Computer Science and Information Systems 00(0):0000–0000 https://doi.org/10.2298/CSIS123456789X

Resource-Aware Object Detection and Recognition Using Edge AI Across the Edge-Fog Computing Continuum*

Dragan Stojanović[1], Stefan Sentić[1], Natalija Stojanović[1] and Teodora Stamenković[1]

Faculty of Electronic Engineering, University of Niš
Aleksandra Medvedeva 14, 18000 Niš
dragan.stojanovic@elfak.ni.ac.rs, stefan.sentic@elfak.rs, natalija.stojanovic@elfak.ni.ac.rs, teodora.stamenkovic@elfak.rs

Abstract. Edge computing and edge intelligence have gained significant traction in recent years due to the proliferation of Internet of Things (IoT) devices, the exponential growth of data generated at the network edge, and the demand for real-time and context-aware applications. Despite its promising potential, the application of Artificial Intelligence (AI) on the edge faces many challenges, such as edge computing resource constraints, heterogeneity of edge devices, scalability issues, security and privacy concerns, etc. The paper addresses the challenges of deploying deep neural networks (DNNs) for edge intelligence and traffic object detection and recognition on a video captured by edge device cameras. The primary aim is to analyze resource consumption and achieve resource-awareness, optimizing computational resources across diverse edge devices within the edge-fog computing continuum while maintaining high object detection and recognition accuracy. To accomplish this goal, a methodology is proposed and implemented that exploits the edge-to-fog paradigm to distribute the inference workload across multiple tiers of the distributed system architecture. The edge-fog related solutions are implemented and evaluated in several use cases on datasets encompassing real-world traffic scenarios and traffic objects' recognition problems, revealing the feasibility of deploying DNNs for object recognition on resource-constrained edge devices. The proposed edge-to-fog methodology demonstrates enhancements in recognition accuracy and resource utilization, validating the viability of both edge-only and edge-fog based approaches. Furthermore, experimental results demonstrate the system's adaptability to dynamic traffic scenarios, ensuring real-time recognition performance even in challenging environments.

Keywords: Resource awareness, Traffic Object Recognition, Edge AI, Distributed Neural Networks, Edge-Fog Computing Continuum.

1. Introduction

Edge computing refers to the paradigm of processing data near its source or point of collection, rather than relying solely on centralized cloud servers. Edge intelligence involves the integration of artificial intelligence (AI) and machine learning (ML) algorithms into edge devices, enabling them to perform data analytics and decision-making tasks locally. By bringing computation and intelligence closer to the data source, edge computing and edge intelligence offer numerous benefits, including reduced latency, improved bandwidth

* The paper is an extension of the paper presented at RAW 2023 workshop

efficiency, enhanced privacy and security, and increased resilience to network failures. AI on edge devices and infrastructure powers real-time, context-aware, and intelligent applications across various fields, such as healthcare, smart cities, industrial automation, transportation, and agriculture. However, its potential comes with challenges, including limited resources, diverse device architectures, scalability difficulties, and concerns around security and privacy. The evolution of edge AI across the edge-fog computing continuum has been extensively explored in recent literature, highlighting the significance of distributed ML and AI in enabling real-time decision-making [20]. It underscores the importance of integrating ML and AI algorithms into edge devices and fog nodes to facilitate intelligent data processing, minimizing latency and network bandwidth usage [8].

In recent years, the rapid proliferation of Internet of Things (IoT) devices, including microcontrollers, single-board computers and smartphones, equipped with built-in or externally connected cameras, has led to their ubiquitous usage across a myriad of applications requiring detection and recognition of objects on the real-time video streams. This widespread adoption has heralded the advent of edge intelligent systems across diverse domains, with a notable emphasis on traffic object detection and recognition. Accurately identifying and classifying traffic objects, such as cars, trucks, motorcycles, bicycles, and pedestrians, is crucial for efficient traffic management, advanced driver assistance systems (ADAS), and autonomous driving. There is an increasing reliance on ML and deep learning (DL) algorithms for video stream analysis for object detection and classification in safety-critical embedded systems and IoT applications, such as autonomous driving systems, surveillance systems and security robots. It emphasizes the need for ML/DL technologies to meet strict timing requirements in real-time systems while maintaining accuracy, given the potentially catastrophic consequences of missed deadlines. Bian et al. in [2] aim to provide a comprehensive exploration of state-of-the-art results in ML/DL-based scheduling techniques, accuracy trade-offs, and security considerations in real-time IoT systems. The potential of Deep Neural Networks (DNNs) to perform efficiently on edge IoT devices is particularly significant, as it harnesses the computational power and availability of these widely used devices. Exploring the intersection of advanced neural network architectures, real-time data processing, and distributed computing paradigms is essential for developing innovative solutions for traffic object detection, classification, and recognition. By combining the proximity and processing capabilities of edge devices, such as smartphones and microcontrollers, with fog servers and cloud infrastructure, research efforts aim to enhance the efficiency, accuracy, and responsiveness of traffic object detection systems.

This paper addresses the challenges of training and deploying DNNs for traffic object detection and recognition across various edge devices, including Android smartphones, microcontrollers, and single-board computers. The focus is on distributing computational and inference tasks between IoT devices at the far edge, edge servers, and fog servers. We examine different deployment strategies, balancing trade-offs between model size, inference speed, power consumption, and accuracy. Specifically, we evaluate two approaches: (1) quantizing and optimizing the DNN model as TensorFlow Lite for direct deployment on edge devices, and (2) deploying the original DNN model on an edge or fog server. We also explore distributing the object recognition task by dividing it into two inference stages: object detection performed at the edge and object recognition of the detected items carried out on the fog server.

Through a series of experiments on traffic object detection and recognition, we evaluate the performance, accuracy, and resource consumption across different platforms, including microcontrollers, smartphones, single-board computers, and commodity servers, under various video data parameters and configurations. The results offer valuable insights into the practicality and efficiency of distributing DNNs for traffic object recognition from the edge to the fog. These findings support resource-aware edge intelligence by optimizing computational resource usage while preserving high recognition accuracy. Furthermore, this research lays the groundwork for developing intelligent transportation systems that harness the potential of edge devices, such as Android smartphones and microcontrollers, along with fog computing infrastructure, to improve traffic safety and management.

This paper represents the extended version of the paper presented at the RAW 2023 Workshop [19].Besides significant extension of the related work section and detailed explanation of the traffic object detection and recognition solutions implemented and presented in the original paper, the main contribution of this paper lies in a new use case for traffic object detection over novel edge devices using a TinyML approach. This involves deploying the traffic object detection solution on a microcontroller (Arduino Nano 33 BLE Sense) and a single-board computer (Raspberry Pi 4). Various DNN models suitable for object detection and recognition tasks have been utilized in their original versions and subsequently optimized, quantified and compressed to enable deployment on mentioned edge devices. Extensive experiments have been conducted to evaluate the performance and accuracy of these applications concerning machine learning tasks. The experiments also assessed resource usage, including memory and processing time. The results have been described and analysed in detail. The extended paper provides thorough insights into the implementation and experimental evaluation of various traffic object detection and recognition tasks in different distributed configurations. It also covers the distribution of tasks across the edge-fog computing continuum, from microcontrollers and single-board computers to smartphones and fog servers (PCs).

The paper is structured as follows. Section II presents related research work in edge computing and edge intelligence related to object detection and recognition from video streams. Section III presents several strategies and corresponding applications for deployment of DNN for traffic object detection and recognition across various edge devices and a fog server. Section IV presents the experimental evaluation for various traffic object detection scenarios and discusses the evaluation results. Section V gives concluding remarks and directions for future research.

2. Related Work

A growing body of research has focused on harnessing the power of edge devices, such as IoT devices and edge servers, to perform real-time object detection and recognition tasks at the network edge, minimizing latency and bandwidth consumption. The convergence of edge computing and DL methods and techniques has enabled the deployment of resource-efficient object detection and recognition models directly on edge devices, facilitating autonomous decision-making and edge AI applications.

Singh and Gill in [18] gives the extensive review of the unique characteristics and advantages of deploying AI algorithms directly on edge devices, enabling real-time inference and decision-making at the network edge. Furthermore, the survey discusses the di-

verse applications of Edge AI across domains such as smart cities, healthcare, autonomous vehicles, and industrial automation, highlighting its transformative potential in enhancing efficiency, scalability, and privacy in distributed computing environments. The article delves into the challenges and open research issues associated with Edge AI, such as resource constraints, security concerns, and algorithmic optimizations.

The need to integrate ML techniques into resource-constrained embedded devices, facilitated by advancements in technologies like the IoT and edge computing has given rise to TinyML, an embedded ML technique, a by enabling ML applications on low-cost, resource-constrained devices. However, implementing TinyML comes with challenges such as processing capacity optimization and maintaining model accuracy [9].

Shuvo et al. in [17] address the challenges of deploying DNNs on edge devices. It highlights the computational complexity and memory requirements of DNNs, which often necessitate cloud-based processing, leading to latency issues and security concerns. The paper explores optimization techniques at both hardware and software levels to enable efficient DNN deployment on edge devices, focusing on four research directions: novel DL architecture and algorithm design, optimization of existing DL methods, algorithm-hardware co-design, and efficient accelerator design. Through a comprehensive review, the paper provides insights into state-of-the-art tools and techniques for efficient edge inference, aiming to facilitate the integration of artificial intelligence capabilities into next-generation edge devices.

The research on the distribution of DNNs for resource-aware systems has gained significant attention in recent years. Several studies have explored different approaches and strategies for optimizing the training and deployment of DNNs in various domains. In the context of object detection and recognition from edge to cloud, several relevant research papers provide valuable insights and inspiration. Bittencourt et al. [3] discuss the integration and challenges of the IoT, fog, and cloud continuum, highlighting the need for efficient resource utilization. Lockhart et al. [12] propose Scission, a performance-driven and context-aware cloud-edge distribution approach for DNNs, emphasizing the importance of considering context and performance in distribution decisions. Cho et al. [4] present a study on DNN model deployment on distributed edges, focusing on distributed inference across edge devices.

Lin et al. [11] propose a distributed DNN deployment approach from the edge to the cloud for smart devices, addressing the challenges of efficient utilization of resources in different computing tiers. McNamee et al. [14] advocate for adaptive DNNs in edge computing, emphasizing the need for dynamic adaptation to optimize resource usage. Ren et al. [16] provide a survey on collaborative DNN inference for edge intelligence, exploring the collaborative aspects of inference across edge devices. Hanhirova et al. [6] characterize the latency and throughput of convolutional neural networks (CNN) for mobile computer vision, providing insights into the perfor-mance aspects of DNNs on resource-constrained devices.

Lee et al. [10] propose Transprecise Object Detection (TOD) for maximizing real-time accuracy on the edge, highlighting the importance of accurate object detection for edge scenarios. Parthasarathy et al. [15]introduce DEFER, a distributed edge inference approach for DNNs, focusing on resource-efficient inference in distributed edge environments. Teerapittayanon et al. [21] investigate distributed DNNs over the cloud, edge, and

end devices, highlighting the trade-offs between resource utilization and computational capabilities across different components of the system architecture.

In line with our research, Dharani et al. in [5] present the utilisation of TinyML and TensorFlow Lite on a mobile phones for image classification, but without more extensive experimental evaluation and discussions. Akhtar et al. in [1] introduce multiple real-time deployable cost-efficient solutions for motorbike detection using state-of-the-art embedded edge devices, addressing the critical need for accurate and real-time traffic surveillance and road safety. The paper presents an improved baseline accuracy of motorbike detection by developing a custom network based on YOLOv5, offering practical insights for real-world implementation.

The aforementioned papers contribute to the understanding of resource aware DNN deployment and optimization techniques in various contexts. Our research aims to provide insights into the efficient training and deployment of DNNs for traffic object detection, classification and recognition, considering the resource utilization from edge to fog in the context of edge devices with various computing capabilities and resources available.

3. Traffic Object Detection and Recognition Across the Edge-Fog Continuum

DNNs have shown promising results in various computer vision tasks, including object detection and recognition. In this section we present three case studies related to distribution of DNN-based software components in the context of traffic object detection and recognition.

3.1. Traffic Object Recognition

To recognize traffic objects captured by a camera on Android smartphones, we implement two approaches: the first executes entirely on the edge device, while the second divides the process between the edge device and a fog node. The training of the DNN model is based on the TensorFlow Object Detection API and specifically utilizes the SSD MobileNet V2 320x320 coco17 tpu-8 pre-trained model. To enhance the training process, we have utilized video data captured from an Android phone camera, as well as publicly available traffic video datasets such as the Udacity Self Driving Car Dataset [22], INRIA Graz-02 (IG02)[13], and the Bike-rider Detector dataset [23]. Manual labeling was applied to these datasets when necessary to ensure accurate annotation of traffic objects, for detection of cars, trucks, motorcycles, bicycles, and pedestrians.

The first approach we propose utilizes the computing power and resources available solely on the smartphone, making it an offline approach. This method does not require a network connection for traffic object recognition within the Android application. To accommodate the limited resources available on the smartphone, the TensorFlow model used for recognition is quantized and converted to TensorFlow Lite form. By optimizing the model, we ensure that it can efficiently operate on the smartphone without compromising its performance. The trained model is integrated into the Android application, enabling real-time traffic object recognition directly on the smartphone without the need for an internet connection (Figure1). The Figure 2 illustrates the execution flow of object

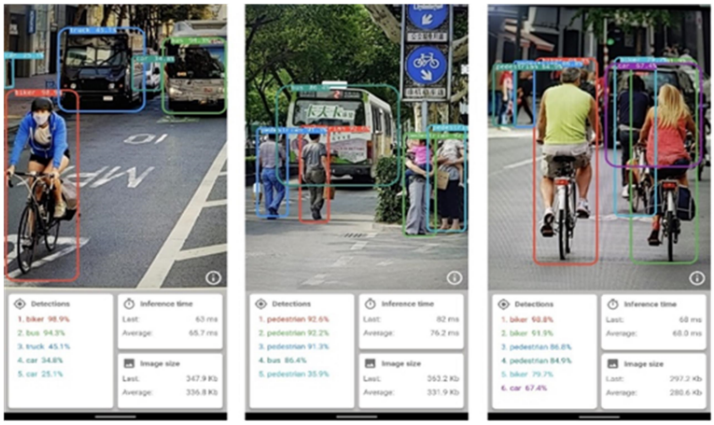

Fig. 1. Android application for traffic object recognition

recognition conducted within an Android application. Once the camera image is available, it is sent to the detector component. In offline mode, the *LocalDetector* component is employed, utilizing a TensorFlow Lite model deployed on the Android smartphone. Prior to inputting the image into the model, specific preparations, including scaling and rotation, must be completed. Furthermore, after detection, it is crucial to parse the results and generate objects that will be used for GUI creation.

The second approach utilizes the advantages of fog computing by offloading part of the processing to a fog server. In this method, the video captured by the smartphone camera undergoes preprocessing within the Android application. These preprocessing steps may include scaling, rotation, and filtering of the captured images to enhance the quality and clarity of the input data. The preprocessed images are then encoded in Base64 format and sent to the fog server through a Web socket using the SocketIO library. The fog server, implemented with Flask/Python, hosts the original TensorFlow model, which conducts object recognition on the received images. The results of this recognition process are returned to the Android application in JSON format, providing real-time feedback and visualization of the recognized traffic objects.

The execution flow of object recognition in edge-fog scenario is illustrated in Figure 3. The client application captures an image from the camera and processes it before sending it to the server. This processing includes scaling, rotation (considering the device's sensor), and encoding the image into Base64 format (as a string). The image is transmitted to the server as an emitted event indicating that it is ready for processing. Upon receipt, the server decodes the Base64 string to reconstruct the image. On the server side, the TensorFlow library is utilized to run the model and perform object detection within the frame. The identified objects are then returned to the client in JSON format.

The second method distributes the computational workload between the smartphone and the fog server, offloading resource-intensive tasks to a more powerful computing infrastructure. This approach enhances the accuracy and robustness of traffic object recognition, particularly in situations where the smartphone's resources are limited. Additionally, leveraging fog computing alleviates the strain on the smartphone's battery and pro-

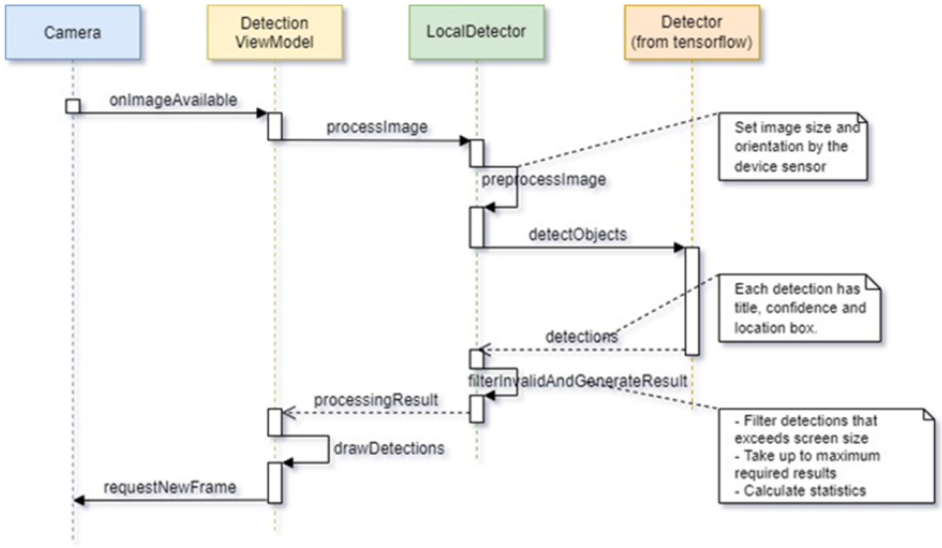

Fig. 2. The sequence diagram during the detection process on Android smartphone.

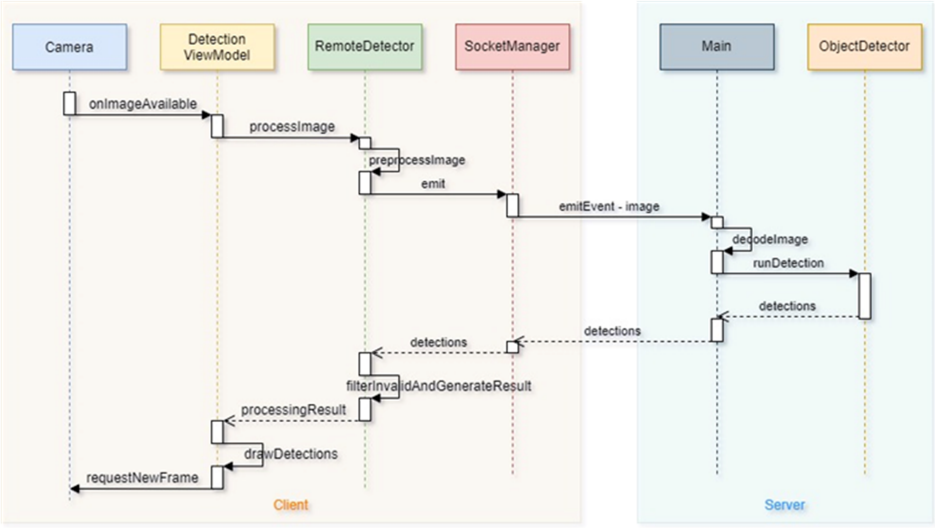

Fig. 3. Sequence diagram for the server-based object detection

cessing capabilities, resulting in better performance and an improved user experience. Both methods provide unique advantages regarding resource utilization, real-time performance, and accuracy, addressing various requirements and constraints, which are experimentally evaluated and discussed in the following section. The source code of Android application implementing traffic object recognition using both method is available at https://github.com/drstojanovic/camera.

3.2. Car Model Recognition

The second use case is related to car model recognition. To tackle the specific challenge of car model recognition, we propose a two-stage approach that leverages both edge and fog computing resources. Our method is built on the pre-trained convolutional neural network MobileNet V2, known for its outstanding performance across various computer vision tasks. For training the model specifically for car model recognition, we employ the Stanford Cars Dataset, which contains over 16,000 images and includes more than 190 different car classes.

The execution flow for recognizing car models is illustrated in Figure 4. As depicted, the process occurs in multiple phases, with some tasks handled on the edge side (client) and others on the fog side (server).

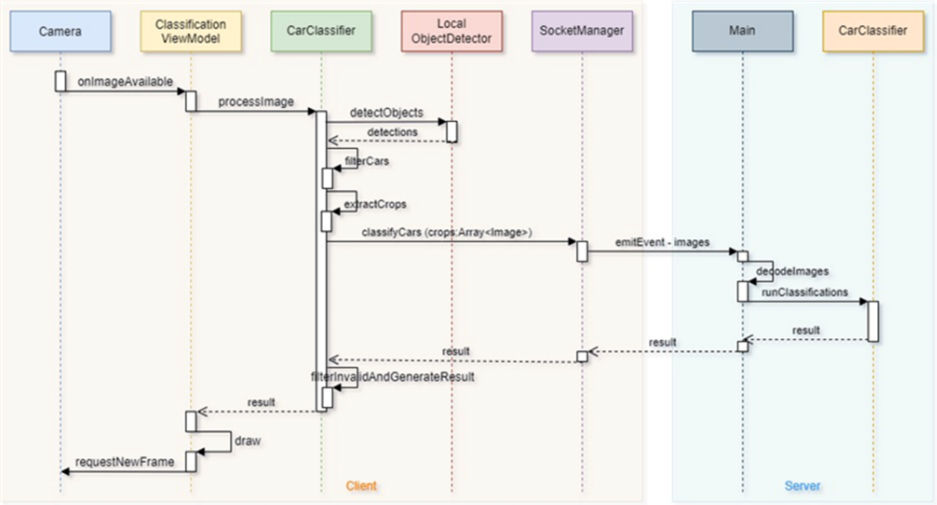

Fig. 4. Execution flow of a car model recognition

Since an existing object detector is used, it will return detections for all five classes of traffic objects: cars, trucks, motorcycles, bikes, and pedestrians. The first step is to filter out only the class of interest—cars. After capturing the image from the camera, the following steps are taken:

1. Object detection is conducted using an edge model, which discards all objects not belonging to the ”car” class.

2. Based on the detections, the image is cropped, creating a list of car crops.
3. Each crop is then encoded in Base64 format, and this list of strings is sent to the server.
4. Upon receipt, the server decodes the list back into images. Each crop is resized to fit the model's input size (192x192 in this application).
5. The model is then executed on each crop, providing a list of potential classes for each image.
6. The results are formatted in JSON, resulting in a list of lists whose length corresponds to the number of detected cars or crops obtained.
7. On the edge side, the detection results are merged with the classification results from the server, and a bounding box is drawn around each car, displaying the recognized class.

The high-level flow of the car model recognition is illustrated in Figure 5. In the first

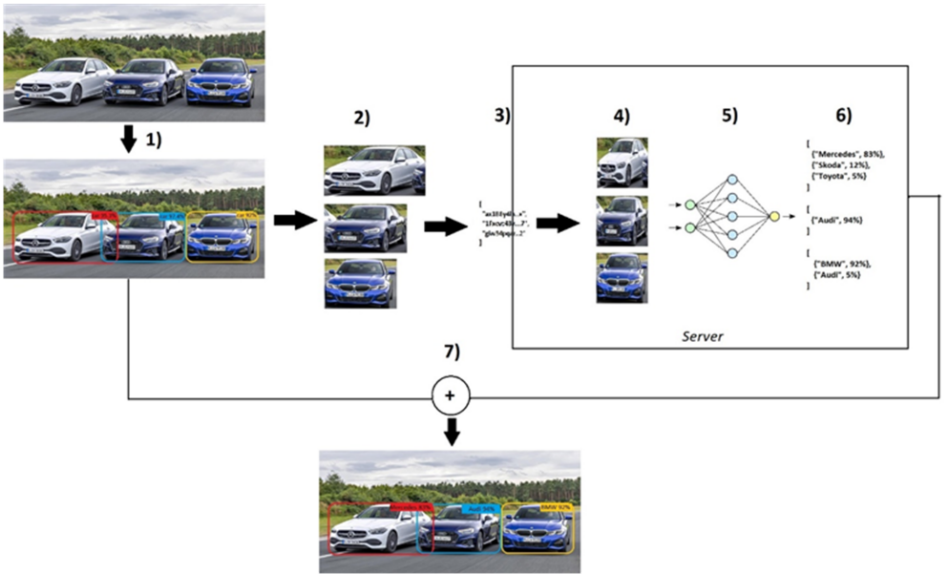

Fig. 5. The steps performed for car model detection.

stage of our approach, the Android application takes advantage of edge computing capabilities to detect cars and determine their positions within the video images. We employ a TensorFlow Lite model, like the one used in the previous implementation, to perform this initial car detection task on the smartphone. Once the cars are identified, the corresponding regions of interest (ROIs) are extracted from the video frames and preprocessed to enhance their quality and suitability for subsequent recognition.

The preprocessed and cropped car images are then sent from the Android application to the fog server. The fog server application handles the second stage of the car model recognition process. It employs a trained TensorFlow model, specifically designed for the

Stanford Cars Dataset, to identify the model of each car in the received images and sends the results back to the Android application (Figure6). The fog server's superior computational resources and processing power allow it to perform more intensive tasks, such as detailed car type classification. To enable communication between the Android appli-

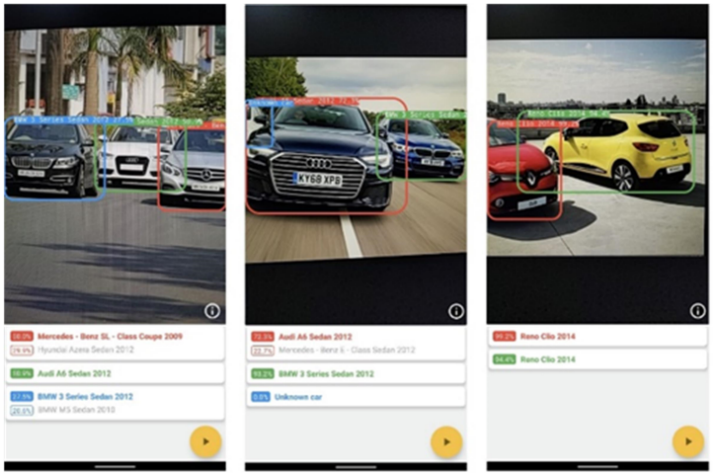

Fig. 6. Car model detection in the Android application

cation and the fog server, we use WebSockets. This approach facilitates real-time, bidirectional communication, allowing for the efficient transfer of preprocessed car images from the smartphone to the server and the return of recognition results to the mobile application. By utilizing WebSockets, we ensure a seamless and responsive user experience throughout the car model recognition process.

By leveraging both edge and fog computing resources, our two-stage approach optimizes the distribution of computational tasks. The edge computing performed on the smartphone efficiently detects cars and extracts relevant regions of interest (ROIs), which reduces the volume of data sent to the fog server. This strategy minimizes bandwidth usage and latency. The more resource-intensive task of car model recognition is offloaded to the fog server, taking advantage of its greater computational capabilities and trained model. This approach maximizes the utilization of computing resources and enhances the overall performance and accuracy of car model recognition in our system. The source code for traffic object recognition server is available at GitHub link https://github.com/drstojanovic/trafficAssistantServer while its docker image can be pulled from https://hub.docker.com/r/stefan2708/ta$_server$.

3.3. Vehicle detection on microcontrollers and single-board computers

The third use case focuses on vehicle detection on low-resource edge devices, such as microcontrollers and single-board computers. The goal is to detect and recognize cars in video streams captured by integrated cameras. This involves implementing various object detection model architectures and optimizing them for execution on devices like the Arduino Nano 33 BLE Sense and Raspberry Pi 4. Part of the model training and evaluation

was conducted using the Edge Impulse platform [7], an online MLOps platform that supports the training, testing, and deployment of ML/DL models across a wide range of edge devices. Additionally, transfer learning was applied using TensorFlow Lite Model Maker. The training dataset, consisting of 499 images with a total of 4,281 vehicles, was sourced from Kaggle. In addition to collecting data directly from the edge devices, datasets were also uploaded to the Edge Impulse platform in YOLO txt format. In this format, each image has an associated .txt file listing the detected objects, where each line in the file represents a single object, containing its class and the normalized coordinates of its bounding box.

The initial implementation of vehicle detection was performed on the Arduino Nano 33 BLE Sense, a resource-constrained edge device. Given the limited capabilities of this development board, the trained models operate on smaller image dimensions. However, increasing image size leads to longer inference times and larger model sizes. Unfortunately, due to the dataset containing small objects, models trained on low-resolution images yielded poor results. For the Arduino Nano application, two models, FOMO MobileNetV2 0.1 and FOMO MobileNetV2 0.35 were trained. The values 0.1 and 0.35 represent the alpha parameters in the MobileNetV2 architecture, indicating the network's width by scaling the number of channels in each layer. These hyperparameters help balance model size, computational efficiency, and accuracy. FOMO models were trained for various numbers of epochs using both RGB and Grayscale images, with various hyperparameter configurations, to find the optimal trade-off between performance and accuracy. The Edge Impulse platform was used to generate binary files for model inference on the Arduino device. Inference can be initiated with the command *edge-impulse-run-impulse*. Object detection results from the OV7675 camera attached to the Arduino Nano are displayed in the browser, as shown in Figure 7. The Raspberry Pi 4 Model B (RPi) offers

Fig. 7. Object detection from Arduino camera shown in browser

greater computing resources, enabling the training of a broader range of models. Beyond the FOMO algorithms, the following models were also trained using the Edge Impulse platform:

1. MobileNetV2 SSD FPN-Lite: This model was pre-trained on the COCO 2017 dataset with images of size 320x320. It consists of three parts:
   - Basic network (MobileNetV2): Provides high-level features for classification or detection. By removing the fully connected and softmax layers and adding a detection network, the model can determine object locations in the image.
   - Detection network (Single Shot Detector, SSD): Detects multiple objects in an image using a single convolutional network. SSD models are faster and more efficient as they simultaneously predict object classes and regions containing objects.
   - Feature Pyramid Network (FPN): Utilizes an input image of a single size to generate feature maps for different sizes, facilitating the detection of objects of various sizes.
2. YOLOv5: This model, part of the You Only Look Once (YOLO) family, performs object detection in a single pass. Introduced in 2020, YOLOv5 incorporates the EfficientDet architecture, built on EfficientNet, to optimize both resource usage and accuracy. Unlike its predecessor, YOLOv5 abandons anchor-based detection, instead relying on a convolutional layer to predict bounding box coordinates directly.

YOLOv5 is used in the transfer learning process, leveraging prior training on a larger dataset. This enables the model to learn from a broader data set and improve its generalization capabilities. During the training of the YOLOv5 model on Edge Impulse, the model size can be selected: XS, S, M, and L. Due to resource limitations, the smallest model with 1.9M parameters, sized at 3.78 MB, was chosen. Models from Edge Impulse are downloaded in the EIM (Edge Impulse Model) format. EIM files are binary files for Linux and macOS that encapsulate the complete impulse built on the Edge Impulse platform, including signal processing, model, and inference blocks. EIM files are architecture-specific and allow direct inference execution on the RPi device.

Inference on the RPi device is performed on images generated using a camera connected to the Arduino development board. Frames are captured on the Arduino and transmitted to the RPi device using serial communication. The OV7675 camera records images sized at 320x240 in RGB565 format. Within the Python script on the RPi device, bytes are read from the serial port, and conversion from RGB565 to RGB888 format is carried out. The converted image is passed to the `run_inference` function, which performs preprocessing and inference (Figure 8). TensorFlow Lite Model Maker [1] is a library designed for training TensorFlowLite models with custom datasets. It leverages transfer learning to minimize the amount of training data required and reduce training time. For object detection, the library offers five versions of EfficientDet-Lite models, each differing in memory usage, detection latency, and mean Average Precision (mAP). These models are deployed on the RPi for real-time vehicle detection. In this setup, frames transmitted from the Arduino Nano's camera are used for detection. However, the inference process differs here, as it utilizes the TensorFlowLite Interpreter, which requires a tensor as input. This necessitates preprocessing the image and generating a tensor with the correct shape and data type. Finally, the function completes the process by drawing bounding boxes and saving the annotated image. The source code of the Arduino Nano and RPi applications are available at https://github.com/drstojanovic/object-detection-rpi.

[1] https://www.tensorflow.org/lite/models/modify/model_maker

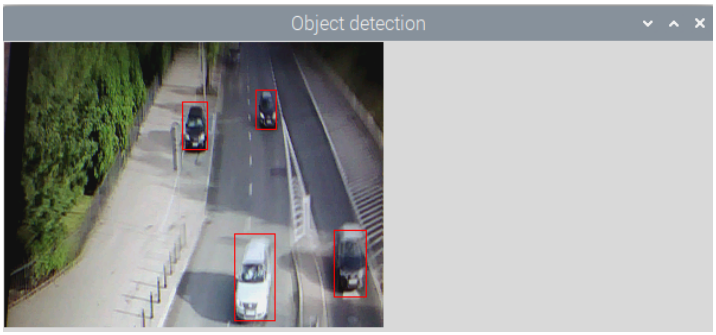

Fig. 8. Detection of vehicles using DL models deployed on RPi

4. Resource-Aware Experimental Evaluation

This section presents an experimental evaluation of previously described use cases and the corresponding methods for distributing DNNs in traffic object detection and recognition. In the first use case, we assess two solutions: one executed entirely within the Android application using a TensorFlow Lite model, and the other performed on the fog server. In the third use case, we evaluate solutions implemented on Arduino Nano and Raspberry Pi (RPi) devices. To measure performance, accuracy, and resource consumption (CPU, memory, and energy), we conduct experiments using various video data parameters and configurations. The objective is to explore the trade-offs between different approaches across the edge-fog computing continuum and analyze their behavior under varying conditions. For evaluation, we use representative datasets that include a variety of traffic scenarios and object types, ensuring coverage of diverse lighting conditions, weather patterns, and traffic densities.

4.1. Smartphone and Server implementation

To evaluate the Android application solution, we measure its performance and accuracy directly on the smartphone. The computational requirements, including CPU usage, memory consumption, and energy consumption, are analyzed using the Android Profiler tool. Furthermore, we assess the accuracy of traffic object recognition by comparing the application's outputs with ground truth annotations from the datasets.

Similarly, for the fog server solution, we assess its performance, accuracy, and resource consumption. The server's computational requirements, such as CPU usage, memory utilization, and energy consumption, are analyzed. Additionally, the recognition results from the fog server are compared with ground truth annotations to evaluate the solution's accuracy. A timeline diagram illustrating CPU and memory consumption, along with energy usage for traffic object recognition performed on the Android smartphone and the fog server, is presented in Figure 9. The maximum CPU utilization during local detection on the Android smartphone reached 33%, while server-based detection peaked at 14%. The maximum RAM usage during local detection was 316 MB, compared to 240 MB for server-based detection. The TensorFlow model size is 11.2 MB; however, with model quantization, it can be reduced to 3.2 MB.

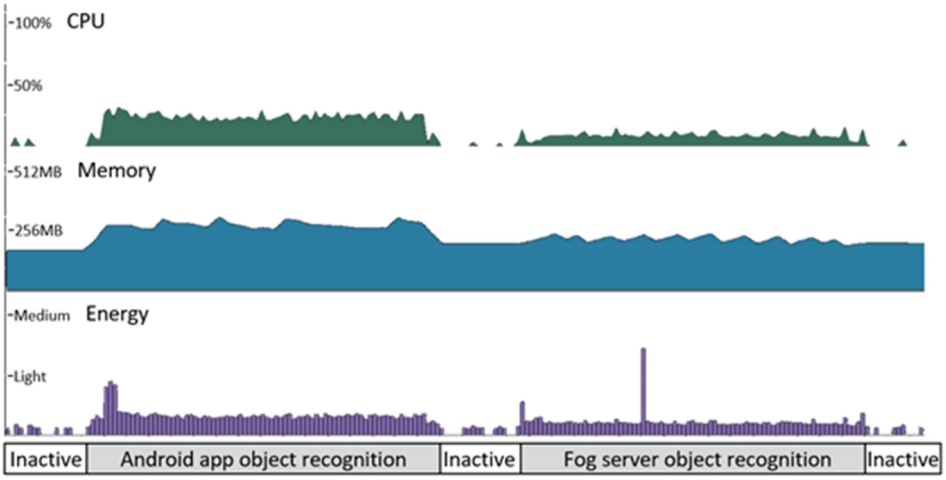

Fig. 9. CPU, memory, and energy usage for edge (smartphone) and edge server solutions

During the experimental evaluation, we varied the video data parameters and configurations to analyze the performance of both solutions under different conditions. This involved adjusting factors such as video resolution, frame rate, lighting conditions, and traffic densities. By conducting experiments across a range of scenarios, we aim to provide a thorough assessment of each solution's performance and resource utilization. The experimental evaluation is carried out using appropriate benchmarking tools and metrics to ensure reliable and meaningful results. We measure the execution time, resource utilization, and accuracy of both solutions across various datasets and configurations. The collected data is analyzed and compared to identify the strengths and weaknesses of each approach. For all experiments, a Google Pixel 4 phone was used, while the server application ran on a Lenovo Legion laptop (CPU: i5-9300H, RAM: 16.0 GB, GPU: NVIDIA GeForce GTX 1650). The server is configured on a local network to facilitate access by the mobile application.

The first experiment aimed to assess the accuracy of recognition with varying image resolutions and object sizes. The phone's camera was directed at a computer monitor displaying an image of a car that consistently shrank in size. Testing was conducted for both operational modes at each of the four available resolutions in the application settings. The image quality was set to the maximum (100 %, with no compression). The reliability of detection, expressed as percentages, is presented in Table 1. This experiment showed that the reliability values for detection, and thus the overall detection quality, are quite comparable for both edge detection and server-based detection. A slight advantage was noted for edge detection at lower image resolutions, likely because of the extra image processing involved in sending the image to the server (such as encoding and decoding in Base64 format). One potential solution to improve performance is to use a more advanced image transport method, like implementing one of the transfer protocols specifically designed for image handling. We also assessed how performance (execution speed) depends on image resolution and image quality, with values expressed in milliseconds (ms). This experiment aimed to evaluate the impact of image compression on detection speed and image size.

Table 1. Dependency of detection accuracy on image resolution and object size

Object size	Resolution							
	640 x 640		512 x 512		300 x 300		160 x 160	
	Edge	Fog	Edge	Fog	Edge	Fog	Edge	Fog
100%	98.9	97.8	98.7	98.7	98.9	98.3	90	84.3
80%	95.4	95	95.1	95.4	94.7	93.1	82.4	82.1
50%	89.2	90	87.4	88.7	89.9	87.9	83.4	80.5
30%	47.1	42.3	40.1	32.8	31.7	19.7	23.1	17.4
15%	12.3	12.7	10.7	12.1	10.2	2.1	7.4	1.8

The phone's camera was directed at a computer monitor displaying a consistent image of a car. Testing was conducted for both operating modes across all four available resolutions and image quality settings of 100 %, 70 %, 50 %, 30 %, and 20 % achieved through JPEG image compression.

Table 2 illustrates how detection speed varies with changes in image characteristics. The rows represent a decrease in image quality, while the columns show a reduction in resolution. The values are presented in milliseconds, and it is evident that local detection operates at a significantly higher speed compared to server-based detection.

Table 2. Dependency of performance (speed of execution) on image resolution and image quality (in ms)

Quality	Resolution							
	640 x 640		512 x 512		300 x 300		160 x 160	
	Edge	Fog	Edge	Fog	Edge	Fog	Edge	Fog
100%	61.4	153.2	58	137	55.9	103.2	45.8	83.9
70%	56.5	112.2	50.4	99.8	43.9	87.7	43	76.9
50%	55.5	108.2	49.8	98.9	43.6	86.7	42.9	76.4
30%	55.3	103.8	49.1	93.1	43.4	84.7	43.4	75
20%	54.4	101.8	48.9	90.2	43.9	84.1	42.8	74.9

Regarding local detection, the difference in detection speed between settings A (640x640 resolution and 100 % quality) and settings B (160x160 resolution and 20 % quality) amounts to 18.6 milliseconds per frame. Although the difference is difficult to measure precisely, it is evident that the detection quality for smaller objects is noticeably lower with settings B.

When it comes to server-based detection, internet connection speed becomes a significant factor. Since a GPU was utilized, the model execution itself was short (45ms), with most of the time being spent on image transportation. During the testing, an internet speed of 55.8 Mbps for download and 7.7 Mbps for upload was used. In the case of edge detection, the byte size of the image does not have as much influence as it does for server-based detection, as each image is transported to the server, and any reduction in image size contributes to increased detection speed. The difference in detection speed between settings A (640x640 resolution and 100 % quality) and settings B (160x160 resolution

and 20 % quality) amounts to 78.3 milliseconds per frame. A significant degradation in detection quality on the server compared to edge detection was observed when decreasing the image resolution.

By assessing the performance, accuracy, and resource consumption of both the Android application and the fog server solutions, we aim to shed light on the trade-offs associated with each approach. This experimental evaluation will enhance our understanding of how these distribution methods perform in traffic object recognition tasks. Ultimately, this analysis will aid in choosing the most suitable solution based on specific requirements, including resource availability, real-time performance, and accuracy.

4.2. Arduino Nano and RPi Solutions for Object Detection

Testing and evaluating vehicle detection models involved comparing key parameters essential for implementing AI at the edge. Resource utilization metrics, such as RAM and flash memory usage, are particularly important for edge devices. Additionally, the time required for object detection is critical for real-time decision-making scenarios. A comparison of model accuracy was also conducted. The objective is to compare models developed using the Edge Impulse platform and TensorFlow Lite tools, exploring various combinations of preprocessing and hyperparameters. Furthermore, this comparison encompasses devices with differing resources, specifically the Arduino Nano and RPi.

The comparison between the FOMO 0.1 and FOMO 0.35 models on the Arduino Nano has been conducted. All results are presented for the quantized versions of the models, utilizing integer 8-bit values. The EON model format was chosen due to its lower resource overhead compared to TFLite models. Figure 10 illustrates the comparison of these models across different image sizes, focusing on model accuracy and inference time. The diagrams indicate that the accuracy of the FOMO 0.1 and FOMO 0.35 models is comparable across all image sizes. However, FOMO 0.1 shows significantly better performance for smaller dimensions, such as 64x64, whereas the advantage shifts toward the FOMO 0.35 algorithm for larger dimensions. Inference time increases with image size, reaching 3 seconds for dimensions of 160x160. Furthermore, the inference time of the FOMO 0.35 model diverges considerably from that of the FOMO 0.1 model as the dimensions increase.

Figure 11 illustrates the memory utilization for the same models. Although flash memory usage remains consistent across various image dimensions, maximum RAM utilization rises sharply as the dimensions increase. The diagram also shows the available RAM on the Arduino development board; models that exceed this limit cannot be executed on the device.

The evaluation of the models executed on the Raspberry Pi includes those generated on Edge Impulse as well as models created using TensorFlow Model Maker. Figure 12 illustrates the changes in accuracy and inference time for different models with varying image sizes. The YOLOv5 model demonstrates superior accuracy in all scenarios, except for the smallest image size. However, in terms of object detection speed, the YOLOv5 algorithm requires up to seven times longer to make decisions compared to the FOMO algorithms.

The resource utilization diagram (Figure 13) reveals that the maximum RAM usage with FOMO algorithms exceeds that of the YOLO algorithm at higher dimensions. Flash

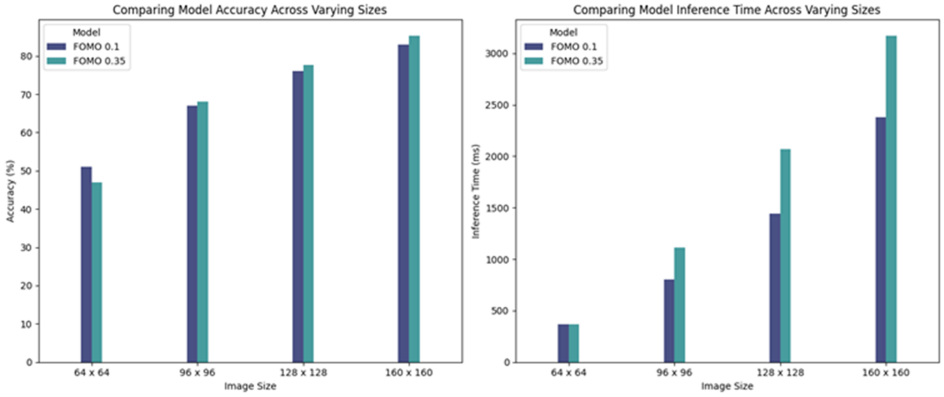

Fig. 10. Comparison of NN models based on accuracy and inference time.

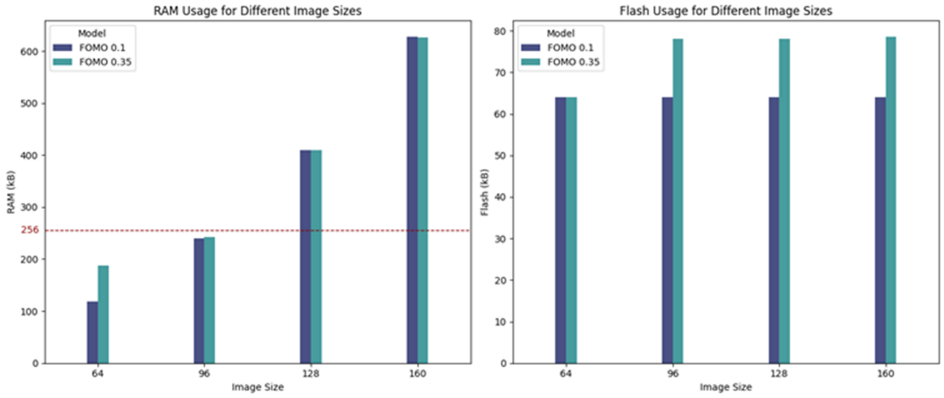

Fig. 11. Comparison of NN models based on memory utilization

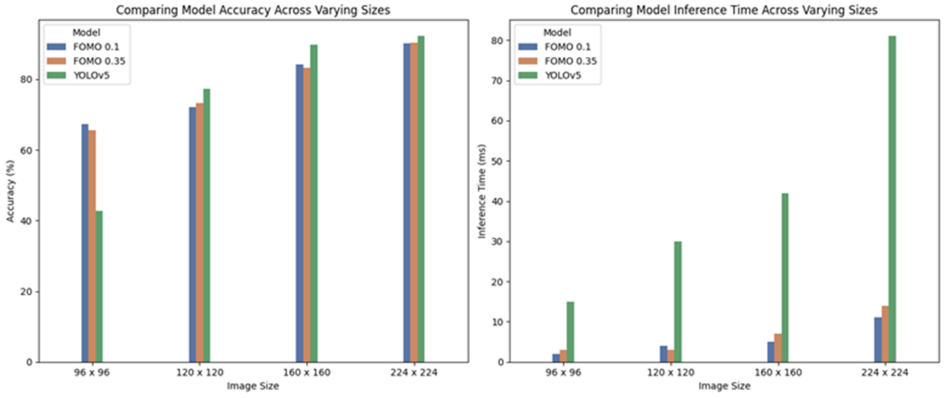

Fig. 12. Comparison of NN models based on accuracy and inference time on RPi

memory utilization remains consistent across all dimensions. Notably, the FOMO algorithms require approximately 70 KB, while the YOLOv5 model occupies 1.8 MB. On

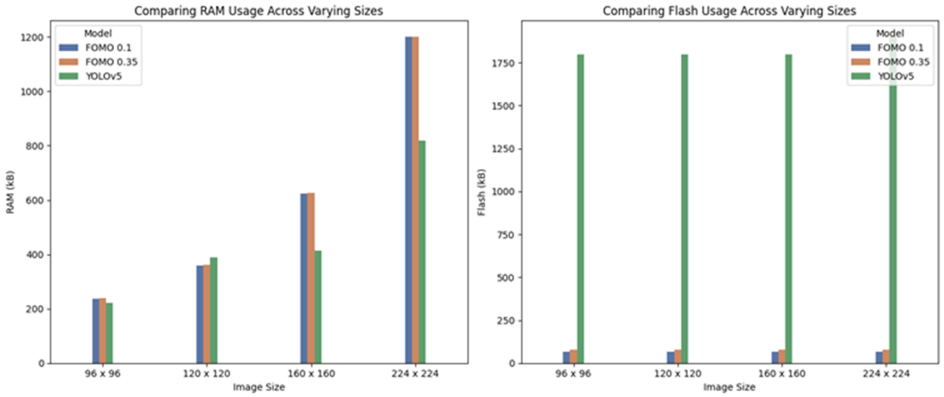

Fig. 13. Comparison of NN models based on memory utilization on RPi

Edge Impulse, several model optimizations are available. Figure 14 illustrates the differences between EON models with float32 values and their quantized versions using int8 values. Notably, quantization has a substantial effect on the size of the MobileNetV2 SSD and YOLOv5 models, as well as on inference time. The numerical comparison of the

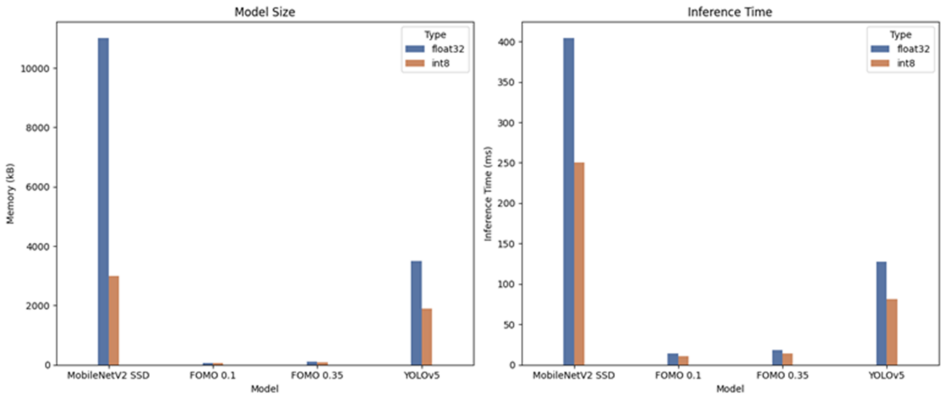

Fig. 14. Comparison between float32 and int8 NN model versions

models' size and inference time values is given in Table 3. The comparison of models compiled using EON and TensorFlow Lite compilers is shown in Figure 15, and presented in Table 4). The objective of the EON compiler is to reduce resource overhead, while maintaining unchanged accuracy and inference time. The values are presented for

Table 3. The NN model size and inference time values

	Model size (float32)	Model size (int8)	Inference time (float32)	Inference time (int8)
MobileNetV2 SSD	11 MB	3 MB	404 ms	249 ms
FOMO 0.1	66.9 KB	64.4 KB	14 ms	11 ms
FOMO 0.35	102.7 KB	78.5 KB	18 ms	14 ms
YOLOv5	3.5 MB	1.9 MB	128 ms	81ms

the int8 versions of the models. Resource optimization using the EON compiler is not supported for YOLOv5.

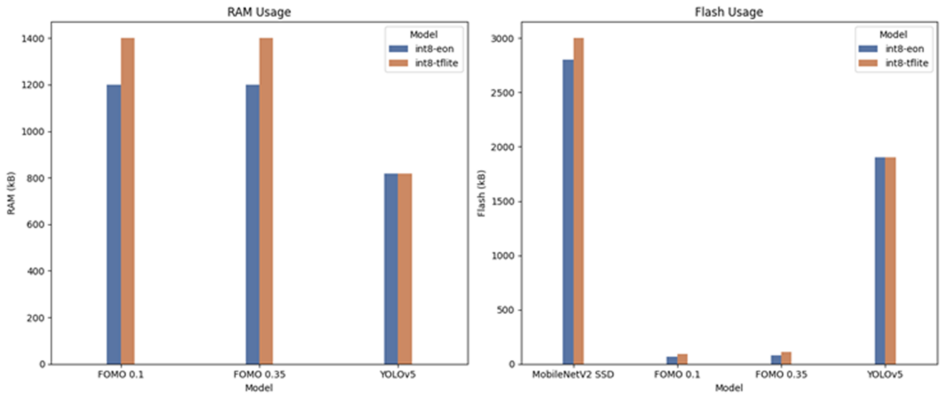

Fig. 15. Comparison of EON and TensorFlowLite models

Table 4. Review of memory utilization for EON and TensorFlow Lite models

	RAM (EON)	RAM (TensorFlowLite)	Flash (EON)	RAM (TensorFlowLite)
MobileNetV2 SSD	/	/	2.8 MB	3 MB
FOMO 0.1	1.2 MB	1.4 MB	64.4 KB	94.4 KB
FOMO 0.35	1.2 MB	1.4 MB	110.5 KB	78.5 KB
YOLOv5	817.5 KB	817.5 KB	1.9 MB	1.9 MB

TensorFlow Model Maker enables training of five versions of EfficientDet models (1-5). The results of the trained EfficientDet2 model are presented in Table 5. Due to the potential loss of model accuracy resulting from optimization, an evaluation of the TensorFlow Lite model is provided using mAP (mean Average Precision) metric for evaluation. The analysis of the results concludes that the differences in accuracy between TensorFlow and TensorFlow Lite models are nearly indistinguishable. The TensorFlow model's precision is 45.35 %. At a 50 % overlap threshold, the model's precision is 81.3 %, while for a 75 % overlap threshold, the value is nearly halved. Detection precision for differ-

Table 5. Evaluation of the TensorFlowLite model

	TensorFlow model	TensorFlow Lite model
AP	0.4535	0.4426
AP50	0.81	0.8
AP75	0.48	0.46
APs	0.003	0.0027
APm	0.35	0.3
APl	0.66	0.65
ARmax1	0.08	0.08
ARmax10	0.5	0.5
ARmax100	0.58	0.54
ARs	0.06	0.045
ARm	0.55	0.49
ARl	0.75	0.71
AP_car/	0.45	0.44

ent sizes (APs – small, APm – medium, APl – large) indicates that the model performs well with larger objects. Additionally, the model exhibits better accuracy when there are multiple objects in the image (ARmax10, ARmax100). There is room for improvement in detecting small objects and under very strict overlap criteria.

EfficientDet 0 and 2 versions were deployed on the RPi device, and a comparison of inference time and model size is given in Figure 16. Memory analysis during the ob-

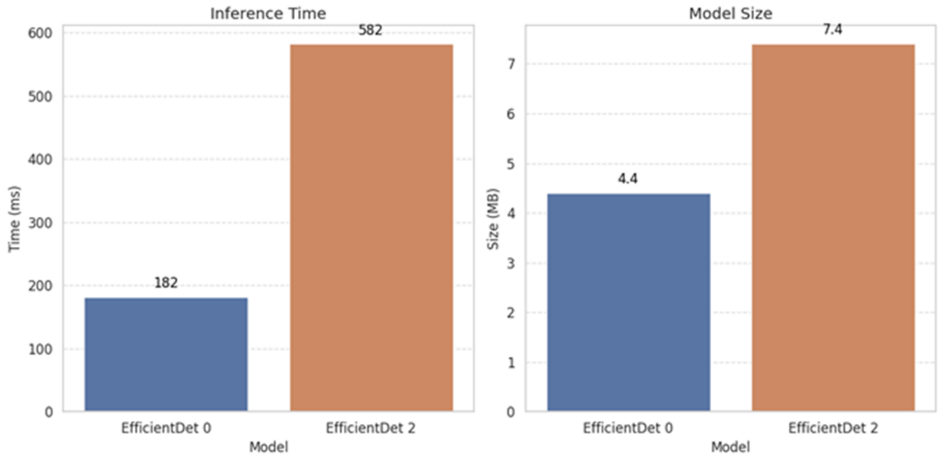

Fig. 16. Comparison of EfficientDet 0 and 2

ject detection process was performed and the results are given in Table 6. Both models require a similar amount of memory, with only minor differences in usage. EfficientDet 0 consumes slightly fewer resources than EfficientDet 2; however, both models function effectively within the available system resources. Through comprehensive testing

Table 6. Memory utilization of EfficientDet models (in KB)

	total	used	free	shared	buff/cache	available
EfficientDet 0	3794	599	1937	75	1256	3043
EfficientDet 2	3794	613	1915	80	1265	3025

and evaluation of NN models created using various tools, significant advancements in technologies for deploying models on edge devices have become apparent. By leveraging advanced optimization techniques, it is feasible to execute complex model architectures on resource-constrained devices. For example, models trained on Edge Impulse occupy considerably less space and consume fewer resources compared to TensorFlow Lite models. Conversely, the strength of the TensorFlow Lite framework lies in its broader selection of model architectures and the ability to utilize established models for object detection.

5. Conclusions

The proposed approach and experimental evaluations provide valuable insights into the challenges and opportunities of deploying DNNs for traffic object detection and recognition across the edge-fog computing continuum. One of the key takeaways is that distributing tasks between edge devices and fog servers offers a balanced trade-off between performance, resource consumption, and accuracy. By leveraging edge devices like smartphones for object detection, we can reduce data transmission and latency. We found that current smartphones perform well in both accuracy and speed, with only 33% CPU utilization during inference. This raises the question of whether offloading detection tasks to the fog is necessary, given that the communication overhead may outweigh any potential performance benefits. The results suggest that, in many cases, performing both detection and recognition on the mobile device itself is a more efficient strategy, especially for real-time applications.

Meanwhile, the fog server, with its higher computational capacity, is well-suited well-suited for more complex tasks such as car model recognition, where detailed feature extraction and processing are required. This delegation not only reduces the computational burden on edge devices like smartphones or embedded platforms but also enables the deployment of heavier models that might otherwise exceed the capabilities of smaller devices. For instance, running computationally intensive architectures such as YOLOv5 or EfficientDet on the fog server ensures that these models can operate without compromising performance or requiring extensive optimization, as would be necessary on edge devices. This division of labor improves the system's overall efficiency by allowing lightweight tasks, such as object detection, to occur locally on edge devices while offloading more demanding tasks to the fog server. Consequently, the combination of fast local inference with sophisticated fog-based recognition creates a robust pipeline that balances speed and accuracy across different layers of the edge-fog continuum.

While the smartphone proved capable, the resource constrained embedded devices, such as RPi or Arduino Nano, could still benefit from fog or cloud offloading, especially for complex models. However, our focus in this study was on evaluating lightweight architectures directly on these devices. Future research could explore hybrid approaches, where RPi or Arduino devices collaborate with fog or cloud systems for more demanding tasks,

striking a balance between local processing and offloading. We found that the quantized and minimized model deployed on Arduino Nano, RPI device, and Android smartphones achieved reasonable performance with efficient resource usage. The use of quantization and model optimization techniques, especially through TensorFlow Lite and Edge Impulse, proved crucial for deploying DNNs on resource-constrained devices like the Arduino Nano and RPi. Our results show that while Edge Impulse models consume fewer resources, TensorFlow Lite offers greater flexibility through access to a wider variety of model architectures. The experiments also highlighted the trade-offs between model size, inference time, and accuracy, emphasizing the importance of fine-tuning hyperparameters based on the specific hardware and use case.

The ability to achieve near real-time detection using optimized models demonstrates that modern AI technologies can effectively run on low-power edge devices. Our findings suggest that combining edge and fog computing provides a scalable and efficient way to implement more demanding AI solutions, such as specific object recognition on video stream. The experiments also revealed that model size and computational overhead must be carefully managed, particularly on microcontrollers, where even slight increases in image resolution or model complexity can lead to significant resource constraints.

However, there are still several avenues for future research in this area. Some potential directions include:

- Exploration of federated learning techniques tailored for object detection and recognition tasks at the edge for aggregating model updates from distributed edge nodes efficiently while accounting for resource constraints.
- Investigation of online continual learning techniques for adaptive object detection and recognition at the edge that enable edge devices to incrementally learn from streaming data while retaining knowledge learned from previous tasks.
- Research resource-efficient model adaptation techniques in edge-fog environments that dynamically adjust model complexity and capacity based on available computational resources and streaming data characteristics.
-

By further exploring these research directions, we can continue to advance the field of DNN model distribution and slicing across the edge-fog-cloud computing continuum, enabling resource-aware and efficient object detection, classification and recognition systems for various real-world applications.

6. Acknowledgements

Research presented in this paper is supported by the Ministry of Science and Technological Development, Republic of Serbia, 451-03-65/2024-03/200102, Erasmus+ Project FAAI: The Future is in Applied AI (WP4 - Artificial Intelligence framework for training in HE) - 2022-1-PL01-KA220-HED-000088359 and COST Action CERCIRAS - Connecting Education and Research Communities for an Innovative Resource Aware Society - CA19135.

References

1. Akhtar, A., Ahmed, R., Yousaf, M.H., Velastin, S.A.: Real-time motorbike detection: Ai on the edge perspective. Mathematics 12(7) (2024), https://www.mdpi.com/2227-7390/12/7/1103
2. Bian, J., Arafat, A.A., Xiong, H., Li, J., Li, L., Chen, H., Wang, J., Dou, D., Guo, Z.: Machine learning in real-time internet of things (iot) systems: A survey. IEEE Internet of Things Journal 9(11), 8364–8386 (2022)
3. Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., Villas, L., DaSilva, L., Lee, C., Rana, O.: The internet of things, fog and cloud continuum: Integration and challenges. Internet of Things 3-4, 134–155 (2018)
4. Cho, E., Yoon, J., Baek, D., Lee, D., Bae, D.: Dnn model deployment on distributed edges. In: Bakaev, M., Ko, I.Y., Mrissa, M., Pautasso, C., Srivastava, A. (eds.) ICWE 2021 Workshops. Communications in Computer and Information Science, vol. 1508. Springer, Cham (2021)
5. Dharani, A., Kumar, S.A., Patil, P.N.: Object detection at edge using tinyml models. SN Computer Science 5(1), 11 (11 2023), https://doi.org/10.1007/s42979-023-02304-z
6. Hanhirova, J., Kämäräinen, T., Seppälä, S., Siekkinen, M., Hirvisalo, V., Ylä-Jääski, A.: Latency and throughput characterization of convolutional neural networks for mobile computer vision. In: Proceedings of the 9th ACM Multimedia Systems Conference (MMSys '18). pp. 204–215. New York, NY, USA (2018)
7. Hymel, S., Banbury, C.R., Situnayake, D., Elium, A., Ward, C., Kelcey, M., Baaijens, M., Majchrzycki, M., Plunkett, J., Tischler, D., Grande, A., Moreau, L., Maslov, D., Beavis, A., Jongboom, J., Reddi, V.J.: Edge impulse: An mlops platform for tiny machine learning. ArXiv abs/2212.03332 (2022), https://api.semanticscholar.org/CorpusID:254366602
8. Iftikhar, S., Gill, S.S., Song, C., Xu, M., Aslanpour, M.S., Toosi, A.N., Du, J., Wu, H., Ghosh, S., Chowdhury, D., Golec, M., Kumar, M., Abdelmoniem, A.M., Cuadrado, F., Varghese, B., Rana, O., Dustdar, S., Uhlig, S.: Ai-based fog and edge computing: A systematic review, taxonomy and future directions. Internet of Things 21, 100674 (2023), https://www.sciencedirect.com/science/article/pii/S254266052200155X
9. Kallimani, R., Pai, K., Raghuwanshi, P., Iyer, S., López, O.L.A.: Tinyml: Tools, applications, challenges, and future research directions. Multimedia Tools and Applications 83(10), 29015–29045 (2024), https://doi.org/10.1007/s11042-023-16740-9, dOI: 10.1007/s11042-023-16740-9
10. Lee, J.K., Varghese, B., Woods, R., Vandierendonck, H.: Tod: Transprecise object detection to maximize real-time accuracy on the edge. In: Proceeding of the 5th EEE 5th International Conference on Fog and Edge Computing. pp. 53–60 (2021)
11. Lin, C.Y., Wang, T.C., Chen, K.C., Lee, B.Y., Kuo, J.J.: Distributed deep neural network deployment for smart devices from the edge to the cloud. In: Proceedings of the ACM MobiHoc Workshop on Pervasive Systems in the IoT Era. p. 43–48. PERSIST-IoT '19, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3331052.3332477
12. Lockhart, L., Harvey, P., Imai, P., Willis, P., Varghese, B.: Scission: Performance-driven and context-aware cloud-edge distribution of deep neural networks. In: Proceedings of the IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). pp. 257–268. Leicester, United Kingdom (2020)
13. Marszałek, M., Schmid, C.: Inria annotations for graz-02 dataset (2007), https://lear.inrialpes.fr/people/marszalek/data/ig02/, accessed: 2024-10-1
14. McNamee, F., Dustdar, S., Kilpatrick, P., Shi, W., Spence, I., Varghese, B.: The case for adaptive deep neural networks in edge computing. In: Proceedings of the IEEE 14th International Conference on Cloud Computing (CLOUD). pp. 43–52 (2021)

15. Parthasarathy, A., Krishnamachari, B.: Defer: Distributed edge inference for deep neural networks. In: Proceedings of the 14th International Conference on COMmunication Systems & NETworkS (COMSNETS). pp. 749–753 (2022)
16. Ren, W., Qu, Y., Dong, C., Jing, Y., Sun, H., Wu, Q., Guo, S.: A survey on collaborative dnn inference for edge intelligence. Machine Intelligence Research 20, 370–395 (2023)
17. Shuvo, M.M.H., Islam, S.K., Cheng, J., Morshed, B.I.: Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. Proceedings of the IEEE 111(1), 42–91 (2023)
18. Singh, R., Gill, S.S.: Edge ai: A survey. Internet of Things and Cyber-Physical Systems 3, 71–92 (2023), https://www.sciencedirect.com/science/article/pii/S2667345223000196
19. Stojanovic, D., Sentic, S., Stojanovic, N.: Towards resource-efficient dnn deployment for traffic object recognition: From edge to fog. In: Zeinalipour, D., Blanco Heras, D., Pallis, G., Herodotou, H., Trihinas, D., Balouek, D., Diehl, P., Cojean, T., Fürlinger, K., Kirkeby, M.H., Nardelli, M., Di Sanzo, P. (eds.) Euro-Par 2023: Parallel Processing Workshops. pp. 30–39. Springer Nature Switzerland, Cham (2024)
20. Taheri, J., Dustdar, S., Zomaya, A., Deng, S.: Edge Intelligence: From Theory to Practice. Springer International Publishing (2023), https://books.google.es/books?id=cCV5zwEACAAJ
21. Teerapittayanon, S., McDanel, B., Kung, H.T.: Distributed deep neural networks over the cloud, the edge and end devices. In: Proceedings of the IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 328–339. Atlanta, GA, USA (2017)
22. Udacity: Udacity self-driving car dataset (2024), https://public.roboflow.com/object-detection/self-driving-car, accessed: 2024-10-1
23. Yong-Hah, R.: Bike-rider detector (2019), https://github.com/yonghah/bikerider-detector, accessed: 2024-10-1