The Journal of Korean Institute of Communications and Information Sciences '24-10 Vol.49 No.10
https://doi.org/10.7840/kics.2024.49.10.1447

Lightweighted Real-Time Object Detection on a Custom Edge Device

Md Javed Ahmed Shanto◆, Dong-Seong Kim*, Taesoo Jun°

ABSTRACT

With recent innovations in AI and software technology, on-device object detection has drawn significant attention. This technique enables real-time processing of visual data without the need for a connection to a distant server. However, deploying these models on resource-constrained edge devices presents several challenges. The primary obstacles stem from the limited processing power, memory, and storage capacity of these devices, as well as software issues. The current constraints make training artificial intelligence inefficient, as it requires substantial storage and computational power. Moreover, the development of devices based on ARM architecture demands the training and implementation of a customized model specifically designed for that edge device. This article discusses the development of a lightweight object recognition model that utilizes a TensorFlow Lite model and achieves a high accuracy rate of 94% on a custom edge device. This study also presents techniques for implementing this method using a custom file, demonstrates new performance metrics, and yields favorable results compared to existing benchmarks.

Key Words : Artificial Intelligence, Custom Edge Device, Edge Computing, Object Detection, Real-time Processing, TensorFlow Lite

Ⅰ. Introduction

Object detection is a crucial component of computer vision with applications spanning autonomous vehicles[1], traffic systems[2], medical technology[3], and robotics[4]. The significance of this technology lies in its ability to accurately recognize and classify various items present in images or videos[5]. Crucially, performing object detection on an edge device facilitates local execution, thus eliminating the need for an internet connection. This approach, tailored to a specific peripheral device, provides efficient processing capabilities, reduces latency, and addresses privacy concerns by avoiding data transmission over the internet. Real-time on-device object detection, as implemented in autonomous vehicles, enables rapid decision-making that enhances safety. Additionally, the local processing model effectively addresses cybersecurity concerns by ensuring that sensitive information remains confined to the device, thus improving both privacy and security standards[6].

Object detection is a task in computer vision that involves identifying and precisely localizing objects within digital images or video streams. The primary goal is to determine the presence and position of various items accurately, often within complex environ-

※ This work was partly supported by Innovative Human Resource Development for Local Intellectualization program through the Institute of Information Communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2024-RS-2020-II201612, 50%) and by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2024-RS-2024-00438430, 50%) supervised by the IITP.

◆ First Author : Kumoh National Institute of Technology, Dept. of IT Convergence Engineering, shantoa729@kumoh.ac.kr, 학생회원
° Corresponding Author : Kumoh National Institute of Technology, Dept. of Computer SW Engineering, taesoo.jun@gmail.com, 정회원
* Kumoh National Institute of Technology, Dept. of IT Convergence Engineeringr dskim@kumoh.ac.kr, 종신회원

논문번호 : 202405-085-0-SE, Received April 29, 2024; Revised June 21, 2024; Revised July 17, 2024; Accepted July 18, 2024

ments[7]. Object detection enhances efficiency, safety, and decision-making by enabling automated analysis and interpretation of visual inputs. Real-time object detection further extends these benefits by providing immediate insights, allowing for swift responses to dynamic environments. However, There are many obstacles[8] to overcome when implementing real-time object detection on specially custom-built edge devices, mainly related to hardware limitations and processing efficiency. Due to the restricted resources of edge devices, real-time processing requires high computational power and low latency, which are challenging to achieve. Composing custom code for these devices introduces a significant difficulty, as programmers have to manage power consumption and thermal concerns while optimizing algorithms to operate well on less capable hardware. Furthermore, because these devices are custom-built, it is frequently necessary to deal with various possibly incompatible hardware and software components, making it more challenging to guarantee reliable and consistent operation. These limits make deploying real-time object detection models on edge devices more difficult, which forces large trade-offs between accuracy, speed, and power consumption. To address these issues, it is crucial to optimize algorithms and models to perform effectively in resource-constrained settings. This optimization can be achieved through techniques such as model compression, optimization, quantization, and hardware acceleration.

The contributions of this study are summarized as follows:

1. This paper demonstrates the implementation of the TensorFlow Lite (TF-Lite) model on a custom-built edge device, offering a more realistic assessment of the model's performance on resource-constrained environments compared to traditional simulation work. This pioneering contribution moves beyond the limitations of simulation-based testing.
2. The study enhances TensorFlow Lite model performance on a custom edge device, providing insights into its behavior in limited-resource environments. It also introduces innovative custom code for comprehensive performance analysis, surpassing previous demonstrations and enhancing the model's resilience, making it more suitable for real-world edge applications.
3. The study improves the existing approaches by deploying a TensorFlow Lite model on an edge device and parameter optimization with comparative assessment. The optimized parameters and comparative evaluations show the proposed methodology outperforms existing approaches, paving the way for real-world edge application breakthroughs.

Ⅱ. Related Work

TinyML, which is an abbreviation for "tiny machine learning," is a trend that is currently being investigated by academics within the field of intelligence[9]. The term "lightweight model" refers to a model that is designed to simplify the deployment of machine learning[10] capabilities on devices that have minimal resources. Examples of such devices are mobile phones and microcontrollers. TinyML makes it possible for these devices to carry out a wide variety of tasks that are associated with artificial intelligence in a variety of fields, including the military, social, medical professions, supply chain operations, and other areas. Tf-lite is used for solving the TinyML deployment problem[11] developed by Google specifically for on-edge devices, addresses the resource limitations often associated with standard TensorFlow[12,13].

In spite of the fact that a great number of academics have made substantial use of TensorFlow Lite for projects on edge devices, none of them have yet adapted it for devices that are specifically designed to meet particular requirements.

According to [11], YOLOv4 was successfully implemented using TensorFlow Lite. Although the implementation was successful, the study did not employ performance evaluation criteria to emphasize its importance, nor did it specify the particular edge device used. In [14], TensorFlow Lite was successfully integrated into a Raspberry Pi 4, enabling real-time dog identification and corresponding notifications. Despite the functionality being implemented, the authors did not provide any performance statistics for comparison, such as accuracy or frames per second (FPS). Additionally, there was no mention of the latency involved in the real-time system throughout their discussion.

Utilizing TensorFlow Lite, which functions on mobile devices, [15] developed enhanced algorithms for executing matrix multiplications of critical kernels, including convolution and matrix multiplication, while optimizing the kernels of the CNN model. However, their research did not evaluate the precision of their work or the minimum FPS that their model could achieve on their system.

[16] have developed a garbage management system utilizing a Convolutional Neural Network (CNN) implemented on an Arduino microcontroller with TensorFlow Lite. The system includes a number of sensors, and the authors provided data on the accuracy and processing time for each type of recognized waste. However, their reported accuracy is lower compared to our study, and the processing time is significantly longer. A notable distinction is that this approach employs a custom edge device.

Ⅲ. Proposed Methodology

This article discusses the utilization of a recently constructed specialized edge device. The complete specifications, including the number of ports and other details of the custom edge device, are provided in Table 1. From the perspective outlined in Figure 1, a bird's-eye view of the device in operation is presented. Additionally, a 4K USB HD webcam module has been incorporated to facilitate the capture of real-time video.

This study consists of two main components, as detailed in Figure 2. The first phase involves developing the model, a critical step in the process. At this stage, it is crucial to construct a dataset tailored to the specific needs. Following this, we will proceed with preprocessing and model training. In this case, the tflite0 model was utilized, which was pre-trained on the COCO dataset by Google. After training, the model will be converted into a TensorFlow Lite format using the TensorFlow runtime.

Table 1. Custom edge device specification

Feature	Description
Architecture	AArch64
CPU Mode	32-bit and 64-bit
CPU Cores	6 (3 per socket, 2 sockets)
Threading	Single thread per core
CPU Base Clock Speed	500 MHz
CPU Boost Clock Speed	2208 MHz
Camera	4K USB webcam
Image Sensor	Sony IMX415 8MP CCD
Video Resolution	Up to 3840x2160 (UHD)
Video Frame Rate	30 FPS
Connectivity	1x HDMI 3x USB 1x Ethernet Wi-Fi (IEEE 802.11b/g/n/ac) Bluetooth

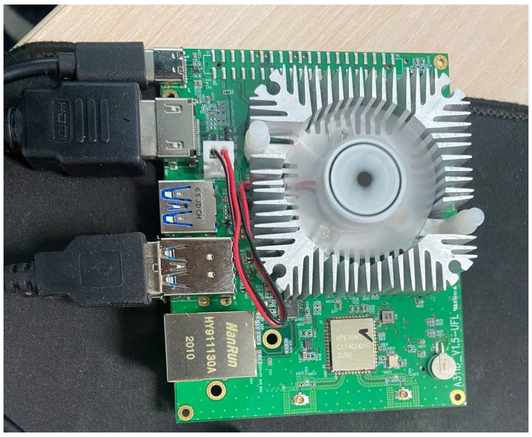
Fig. 1. Perspective top view of the custom edge device in the study

In the second phase, known as the deployment stage, our objective is to effectively implement the model on the edge device. Initially, we install all required libraries and camera modules. Subsequently, we adapt the model to align with the specifications of both the board and the model itself. Once we achieve satisfactory performance results through our custom code, we proceed with deploying the model on the edge device. If the results are insufficient, we en-

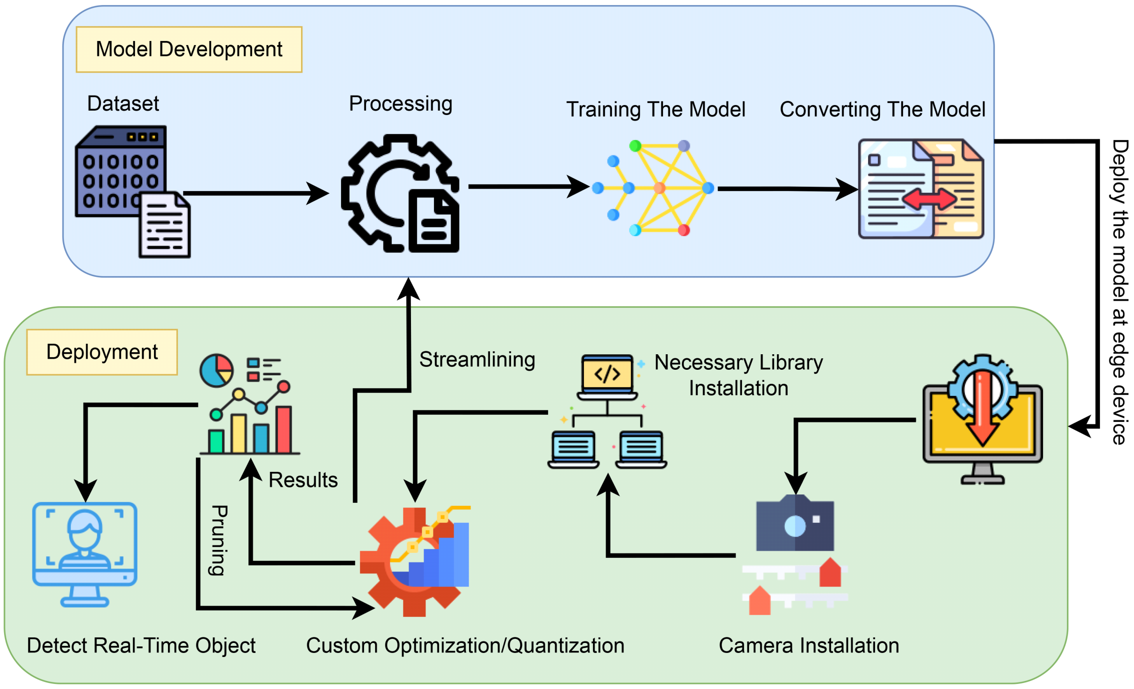

Fig. 2. Architechcural workflow of the proposed study

hance the performance through parameter optimization, such as adjusting the number of CPU threads, camera resolution, maximum detected items, and their thresholds based on performance metrics observed in the terminal. Should the model still underperform, we revert to the model development phase to modify processing steps, retrain the model, and iterate through the stages again. This iterative process allows us to develop a robust system that can be adjusted without the need for a complete rebuild.

By employing this approach, we can make regular modifications to the model as needed, ensuring that the proposed scheme remains both effective and efficient.

Ⅳ. Experimental Setup

Due to the novelty and customized nature of our device, it is essential to initially install the operating system. We have chosen Ubuntu 20.04 from Khadas for this purpose. To transfer the vim3-ubuntu-20.04-gnome-linux-4.9-fenix-1.3-221118-emmc, a customized version of Ubuntu developed by Khadas, we use the USB-burning tool. The first step of the installation process involves using an A-type port and a C-type port to establish a connection between the edge device and the host computer. After the operating system is successfully installed, we disconnect the power cable and then use an adapter to power the device. Following this, the Ubuntu operating system will begin its startup process. To run the TensorFlow Lite model on the edge device, acquiring the appropriate libraries is crucial. This task can be completed seamlessly once Ubuntu is up and running. Subsequently, we require tensorflow, tensorflow inference, python 3, and other essential libraries to execute the suggested object identification model on the edge device.

To determine if the TensorFlow Lite (tf-lite) model is compatible with the edge device developed by Google, this article examines the Convolutional Neural Network (CNN) model, specifically the lite-model_efficientdet_lite0_detection_metadata_1.tflite. This Google Edge device model has been trained using the COCO dataset, which includes 80 object categories and 330,000 images. The model operates with a default configuration of fifty epochs and a batch size of sixty-four, and each execution involves just one step. Although TPU is an option, it is not utilized in this instance. Moreover, the default verbosity level is set to 0, but we can adjust it according to our needs. After executing the update and upgrade commands on Ubuntu, we cloned one of the official Google scripts and made specific modifications to it. We emulated the code seen in TensorFlow Lite examples to achieve this. Subsequently, we installed Python 3 to run the code or make any necessary adjustments. In order to execute our code, we created a virtual environment, referred to as a venv in Python, to separate it from the main libraries and guarantee that all necessary libraries are contained within the venv. This prevents any interference with the dependencies of the main library.

The remaining steps are illustrated in Figure 3, which presents the flow diagram guiding the deployment of this method. This paper identifies two areas requiring optimization: support for TensorFlow Lite

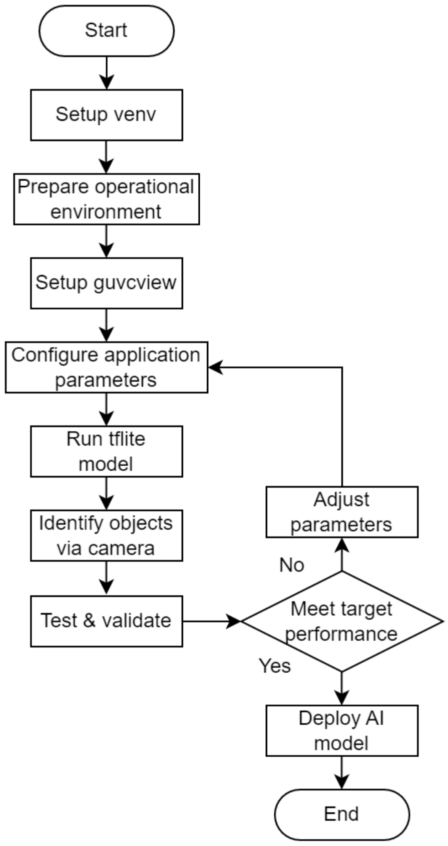

Fig. 3. Procedure for deploying the TensorFlow lite model in the edge device.

(tflite) and the camera port, which necessitate modifications specific to the device. For this particular case, we reverted the tflite-support file to version 0.4.3 and conducted multiple tests to identify the camera port correctly. We discovered that neither the default value of 1 nor any other values functioned correctly, leading us to set it to 0. Nevertheless, in the event that the camera undergoes a change, it will be necessary to modify the port settings once more. Additionally, we developed our own primary Python script to execute the tflite model, incorporating functionalities to measure performance metrics such as memory consumption and latency distribution. We may optimize the script to suit our requirements for assessing and adjusting the parameters of the model. The TensorFlow Lite model used in this study is named 'efficientdet_lite0.tflite'. Upon activation, the camera automatically opens and begins object detection. Analysis of the data highlights the exceptional nature of the work documented in this paper, surpassing other efforts. The system achieves an accuracy of 94.17%, operates at an average frame rate of 15 FPS, and maintains a system latency of just 37 milliseconds, with real-time results visible in Figure 4, which also proves computational efficiency.

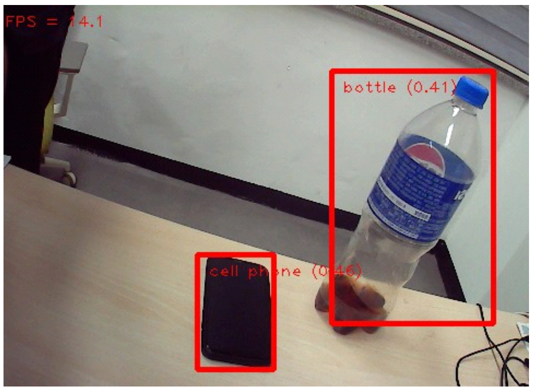

Fig. 4. Detection of the objects using the edge device

Ⅴ. Performance Evaluation

5.1 Parameter Optimization

By modifying parameters, the research emphasizes a significant enhancement in the system's performance and efficiency. The optimized key parameters include the number of CPU threads (num_threads), the maximum number of detection results (max_results), and the score threshold for detections (score_threshold)[18]. The num_threads parameter determines the number of concurrent CPU threads utilized, impacting the system's computational velocity. Augmenting this numerical value can enhance performance by capitalizing on multi-core processors, although it must be carefully weighed against potential rises in power consumption. The max_results option restricts the quantity of identified objects in each frame, thereby decreasing computing burden and enhancing real-time efficiency. The score_threshold parameter establishes the lowest degree of confidence required for detections to be valid, thus managing the trade-off between accuracy and false positive rates. Increasing the threshold levels decreases the occurrence of false positive results, whilst decreasing the threshold levels enhances the sensitivity.

The enable_edgetpu parameter and camera resolution (width and height) were left at their default settings to ensure compatibility and optimize performance. The Edge TPU is a specialized hardware device created to enhance the performance of executing machine learning models in edge computing environments, which this device does not have. The default camera resolution provides sufficient detail for object detection without overwhelming the machine. This research achieves optimal performance in real-time object detection on a custom edge device by adjusting the parameters, highlighting parameter optimization's importance in achieving a balance between computing efficiency, accuracy, and responsiveness.

5.2 Performance Metrics

This work evaluates performance using the following metrics, utilizing a custom-created Python file to provide a more versatile insight into the model and its performances:

5.2.1 Accuracy

Accuracy is determined by the proportion of correctly categorized objects among all detected objects. However, the model is limited to recognizing only those objects that are within the scope of the training image dataset on which it was trained, for this case, the COCO dataset. If the image displayed in front of the camera is not included in the training dataset, it may result in the incorrect identification of the object or the failure to recognize the object altogether.

5.2.2 Frames Per Second

FPS measures how quickly the model can process frames. A higher FPS value signifies shorter inference times and reduced delay. In this work, FPS is measured by counting the number of frames the model processes in one second during real-time operation. This is achieved by setting a counter while the model processes the video stream and averaging the total count over a designated test period.

5.2.3 Frame Processing Rate

FPR (Frame Processing Rate) calculates the proportion of frames processed incorrectly among all frames handled by the system. A lower FPR indicates more efficient and accurate frame processing, reflecting superior system performance. It is calculated by dividing the number of frames incorrectly processed by the total number of frames processed during the operation. In this work, FPR is measured by running the model on a dataset where accurate frame processing is crucial. Each frame' s processing outcome is meticulously logged, and the percentage of frames incorrectly processed is subsequently determined.

5.2.4 Inference Time

Inference time measures how long the model takes to process and make predictions on a single frame. A shorter inference time indicates faster processing and reduced latency, showcasing a more efficient model. It is measured by the amount of time the model requires to process one input before producing a prediction. To quantify this, the Python script in this work records the start and finish times for each frame or input during the model's inference call and then calculates the average of these times.

5.2.5 Latency Distribution

Latency distribution assesses how delays are spread across various frames. A skewed distribution may indicate performance inconsistencies, while a uniform latency distribution signifies consistent performance across all frames. This metric is evaluated by recording the inference times in real-time for multiple inputs and then plotting these times to visualize the distribution. This approach facilitates comprehension of the model's variability and enables the identification of the most prevalent inference times in this experiment.

5.2.6 Memory Usage

This metric measures the amount of memory the model consumes during inference or while the model is running. Lower memory usage signifies a more efficient model architecture and reduced computational overhead. To monitor the system's memory use, it is examined both before and during the model's operation. In this study, the memory_usage function from the Python resource library was utilized to track memory consumption while the video actively detected objects.

5.3 Performance Analysis

Based on the data presented in Table 2, it is evident that[11] lacks statistical data regarding accuracy, average FPS, or latency. Moreover, the authors do not specify the type of edge device used for object detection. In contrast, the paper[14] specifies the use of a Raspberry Pi 4 for dog identification but fails to include performance assessment measures such as device latency, FPS, accuracy, or precision.

Further investigation into another study[16] reveals its use of TensorFlow Lite for trash detection, specifically identifying materials such as glass, paper, metal, plastic, and even cupboards. This model reports an accuracy of 91.76%, which is lower than the accuracy achieved by the proposed work. Regrettably, the study omits details on FPS or inference time, although it notes an exceedingly long inference time of 358.95 milliseconds when using an Arduino, which is a significant delay.

The work[17] utilizes TensorFlow on a Raspberry Pi 4 to aid visually impaired individuals but does not provide statistics on the model' s accuracy or latency. However, they report successfully establishing a delay of fifteen milliseconds, achieving their objective.

In comparison, this article's findings demonstrate exceptional performance with an accuracy rate of 94.17%, impressively low latency of 0.057 seconds, and a high average FPS of 30, surpassing the results of the aforementioned studies.

The performance findings detailed in Table 2 have not been documented or evaluated in previously published works. This study conducted extensive research to enhance understanding of the model and the custom edge device, obtaining these significant results. When executed on the board, the model requires a total of 424,996 bytes, and the FPR is 3.4277, indicating high efficiency and accuracy.

The histogram in Figure 5 depicts latency distribution across several CPU threads. It demonstrates

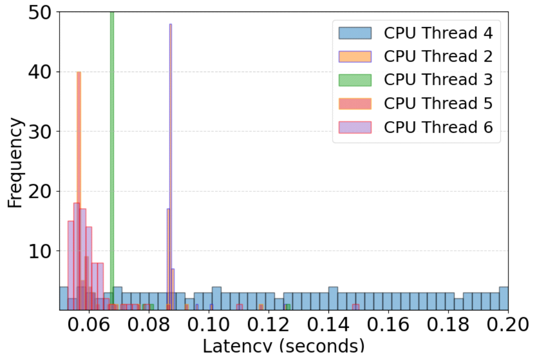

Fig. 5. Latency distribution over CPU threads.

Table 2. Comparative analysis of the proposed work with the existing models

Researches	Accuracy	Average FPS	Inference time (s)	Detected object	Implemented Device
[11]	Not specified	Not specified	Not specified	Any Objects	Not specified
[14]	Not specified	Not specified	Not specified	Only Dog	Raspberry Pi 4
[16]	91.76	Not specified	0.35895	Cardboard, Paper, Metal, Plastic, Glass	Arduino
[17]	Not specified	4.5	Not specified	Object detection for blind people	Raspberry Pi 4
This work	**94.17%**	**38**	**0.056**	**80 different objects**	**Custom Edge Device**

that "CPU thread 4" constantly exhibits lower latency values, reaffirming its efficiency. Table 3 compares CPU threads, emphasizing memory utilization, average latency, frame processing rate, mean latency, median latency, and inference time. The "CPU thread 4" has superior performance, minimal memory usage of 424996.98, low average latency of 0.057, and fast inference time of 0.056. Additionally, it achieves the greatest FPR of 3.4277, showing an optimal balance and high efficiency. Although the mean and median latency of CPU threads 5 and 6 are better, they are inconsistent, as shown in Figure 5. Additionally, CPU thread 4 has all the upper hand in other aspects, and we wanted to set aside some CPU thread if some emergency arises, which could be of significant help.

Table 4 evaluates the upper limit of recognized entities. The ideal configuration involves identifying three objects, achieving the lowest latency of 0.0573, the highest FPR of 3.4277, the lowest inference time of 0.0561, and the highest FPS of 38. This arrangement effectively optimizes the trade-off between velocity and precision. Table 5 assesses the threshold values, determining that 0.3 is the most favorable. The model achieves the lowest memory use of 424996, with an average latency of 0.0573 and an inference time of 0.0561. Furthermore, it achieves a maximum FPR of 3.4277 and FPS of 41, optimizing processing efficiency and accuracy, respectively.

Table 3. Performance analysis with CPU threads

No. of CPU thread	Memory usage (avg)	Latency (avg)	FPR (avg)	Mean latency	Inference time (s)
2	436742.48	0.088	3.053	0.088	0.084
3	436147.14	0.068	3.246	0.068	0.067
5	437356.88	0.060	3.337	**0.060**	0.058
6	436423.60	0.060	3.311	**0.060**	0.060
4	**424996.98**	**0.057**	**3.4277**	0.070	**0.056**

Table 4. Analysis with the max identified object

Max identified object	Memory usage	Latency (avg)	FPR (avg)	Inference time (s)	FPS (avg)
1	437076	0.0583	3.3591	0.0581	32
2	438372	0.0581	3.2576	0.0579	34
3	**424996**	**0.0573**	**3.4277**	**0.0561**	**38**
4	436660	0.0578	3.3633	0.0574	36
5	435868	0.0580	3.3643	0.0580	32
6	435868	0.0575	3.3628	0.0574	33

Table 5. Performance comparison with threshold values

Threshold	Memory usage (avg)	Latency (avg)	FPR (avg)	Inference time (s)	FPS (avg)
0.1	437153	0.0583	3.2168	0.0581	32
0.2	437154	0.0581	3.2264	0.0578	33
0.3	**424996**	**0.0573**	**3.4277**	**0.0561**	41
0.4	437168	0.0582	3.3289	0.0574	41
0.5	436440	0.0578	3.3287	0.0575	43
0.6	437104	0.0582	3.3357	0.0574	**48**

The ideal and best outcomes are attained by the equitable distribution of computational resources and effective parameter configurations that minimize latency and memory consumption while maximizing processing speed and accuracy. More precisely, the setup, including "CPU thread 4" and a threshold of 0.3, optimally utilizes the CPU threads to process data swiftly, minimizes processing delays, and ensures consistently high frame rates. Moreover, the configuration for recognizing three objects achieves a harmonious equilibrium between the detection accuracy and the processing speed, guaranteeing that the system can successfully manage several objects without overburdening the device. The adjusted settings improve the system's performance by effectively controlling computational load and achieving faster and more accurate object detection.

The tradeoffs entail carefully managing memory utilization, minimizing latency, and optimizing processing speeds. Reduced latency and inference durations are directly related to increased FPS, improving real-time performance. The optimized parameters are shown in the tables and selected, exhibiting efficient setups by ensuring minimal delay and inference times while optimizing processing rates and FPS, which are vital for real-time applications that require speed and accuracy. The results show no distinct performance tendency relative to parameters. Thus, optimal settings must be manually identified using a brute-force approach, balancing faster inference times with accuracy and FPS as needed.

Ⅵ. Conclusion

In this study, TensorFlow Lite was employed to facilitate real-time object recognition on a resourceconstrained edge device specifically designed for this purpose -a commendable achievement. The study also outlines effective procedures and guidelines for implementing and assessing performance. It achieved a 94% accuracy rate while maintaining minimal latency and delivering 30 FPS. Furthermore, a thorough comparison with previous studies clearly indicates that this effort has surpassed earlier outcomes. Enhancing accuracy and FPS will be the primary focus of our future endeavors to meet our objectives. This will involve optimizing the FPS and integrating the system into practical scenarios, such as a manufacturing execution system.

References

[1] N. Ding, C. Zhang, and A. Eskandarian, "Saliendet: A saliency-based feature enhancement algorithm for object detection for autonomous driving," *IEEE Trans. Intell. Veh.*, 2023. (https://doi.org/10.1109/TIV.2023.3287359)

[2] I. García-Aguilar, J. García-González, R. M. Luque-Baena, and E. López-Rubio, "Object detection in traffic videos: An optimized approach using super-resolution and maximal clique algorithm," *Neural Comput. and Appl.*, vol. 35, no. 26, pp. 18999-19013, 2023. (https://doi.org/10.1007/s00521-023-08741-4)

[3] A. Kaur, Y. Singh, N. Neeru, L. Kaur, and A. Singh, "A survey on deep learning approaches to medical images and a systematic look up into real-time object detection," *Archives of Computational Methods in Eng.*, pp. 1-41, 2021. (https://doi.org/10.1007/s11831-021-09649-9)

[4] D. Horváth, G. Erdős, Z. Istenes, T. Horváth, and S. Földi, "Object detection using sim2real domain randomization for robotic applications," *IEEE Trans. Robotics*, vol. 39, no. 2, pp. 1225-1243, 2022. (https://doi.org/0.1109/TRO.2022.3207619)

[5] D. A. Forsyth, J. Malik, M. M. Fleck, et al., "Finding pictures of objects in large collections of images," in *Object Representation in Comput. Vision II: ECCV 96 Int. Wkshp. Cambridge, UK, April 13-14, 1996 Proc. 2*, pp. 335-360, Springer, 1996. (https://doi.org/10.1007/3-540-61750-7_36)

[6] S.-W. Kim, K. Ko, H. Ko, and V. C. Leung, "Edge-network-assisted real-time object detection framework for autonomous driving," *IEEE Netw.*, vol. 35, no. 1, pp. 177-183, 2021. (https://doi.org/10.1109/MNET.011.2000248)

[7] A. B. Amjoud and M. Amrouch, "Object detection using deep learning, cnns and vision transformers: A review," *IEEE Access*, 2023. (https://doi.org/10.1109/ACCESS.2023.3266093)

[8] H. Naeem, J. Ahmad, and M. Tayyab, "Real-time object detection and tracking," *INMIC*, pp. 148-153, 2013. (https://doi.org/10.1109/INMIC.2013.6731341)

[9] V. Rajapakse, I. Karunanayake, and N. Ahmed, "Intelligence at the extreme edge: A survey on reformable tinyml," *ACM Comput. Surv.*, vol. 55, no. 13s, pp. 1-30, 2023. (https://doi.org/10.1145/3583683)

[10] M. J. A. Shanto, R. Akter, D.-S. Kim, and T. Jun, "Predicting bike-sharing demand: A machine learning approach for urban mobility analysis," in *2023 14th Int. Conf. ICTC IEEE*, pp. 1079-1081, 2023. (https://doi.org/10.1109/ICTC58733.2023.10393175)

[11] R. S. Praneeth, K. C. S. Akash, B. K. Sree, P. I. Rani, and A. Bhola, "Scaling object detection to the edge with yolov4, tensorflow lite," in *2023 7th IC-CMC, IEEE*, pp. 1547-1552, 2023. (https://doi.org/10.1109/ICCMC56507.2023.10084319)

[12] R. David, J. Duke, A. Jain, et al., "Tensorflow lite micro: Embedded machine learning for tinyml systems," in *Proc. Mach. Learn. and*

Syst., vol. 3, pp. 800-811, 2021. (https://shorturl.at/8Yoej)

[13] V. AI, *Tensorflow lite*, VISO AI, Accessed: 2024-04-22, 2024. [Online] Available: https://shorturl.at/PTCS4

[14] S. Swain, A. Deepak, A. K. Pradhan, S. K. Urma, S. P. Jena, and S. Chakravarty, "Real-time dog detection and alert system using tensorflow lite embedded on edge device," in *2022 1st IEEE Int. Conf. Industrial Electr.: Developments & Appl.* (*ICIDeA*), pp. 238-241, 2022. (https://doi.org/10.1109/ICIDeA53933.2022.9969906)

[15] M. S. Louis, Z. Azad, L. Delshadtehrani, et al., "Towards deep learning using tensorflow lite on risc-v," in *Third Wkshp. CARRV*, vol. 1, p. 6, 2019.

[16] N. C. A. Sallang, M. T. Islam, M. S. Islam, and H. Arshad, "A cnn-based smart waste management system using tensorflow lite and loragps shield in internet of things environment," *IEEE Access*, vol. 9, pp. 153 560-153 574, 2021. (https://doi.org/10.1109/ACCESS.2021.3128314)

[17] M. Konaite, P. A. Owolawi, T. Mapayi, et al., "Smart hat for the blind with real-time object detection using raspberry pi and tensorflow lite," in *Proc. Int. Conf. Artificial Intell. and its Appl.*, pp. 1-6, 2021. (https://doi.org/10.1145/3487923.3487929)

[18] T. Authors, *Object detection example on raspberry pi*, https://github.com/tensorflow/examples/blob/master/lite/examples/object_detection/raspberry_pi/detect.py Accessed: 2024-06-19, 2024.

Md Javed Ahmed Shanto

He has currently completed his Masters's degree and working as a graduate researcher at Networked System Laboratory, IT-Convergence Engineering in Kumoh National Institute of Technology Gumi, South Korea. He received his Bachelor of Science and Engineering degree in Computer Science and Engineering from the International University of Business Agriculture and Technology, Bangladesh. His research interests are Artificial Intelligence, Machine Learning, Deep Learning, Federated Learning, Edge AI and Blockchain.
[ORCID:0000-0002-4592-0041]

Dong-Seong Kim

He received his Ph.D. degree in electrical & computer engineering from the Seoul National University (SNU), Korea, in 2003. From 1994 to 2003, he worked as a full-time researcher in ERC-ACI at SNU, Korea. From March 2003 to February 2005, he worked as a post-doctoral researcher at the Wireless Network Laboratory in the School of Electrical & Computer Engineering at Cornell University, NY. From 2007 to 2009, he was a visiting professor with Department of Computer Science, University of California, Davis, CA. He is currently a Dean of Industrial Academic Cooperation Foundation & Director ICT Convergence Research Center (ITRC & NRF advanced research center program) supported by Korean government at Kumoh National Institute of Technology. He is a senior member of IEEE & ACM. His current main research interests are real-time IoT, smart platform, industrial wireless control network, networked embedded system & Field-bus.
[ORCID:0000-0002-2977-5964]

Taesoo Jun

He earned his B.S. degree in Electrical Engineering from Seoul National University in February

1998, followed by an M.S. degree from the same institution in February 2000. In December 2009, he successfully obtained his Ph.D. in Computer Engineering from the University of Texas at Austin. Demonstrating his commitment to research and development, he served as the Director and Principal Engineer for the SW Platform team/Global AI Center at Samsung Research, Samsung Electronics, Seoul, Korea, from 2010 to 2022. Since March 2022, he has been contributing to academia as an Assistant Professor in the Department of Computer SW Engineering at Kumoh National Institute of Technology. His diverse research interests encompass distributed computing in smart environments, intelligent systems for pervasive computing, AI applications, SW platforms for intelligent systems, and real-time systems.

[ORCID:0000-0002-1435-3769]