithm, such as response time, precision in detection, adaptability, and hardware dependency, among others [5]. These challenges of improving precision, reducing complexity, and hardware dependence have encouraged the development of more robust algorithms that allowed the adaptation and improvement in recent results. Through the years, intelligent algorithms began to be used in state-of-the-art, specifically, machine learning [6], [7]. The expected results were improved due

The associate editor coordinating the review of this manuscript and approving it for publication was Marco Martalo.

to better extraction of more descriptive characteristics, and using classifiers, classification, and detection of traffic signs could be performed.

The detection task can be solved by implementing DL algorithms, which have progressed continuously through time, improving their performance among the various tasks in detection, segmentation, and classification of objects [8], [9]. DL algorithms have been increasingly used to solve computer vision problems in autonomous vehicles; this is due to the extraction of characteristics and classification employing Convolutional Neural Network (CNN) architectures. Liu in [10], reported that for the case of the TSD problem, they started using CNN for the background's objects classification and they needed extraction methods for the regions of interest (ROI) to obtain the candidates.

These network architectures have evolved into two different types of methods: one-stage methods and two-stage methods. One-stage methods contain a single feed-forward fully convolutional network that directly provides the bounding boxes and the object classification. The two-stage methods divide the detection process into the region proposal and the classification stage [11]. An important one-stage method is the YOLO model [12]. This model had a great impact on the state-of-the-art real-time object detection methods. It has been evolved to reach a new YOLO v5 version [13], which improves its object detection results. The two-stage methods began with the Region-Based Convolutional Neural Networks (R-CNN) [14], being the pioneering deep learning-based work. Newer studies as the Cascade R-CNN method [15] can yield more precise detections, but it requires more computation, making it less suitable for real-time applications. However, these methods (one-stage and two-stage) have specific limitations or are dependent on hardware accelerators to take advantage of their processing power.

In this article, we continue our previous works [16], [17] for the traffic sign detection problem. We propose a comparative evaluation of two different hardware accelerators to measure the detection system's performance on traffic sign detection. We evaluated these detection systems for different situations to implement them in embedded devices for real-time applications. Different metrics are considered for the proper evaluation of detection systems in embedded devices. The metrics used in this work are *Precision*, *Recall*, *mean Average Precision* (*mAP*), *Memory consumption*, *Latency*, *Throughput*, and *Energy consumption*. The main contributions of this paper can be summarized as follows:

- We contribute with comparisons based on a descriptive statistical analysis of detection system combinations based on deep learning and hardware accelerators. This will help to elucidate some advantages and disadvantages when using a Central Processing Unit (CPU), Graphics Processing Unit (GPU), or a Tensor Processing Unit (TPU). This analysis can be used to identify some bottlenecks in the whole system since it tells which combination helps to reduce or increase the training time while improving the detection results.
- We proposed a methodology to evaluate a DL model's performance. This methodology is made up of four stages: data selection, model selection, hardware selection, and the embedded application. Each of these stages uses different objective metrics to select, characterize and analyze the final application approach. It can be used for hardware architectures with ample resources such as in a workstation or limited architectures like in an embedded system. The methodology allows producing a balanced set of examples to improve the training. The model selection stage evaluates the training time required by different hardware accelerators, and the detection results providing useful information to choose the most appropriate hardware for the model. Moreover, it evaluates the *Memory consumption* and the required-time for the different embedded applications. This methodology experimentally demonstrates that the use of TPU reduces training time without significantly affecting the accuracy of the models.
- We contribute with an algorithm for the proposed methodology to evaluate the different combinations of the selected DL models. The algorithm is immersed in the methodology mentioned above. It configures and connects Google Cloud and Google Colab services to use TPU processing, loads the dataset, the DL model, and the model's hyperparameters. The configuration files that contain all the required parameters of datasets, the model, and its parameters, and hyperparameters are in a bucket in the Google Cloud Storage service. The training results are saved in a checkpoint file in the mentioned bucked.

The remainder of this paper contains the following topics. In Section II, we present a review of a selection of related works that deal with the evaluation of hardware accelerators for deep learning algorithms, the works focused on the evaluation of traffic sign detection systems, as well as the Single Shot Multibox Detector (SSD) meta-architecture and Feature Pyramid Network (FPN), and the feature extractors are presented. In Section III, the proposed methodology for evaluating traffic sign detection systems in hardware accelerators is described. In Section IV, the experiments and results obtained from the TSD systems are presented. In Section V, the conclusions and future work are presented.

II. FUNDAMENTALS

Currently, accidents in autonomous vehicles keep happening [18]–[20]. Hence, the demand for solutions has been on the rise, pressing research lines and the automotive industry in order to accelerate the design and development of algorithms to reach full autonomy of vehicles. At present, to achieve this feat, researchers around the globe have based their work on the implementation of DL algorithms to contribute to the various tasks involved in different levels of autonomy [21]. However, these algorithms present another problem besides requiring an adequate dataset for the training stage. The reliance on hardware, specifically, hardware accelerators, was created by DL algorithms to exploit their specific hardware architectures for more efficient processing, inference, and training by reducing the time necessary to implement DL models [22].

DL algorithms rely on big data to accomplish appropriate performance in a model's training and testing stages. The results are sometimes dependent on the quality and quantity of data, being a problem when it has a small dataset or when it has a lot of noise in the data. This indicates that an extra step is necessary before processing the data. This stage is called ''preprocessing'', which allows transforming the data into a more understandable format, as well as fixing data problems. Some of the problems are incomplete, inconsistent, or patterned data, which ultimately call to the stage of processing [23].

Although data and preprocessing are relevant problems for DL algorithms, they have not been the only trend or challenge we have today. Mobile applications are still a current challenge, mainly online mode, this refers to when the cloud carries out training and data inference, while mobile devices only send and receive data due to their battery, power of computation, and data storage. However, this online mode depends on internet connectivity, which can cause a long delay. On the other hand, in offline mode, the training task is still performed by the cloud, but the trained model is sent to the mobile device to make inferences locally to preserve user privacy. However, trained deep models can have a large number of parameters and complicated calculation requirements, posing great challenges for the limited resources of mobile devices [24].

New mobile and embedded devices are more powerful, featuring dedicated hardware accelerators, multi-core processors, and gigabytes of RAM. So, the new emerging trend is to take advantage of offline mode to directly implement deep learning models on mobile and embedded devices for inference. Offline mode is the preferred mode for various applications, especially for real-time applications; however, factors such as computing power, memory, memory bandwidth, and battery are too limited to support modern deep learning models that require both computation and memory [24]. These computational cost and memory limitations have prompted the development of new hardware architectures and DL algorithms for more efficient processing by optimizing the use of computing resources. This has promoted the investigation of DL algorithms applied to embedded systems, optimization, and compression of DL models, among other topics. Therefore, different researchers have carried out various studies that address these problems to find solutions [22], [25]. Other more specific investigations have focused on certain applications such as autonomous vehicles [26], [27], and more specifically their tasks such as object detection, semantic segmentation, among others [28]–[30].

These DL algorithms designed for real-time applications are based upon a proper trade-off between the essential properties for real-time implementation, like *Latency*, model size, *Energy consumption*, Floating Point Operations per second (FLOPs), among other relevant hardware factors. On the other hand, lie the qualities of object detection, centered in the model's inference results. In this case, localization, and classification of objects, ensuring *Precision*, *Recall*, and *Average Precision* to fulfill these two objectives [31], [32].

Researchers such as Arcos-Garca *et al.* [33] and Ayachi *et al.* [34] developed a TSD system utilizing a GPU, where the results were evaluated through *mean Average Precision* (*mAP*). Ayachi implemented a database created by their team, while Arcos uses the German Traffic Sign Detection Benchmark (GTSDB). On the other hand, researchers like Bangquan and Xiong [35] and William *et al.* [36] achieved important advancements in the embedded system implementation field, evaluating the performance of detection systems in these devices. Both, Bangquan as well as William utilize meta-architectures and feature extractors to develop models, as well as *mAP* and accuracy metrics. Research from Jouppi *et al.* [37] compared CPU and TPU hardware accelerators to evaluate the difference between *Throughput* and *Latency*, exploitation on neural networks of these specific devices.

DL algorithms are employed in computer vision applications, where object detection is one of the revolutionizing tasks of state-of-the-art through these algorithms. These systems are composed of meta-architectures and feature extractors, which allow improving detection results over different combinations.

Arcos in [33] states that combinations rely on the application's approach to development and performance of Traffic Sign Detection (TSD) systems can be enhanced through said combinations. Comparing several combinations helps find the proper trade-off among the application systems' performance on embedded devices and their detection results.

These characteristics allow the development of more efficient systems, requiring an appropriate performance of the detection results and the application's real-time functionality, which are fundamental in developing autonomous vehicle systems. Nevertheless, this process depends on the employed hardware accelerator, both in the training and test phase and in the system implementation phase [34], [35]. Thus, this work centers on comparing advantages and disadvantages in the use of hardware accelerators implemented in DL algorithms.

A. MACHINE LEARNING ARCHITECTURES

Deep Learning Neural Networks are a subset of machine learning algorithms, which simplify feature extraction and description through a multi-layer Convolutional Neural Network. CNN intends to transform the high-dimension input image into a low-dimension but highly abstracted semantic output. Powered by massive amounts of data and modern CPUs and GPUs, methods based on deep neural networks obtain state-of-the-art performance and help the computer vision field in algorithms development. At present, there is an increase in the design and development of deeper networks to make CNN's provide near-human accuracy in different computer vision applications, for instance, classification, segmentation, and detection. However, a high computational cost is required in order to achieve these levels of high accuracy. Dedicated hardware like GPUs and application-specific CPUs are studied to optimize Deep Neural Networks (DNN) [38].

Convolutional Neural Network (CNN) is a well-known deep learning architecture inspired by the natural visual perception of living beings. In 1959, Hubel and Wiesel [39] found that cells in the animal visual cortex are responsible for detecting light in receptive fields. Inspired by this, Kunihiko Fukushima proposed the neocognitron in 1980 [40], which can be considered as the precursor of CNN. In 1990, Le-Cun [41] established the modern framework of CNN and subsequently improved it in [42]; they developed a multi-layer artificial neural network named LeNet-5, which could classify handwritten digits.

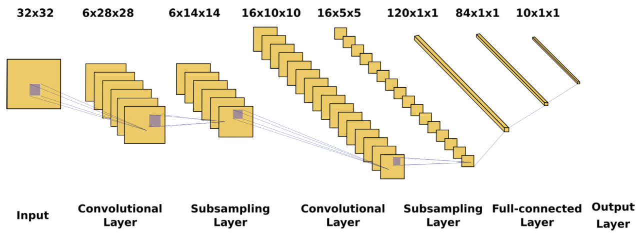


FIGURE 1. LeNet-5 network architecture.

There are several variants of CNN architectures. Nonetheless, their essential elements are very similar. LeNet-5, for example, consists of three types of layers: convolutional, pooling, and fully connected layers. The convolutional layer is in charge of learning the representations of input characteristics. Each layer is composed of a convolution kernel. This kernel calculates the following input feature maps of the next layer, see Fig. 1.

Each neuron of the feature map is connected to a region of adjacent neurons in the previous layer. This neighborhood is named as the neuron's receptive field in the past layer. Convolving the input with a learned kernel and applying a nonlinear activation function through the element on the convolved results produces a new feature map. The kernel is shared in spatial locations of the input in order to generate each feature map. The complete set of feature maps are obtained by using several different kernels. Mathematically, the feature value at location (i, j) in the k-th feature map of l-th layer, $z^l_{i,j,k}$, is calculated by:

$$z^l_{i,j,k} = (w^l_k)^T x^l_{i,j} + b^l_k \tag{1}$$

where $(w^l_k)^T$ and b^l_k are the weight vector and bias term of the k-th filter of the l-th layer respectively, and $x^l_{i,j}$ is the input patch centered at the location (i, j) of the l-th layer. The kernel w^l_k that generates the feature map $z^l_{i,j,k}$ is shared. Such a weight-sharing mechanism has several advantages: reducing the model complexity and make the network easier to train. The activation function introduces nonlinearities to CNN, which are desirable for multi-layer networks to detect nonlinear features. Let $a(\cdot)$ denotes the nonlinear activation function. The activation value $a^l_{i,j,k}$ of convolutional feature $z^l_{i,j,k}$ can be computed as:

$$a^l_{i,j,k} = a\left(z^l_{i,j,k}\right) \tag{2}$$

Common activation functions are sigmoid, tanh, and Rectified Linear Unit (ReLU) [43]. The pooling layer's goal is to achieve shift-invariance by reducing the resolution of the feature maps. It is usually placed between two convolutional layers. Each feature map of a pooling layer is linked to its corresponding feature map of the preceding convolutional layer. Defining the pooling function as $pool(\cdot)$, for each output of the feature map $z^l_{i,j,k}$ processed with its activation function $a^l_{i,j,k}$ we get:

$$y^l_{i,j,k} = pool\left(a^l_{m,n,k}\right), \quad \forall(m, n) \in R_{i,j} \tag{3}$$

where $R_{i,j}$ is a local neighborhood around location (i, j). The common pooling operations are average pooling and max pooling [44]. After numerous convolutional and pooling layers, there may be one or more fully connected layers that aim to perform high-level reasoning [45]. The fully connected layers take all neurons in the prior layer and connect them to every neuron of the current layer to produce global semantic information. The last layer of CNNs is an output layer. Let θ denote all the parameters for a CNN. The optimal parameters of a specific task can be obtained by minimizing a suitable objective function in that task, in this case, C is defined as the cost function and l would be the loss function to optimize for each N relation. Therefore, if we have N desired input-output relations $\{(x^{(n)}, y^{(n)}); n \in [1, \ldots, N]\}$, where $x^{(n)}$ is the n-th input data, $y^{(n)}$ is its corresponding target label and $o^{(n)}$ is the output of the CNN. The loss of CNN can be calculated by:

$$C = \frac{1}{N}\sum_{n=1}^{N} l\left(\theta; y^{(n)}, o^{(n)}\right) \tag{4}$$

Deep Convolutional Neural Networks (DCNNs) have a significant number of advantages: a hierarchical structure to learn representations of data with multiple levels of abstraction, the capacity to learn very complex functions, and learning feature representations directly and automatically from data with minimal domain knowledge. What has made DCNNs successful has been the availability of large, labeled datasets and hardware accelerators like GPUs with considerably high computational capabilities [9], [46].

The Single Shot Multibox Detector (SSD) [47] is an approach based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections, as seen in Fig. 2. Default boxes represent carefully selected bounding boxes based on their sizes, aspect ratios, and positions across the image. The core of SSD is to predict category scores and box offsets, aiming to determine which of the set of default boxes will be used to apply convolutional filters to the feature maps. The early network layers are based on a standard architecture used for high-quality image classification. A key feature of the model is the use of multi-scale convolutional bounding box outputs attached to multiple feature maps at the top of the network. This representation allows us to model the space of possible box shapes efficiently. The key difference between training SSD and training a typical detector that uses region proposals is that ground truth information needs to be assigned to specific outputs in the fixed set of detector outputs.

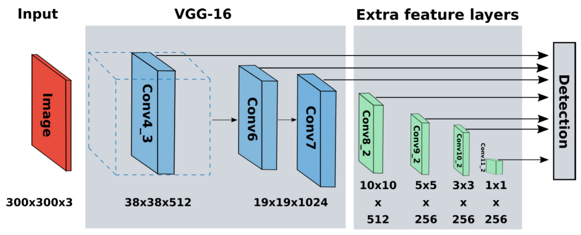


FIGURE 2. Single Shot Multibox Detector (SSD) model.

When deeper networks start converging, a degradation problem arises; as network depth increases, accuracy gets saturated and degrades rapidly. This degradation indicates that not all systems can be optimized similarly and easily. For example, if we consider a shallow architecture and add more layers to it, one would suppose that the performance obtained from the model would be better. However, this can lead to higher complexity when optimizing the network parameters causing higher training errors. A solution is that the added layers focus on identity mapping, while the other layers extract the residuals from previous layers of the shallower part of the model. Due to this, small residuals are added to the input during the learning process; instead of transforming the whole input [48]. The ResNet50 model [48] is based on a deep residual learning framework consisting of multiple subsequent residual modules, see Fig. 3. The residual blocks allow the network to retain what it has previously learned. Using an identity mapping function where the output equals the input to retain what it previously learned, will add what the network has already learned if there is nothing new to learn.

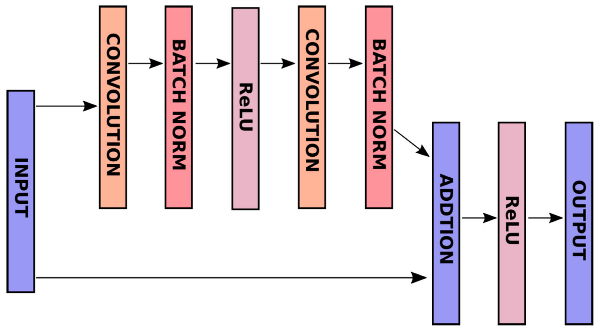


FIGURE 3. Residual ResNet50 v1 block.

The MobileNet model [49] is based on depthwise separable convolutions, which are a form of factorized convolutions that factorize a standard convolution into a depthwise convolution, and a 1 × 1 convolution called a pointwise convolution. The result of the depthwise convolutions significantly reduces the number of parameters compared to the network with normal convolutions with the same depth. It reduces the total number of floating-point multiplication operations, which is favorable in mobile and embedded vision applications with less computer power. This factorization has the effect of drastically reducing computation and model size. MobileNet's essential block is defined in Fig. 4.

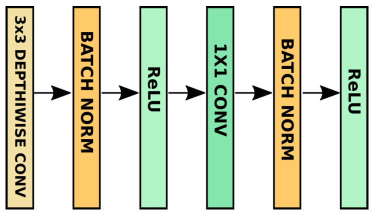


FIGURE 4. Depthwise separable convolution MobileNet v1 block.

Feature pyramids are a necessary component in recognition systems for detecting objects at different scales. These pyramids are scale-invariant. An object's scale change is offset by shifting its level in the pyramid, enabling models to detect objects across a large range of scales by scanning over pyramid levels and positions. The main advantage of featuring each level of an image pyramid is that it produces a multi-scale feature representation in which all levels are semantically strong, including high-resolution levels. Nonetheless, featuring each level increases inference time, requires considerable memory usage, and produces inconsistency between train/test-time inference. The Feature Pyramid Network (FPN) [50] shown in Fig. 5 is an architecture that combines low-resolution, semantically strong features with high-resolution, semantically weak features via a top-down pathway, and lateral connections.

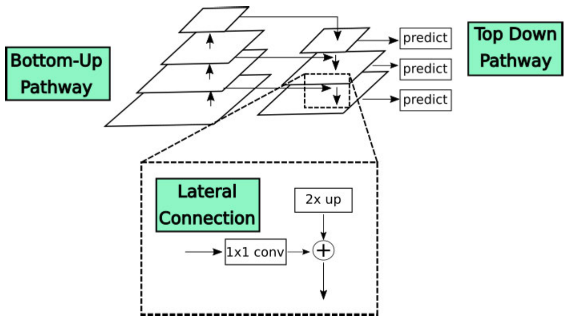


FIGURE 5. FPN block showing the top-down pathway and lateral connection merged by addition.

This method takes a single-scale image of arbitrary size as input and delivers proportionally sized feature maps at multiples levels as an output. The bottom-up pathway represents the feed-forward computation of the backbone ConvNet, which computes a feature hierarchy consisting of feature maps at several scales with a scaling step of two. The top-down pathway visualizes higher resolution features by upsampling spatially rough to extract feature maps from higher pyramid levels. These features are enhanced with

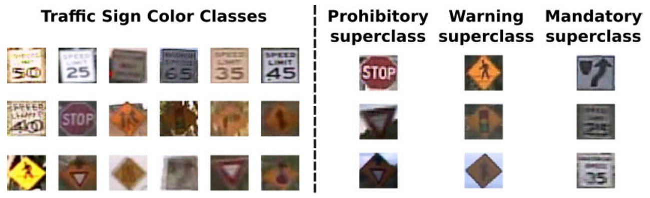


FIGURE 6. Examples of classes and superclasses in the LISA dataset.

features from the bottom-up pathway via lateral connections. Each lateral connection merges feature maps of the same spatial size. As seen in Fig. 5, with a coarser resolution feature map, the spatial resolution is upsampled by a factor of two and then merged with the corresponding bottom-up by element-wise addition. This process is repeated until an adequate resolution map is obtained.

B. TRAFFIC SIGN DETECTION DATABASE

The Laboratory for Intelligent & Safe Automobiles (LISA) developed a traffic sign dataset comprising 6,610 images with several distinct characteristics. Variables such as occlusion, blur, illumination variation, multiple signs, and distance discrepancy are some of the events present in the dataset. There are 47 unique traffic signs in the database. In this work, it is chosen to divide these 47 classes of signs into three superclasses: warning, prohibitory, and mandatory, as can be seen in Fig. 6. These are divided into a subset of 2,641 color images containing 18 traffic sign classes, as also shown in Fig. 6. In this work, we used only 900 samples divided by their corresponding superclass, which are also labeled by their traffic sign class, as shown in Table 1. The purpose of superclasses is to encompass similar properties of signs, as well as their parent category. For example, the warning superclass mostly shares the geometric shape of a diamond, as well as its characteristic color is commonly yellow. On the other hand, the mandatory ones, mostly their geometric shape is a rectangle, and they are white, as they are also commonly speed limit signs. Finally, the prohibitory ones have a geometric shape from circular to octagonal, this value is variable, and its characteristic color is red, an example of this superclass is the stop sign. The final objective of this categorization of superclasses is to be able to facilitate the final classification of the signs, reducing the multi-class classification problem from 47 to 3 classes.

Besides, each annotation provides data that are more relevant for the training phase. The database has different resolutions, ranging from 640 × 480 to 1,024 × 522 pixels, as well as color samples and others at gray scales. Other characteristics are the type of sign, position, size, if it has occlusion or not, and if it is on the way of the vehicle or not.

A descriptive example of a scene from the database would be the following: in video number 0 is sample number 0, this sample is in color and has a resolution of 704 × 480 pixels, as well as in PNG format. This scene has a stop sign, which has the following coordinates of the bounding box of the sign: in the upper-left corner (x_1) it has the first point at pixel 485, in the upper-right corner (y_1) it has the second point

TABLE 1. Traffic sign classes and superclasses in the LISA dataset.

Superclasses	Number of samples
Prohibitory	**(300)**
Stop	210
Yield	45
YieldAhead	45
Warning	**(300)**
Merge	100
PedestrianCrossing	100
SignalAhead	100
Mandatory	**(300)**
KeepRight	110
SpeedLimit35	110
SpeedLimit25	80

in pixel 41, in the lower-left corner (x_2) there is the third point in pixel 504 and the lower-right corner (y_2) there is the fourth point in pixel 59. This sign has no occlusion that is represented by the value 0 and is in the path of the vehicle which is also represented by the value 0. Another important piece of information in the traffic sign dataset is their location since their geometry and color vary by continent and country [51].

C. HARDWARE ACCELERATORS

The deployment of a DNN on a real-world application consists of two phases: training and inference. Network training is expensive in terms of speed and memory; consequently, it is commonly carried out on GPUs off-line. Recently, hardware accelerators [52], [53] for DNN have received widespread attention, particularly for the training phase. An accelerator comprises five parts: data buffers, parameter buffers, processing elements, global controller, and off-chip transfer manager [31].

The data buffers are used to cache input images, intermediate data, and output predictions, while the weight buffers are used mainly to cache convolutional filters. The processing elements are a set of basic computing units that execute multiply-add, non-linearity, and other functions such as normalization and quantization. The global controller is employed to arrange computing flow on-chip, while the manager organizes off-chip instructions and data transfer. The most significant difference among all hardware accelerators is in the processing elements. They are designed for most intensive computing tasks in deep networks, such as massive multiply-add operations, normalization (batch normalization, or local response normalization), and calculating non-linearities in common activation functions like ReLU, sigmoid, and hyperbolic tangent [31].

A hardware accelerator for DNNs implemented on an Application-Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) typically consists of an array of Processing Elements (PE) for computation, as shown in Fig. 7. The PEs are interconnected by a Network-on-Chip (NoC) designed to achieve the desired data movement scheme. The three levels of the memory hierarchy are the Register Files (RFs) in the PEs, which store data for inter-PE movements or accumulations, and the Global Buffers (GBs), which stores enough values to feed the PEs, and the off-chip memory, usually a DRAM [54]. The real bottleneck in DNNs computation is the huge number of memory access by unit of time. Therefore, one of the key design issues for memory hierarchy is to reduce the DRAM accesses since they have a high *Latency* and energy cost. Hence, the reuse of the data stored in smaller, faster, and low-energy memories (GBs and RFs) is favored, as seen in Fig. 7.

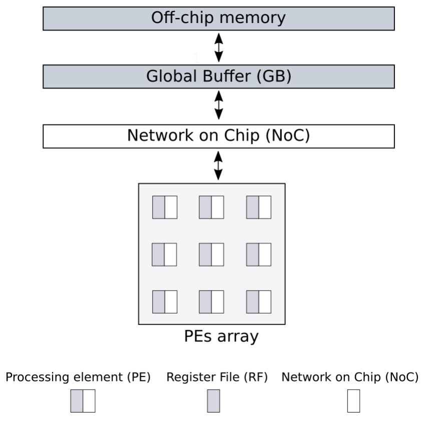
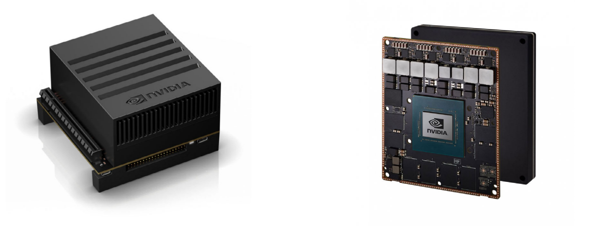


FIGURE 7. The architecture of hardware accelerator in DNNs.

An embedded real-object detection system has numerous real-world applications; however, an on-board system that supports weight, power consumption, and computational resources are necessary for these tasks. Furthermore, while GPUs enhance deep learning algorithm's performance, their power consumption and cost are greater than an embedded CPU system. Here is where the NVIDIA Jetson AGX Xavier, which is illustrated in Fig. 8, exceeds, in terms of performance and efficiency in deep learning and computer vision. This device enables the development of prototypes and artificial intelligence applications for autonomous machines and robots with a GPU workstation's performance in an integrated module of less than 30W [55]. This powerful AI computing can provide 32 Tera Operations Per Second (TeraOPS) of peak computation in a compressed module of 100-mm × 87-mm.



(a) Jetson AGX Xavier Developer Kit. (b) Jetson AGX Xavier module.

FIGURE 8. NVIDIA Jetson AGX Xavier Developer Kit and module.

III. PROPOSED METHODOLOGY AND EVALUATION METHOD

In this section, a methodology for traffic sign detection systems using hardware acceleration is proposed. At present, different hardware accelerators exist and provide the ability to accomplish more difficult challenges through their power capabilities. In this methodology, three different hardware accelerators are utilized (CPU, GPU, and TPU). The selection of hardware accelerators relies on data properties such as type, size, quantity, and application. This information assists in selecting the hardware accelerator and model, allowing us to choose the right combination based on the data characteristics [23]. Thus, this methodology is proposed regarding a specific database and two models and three hardware accelerators, see Fig 9.

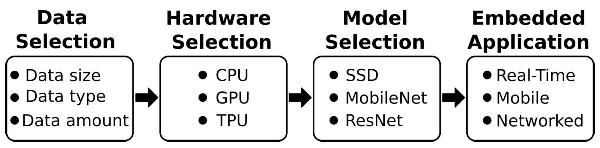


FIGURE 9. Proposed methodology for hardware accelerators in DL.

A. DATA SELECTION

An essential stage in choosing and implementing DL algorithms is the selection of data. DL algorithms require large amounts of data to attain better performance. Several works [23], [36] consider transcendent the size and type of data when working on a computer vision problem. Since processing is done using images, the resolution must be considered in order to determine the input size of the algorithm. Data quantity must also be contemplated; if the dataset is too small, training will be affected negatively depending on the number of samples per class. Considering the previous properties, we performed different steps for the data selection stage. The LISA database (see subsection II-B for more details) contains 6,610 images and 47 classes of traffic signs; these samples are composed of some examples in color and others in grayscale. The first step considers only the color samples because they contain more information for the feature extraction stage of the DL models. In addition, the superclass design is based on the color and geometric shape of the traffic signs. This process discarded 3,969 samples and 29 classes from the dataset, reducing their number of samples by 60%, see Table 2.

In the next step, we consider the classes with the greatest number of examples in each superclass. The objective of this process was to balance the superclasses (see Table 1). Each superclass contains three intraclasses and 300 examples per superclass, giving a total of 900 samples for the dataset. We called this reduced dataset Tiny LISA and made it available on the Kaggle site.[1]

B. HARDWARE SELECTION

Due to the dependency of DL algorithms in hardware accelerators, specific hardware that meets the model's application and data requirements must be picked, prompting

[1]https://www.kaggle.com/mmontiel/tiny-lisa-traffic-sign-detection-dataset

TABLE 2. Traffic sign classes in the color LISA dataset.

Classes	Number of samples
Stop	929
SignalAhead	339
PedestrianCrossing	231
SpeedLimit35	208
KeepRight	175
Merge	128
SpeedLimit25	80
AddedLane	77
StopAhead	76
SpeedLimit40	73
SpeedLimit45	68
YieldAhead	53
SpeedLimit50	47
RightLaneMustTurn	44
Yield	43
TurnRight	34
SpeedLimit65	18
LaneEnds	18

the seeking of the proper hardware for the task through evaluation. The evolution of deep neural networks has increased the demand for computational complexity and, consequently, its resources consumption, posing hardware implementation challenges for these deep neural networks. Some works, such as the one from Cheng *et al.* [31], provide an analysis focused on essential topics such as network pruning, low-rank approximation, network quantization, teacher-student networks, compact network design, and hardware accelerators. Other researchers like Sharma *et al.* [56] and Chen *et al.* [57] generated hardware accelerators both in the hardware and software fields to improve the performance and take advantage of the use of DNNs and CNNs and optimizing and maximizing data reuse, data movement, and statistical exploitation.

In this methodology, the performance of different hardware accelerators is compared for the models' training and execution time of the algorithm phases to determine the most appropriate hardware accelerator for the selected database. The significance of hardware accelerator selection is reflected in training time and implementation of the algorithm [22] due to each hardware accelerator's different specifications. Several works [58]–[60] consider *Memory consumption*, *Energy consumption*, *Latency*, and *Throughput* as the most relevant metrics for DL implementation; thus, these metrics are evaluated in this methodology.

C. MODEL SELECTION

The selection of a model relies on its properties, and it is indispensable to identify in the approach of the model the adequate features that constitute the model [61], [62]. According to the application, a balance between the properties is required to attain the desired performance. The main property considered for the model was its high-quality image classification capabilities in the early layers of the network. The TSD system's construction is established on the use of meta-architectures and feature extractors [63], [64]. Modern meta-architectures implement CNNs for object detection; hence, they must be carefully selected since they affect object detection speed and performance. Feature extractors focus on attaining the substance of the infrastructure. In conjunction with meta-architectures, high-level image information extraction is simplified. Thus, capacity type selection impacts feature extractors; hence, model selection for meta-architectures and feature extractors depends on the application. Good performance in TSD systems depends on different properties. The *Latency*, *Memory consumption*, and *mean Average Precision* are the ones considered in this work. The model's inference time rests in how quickly the system delivers a result, and the time frame for it determines its potential in real-time applications. *Memory consumption* refers to how much hardware resources are required by the system, which helps ascertain if the embedded device is capable to support it. The *mean Average Precision* identifies the excellent performance of a system through its sensitivity and *Precision*. Considering all these properties define whether a system can be implemented in embedded devices for real-time applications [65].

D. EMBEDDED APPLICATION

Specific hardware characteristics are necessary in order to implement a system on an embedded device [66]. Factors like RAM, memory bandwidth, and hardware accelerator are considered to select an adequate embedded system. Embedded devices with GPU are necessary for DL-based systems due to the massive data processing performed in the training and validation stages. A high number of cores and memory bandwidth allow the GPU to process a large amount of data required in DL algorithms. Thereby, embedded devices require a minimum of hardware specifications so that DL-based systems can be implemented efficiently on them. However, hardware requirements must be reduced as well to make up for the limitations of embedded systems.

E. EVALUATION METHOD

Algorithm 1 presents the Traffic Sign Detection Evaluation Method (TSDEM), which consists of performing the steps mentioned in the previous sections.

The main objective of Algorithm 1 is to carry out each of the stages mentioned in the proposed methodology, see Fig. 9. Algorithm 1 performs the following steps:

- Select workstation for training (CPU, GPU, or TPU).
- Configure and connect the workstation or the Google Cloud Service.
- Load the dataset, the DL model, and its hyperparameters.
- Train the DL model and save it to a checkpoint file.
- Evaluate the DL model on CPU, GPU, or Jetson.
- Generate and save training, detection, and performance metrics.

The TSDEM algorithm has the following **input parameters**: the D parameter is the original dataset that contains all the samples in the database. The M_{in} parameter is the combination of the selected model. The parameter H is

Algorithm 1 TSDEM

```
Input: database $D$, DL model $M_{in}$, hyperparameters of the model $H$,
       and system configuration parameters $P$
Output: file $F_{model}$ with trained model and $F_{mt}$, $F_{md}$, and $F_{mp}$ for
        train, detection, and performance metrics
1  $CKPT \leftarrow P_e$   Every few epochs the model is saved
2  $B \leftarrow P_s$   Number of samples per class
3  $C \leftarrow P_c$   Specific classes
4  $E \leftarrow H_e$   Total number of epochs
5  Initialize files $F$ to store model and metrics
6  Initialize superclasses $SC$ to store class samples
7  Initialize $Mt$ to store train metrics
8  Initialize $Md$ to store detection metrics
9  Initialize $Mp$ to store performance metrics
10 Flag $HW$ to select the hardware accelerator to train or implement
11 init
12     $i \leftarrow 0$
13     foreach sample in $D$ do
14         Extract image, and annotation of sample
15         $SC(i+1) \leftarrow sample \cap C$
16         $i \leftarrow i+1$
17     Create subsets $D_{train} = (90\%)(SC)$ and $D_{test} = (10\%)(SC)$
18     Sample balancing $S_{train} = \{D_{train} \cap B\}$ and $S_{test} = \{D_{test} \cap B\}$
19     Create TFrecords $TFR_{train} \leftarrow S_{train}$ and $TFR_{test} \leftarrow S_{test}$
20     if $HW$ is equal to CPU or $HW$ is equal to GPU then
21         Set $P$ workstation settings
22     else
23         Set $P$ cloud settings
24     $j \leftarrow 0$
25     while $j \leq E$ do
26         $M_{out} \leftarrow$ Train $M_{in}$ with $H$ and $S_{train}$
27         if $j$ is equal to CKPT then
28             Save model $M_{out}$ to $F_{model}$
29         Generate training metrics for $Mt_j$ $Mt(j+1)$
30         $j \leftarrow j+1$
31     Saved training metrics $Mt$ to $F_{mt}$
32     $k \leftarrow 0$
33     while $k \leq E$ do
34         Eval $M_{out}$ with $S_{test}$
35         Generate detection metrics for $Md_k$ $Md(k+1)$
36         $k \leftarrow k+1$
37     Saved detection metrics $Md$ to $F_{md}$
38     if $HW$ is not equal to TPU then
39         $l \leftarrow 0$
40         while $l \leq E$ do
41             Eval $M_{out}$ with $S_{test}$
42             Generate performance metrics for $Mp_l$
43             $Mp(l+1)$
44             $l \leftarrow l+1$
45         Saved performance metrics $Mp$ to $F_{mp}$
46 end
```

all the hyperparameters of the M_{in} model, which are: the number of epochs H_e, the batch size, the initial learning rate, the decay factor, and the values of the two L2 regulators. The P parameter contains the following system configuration variables: the P_{ckpt} variable is every few epochs that the model is saved, the P_s variable is the number of examples selected per class, and the P_c variable is the selected classes.

The **output parameters** of the TSDEM algorithm are as follows: the F_{model} parameter is a file containing the trained model, the F_{mt} parameter is a file containing the training results, the parameter F_{md} is a file that contains the detection results, and the parameter F_{mp} is a file that contains the performance results.

From line 1 to line 10 is the assignment and initialization of the variables used in the algorithm, such as the initialization of the F files to store the model and metrics, as well as the HW flag to indicate the hardware accelerator to use.

From line 11 to line 18, the data selection process is carried out, where the examples of the specified classes C are extracted to generate the superclasses SC; From the examples of each superclass, the training set D_{train} and test D_{test} are created, then each class and superclass are balanced, obtaining the training set S_{train} and the test set S_{test}. And finally, the TensorFlow Records for training TFR_{train} and TFR_{test} of the balanced sets S_{train} and S_{test} are generated; these records facilitate and optimize the storage and loading of data, in a simple format, based on a sequence of binary records.

From lines 19 to 22, the hardware accelerator with the HW flag is selected to perform the training stage. Depending on whether the workstation or cloud computing is used, different configurations are made through parameter P. In the case of CPU or GPU, it is configured for the workstation based on different scripts, and in the case of the TPU, it is configured for cloud computing with other scripts.

From lines 23 to 29, the selection and training process of the M_{in} model is carried out to obtain the M_{out} trained model and its M_t training metrics. This cycle trains the M_{in} model up to the hyperparameters H up to many epochs indicated by E. Every certain number of epochs the model is saved in a $CKPT$ checkpoint file, and every epoch the training metrics are generated and saved in F_{mt}.

From lines 30 to 34, the evaluation process of the M_{out} trained model is carried out, to obtain its M_d detection metrics. This cycle evaluates the M_{out} model up to the number of epochs indicated by E. Each epoch the detection metrics are generated and stored in M_d. In the end, all the metrics are collected in the file F_{md}.

From lines 35 to 41, the process of implementation and performance evaluation of the trained model M_{out} is carried out, to obtain its M_p performance metrics. This process is focused on CPU, GPU, or Jetson, to know the difference in the *Precision* of the training in TPU, so TPU is not used at this stage. This cycle evaluates the performance of the M_{out} model by the number of epochs indicated by E. In each epoch, the performance metrics are generated and stored in M_p. In the end, all the metrics are collected in the F_{mp} file.

The technical details are described below: the configuration files that contain all the required parameters of datasets, the model and its parameters, and hyperparameters are in a bucket in the Google Cloud Storage Service. The execution of the training of the selected model and the trained model is saved from every certain number of epochs in a checkpoint file, in the same mentioned bucket. The model is downloaded to be tested in any of the selected architectures (CPU, GPU, or Embedded System); and the evaluation scripts are executed in the architecture to generate the metrics, graphs and to be able to perform the analysis of the results. This algorithm allows assigning various user parameters such as model name, dataset, hyperparameters such as batch size, number of steps, learning rate, L2 regulator value, and also in which architecture it will be tested (CPU, GPU, or Embedded System). For the testing and implementation stage, scripts were developed to evaluate the performance of the models both on the workstation and on the Jetson Xavier Embedded System. Various tools such as Google Colab, TensorFlow Object Detection API, NVIDIA System Management Interface, Jetsonstats, and Tegrastats are used.

F. EVALUATION METRICS

The following describes the metrics used to evaluate each stage of detection for the DL models. In the detection stage, Bounding-Box, *Recall*, *Precision*, and *Average Precision* metrics are used to evaluate the sensitivity and *Precision* of the TSD systems. For the implementation stage, the metrics for system evaluation are *Memory consumption*, *Energy consumption*, *Latency*, and *Throughput*. The purpose is to compare different hardware accelerators.

In the detection task, the model's predictions can be evaluated through the Bounding-Box measure, in which the overlap ratio between the predicted Bounding-Box B_p and the ground truth box B_{gt} is calculated. A correct detection is obtained when the overlap ratio *Intersection over Union* (*IoU*) surpasses 0.5 using the equation:

$$IoU = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})} \tag{5}$$

where $B_p \cap B_{gt}$ denotes the intersection area, in pixels, of both predicted and ground truth bounding boxes. The term $B_p \cup B_{gt}$ denotes the union area in pixels. The possible values for the threshold range from 0 to 1 [67].

The classification results were divided into three groups: true positives (TP), which are examples correctly labeled as positives; false positives (FP) refer to negative examples incorrectly labeled as positive; and false negatives (FN), which are positive examples labeled incorrectly as negative. In [68] were defined the metrics *Precision* and *Recall* based on the groups mentioned above. The *Precision* metric in Eq. 6 is defined as the number of true positives divided by the sum of true positives and false positives, and the *Recall* metric in Eq. 7 is the number of true positives divided by the sum of true positives and false negatives (the sum is just the number of ground-truths).

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

Although a curve for *Precision* and *Recall* relation can be drawn, a better way to evaluate a detector is using the *Average Precision* (*AP*) metric [67]. The *AP* summarizes the shape of the *Precision/Recall* curve and is defined as the mean precision at a set of eleven equally spaced *Recall* levels [0, 0.1, . . . , 1]. The *AP* evaluates the *Average Precision* across all unique *Recall* values, and its equation is defined as:

$$AP = \frac{1}{n} \sum_{r \in [0,0.1,\ldots,1]} P_{interp}(r) \tag{8}$$

where n represents the number of *Recall* levels used. In this work, the PASCAL VOC challenge's [67] new standard is chosen, rather than the traditional way, which implements only 11 *Recall* levels. The *Precision* at each *Recall* level r is interpolated by taking the maximum *Precision* measured for a method for which the corresponding *Recall* exceeds r:

$$P_{interp}(r) = \max_{\tilde{r}:\tilde{r} \geq r} p(\tilde{r}) \tag{9}$$

where $p(\tilde{r})$ is the measured *Precision* at *Recall* $\tilde{r}$. The *Precision/Recall* curve is interpolated to reduce the impact of the "wiggles" caused by small variations in the ranking of examples. To obtain a high level of *AP*, a high *Precision* value must be obtained for each *Recall*, because its mean is calculated.

Memory consumption is defined as the maximum memory usage during the training [59], [60]. Intermediate data is the most memory-consuming among the different variables (DL model, hardware accelerator, or application). This data includes the machine learning framework, also known as the workspace and the feature map.

Energy consumption (E) is crucial in the inference process of deep learning models. This consumption determines if the model will be feasible within the battery capacity of the mobile device [59], [60]. In Eq. 10, the time interval (ti) defines how frequently sample data j is collected, and power consumption (pwr) represents the whole system power of the mobile device collected in a power sample data j.

$$E = \sum_j ti_j \times pwr_j \tag{10}$$

The *Latency* of a model, also known as inference time, is the amount of time it takes to train a single sample of the batch size [59], [60]. Specifically, considering the time required by the model to deliver an output response upon receiving the input image, smaller time values mean a better model speed is attained. Eq. 11 describes *Latency* calculation, where the number of samples (num_spl) defines the total number of samples to be processed, the start of the inference ($start\_infer$) indicates the time when the model started processing the sample, and the end of the inference (end_infer) represents the time when the model finished processing the sample. The time is measured in milliseconds.

$$Latency = \frac{num\_spl}{end\_infer - start\_infer} \tag{11}$$

To evaluate the efficiency of the DL model processing, the metric of *Throughput* is utilized. *Throughput* describes how many inferences can be delivered given the size of the deep learning network created or deployed. DL inference *Throughput* is generally expressed as images-per-second for image-based networks and the system must achieve *Throughput* within a specified *Latency* threshold [69]. In based on [59] this metric it is defined as follows, the number of samples (images) that can be processed in the *Latency* of the DL model. Eq. 12 defines the calculation of *Throughput* metric.

$$Throughput = \frac{num\_batches \times batch\_sz}{Latency} \qquad (12)$$

where the *number of batches* (*num_batches*) refers to the number of batches to be processed, and the *batch size* (*batch_sz*) establishes the number of samples that the model feeds, and the *Latency* refers to the time calculated in Eq. 11 and establishes the time required for the model to process a single sample. For example, if we have 100 batches, a batch size of 32, and a model inference time of 5 seconds, the *Throughput* is 640 samples per second.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

We divided the experimental phase into two parts: training of DL algorithms and embedded system implementation. For training the DL algorithms, we used a subset of the LISA database [51]. It is constituted of 900 samples and is divided into three different subclasses (prohibitory, mandatory, and warning) with 300 samples each. Each superclass contains three classes subsequently, as shown in Table 1. The dataset is separated by 90% for the training phase and the remaining 10% for the testing phase. Training is done using a CPU (AMD Ryzen 3600), a GPU (NVIDIA 2060 SUPER card), and a Google Colab TPU. For the embedded system implementation, we used the NVIDIA Jetson AGX Xavier embedded system board. The detection algorithm was implemented using the TensorFlow Object Detection API [70], where COCO [71] database pre-trained models that support TPU hardware are selected. Evaluation of traffic sign detection is done using COCO Object Detection metrics, where *mAP* with different sized objects (small, medium, and large) is chosen explicitly for detection model evaluation. The combination of detection systems is composed of a meta-architecture (SSD) and two convolutional feature extractors (MobileNet v FPN and ResNet50 v1 FPN). System 1 (S1) is formed by SSD + MobileNet v1 FPN, and System 2 (S2) is composed of SSD + ResNet50 v1 FPN architectures. The hyperparameters used for both S1 and S2 are a batch size of 32, with 2,600 steps, an Adaptive Momentum Estimation (ADAM) optimizer, an initial learning rate of 0.000999, a decay factor of 0.950, and a refresh every 26 steps (1 epoch). Two L2 regulators with a value of 0.000025 were used for both the box-predictor and the feature extractor.

A. TRAINING EVALUATION

The following results are from the training systems S1 and S2, with each hardware accelerator selected. The training stage was composed of 809 samples, a batch size of 32, with 100 epochs, and 2,600 steps.

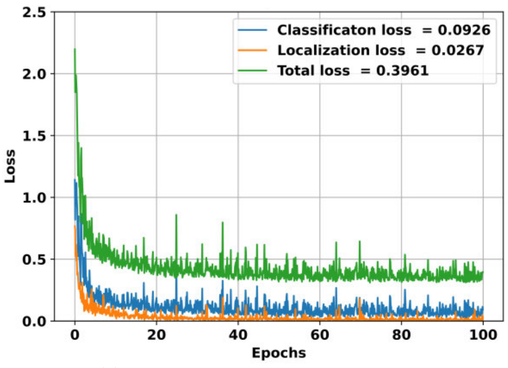
Table 3 shows training time as well as the number of training steps per second for each hardware accelerator, where it can be observed that CPU is the one that requires the most amount of training time and TPU attains the best results on both metrics. This is due to three important elements that differ depending on the type of hardware accelerator; these elements are primitive calculation, memory architecture, and its design purpose. TPU utilizes tensor calculation and can handle up to 128,000 operations per cycle and have a specific purpose of accelerating the development of DL tasks employing TensorFlow for attaining a faster training process.

TABLE 3. System training time results. The best results are in bold.

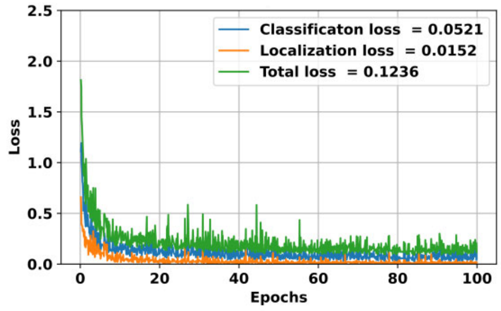
Hardware accelerators	System	Step execution time (secs)	Training time (hrs)
CPU	**S1**	**46.11**	**33.30**
CPU	S2	79.85	57.67
GPU	**S1**	**2.26**	**1.63**
GPU	S2	2.77	2
TPU	**S1**	**0.14**	**0.10**
TPU	S2	0.20	0.15

Generally, the most critical factor in the overall performance of a DNN model is the architecture. The hardware accelerator is significantly important. Most DNN models are based on 32-bit floating-point matrix multiplication [22]. These operations require floating-point arithmetic units for vector processing in a combination of bandwidth memories. Due to this, hardware accelerators are generally required to allow massive parallel processing of data, where the GPU is generally selected instead of the CPU. Currently, the custom-designed TPU hardware accelerator for Machine Learning is considered. This selection is due to the computing power of the TPU, which is higher than that of the GPU. The TPU has multiple array units (MXUs) implemented in systolic arrays. These units allow higher performance for larger batches, and this reduces model training time. On the other hand, the GPU is a better option for small batches and large models. This is because the memory system is better optimized in the transmission and memory reuse operations. Also, the GPU has a higher bandwidth than the TPU, which allows it to better adapt to the memory requirements of large models [63].

Therefore, it can be seen that the hardware selection will depend on several factors, such as model size, batch size, model architecture, and its operations. An efficient combination of a hardware accelerator and a DNN architecture will help to improve the convergence of the cost functions of the models. This combination will provide greater stability, as well as lower *Latency* in the model steps. On the other hand, an inefficient combination could cause bottlenecks, causing the opposite effect, and affecting the inference of the model. These bottlenecks will depend on the workload, hardware memory resources, and the design of the DNN. Fig. 10 shows the losses related to the CPU hardware accelerator, where the total loss for S2 is lower than in S1. This is because the architecture of this system uses waste to improve its performance. It can also be noted that there is a type of shattering because there are not enough processors to avoid *Throughput* bottleneck. In Fig. 11, the losses related to the GPU hardware accelerator are shown. Where the total loss for S2 is also lower than in S1, we notice a smoother loss (without shattering). In this case, the number of cores in the

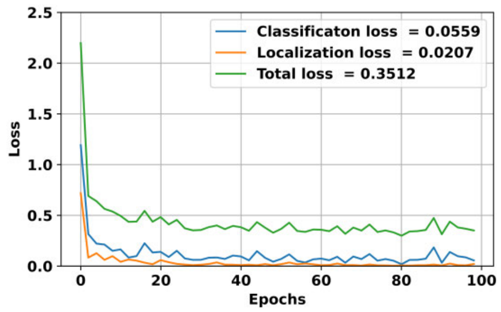


(a) System S1 MobileNet v1 FPN.

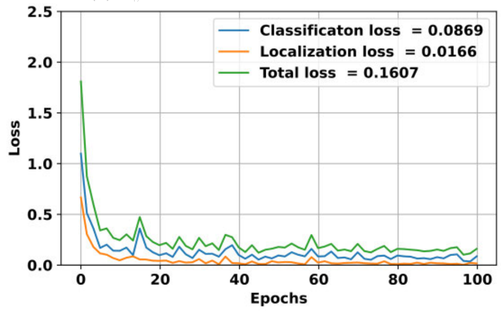


(b) System S2 ResNet50 v1 FPN

FIGURE 10. Training loss on CPU of Systems S1 and S2.



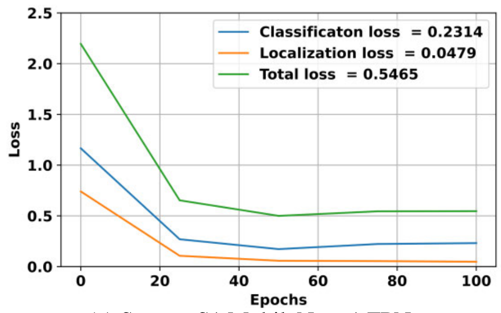
(a) System S1 MobileNet v1 FPN.



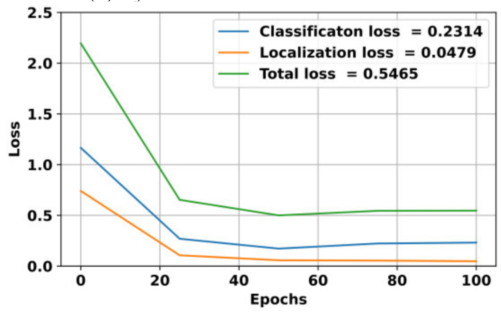
(b) System S2 ResNet50 v1 FPN

FIGURE 11. Training loss on GPU of Systems S1 and S2.

accelerator is 182 times higher, having a lower *Throughput* and avoiding bottlenecks. In Fig. 12, the losses related to the TPU hardware accelerator are shown; the total loss for S2 continues to be less than in S1 as in the previous accelerators, but in this case, we can see that both the losses of S1 and S2 are more significant than in the other accelerators, the loss that is affected or the greatest is the classification loss.



(a) System S1 MobileNet v1 FPN.



(b) System S2 ResNet50 v1 FPN

FIGURE 12. Training loss on TPU of Systems S1 and S2.

These higher TPU losses are due to the type of operations they use. TPU uses bfloat (Brain Floating Point Format). A regular TPU Cloud Service has two 128 × 128 systolic arrays, adding 32,768 ALUs for 16-bit floating-point values on a single processor [72].

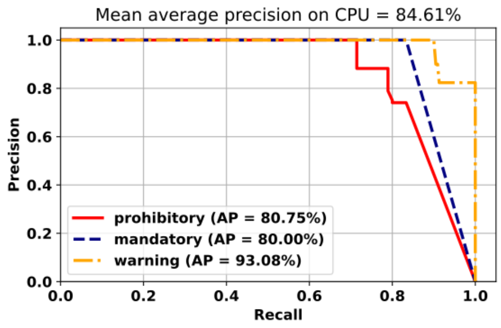
B. DETECTION EVALUATION

The following detection results are from S1 and S2, with each hardware accelerator selected. *Precision*, *Recall*, and *mAP* were the selected metrics and the COCO metrics for different sized objects (small, medium, and large) to evaluate the systems under different types of traffic signs.

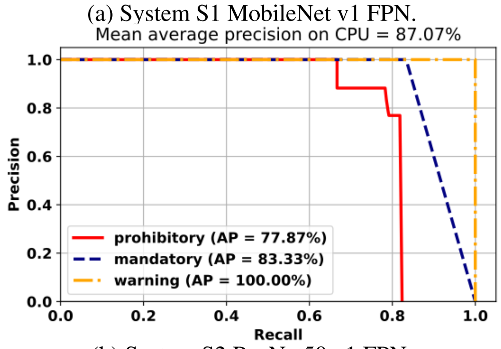
Table 4 shows superclass detection results for each hardware accelerator system, where a noticeable difference can be seen among the implemented devices. Even though the same hyperparameters are used (training sample number, epoch number, batch size, type of optimizer, and regulation constants) different results were obtained. As can be seen in the combinations **(S1 + TPU)**, **(S2 + GPU)**, and **(S2 + CPU)** have better results in some superclasses than other combinations with the same model but with different hardware accelerators. This difference is owed to the primitive operations mentioned in section IV-A, proving that the use of different hardware accelerators has an impact on the results of DL-based systems [73]. Fig. 13 shows the *mean Average Precision* results of the CPU hardware accelerator. The S2 system has better detection results in the *mAP* (87.07%). Obtaining better results for the warning (100%) and mandatory (88.33%) class; however, in the prohibitory class (77.87%), it is not better than that of the S1 system (80.75%). Fig. 14 shows the results of the *mean Average Precision* of the GPU hardware accelerator. In this case,

TABLE 4. ***Precision*, *Recall*, and *AP* results for each System with different hardware accelerators (*IoU* = 0.5). The best results are in bold.**

Hardware accelerator	System	Classes	*Precision*	*Recall*	*AP*
CPU	S1	Prohibitory	0.7404	0.8333	0.8075
		Mandatory	1.0	0.8	0.8
		Warning	0.7857	0.9167	0.9308
CPU	**S2**	Prohibitory	0.7404	0.8333	0.8075
		Mandatory	**1.0**	**0.8333**	**0.8333**
		Warning	**0.7667**	**1.0**	**1.0**
GPU	S1	Prohibitory	0.7931	0.92	0.9259
		Mandatory	1.0	0.7667	0.7667
		Warning	0.7586	0.9565	0.9566
GPU	**S2**	Prohibitory	0.7778	0.84	0.8256
		Mandatory	0.9583	0.7931	0.7934
		Warning	**0.7667**	**1.0**	**1.0**
TPU	**S1**	**Prohibitory**	**0.8148**	**0.8462**	**0.8394**
		Mandatory	0.9231	0.8571	0.8525
		Warning	0.7586	0.9565	0.9631
TPU	S2	Prohibitory	0.7826	0.6923	0.6679
		Mandatory	0.9565	0.7586	0.7546
		Warning	0.7586	0.9565	0.9565

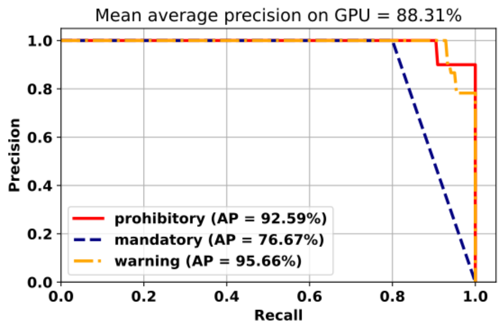


(a) System S1 MobileNet v1 FPN.

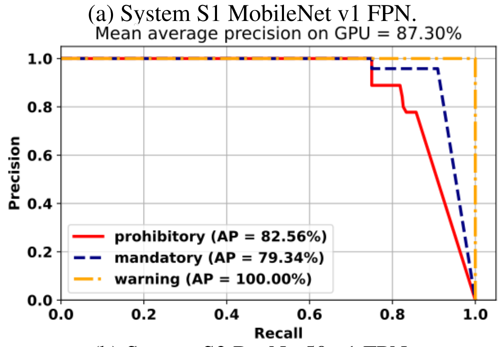


(b) System S2 ResNet50 v1 FPN.

FIGURE 13. ***mean Average Precision* on CPU of Systems S1 and S2.**

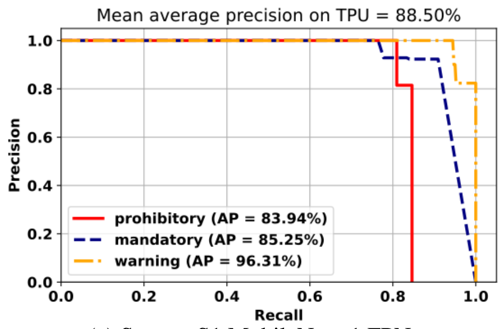


(a) System S1 MobileNet v1 FPN.

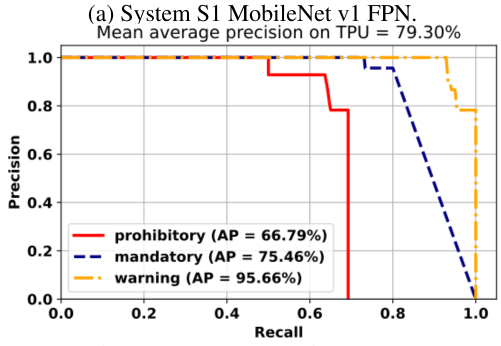


(b) System S2 ResNet50 v1 FPN.

FIGURE 14. ***mean Average Precision* on GPU of Systems S1 and S2.**



(a) System S1 MobileNet v1 FPN.



(b) System S2 ResNet50 v1 FPN.

FIGURE 15. ***mean Average Precision* on TPU of Systems S1 and S2.**

it is observed that the S1 system is the one that obtains the best detection results in the *mAP* (88.31%). Obtaining better results for the warning class (95.66%) and the prohibitory class (92.59%); however, in the mandatory class (76.67%) it is worse than that of the S1 system (79.34%); though to MobileNet v1 architecture (S1) was developed for embedded vision applications and mobile devices, and ResNet50 v1 (S2) focuses on computational accuracy and also has more parameters; the pointwise convolution and depthwise convolution operations of S1 create new features that with more epochs it gets better results than S2 combination. Fig. 15 shows the *mean Average Precision* results of the TPU hardware accelerator, where the S1 system gets better detection results in the *mAP* (88.50%); it showed better results for all classes, warning (96.31%), mandatory (85.25%), prohibitory (83.94%).

The difference between the *mAP* results can be seen more clearly through the COCO metric evaluation, considering that the size of the objects does not vary dramatically for the S1 system. However, for the S2 system, the results of the TPU are worse than those of the CPU and GPU, and this is

TABLE 5. ***mAP* results for S1 and S2 using COCO metrics with different hardware accelerators. The: represents a range with an increment 0.5 and the best results are in bold.**

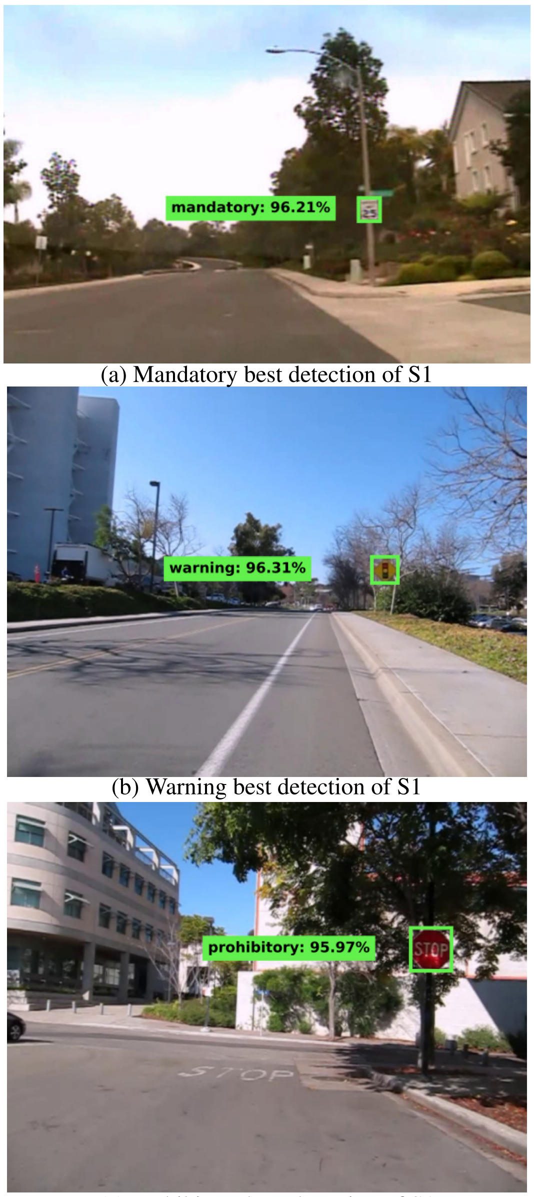
Hardware accelerator	System	Area	*IoU*	*mAP*
CPU	S1	ALL	0.50:0.95	0.8622
			0.50	0.8461
			0.75	0.8901
		SMALL	0.50:0.95	0.8435
		MEDIUM	0.50:0.95	0.8862
		LARGE	0.50:0.95	0.8451
CPU	**S2**	**ALL**	**0.50:0.95**	**0.8778**
			0.50	**0.8707**
			0.75	**0.8428**
		SMALL	**0.50:0.95**	**0.9221**
		MEDIUM	**0.50:0.95**	**0.8834**
		LARGE	**0.50:0.95**	**0.8933**
GPU	**S1**	**ALL**	**0.50:0.95**	**0.8985**
			0.50	**0.8831**
			0.75	**0.9223**
		SMALL	**0.50:0.95**	**0.9011**
		MEDIUM	**0.50:0.95**	**0.8886**
		LARGE	**0.50:0.95**	**0.8362**
GPU	S2	ALL	0.50:0.95	0.8824
			0.50	0.8730
			0.75	0.9018
		SMALL	0.50:0.95	0.9033
		MEDIUM	0.50:0.95	0.9036
		LARGE	0.50:0.95	0.7545
TPU	**S1**	**ALL**	**0.50:0.95**	**0.8959**
			0.50	**0.8850**
			0.75	**0.8716**
		SMALL	**0.50:0.95**	**0.8864**
		MEDIUM	**0.50:0.95**	**0.9410**
		LARGE	**0.50:0.95**	**0.8196**
TPU	S2	ALL	0.50:0.95	0.8091
			0.50	0.7930
			0.75	0.7933
		SMALL	0.50:0.95	0.8890
		MEDIUM	0.50:0.95	0.8163
		LARGE	0.50:0.95	0.7586

due to the bottleneck mentioned above in its feature extractor (ResNet50), which requires memory-bound operations [63].

Table 5 shows detection results through COCO metrics considering two cases. First, all sizes of objects are contemplated, and second selecting different categories depending on the number of pixels for the object area, like small (0 to 32^2 pixels), medium (0 to 46^2 pixels), and large (46^2 to 92^2 pixels). The previously mentioned results in Table 5 have been corroborated once again; combinations of S1 with TPU and GPU attain the best performance; on the other side, the combination of S1 with CPU attains better results than GPU and TPU, due to the bottleneck caused by the feature extractor ResNet50 and its memory capacity dependency.

Fig. 16 shows illustrations of the detection of the S1 system in the CPU. In the scene of the mandatory and warning class the size of the objects is smaller due to the distance that was used, and yet acceptable results were obtained. In the prohibitory class scene, it can be observed that there is a lower lighting condition than the one in the environment due to the shadow generated by the tree on the traffic sign, and the algorithm can detect it with this variation.

Fig. 17 shows illustrations of the detection of the S2 system in the CPU. In the mandatory class scene, the algorithm has great robustness since the traffic sign is attached to a gray post which has a similar color; however, the algorithm manages to properly locate the traffic sign. In the scene of the warning class, there are many cars with different colors that could confuse the algorithm, however, the systems yield good



(a) Mandatory best detection of S1
(b) Warning best detection of S1
(c) Prohibitory best detection of S1

FIGURE 16. Illustrations of the best detection results of the S1 System on CPU.

detection performance. In the prohibitory class scene, another case of illumination can be seen, the lens is being affected by a flash of light, having a high illumination condition, however, the algorithm demonstrates its robustness in traffic sign detection under challenging lighting conditions.

Fig. 18 shows illustrations of the detection of the S1 system in the GPU. In the scene of the mandatory class two objects could cause noise to the algorithm, the sidewalk where the traffic sign already has a gray color and a rectangular shape and the wall on the right that separates the trees, which has a rectangular shape and a beige color, having a similarity with the geometric and color characteristics of the traffic sign, however, the algorithm yields acceptable results in traffic sign detection. In the scene of the warning class, it is noted

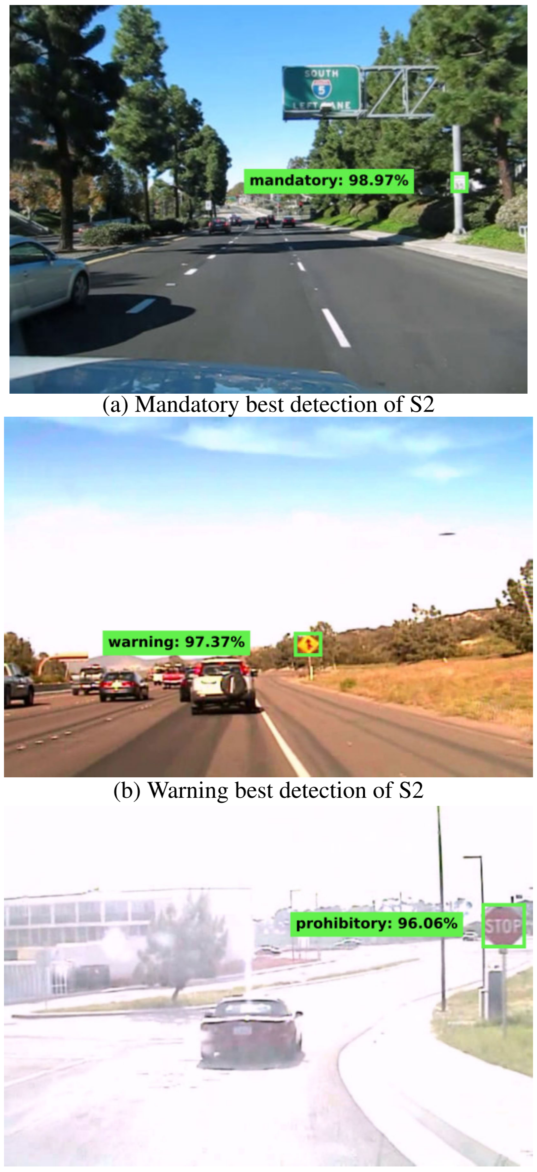

mandatory: 98.97%
(a) Mandatory best detection of S2
warning: 97.37%
(b) Warning best detection of S2
prohibitory: 96.06%
(c) Prohibitory best detection of S2

FIGURE 17. Illustrations of the best detection results of the S2 System on CPU.

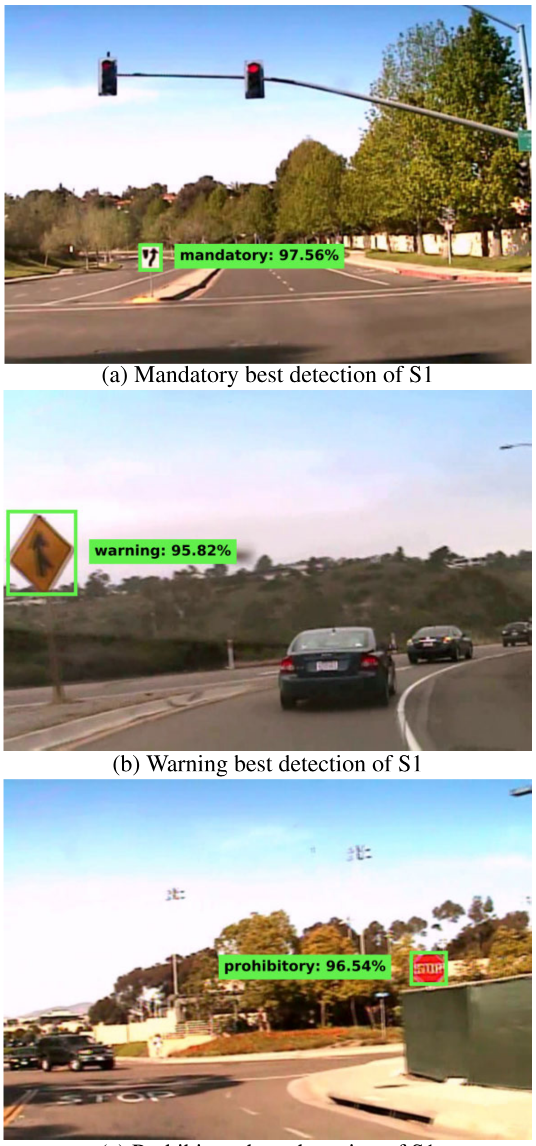
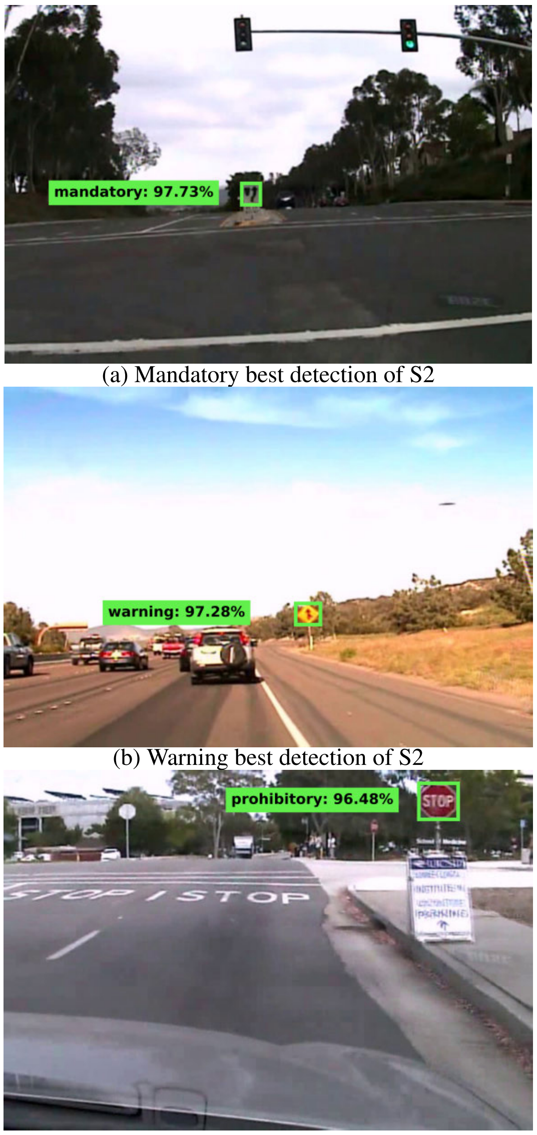

mandatory: 97.56%
(a) Mandatory best detection of S1
warning: 95.82%
(b) Warning best detection of S1
prohibitory: 96.54%
(c) Prohibitory best detection of S1

FIGURE 18. Illustrations of the best detection results of the S1 System on GPU.

that there is a certain blur generated by the acquisition of the moving image, however, the algorithm presents good performance to detect the traffic sign under this condition. In the scene of the prohibitory class, there is also a blur, but the algorithm manages to detect under this condition. Fig. 19 shows illustrations of the detection of the S2 system in the GPU, where the scene of the mandatory class is taken with a low-resolution camera and presents some blur in the captured scene, since some details of the borders of the paving are blurred, however, the algorithm has no problem detecting the traffic sign. In the warning class scene, as it is shown, it is surrounded by several cars with different shapes and colors, and the algorithm yields high performance in traffic sign detection. In the prohibitory class scene, it is possible to appreciate that there is a blur, and a stop sign is behind in a lateral position, and on the left side there is another stop sign facing away, having some opportunity to confuse the algorithm with its similar properties, however, the algorithm presents outstanding robustness under these conditions and detects the traffic sign. Fig. 20 shows illustrations of the detection of the S1 system in the TPU, where it can be observed that in the scene of the mandatory class there are too many objects of different sizes and colors, where some of these share colors of the traffic sign such as vehicles and there is also a traffic sign from behind that could confuse the algorithm due to its similar geometric shape, however, the algorithm demonstrates its robustness. Also, the presented algorithm yields good detection performance in the scene of the warning class, which also presents another case of blurring due to the acquisition in motion in the scene capture, as well as cluttered background containing objects

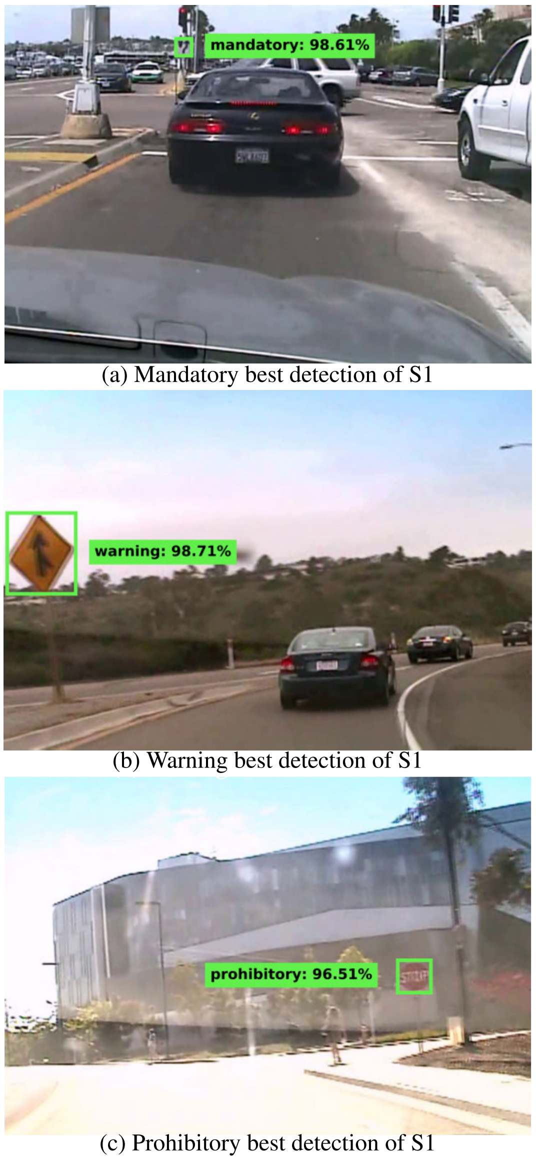


(a) Mandatory best detection of S2

(b) Warning best detection of S2

(c) Prohibitory best detection of S2

FIGURE 19. Illustrations of the best detection results of the S2 System on GPU.



(a) Mandatory best detection of S1

(b) Warning best detection of S1

(c) Prohibitory best detection of S1

FIGURE 20. Illustrations of the best detection results of the S1 System on TPU.

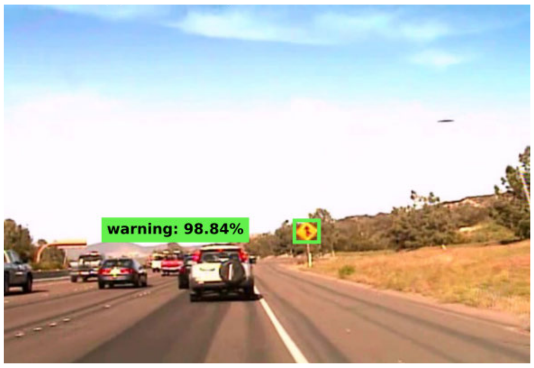
and vehicles. In the scene of the prohibitory class, the image scene presents a challenging lighting condition, which that affects the color properties of the traffic sign; however, the algorithm has no problem detecting the traffic sign under this condition. Fig. 21 shows illustrations of the detection of the S2 system in the TPU, where it can be seen that in the scene of the mandatory class two objects could cause noise to the algorithm, the sidewalk where the traffic sign is already has a gray color and a rectangular shape and the wall on the right that separates the trees, which has a rectangular shape and a beige color, having a similarity with the geometric and color characteristics of the traffic sign, however, the algorithm yields high detection results. In the warning class scene, as it is shown, it is surrounded by several cars with different shapes and colors, and the algorithm manages to detect the traffic sign. In the prohibitory class scene, it is observed that there is a lower lighting condition than the environment due to the shadow generated by the tree on the traffic sign, and the algorithm presents robustness of traffic sign detection with this variation.

As shown in the illustrations above, different detection results are obtained for current scenarios from the LISA database. Considering that the detection is achieved under different lighting conditions, blur, multiple objects, or with similar characteristics, different sizes, or distances of the objects.

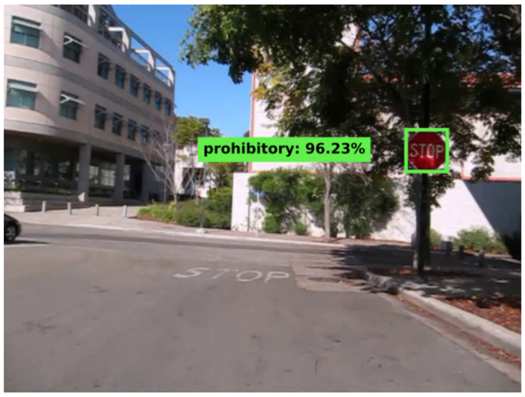
C. PERFORMANCE EVALUATION

The results of the performance evaluation of the S1 and S2 systems for each hardware accelerator are shown below, Table 6 shows the characteristics of the hardware accelerators used [73]–[76].

(a) Mandatory best detection of S2



(b) Warning best detection of S1



(c) Prohibitory best detection of S2

FIGURE 21. Illustrations of the best detection results of the S2 System on TPU.

TABLE 6. Hardware accelerators specifications. HBM: amount of high-bandwidth memory. MXU: scalar, vector, and matrix units.

Hardware accelerators	Architecture	Memory	FP32 (float) performance
Cloud TPU v2-8	8 GiB of HBM for each TPU core One MXU for each TPU core	64 GiB	180 TFLOPS
NVIDIA GeForce RTX 2060 SUPER	2176-core NVIDIA Turing-TU106	8 GB GDDR6	7.181 TFLOPS
NVIDIA Jetson AGX Xavier	512-core NVIDIA Volta-GPU	32 GB LPDDR4x	1,410 GFLOPS
AMD Ryzen 5 3600	6-core Zen 2	16 GB DDR4 DRAM	1,209.6 GFLOPS

In the performance evaluation, *Latency*, *Throughput*, *Memory consumption*, and *Energy consumption* metrics were considered.

TABLE 7. Performance results for S1 and S2 with *Latency*, and *Throughput* metrics with different hardware accelerators. The best results are in bold.

Hardware accelerators	System	*Latency* (ms)	*Throughput* (samples / *Latency*)
CPU	**S1**	**1,440.94**	**2.22**
CPU	S2	2,495.31	1.28
GPU	**S1**	**70.63**	**45.31**
GPU	S2	86.56	36.97
TPU	**S1**	**4.38**	**730.59**
TPU	S2	6.25	512

TABLE 8. Performance results for S1 and S2 with average *Memory consumption* and average *Energy consumption* metrics with different hardware accelerators. The best results are in bold.

Hardware accelerators	System	*Memory consumption* (GB)	*Energy consumption* (Watts)
GPU	**S1**	**5**	**48.57**
GPU	S2	5	50.8
Jetson Xavier	**S1**	**9**	**2.38**
Jetson Xavier	S2	10	2.44

Table 7 shows the performance results for TPU, GPU, and CPU. These results are focused on the efficiency analysis; therefore, only the *Latency* and *Throughput* metrics are used. Statistics show that S1 is the best match across all architectures, both in *Latency* and *Throughput*. The description of the experiment to obtain the above results is as follows:

- In the case of the *Latency*, metric, 256 samples were used, and a timeline tracker was used to know the start and end of the algorithm's execution time. The times that do not involve the inference time were ignored, they remained as the initial time when the sample enters the network and the final time when the detection result is delivered. For the GPU, 300 samples were used to perform the warmup and the warmup time was leftover.
- In the case of the *Throughput* metric, the number of batches was 100, with a batch size of 32, and the inference time was previously calculated.

The results above show that the TPU architecture with S1 is the best performing combination. Although the network architecture is not optimized, acceptable performance is obtained due to the characteristics of the models such as the depth of the layers and the operations they perform, which impacts the results [77].

Table 8 shows the performance results for the GPU and Jetson Xavier hardware accelerators. These results are focused on the resources consumption analysis; therefore, only the *Memory consumption* and *Energy consumption* metrics are used. In this analysis, we only consider the most feasible implementations (generic GPU and specific Jetson embedded device); for both the training and inference stage. We discard the CPU and TPU hardware architectures in this analysis due to the following:

- In the case of the CPU, both training and inference results show (see Table 3 and Table 7) that both proposed combinations have bottlenecks. This is because this hardware has not enough cores to take advantage of

TABLE 9. Results obtained using the LISA database. The −− represents missing results and the best results are in bold.

Methods	Classes	*Precision*	*Recall*	*AP / mAP*	COCO *mAP* (Area ALL)	*Latency* (ms)	*Throughput* (samples / *Latency*,)	*Memory consumption* (GB)	*Energy consumption* (Watts)
ICF in [78]	Prohibitory	−−	−−	0.8732 (*AP*)					
	Mandatory	−−	−−	0.9109 (*AP*)					
	Warning	−−	−−	0.9603 (*AP*)					
	All	−−	−−	0.9148 (*mAP*)	−−	−−	−−	−−	−−
ACF in [78]	Prohibitory	−−	−−	0.9898 (*AP*)					
	Mandatory	−−	−−	0.9617 (*AP*)					
	Warning	−−	−−	0.9611 (*AP*)					
	All	−−	−−	**0.9708** (*mAP*)	−−	−−	−−	−−	−−
Multiscale cascaded R-CNN in [79]	Prohibitory	−−	−−	−−					
	Mandatory	−−	−−	−−					
	Warning	−−	−−	−−					
	All	**0.9890**	0.8560	−−	−−	−−	−−	−−	−−
Ours (S1 + CPU)	Prohibitory	0.7404	0.8333	0.8075 (*AP*)					
	Mandatory	1.0	0.8	0.8 (*AP*)					
	Warning	0.7857	0.9167	0.9308 (*AP*)					
	All	0.8420	0.85	0.8461 (*mAP*)	0.8622	1,440.94	2.22	−−	−−
Ours (S2 + CPU)	Prohibitory	0.7404	0.8333	0.8075 (*AP*)					
	Mandatory	1.0	0.8333	0.8333 (*AP*)					
	Warning	0.7667	1.0	1.0 (*AP*)					
	All	0.8357	**0.8888**	0.8802 (*mAP*)	0.8778	2,495.31	1.28	−−	−−
Ours (S1 + GPU)	Prohibitory	0.7931	0.92	0.9259 (*AP*)					
	Mandatory	1.0	0.7667	0.7667 (*AP*)					
	Warning	0.7586	0.9565	0.9566 (*AP*)					
	All	0.8506	0.8810	0.8830 (*mAP*)	**0.8985**	70.63	45.31	**5**	**48.57**
Ours (S2 + GPU)	Prohibitory	0.7778	0.84	0.8256 (*AP*)					
	Mandatory	0.9583	0.7931	0.7934 (*AP*)					
	Warning	0.7667	1.0	1.0 (*AP*)					
	All	0.8342	0.8777	0.8730 (*mAP*)	0.8824	86.56	36.97	**5**	50.8
Ours (S1 + TPU)	Prohibitory	0.8148	0.8462	0.8394 (*AP*)					
	Mandatory	0.9231	0.8571	0.8525 (*AP*)					
	Warning	0.7586	0.9565	0.9631 (*AP*)					
	All	0.8321	0.8866	0.8850 (*mAP*)	0.8959	**4.38**	**730.59**	−−	−−
Ours (S2 + TPU)	Prohibitory	0.7826	0.6923	0.6679 (*AP*)					
	Mandatory	0.9565	0.7586	0.7546 (*AP*)					
	Warning	0.7586	0.9565	0.9565 (*AP*)					
	All	0.8325	0.8024	0.7930 (*mAP*)	0.8091	6.25	512	−−	−−

parallel processing and the DL architecture design, causing bottlenecks in the data processing.

- In the case of TPU, although the training results show (see Table 3) that it is the best option, this hardware is not feasible for the inference stage. This is because TPU relies on cloud services, and the only available embedded devices that have TPUs do not have enough RAM (currently) for our proposed combinations.

The results show that the S1 combination is the best, both on the GPU and the embedded system. The experiment carried out to obtain the previous results is the same as that in Table 7. In this case, the tools mentioned in the evaluation Algorithm 1 were used. The *Memory consumption* and power consumption between the GPU and the Jetson show that each hardware accelerator has a specific purpose to obtain the best possible performance based on its task. In the case of the GPU, it gets higher power consumption with lower *Memory consumption*, due to its purpose of maximizing processing performance. In the case of the Jetson Xavier, there is a higher *Memory consumption* with lower power consumption due to its purpose of minimizing power consumption for use in applications with low energy demand. The results show that the embedded system achieves 20 times less power consumption than the GPU; however, its *Memory consumption* is higher because it shares memory for CPU and GPU when the model is loaded, the samples are loaded, and the inference is performed [46].

D. RESULTS COMPARISON

In this subsection, we show our testing stage results against the results of previous works in the LISA database. Because not all works show results based on the metrics used in this work, we put all our metrics in Table 9.

Table 9 shows the results obtained from different methods used in the LISA database. To make an objective comparison of the *AP* and *mAP* results of the evaluated works, we added the class (All); this class represents "all the superclasses" considered on different works. As it is shown, each work considered different classes for each superclass. Therefore, to compare the methods, the *mAP* results of the class "All" are used.

As it can be seen, the results of the method of Mogelmose *et al.* in [78] are better than the other methods. However, it can also be noted that state-of-the-art works of the LISA database do not have *Latency*, *Memory consumption*, and *Energy consumption* metrics. Because of this, we do not know if it can be applied to a real-time environment. Therefore, although our proposed method does not obtain the best results, our methodology, experimental tests, metrics, and results obtained allow us to carry out a preliminary analysis for its application in real-time, being essential for the deployment of an autonomous vehicle.

V. CONCLUSION

Recent reports about accidents in autonomous vehicles can identify some challenging problems that it is important to solve to avoid accidents. This work is intended to contribute to advances in safety for autonomous vehicles. Therefore, we focused on developing an evaluation methodology based on statistical analysis and algorithms for Deep Learning to be implemented into embedded systems applied to the Traffic Sign Detection (TSD) task for autonomous vehicles.

Our first contribution provides a descriptive statistical analysis where different combinations of models and deep learning hardware architectures for a workstation and an embedded system were evaluated. This is important because deep learning systems are complex, and the number of combinations that may form a system is large, being easy to have bottlenecks and other design problems. The descriptive statistical analysis provides information about different deep learning models and hardware architecture combinations. For example, for workstations, we have found that the models MobileNet v1 FPN (S1) and ResNet50 v1 FPN (S2) have the shortest training times. The achieved TPU speedup of the S1 system training is 333 compared to CPU training and 16.3 times compared with GPU training. The TPU speedup training of the S2 system is 384.46 compared with the CPU training and 13.33 compared with the GPU. Using the *mean Average Precision* metric (*mAP*) for the ALL size cases of objects, we found that the S1 in TPU has a difference in its favor of 0.0337 when compared against the CPU, and a disfavorable difference of 0.0026 when compared with the GPU; in the same way, *mAP* for the S2 in TPU is 0.0687 better than in CPU and 0.0733 worse than in GPU. The implementation performance was evaluated with the *Latency* and *Throughput* metrics, which show that:

- The case of the S1 combination system implemented in TPU has 1,436.56 ms of less *Latency* than in CPU, and the TPU *Throughput* is 328.98 better. When comparing the TPU vs GPU implementation, we obtained that the TPU has 66.25 ms less *Latency* than the GPU, and the TPU *Throughput* is 16.13 better.
- For the case of the S2 combination, we found that the TPU has 2,489.06 ms less *Latency* than the CPU, and the TPU *Throughput* is 399.25 better. When comparing the TPU vs GPU implementation, we obtained that the TPU has 80.31 ms less *Latency* than the GPU, and the TPU *Throughput* is 13.85 better.

Memory consumption and *Energy consumption* are also important information that can be obtained with statistical analysis. The *Memory consumption* metric tells us that the S1 combination implemented into an embedded system requires 4 GB more of memory than the same combination when implemented into a workstation; the S2 combination implemented into an embedded system requires 5 GB more of memory than when the same application is implemented into a workstation. The *Energy consumption* metrics indicate that the power consumption of both system combinations, S1 and S2, implemented into an embedded system is less than when implemented into a workstation; in these cases, 46.19 and 48.32 fewer watts, correspondingly.

Therefore, the descriptive statistical analysis helps to identify the strengths and weaknesses of deep learning models for different system targets. In our case, we identify that using the TPU provides a substantial reduction of training time, obtaining a good approximation of the desired results; however, one of its main disadvantages is that technical factors such as the use of Google Colab and Google Cloud TPU Cloud Services or acquiring an integrated device from Google Coral is required. Also, the support of the TPU models for DL is less than the CPU and GPU support, and similarly for the TensorFlow versions. Now on the side of the implementation in embedded systems with this method, we see that the advantage of a workstation is that it distributes the *Memory consumption* better due to its type of hardware architecture so that it will not have any problem with the data load, model, and processing concerning *Memory consumption*, however, this will be reflected in the power consumption. In the case of the Jetson Xavier embedded system, it is observed that the *Memory consumption* is almost double, so for other embedded systems that do not have this capacity, it would be a problem; therefore, this evaluation method supports us to confirm which model is more appropriate in order to avoid such implementation problems.

The second contribution of the paper is regarding the methodology, which was divided into four stages. The first stage (data selection) produces a balanced set of examples to improve the training, enclosing the 47 classes of traffic signs in three superclasses to generalize and facilitate the object detection problem. The second and third stages (Hardware and model selection) focus on selecting the best combination of hardware architecture and DL model, covering training time and detection results. The last stage (embedded application) focuses on knowing if the best combination proposed in the previous stage is feasible to implement in an embedded system or is better in a workstation. This methodology uses a simple and practical framework to know the minimum requirements before implementing a DL model into an embedded system. This is an important issue since, at present, there is no standardized methodology that guides users to achieve an appropriate model and hardware selection for a DL application.

Our third contribution provided an algorithm for the proposed methodology. The purpose of this algorithm is to divide each stage into simple repetitive scripts. For example, for the TPU training, we used a Google Colab Notebook for the connection and configuration for cloud computing. Furthermore, for the Workstation, we developed separate python scripts to train, evaluate, and measure any stage of the methodology. Also, for the testing of both Workstation and an Embedded System, we developed a bash script to measure both hardware and model DL performance. This algorithm will help to easily reproduce and perform possible improvements to the results of this work.

Overall, all the work provides a global methodological framework based on descriptive statistical analysis and an evaluation algorithm. This descriptive statistical analysis and evaluation algorithm helped to select the most suitable combination between models and hardware architectures for DL vision applications. This methodological framework allows knowing the minimum requirements before the implementation of a DL model in an embedded system. The evaluation algorithm covers the connection and configuration of cloud computing with TPU to the implementation in the embedded system. The algorithm is divided into small, simple, and reusable scripts for each stage, allowing easy reproduction, making possible future improvements to the algorithms and results of this work.

The obtained results indicate that the combination S1 (SSD + MobileNetv1 FPN) in TPU was the best combination. This combination obtained better results on all stages and almost all metrics, getting a speedup of 16.3 for the training and a minimum difference between accuracies of 0.0363 for both CPU and GPU.

This work allowed us to analyze the methodology, methods, and metrics used in DL for the TSD problem. We considered the performance results of the DL models, such as *Latency*, *Throughput*, *Memory consumption*, and *Energy consumption*, which in other works were not considered as shown in Table 9. Therefore, based on the analysis and results obtained, we proposed to include these metrics, to have a broader analysis when implementing the methods in real-time environments, in this case to autonomous vehicles.

REFERENCES

[1] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, Apr. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231216315533

[2] E. Dickmanns and A. Zapp, "Autonomous high speed road vehicle guidance by computer vision1," *IFAC Proc. Volumes*, vol. 20, no. 5, pp. 221–226, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667017553203

[3] G. Piccioli, E. De Micheli, and M. Campani, "A robust method for road sign detection and recognition," in *Computer Vision—ECCV*, J.-O. Eklundh, Ed. Berlin, Germany: Springer, 1994, pp. 493–500.

[4] D. Temel, T. Alshawi, M.-H. Chen, and G. AlRegib, "Challenging environments for traffic sign detection: Reliability assessment under inclement conditions," 2019, *arXiv:1902.06857*. [Online]. Available: https://arxiv.org/abs/1902.06857

[5] S. B. Wali, M. A. Abdullah, M. A. Hannan, A. Hussain, S. A. Samad, P. J. Ker, and M. B. Mansor, "Vision-based traffic sign detection and recognition systems: Current trends and challenges," *Sensors*, vol. 19, no. 9, p. 2093, May 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/9/2093

[6] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Netw.*, vol. 32, pp. 323–332, Aug. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608012000457

[7] S. Lafuente-Arroyo, P. Gil-Jimenez, R. Maldonado-Bascon, F. Lopez-Ferreras, and S. Maldonado-Bascon, "Traffic sign shape classification evaluation I: SVM using distance to borders," in *Proc. IEEE Intell. Vehicles Symp.*, Jun. 2005, pp. 557–562.

[8] Y. Saadna and A. Behloul, "An overview of traffic sign detection and classification methods," *Int. J. Multimedia Inf. Retr.*, vol. 6, no. 3, pp. 193–210, Sep. 2017, doi: 10.1007/s13735-017-0129-8.

[9] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep learning for generic object detection: A survey," *Int. J. Comput. Vis.*, vol. 128, no. 2, pp. 261–318, Jan. 2020, doi: 10.1007/s11263-019-01247-4.

[10] C. Liu, S. Li, F. Chang, and Y. Wang, "Machine vision based traffic sign detection methods: Review, analyses and perspectives," *IEEE Access*, vol. 7, pp. 86578–86596, 2019.

[11] M. Carranza-García, J. Torres-Mateo, P. Lara-Benítez, and J. García-Gutiérrez, "On the performance of one-stage and two-stage object detectors in autonomous vehicles using camera data," *Remote Sens.*, vol. 13, no. 1, p. 89, Dec. 2020. [Online]. Available: https://www.mdpi.com/2072-4292/13/1/89

[12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[13] G. Jocher *et al.* (Apr. 2021). Ultralytics/YOLOv5: V5.0—YOLOv5-P6 1280 Models, AWS, Supervise.ly and YouTube Integrations. [Online]. Available: https://doi.org/10.5281/zenodo.4679653

[14] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.

[15] Z. Cai and N. Vasconcelos, "Cascade R-CNN: Delving into high quality object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2017, pp. 6154–6162.

[16] M. Lopez-Montiel, Y. Rubio, M. Sánchez, and U. Orozco-Rosas, "Evaluation of algorithms for traffic sign detection," *Proc. SPIE*, vol. 11136, Proc. 2019, Art. no. 111360M. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11136/2529709/Evaluation-of-algorithms-for-traffic-sign-detection/10.1117/12.2529709.full

[17] M. Lopez-Montiel, U. Orozco-Rosas, M. Sánchez-Adame, K. Picos, and O. Montiel, "Evaluation of deep learning algorithms for traffic sign detection to implement on embedded systems," in *Studies in Computational Intelligence*, vol. 915. Cham, Switzerland: Springer, 2021, pp. 95–115. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-58728-4_5

[18] M. McFarland, "UBER self-driving car operator charged in pedestrian death," in *Proc. CNN*, 2020. [Online]. Available: https://edition.cnn.com/2020/09/18/cars/uber-vasquez-charged/index.html

[19] K. Wiggers. (2020). *Waymo's Driverless Cars Were Involved in 18 Accidents Over 20 Months*. VentureBeat. [Online]. Available: https://venturebeat.com/2020/10/30/waymos-driverless-cars-were-involved-in-18-accidents-over-20-month/

[20] A. Cuthbertson. (2020). *Tesla Autopilot Crash Driver Killed After Playing Video Games, Investigation Reveals*. The Independent. [Online]. Available: https://www.independent.co.uk/life-style/gadgets-and-tech/news/tesla-crash-death-autopilot-video-model-x-self-driving-a9358971.html

[21] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *J. Field Robot.*, vol. 37, no. 3, pp. 362–386, Apr. 2020, doi: 10.1002/rob.21918.

[22] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[23] H.-J. Jeong, K.-S. Park, and Y.-G. Ha, "Image preprocessing for efficient training of YOLO deep learning networks," in *Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp)*, Jan. 2018, pp. 635–637.

[24] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Comput. Surveys*, vol. 53, no. 4, pp. 1–37, Sep. 2020, doi: 10.1145/3398209.

[25] A. Berthelier, T. Chateau, S. Duffner, C. Garcia, and C. Blanc, "Deep model compression and architecture optimization for embedded systems: A survey," *J. Signal Process. Syst.*, pp. 1–16, Oct. 2020, doi: 10.1007/s11265-020-01596-1.

[26] F. J. Belmonte, S. Martin, E. Sancristobal, J. A. Ruiperez-Valiente, and M. Castro, "Overview of embedded systems to build reliable and safe ADAS and AD systems," *IEEE Intell. Transp. Syst. Mag.*, early access, Feb. 12, 2020, 10.1109/MITS.2019.2953543.

[27] K. Muhammad, A. Ullah, J. Lloret, J. D. Ser, and V. H. C. de Albuquerque, "Deep learning for safe autonomous driving: Current challenges and future directions," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 7, pp. 4316–4336, Jul. 2021.

[28] W. Wang, Y. Fu, Z. Pan, X. Li, and Y. Zhuang, "Real-time driving scene semantic segmentation," *IEEE Access*, vol. 8, pp. 36776–36788, 2020.

[29] M. M. D. Santos, J. E. Hoffmann, H. G. Tosso, A. W. Malik, A. U. Rahman, and J. F. Justo, "Real-time adaptive object localization and tracking for autonomous vehicles," *IEEE Trans. Intell. Vehicles*, early access, Nov. 16, 2020, doi: 10.1109/TIV.2020.3037928.

[30] D.-H. Lee, K.-L. Chen, K.-H. Liou, C.-L. Liu, and J.-L. Liu, "Deep learning and control algorithms of direct perception for autonomous driving," *Int. J. Speech Technol.*, vol. 51, no. 1, pp. 237–247, Jan. 2021, doi: 10.1007/s10489-020-01827-9.

[31] J. Cheng, P.-S. Wang, G. Li, Q.-H. Hu, and H.-Q. Lu, "Recent advances in efficient computation of deep convolutional neural networks," *Frontiers Inf. Technol. Electron. Eng.*, vol. 19, no. 1, pp. 64–77, Jan. 2018, doi: 10.1631/FITEE.1700789.

[32] M. Shafique, R. Hafiz, M. U. Javed, S. Abbas, L. Sekanina, Z. Vasicek, and V. Mrazek, "Adaptive and energy-efficient architectures for machine learning: Challenges, opportunities, and research roadmap," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2017, pp. 627–632.

[33] Á. Arcos-García, J. A. Álvarez-García, and L. M. Soria-Morillo, "Evaluation of deep neural networks for traffic sign detection systems," *Neurocomputing*, vol. 316, pp. 332–344, Nov. 2018.

[34] R. Ayachi, M. Afif, Y. Said, and M. Atri, "Traffic signs detection for real-world application of an advanced driving assisting system using deep learning," *Neural Process. Lett.*, vol. 51, no. 1, pp. 837–851, Feb. 2020, doi: 10.1007/s11063-019-10115-8.

[35] X. Bangquan and W. X. Xiong, "Real-time embedded traffic sign recognition using efficient convolutional neural network," *IEEE Access*, vol. 7, pp. 53330–53346, 2019.

[36] M. M. William, P. S. Zaki, B. K. Soliman, K. G. Alexsan, M. Mansour, M. El-Moursy, and K. Khalil, "Traffic signs detection and recognition system using deep learning," in *Proc. 9th Int. Conf. Intell. Comput. Inf. Syst. (ICICIS)*, Dec. 2019, pp. 160–166.

[37] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, 2017, doi: 10.1145/3140659.3080246.

[38] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," *Integration*, vol. 69, pp. 309–320, Nov. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167926019301762

[39] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *J. Physiol.*, vol. 195, no. 1, pp. 215–243, 1968.

[40] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biol. Cybern.*, vol. 36, no. 4, pp. 193–202, Apr. 1980.

[41] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Proc. NIPS*, 1989, pp. 396–404.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[43] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn.* Madison, WI, USA: Omni Press, 2010, pp. 807–814.

[44] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proc. 27th Int. Conf. Mach. Learn.* Madison, WI, USA: Omni Press, 2010, pp. 111–118.

[45] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision—ECCV*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham, Switzerland: Springer, 2014, pp. 818–833.

[46] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen, "Recent advances in convolutional neural networks," *Pattern Recognit.*, vol. 77, pp. 354–377, May 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320317304120

[47] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," *Computer Vision—ECCV* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2016, pp. 21–37, doi: 10.1007/978-3-319-46448-0_2.

[48] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[49] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," Jul. 2017, *arXiv:1704.04861*. [Online]. Available: http://arxiv.org/abs/1704.04861

[50] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 936–944.

[51] A. Mogelmose, M. M. Trivedi, and T. B. Moeslund, "Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 13, no. 4, pp. 1484–1497, Dec. 2012.

[52] H. Huang, Z. Liu, T. Chen, X. Hu, Q. Zhang, and X. Xiong, "Design space exploration for YOLO neural network accelerator," *Electronics*, vol. 9, no. 11, p. 1921, Nov. 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/11/1921

[53] X. Zhang, X. Wei, Q. Sang, H. Chen, and Y. Xie, "An efficient FPGA-based implementation for quantized remote sensing image scene classification network," *Electronics*, vol. 9, no. 9, p. 1344, Aug. 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/9/1344

[54] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, vol. 12, no. 7, p. 113, Jul. 2020. [Online]. Available: https://www.mdpi.com/1999-5903/12/7/113

[55] S. Hossain and D.-J. Lee, "Deep learning-based real-time multiple-object detection and tracking from aerial imagery via a flying robot with GPU-based embedded devices," *Sensors*, vol. 19, no. 15, p. 3371, Jul. 2019.

[56] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[57] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[58] V. Mazzia, A. Khaliq, F. Salvetti, and M. Chiaberge, "Real-time apple detection system using embedded systems with hardware accelerators: An edge AI application," *IEEE Access*, vol. 8, pp. 9102–9114, 2020.

[59] J. Liu, J. Liu, W. Du, and D. Li, "Performance analysis and characterization of training deep learning models on mobile devices," Jul. 2019, *arXiv:1906.04278*. [Online]. Available: http://arxiv.org/abs/1906.04278

[60] J. Hanhirova, T. Kämäräinen, S. Seppälä, M. Siekkinen, V. Hirvisalo, and A. Ylä-Jääski, "Latency and throughput characterization of convolutional neural networks for mobile computer vision," Jul. 2018, *arXiv:1803.09492*. [Online]. Available: http://arxiv.org/abs/1803.09492

[61] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive deep learning model selection on embedded systems," *ACM SIGPLAN Notices*, vol. 53, no. 6, pp. 31–43, Dec. 2018, doi: 10.1145/3299710.3211336.

[62] M. Loni, M. Daneshtalab, and M. Sjodin, "ADONN: Adaptive design of optimized deep neural networks for embedded systems," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 397–404.

[63] J. Wang, S. Jiang, W. Song, and Y. Yang, "A comparative study of small object detection algorithms," in *Proc. Chin. Control Conf. (CCC)*, 2019, pp. 8507–8512.

[64] S. You, Q. Bi, Y. Ji, S. Liu, Y. Feng, and F. Wu, "Traffic sign detection method based on improved SSD," *Information*, vol. 11, no. 10, p. 475, Oct. 2020. [Online]. Available: https://www.mdpi.com/2078-2489/11/10/475

[65] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE Access*, vol. 6, pp. 24411–24432, 2018.

[66] J. Xu, B. Wang, J. Li, C. Hu, and J. Pan, "Deep learning application based on embedded GPU," in *Proc. 1st Int. Conf. Electron. Instrum. Inf. Syst. (EIIS)*, 2017, pp. 1–4.

[67] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal visual object classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, Sep. 2009.

[68] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *Proc. 23rd Int. Conf. Mach. Learn.* New York, NY, USA: Association for Computing Machinery, 2006, pp. 233–240, doi: 10.1145/1143844.1143874.

[69] A. D. Teich and R. P. Teich, "PLASTER: A framework for deep learning performance," NVIDIA, Santa Clara, CA, USA, Tech. Rep., 2018. [Online]. Available: https://images.nvidia.com/content/pdf/plaster-deep-learning-framework.pdf

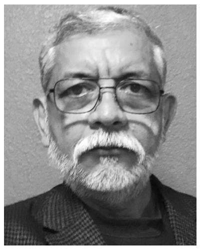
[70] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7310–7311.

[71] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, 2015, pp. 740–755.

[72] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*. [Online]. Available: https://arxiv.org/abs/1905.12322
[73] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking TPU, GPU, and CPU platforms for deep learning," 2019, *arXiv:1907.10701*. [Online]. Available: https://arxiv.org/abs/1907.10701
[74] TechPowerUp. (2021). *AMD Ryzen 5 3600 Specs*. Accessed: Mar. 10, 2020. [Online]. Available: https://www.techpowerup.com/cpu-specs/ryzen-5-3600.c2132
[75] (2021). *Nvidia Geforce RTX 2060 Super Specs*. Accessed: Mar. 10, 2020. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-super.c3441
[76] (2021). *Nvidia Jetson AGX Xavier GPU Specs*. Accessed: Mar. 10, 2020. [Online]. Available: https://www.techpowerup.com/gpu-specs/jetson-agx-xavier-gpu.c3232
[77] V. J. Reddi *et al.*, "MLPerf inference benchmark," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, May 2020, pp. 446–459.
[78] A. Mogelmose, D. Liu, and M. M. Trivedi, "Detection of U.S. traffic signs," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 6, pp. 3116–3125, Sep. 2015.
[79] J. Zhang, Z. Xie, J. Sun, X. Zou, and J. Wang, "A cascaded R-CNN with multiscale attention and imbalanced samples for traffic sign detection," *IEEE Access*, vol. 8, pp. 29742–29754, 2020.

MIGUEL LOPEZ-MONTIEL received the B.Eng. degree in computer engineering from the Universidad Autónoma de Baja California (UABC), Tijuana, México, in 2017, and the M.Sc. degree in digital systems from the Centro de Investigación y Desarrollo de Tecnología Digital (CITEDI), Tijuana, in 2020, where he is currently pursuing the Ph.D. degree. His current research interests include automated machine learning, deep learning, neural networks, neural architecture search, and computer vision applications for autonomous vehicles and robotics.

ULISES OROZCO-ROSAS (Member, IEEE) received the B.Eng. degree in electronics engineering from the Universidad Autónoma de Baja California, Mexico, in 2004, and the M.Sc. and Ph.D. degrees in digital systems from the Instituto Politécnico Nacional, Mexico, in 2014 and 2017, respectively. He held a postdoctoral position. From 2017 to 2018, he was appointed with the Department of Computer Science, ETSII, Universidad Rey Juan Carlos, Spain. He is currently an Associate Professor with the School of Engineering, CETYS Universidad. He is a member of the National System of Researchers–CONACYT. His research activities include the design of algorithms and software development for autonomous vehicles. His research interests include machine learning, computational and artificial intelligence, parallel and heterogeneous computing, autonomous vehicles, and mobile robots.


MOISÉS SÁNCHEZ-ADAME received the M.Sc. degree in digital systems from the Instituto Politécnico Nacional (IPN), in 1997, and the Ph.D. degree in computer networks and communication systems from the CETYS Universidad, Campus Mexicali, Baja California, Mexico, in 2005. He is currently a full-time Research Professor with the Center for Research and Development of Digital Technology (CITEDI), IPN. He has published articles about wireless communications, wireless sensor networks, and machine learning. His current research interests include machine learning, deep learning, quantum computing, and quantum cryptography.

KENIA PICOS received the Ph.D. degree from the Instituto Politécnico Nacional, Centro de Investigación y Desarrollo de Tecnología Digital, in 2017. She is currently a full-time Professor with the School of Engineering, CETYS Universidad Campus Tijuana, Mexico. She is a member of the National System of Researchers (Sistema Nacional de Investigadores), CONACYT. Her current research interests include computer vision, object recognition, three-dimensional object tracking, pose estimation, and parallel computing with graphics processing units.

OSCAR HUMBERTO MONTIEL ROSS (Senior Member, IEEE) received the Bachelor of Science degree in electrical engineering and electronics from the Universidad Autónoma de Baja California (UABC), México, in 1985, the M.Sc. degree in digital systems from the Instituto Politécnico Nacional (IPN), México, in 1999, the M.Sc. degree in computer from the Instituto Tecnológico de Tijuana, México, in 2000, and the D.Sc. degree in computer science from the UABC, in 2006. He is currently a Researcher with the Centro de Investigación y Desarrollo de Tecnología Digital (CITEDI-IPN). He has leading 17 research projects and participated in about 20 others, most of them in the computational intelligence field. He has published articles, books, and book chapters about quantum computing, evolutionary computation, mobile robotics, mediative fuzzy logic, ant colonies, type-2 fuzzy systems, and embedded systems. He has experience editing books and journal's special issues. He is a member of the International Association of Engineers (IANG) and the Mexican Science Foundation Consejo Nacional de Ciencia y Tecnología (CONACYT). He received the Research Award 2016 from IPN. He is a Co-Founder and an Active Member of the Hispanic American Fuzzy Systems Association (HAFSA) and the Mexican Chapter of the Computational Intelligence Society (IEEE).