Comparative analysis of neural network models performance on low-power devices for a real-time object detection task

A. Zagitov [1], E. Chebotareva [1], A. Toschev [1], E. Magid [1,2]
[1] Institute of Information Technology and Intelligent Systems, Kazan Federal University, 420008, Kazan, Russian Federation, Kremlevskaya St. 35
[2] School of Electronic Engineering, Tikhonov Moscow Institute of Electronics and Mathematics, HSE University, 123592, Moscow, Russian Federation, Tallinskaya street 34

Abstract

A computer vision based real-time object detection on low-power devices is economically attractive, yet a technically challenging task. The paper presents results of benchmarks on popular deep neural network models, which are often used for this task. The results of experiments provide insights into trade-offs between accuracy, speed, and computational efficiency of MobileNetV2 SSD, CenterNet MobileNetV2 FPN, EfficientDet, YoloV5, YoloV7, YoloV7 Tiny and YoloV8 neural network models on Raspberry Pi 4B, Raspberry Pi 3B and NVIDIA Jetson Nano with TensorFlow Lite. We fine-tuned the models on our custom dataset prior to benchmarking and used post-training quantization (PTQ) and quantization-aware training (QAT) to optimize the models' size and speed. The experiments demonstrated that an appropriate algorithm selection depends on task requirements. We recommend EfficientDet Lite 512×512 quantized or YoloV7 Tiny for tasks that require around 2 FPS, EfficientDet Lite 320×320 quantized or SSD Mobilenet V2 320×320 for tasks with over 10 FPS, and EfficientDet Lite 320×320 or YoloV5 320×320 with QAT for tasks with intermediate FPS requirements.

Keywords: computer vision, image analysis, object detection, deep learning, benchmarking, optimization techniques, edge devices.

Citation: Zagitov A, Chebotareva E, Toschev A, Magid E. Comparative analysis of neural network models performance on low-power devices for a real-time object detection task. Computer Optics 2024; 48 (2): 242-252. DOI: 10.18287/2412-6179-CO-1343.

Introduction

Visual sensors based machine vision is one of key technologies of the Industry 4.0 paradigm [1]. In particular, object detection with mono and stereo cameras is often used in modern industrial robotic systems for orientation in space [2] and interaction with surrounding objects [3]. Objects to be detected by machine vision systems could have different shapes, sizes and colors, making it difficult to detect and classify them accurately. Target objects may be partially hidden or not fully visible, or have different shapes in perspective depending on their position. Furthermore, environmental factors such as lighting conditions can further complicate a detection process.

For these reasons, convolutional neural network (CNN) based object detection methods, which are a type of deep neural network, have gained a particular popularity in recent years due to their efficiency over traditional methods [4]. However, a high computational power required for most real-time object detection methods involving neural networks can be a limiting factor for their use on low-power devices [5]. In many cases, a choice of a particular solution method is influenced by low resource requirements while still maintaining an acceptable level of accuracy in a machine vision system.

Yet, despite a demanding nature of deep learning algorithms, there are numerous techniques that can enhance their performance. One such method is to employ models specifically crafted for mobile computing, such as the renowned MobileNet [6]. Additionally, applying frameworks that optimize a device resource utilization (e.g., Tensorflow Lite [7], ncnn [8], or MNN [9]) could significantly boost algorithms' efficiency.

This paper compares a number of the most popular single-stage neural network models designed for object detection that are suitable for use on low-performance real-time devices. We run benchmarks using Raspberry Pi 4B, Raspberry Pi 3B, and NVIDIA Jetson Nano on a selection of models that were fine-tuned for our dataset. Different model sizes were examined and optimization techniques were used to speed up an inference process. We believe that our experimental results and their comparative analysis provide useful insights into trade-offs between the accuracy, speed, and computational efficiency of these models.

1. Related work

Machine vision with CNNs has been extensively researched and applied to a wide range of tasks. In robotics, this technology has proven to be a crucial component in navigation, manipulation, and perception tasks. For example, Myrzin et al. [10] developed a human detection framework for Servosila Engineer rescue robot camera using Rotation-Invariant Histogram of Oriented Gradients (RIHOG) features along with binarized normed gradients (BING) pre-processing and skin segmentation steps. In a related study, Buyval et al. [11] used CNN filtering to exclude dynamic objects in the process of visual based self-

localization for unmanned aerial vehicles (UAVs). Among the most common tasks in robotics that utilize a machine vision, an object detection stands out. CNNs have been proven to be highly effective for this task, with numerous architectures and techniques being proposed over the past years.

One of the first neural networks used to detect objects was the Region-Based CNN (R-CNN) [12]. The algorithm operated in two stages: the first stage utilized a selective search method to identify regions in an image that may contain an object, while the second stage employed a CNN and SVM to classify these regions and linear regression to refine their boundaries. On the other hand, the Single-Shot MultiBox Detector (SSD) [13] proposed by Wei Liu et al. operated with a single pass over an image to identify multiple objects and generate corresponding bounding boxes. While this approach resulted in a lower accuracy compared to two-stage detectors, it offered a faster detection speed and became a preferred choice for low-power devices.

In the original SSD paper, the authors used VGG-16 as a feature extractor. Later, Howard et al. introduced a set of mobile and embedded image classification models called MobileNets [6]. These models are based on an optimized architecture that uses depth-separated convolutions to create lightweight deep neural networks. Huang et al. [14] demonstrated that combining SSD with MobileNet as a feature extractor could achieve a similar accuracy to VGG-16 on ImageNet with only 1/30 of the computational cost and model size. This led to the MobileNet-SSD model becoming a highly researched topic with numerous applications for resource-constrained tasks. For example, in [15], MobileNet-SSD was utilized and compared against several other models in a surface defect detection, achieving not only the highest correct detection rate of 95 % but also the lowest detection time per image on GPU. Later, it was implemented and verified in an oil chili filling production line in Guizhou, China.

The work on MobileNet was continued by Sandler et al. [16]. Their proposed MobileNetV2 model greatly enhanced computer vision applications on mobile platforms. By optimizing the architecture of MobileNetV1 and adding linear bottlenecks between layers along with short-cut connections between them, the authors were able to increase the inference speed by almost 30 % on classification tasks while maintaining the same level of accuracy. In addition, the SSDLite architecture described in their paper enabled the same speedup to be achieved for object detection tasks.

MobileNetV2 with SSD has been widely used in a variety of computer vision tasks, including expression recognition and surveillance systems. In [17] Zhang et al. applied MobileNetV2-SSD for expression recognition on Nao robot equipped with ATOM Z530 1.6 GHz CPU and 1 GB RAM, and was able to achieve real-time performance with 68.97 % accuracy on FER2013 dataset and 89.2 % accuracy on CK+ dataset. Ahmed et al. used the MobileNetV2-SSD model in a real-time crowd surveillance system [18]. They sent video sequences from IP cameras to a remote server over the Internet and used a deep learning model. Their results showed a mean average precision and counting accuracy of 95 %.

According to [19], the MobileNetV2-SSD [16] and the EfficientDet [20] are the most commonly used models in research papers, followed by YOLO models [21]. Additionally, CenterNet's [22] unique architecture and approach enabled it to achieve the best results among all single-stage detectors on the MS COCO dataset [38], while maintaining a high computational speed. Therefore, we selected all the aforementioned models for a comparison in our study.

EfficientDet [20] followed the one-stage detector architecture and was built upon EfficientNet [23], which was pre-trained on ImageNet. To enhance its performance, it included a weighted bi-directional feature pyramid layer (BiFPN), which enabled an efficient multi-scale feature fusion using a bi-directional information flow – a new fast normalized fusion technique that adjusted a weight of each input feature based on its contribution to an output, and fast depthwise separable convolutions [24]. Next, the BiFPN output was processed by a class network and a block network, which generated predictions for an object class and bounding boxes, respectively.

Several studies applied EfficientDet to different object detection tasks and evaluated its performance on various hardware. For instance, Nguyen et al. [25] used EfficientDet on Nvidia Jetson TX2 for a real-time vehicle detection and achieved 47.3 % mean average precision (mAP) at Intersection over Union (IoU) of 75 % (mAP:0.75) with EfficientDet-D0 512x512 model and 16.5 frames per second on the KITTY dataset. An application of EfficientDet in fabric defect detection was used to enhance an accuracy of an industrial defect detection for textile production lines. A recent study by Song et al. [26] implemented EfficientDet-D0 512x512 model on NVIDIA Jetson TX2 for this task. Among several other models, EfficientDet achieved a highest accuracy and detection speed on five different datasets.

The single-stage YOLO architecture was first introduced by Redmon et al. [21]. Currently, several modifications and versions of YOLO family models were implemented providing a substantial increase in a detection speed and accuracy. A number of research papers have adopted the YOLO architecture for a variety of tasks. For example, YoloV3 was used by Abdulganeev et al. [27] for a door handle detection for mobile robots. Lyu et al. [28] utilized a modified version of YoloV5 model on Nvidia Jetson Xavier NX for detecting and counting green citrus in orchards. They achieved a high mean average precision of 98.23 % at 0.5 intersection over a union threshold and 28 frames per second on a custom dataset comprising 620 test images with a size of 416×416.

This paper focuses on YoloV5 [29], YoloV7 [30] and YoloV8 [31] models, which are considered to be the most

recent and widely used within the YOLO family. YoloV5 is much easier to train and is more lightweight then previous YOLO models, while YoloV7 promises a significant improvement in speed and accuracy over YoloV5. However, some studies (e.g., [32], [33]) suggested that this improved computational speed is only observable on high-speed GPUs such as Nvidia RTX 3090 or Tesla A100, while on standard GPU or CPU systems YoloV7 may perform slower than YoloV5. On the other hand, in addition to the YoloV7 model, the authors in [30] introduced a lightweight version called YoloV7-tiny. This version is optimized for edge devices and is also included in our comparison. Finally, the latest addition to the YOLO family, YoloV8, promises an improved performance and accuracy when compared to YoloV5 and YoloV7.

CenterNet [22] was introduced in 2018 and is known for its high accuracy and fast inference speed compared to two-stage object detection networks like Faster R-CNN. CenterNet used a keypoint detection approach, where a single point was assigned to each object instead of a bounding box. This approach allowed the network to focus on a center of objects, which was crucial for detecting small and dense objects. Additionally, the network utilized a regression network to estimate an object size and pose. In a recent study by Xia et al. [34], the authors used CenterNet with MobileNetV1 backbone for an insulator defect detection during a power lines inspection task with a UAV. Their model achieved a mean average precision of 90 % and a frame rate of 15 frames per second (FPS) on an Intel Core i7-8700 CPU, outperforming the original CenterNet with the ResNet-50 backbone.

Various transformation methods are available for optimizing a size and a speed of neural network models. One of the most effective techniques is a quantization, which targets for reducing a number of bits that represent weights and activations in a model [35] [36]. There are two primary approaches to the quantization: a post-training quantization (PTQ) and a quantization-aware training (QAT). The first approach involves a neural network quantizing after it has been trained with floating-point computations. This approach is straightforward to use but it often leads to a high loss of accuracy. To mitigate this, QAT is employed. This approach retrains a neural network using a forward pass quantization simulation, which enables a model to maintain its accuracy even with a full-integer quantization.

Challenges and limitations of running object detection models on low-constrained devices as well as benchmarking results were explored by Cantero et al. [37]. They demonstrated a performance of several models on the i-MX8M-PLUS processor and Coral Dev Board. Our research extends the evaluation to the Raspberry Pi 4B, Raspberry Pi 3B and Jetson Nano, and incorporates three additional models, YoloV5, YoloV7 and YoloV8, and examines different model sizes. Moreover, we expand the analysis by fine-tuning and evaluating models on a custom dataset. This allows us to investigate the models' capability to achieve a high accuracy on a smaller training dataset.

2. Instruments

To evaluate a performance of neural networks in an object recognition, a number of tools and instruments are required for conducting experiments and collecting data. This section describes instruments and software used in our experiments, including hardware specifications, configuration details, software libraries, and frameworks.

2.1. Models

The following 6 model architectures for object detection were used in our research:

- SSD MobilenetV2 FPN-lite [16];
- EfficientDet Lite D0/D3 [20];
- CenterNet MobileNetV2 FPN [22];
- YoloV5 Small [29];
- YoloV7 + YoloV7 Tiny [30];
- YoloV8 Small [31].

Each model (except YoloV7) was trained and evaluated with two different input layer sizes: 320×320 and 512×512. Next, each model (except YoloV7 and CenterNet) was optimized using int8 post-training quantization. It was observed that due to the YoloV7 and CenterNet models' architecture, the TFLiteConverter tool failed to perform a quantization on these models correctly. A similar issue arose with the YoloV8 model, but it was successfully resolved by manually removing an extra layer of dequantization that the tool had added automatically. Lastly, YoloV5 has a version obtained via quantization-aware training. In total, 22 models were covered in the comparison. All of them were pre-trained on the Microsoft COCO (Common Object in Context) dataset [38] and fine-tuned using the custom dataset, which is further described in subsection 1.4.

We applied various data augmentations during the training for different architectures: a random horizontal flip (with 50 % probability) and a random crop (ensuring at least 75 % of an initial image area is retained) for SSD, CenterNet, and EfficientDet; default YOLO augmentations of a mosaic augmentation, a horizontal flip, a HSV augmentation, a scale, and a translate for YOLO models. Additionally, each model was trained using its default hyperparameters from the pretraining on the COCO dataset, as provided in the official Tensorflow repository for SSD, CenterNet, and EfficientDet, and in the corresponding YOLO repositories for YOLO models. The training was stopped if a model failed improving mAP on a validation sample for over 300 epochs for the YOLO models and 5000 epochs for all other models.

2.2. Hardware

Three microcomputers were used in the experiments:

- Raspberry Pi 4B, equipped with ARM Broadcom BCM2711 Cortex-A72 4-core processor running

at 1.5GHz, 8 GB of RAM, and Debian 10 Buster 64-bit OS installed;
- Raspberry Pi 3B, equipped with ARM Broadcom BCM2837 Cortex-A53 4-core processor running at 1.2 GHz, 1 GB of RAM, and Debian 10 Buster 64-bit OS;
- NVIDIA Jetson Nano Developer Kit, equipped with ARM Cortex-A57 4-core processor running at 1.43 GHz, 4 GB of RAM, and Ubuntu 20.04 64-bit OS installed.

The Raspberry Pi 4B, Raspberry Pi 3B, and Jetson Nano are microcomputers that can be used for a variety of applications and are used in the majority of studies, according to [19]. The Raspberry Pi 4B is versatile and cost-effective, while the Raspberry Pi 3B is an older version that is still widely used. The Jetson Nano is designed specifically for AI and machine learning applications and is known for its high performance and low power consumption.

The PC for model trained had AMD Ryzen 7 3700X 8-core processor, 32 GB of RAM, NVIDIA GeForce RTX 3060 12 GB GPU and Windows 10 OS with WSL2 Ubuntu 20.04. The tests were conducted using the same PC and operating system.

2.3. Frameworks

TensorFlow 2 Object Detection API [14] was used as a framework for training MobilenetV2, EfficientDet and CenterNet SSD models. PyTorch [39] was used to train YoloV5, YoloV7 and YoloV8.

There are a variety of runtime frameworks for deep learning models, designed for low-power devices and supported by Raspberry Pi 4B, Raspberry Pi 3B and Jetson Nano, such as ncnn, MNN or TensorFlow Lite runtime. This paper uses the latter due to its ease of use, detailed documentation, and many accompanying useful tools for working with *.tflite* models. Additionally, Tensorflow Lite is a common choice for TinyML applications [40]. It's worth noting that Tensorflow Lite is designed to support an inference only on CPU and may not be the optimal choice for devices with powerful GPUs, such as Jetson Nano.

2.4. Datasets

Often, object detection models are trained and tested on large public datasets, such as MS COCO. However, when it comes to a specific task, only a small number of object classes may be relevant. Additionally, collecting a suitable dataset can be one of the most expensive aspects of implementing deep learning algorithms.

For the model fine-tuning we use our custom dataset of 913 images of three unique household objects, each forming a separate class: a toy, a can opener, and a tonometer. Additionally, the dataset includes some images with no objects present (forming a background).

A selection of such dataset was driven by a desire to evaluate the model quality using non-typical objects that are not commonly found in existing datasets. Secondly, we wanted to test the models' ability to achieve a high accuracy with a relatively small amount of training data, which can be valuable in real-world scenarios for low-cost robots with limited resources, e.g., swarm applications. Initially, the dataset idea derived from an educational robotics project that had been focused on a detection of several specific objects' using a small self-constructed dataset (while a usage of existing open source datasets was forbidden).

The dataset was divided into 3 parts:
- A training set: 595 images;
- A validation set: 148 images;
- A test set: 170 images.

The images in the dataset were captured under various lighting conditions and from different angles and distances from the objects. Examples of training data are shown in Fig. 1. Table 1 specifies a number of labelled objects in each set and demonstrates that all parts of the dataset are mostly balanced. Note that a single image may contain multiple labeled objects.

Tab. 1. Number of labeled objects

Object	Train	Validation	Test
Toy	209	56	67
Can opener	236	43	67
Tonometer	211	60	66
Background	35	8	10

Fig. 1. Some examples of the training data: a) toy; b) can opener; c) tonometer; d) background (no objects)

2.5. Metrics

The metrics chosen for evaluating a quality of detections were mAP with an Intersection-over-Union (IoU) threshold of 0.5 and an average mAP over a range of IoU levels from 0.5 to 0.95 with a step frequency of 0.05 (mAP:0.5-0.95). To evaluate an inference speed of each model, we measured Latency, which is the average inference time per frame in milliseconds. For a reader convenience, we also provide an average expected FPS, calculated using equation (1).

$$FPS = \frac{1000}{Latency}. \quad (1)$$

2.6. Additional instruments

The dataset was annotated using the Microsoft Visual Object Tagging Tool (VoTT). The official TFLiteConverter tool allowed us to obtain a model in *.tflite* format supported by TensorFlow Lite and apply PTQ to the completed model. We also utilized the TFLite Model Benchmark Tool, another official tool by the TensorFlow developers, which allows for easy and accurate measurement of a speed of TFLite models. Furthermore, we optimized YoloV5 using QAT with SparseML [41]. SparseML is a set of tools designed to simplify an implementation of advanced sparsification techniques, such as pruning and quantization, on any type of neural networks.

3. Experimental results

Tab. 2 displays an evaluation of the quality metrics for models of varying sizes on the test set, with the *[int8]* label indicating that the PTQ was used, and *[QAT]* indicating the model was optimized through the quantization-aware training. Examples of successful and incorrect detections for quantized EfficientDet 320×320 are shown accordingly in Fig. 2 and Fig. 3.

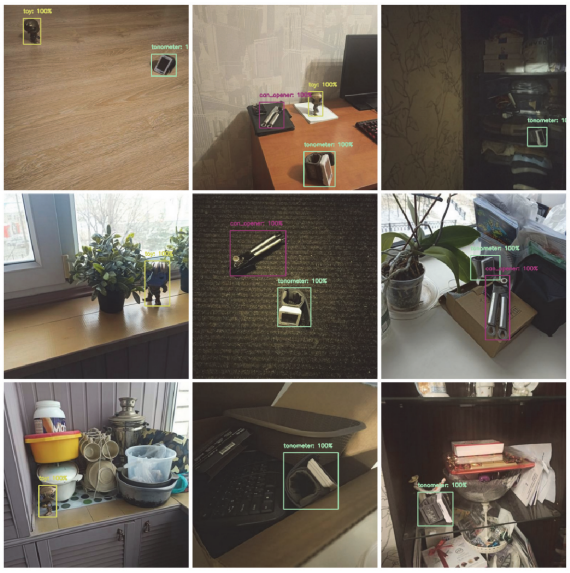

Fig. 2. Examples of successful detections on test data with int8 quantized EfficientDet for 320×320 model size

Tab. 3 presents an average processing time of a single image for each model in milliseconds; the average processing time was evaluated based on a sample of 1000 randomly selected images. It is important to note that the latency and FPS values shown are only for the model's inference time and the application of Non-Maximum Suppression to the output. The time required for preprocessing an image is typically constant for a given task, but can vary between different tasks. For these reasons, time for an image or a frame pre-processing and any additional post-processing of the results were not included in these values. The table is divided into two vertical and two horizontal blocks. Within the vertical blocks, the left block displays the latency in milliseconds and the right one displays FPS. Within the horizontal blocks, the top block contains results for 512x512 model size, while the bottom block stands for 320x320 model size.

Fig. 3. Examples of incorrect detections on test data with int8 quantized EfficientDet for 320×320 model size

Tab. 2. Models accuracy

Model	Accuracy			
	Input layer 320×320		Input layer 512×512	
	mAP:0.5-0.95	mAP:0.5	mAP:0.5-0.95	mAP:0.5
CenterNet	0.406	0.595	0.494	0.704
EfficientDet Lite	0.557	0.781	0.707	0.860
EfficientDet Lite [int8]	0.541	0.782	0.700	0.860
SSD	0.451	0.674	0.510	0.719
SSD [int8]	0.401	0.647	0.498	0.720
YoloV5	0.585	0.799	0.703	0.893
YoloV5 [int8]	0.355	0.607	0.3	0.609
YoloV5 [QAT]	0.549	0.852	0.586	0.894
YoloV7	0.717	0.93	-	-
YoloV7 Tiny	0.609	0.857	-	-
YoloV8	0.586	0.776	0.662	0.825
YoloV8 [int8]	0.45	0.69	0.552	0.792

A relationship between the model quality and the computation speed for each platform is illustrated by Fig. 4. The vertical axis of the graph represents the mean average precision of each model, which is calculated over a range of IoU thresholds from 0.5 to 0.95. The horizontal axis represents the latency of each model, which is the average amount of time it takes to process a single image. The dashed arrows show changes in models after quantization. The quantized

models using PTQ are marked with circles around points, while those optimized using QAT are marked with squares around points. Different models have different colors; different model sizes are marked with different symbols.

Tab. 3. Inference time comparison

Model	Inference Time							
	Latency (ms)				FPS			
	PC	Jetson Nano	Raspberry Pi 4B	Raspberry Pi 3B	PC	Jetson Nano	Raspberry Pi 4B	Raspberry Pi 3B
512×512								
CenterNet MobileNetV2	17	262	490	1802	58	3.8	2	0.5
EfficientDet Lite D3	65	756	1041	4110	15	1.3	0.9	0.2
EfficientDet Lite D3 [int8]	41	491	503	2687	24	2	1.9	0.4
SSD Mobilenet V2	23	254	416	1674	43	3.9	2.4	0.6
SSD Mobilenet V2 [int8]	15	177	221	1094	66	5.6	4.5	0.9
YoloV5	40	690	925	3281	25	1.4	1	0.3
YoloV5 [int8]	24	345	369	1627	41	2.9	2.7	0.61
YoloV5 [QAT]	27	358	378	1857	37	2.8	2.6	0.5
YoloV8	67	1107	1551	5515	14	0.9	0.6	0.18
YoloV8 [int8]	33	518	557	2396	30	1.9	1.8	0.41
320×320								
CenterNet MobileNetV2	7	104	189	624	142	9.5	5	1.6
EfficientDet Lite D0	10	127	176	685	100	7.8	5.6	1.4
EfficientDet Lite D0 [int8]	7	80	85	412	142	12	11.7	2.4
SSD Mobilenet V2	9	140	158	661	111	7	6.3	1.5
SSD Mobilenet V2 [int8]	6	70	71	345	166	14	14	2.9
YoloV5	16	272	370	1525	62	3.6	2.7	0.65
YoloV5 [int8]	10	135	146	647	100	7.3	6.8	1.5
YoloV5 [QAT]	10	140	155	674	100	7.1	6.4	1.5
YoloV7	254	1716	2642	9279	3	0.6	0.37	0.1
YoloV7 Tiny	42	242	385	1211	23	4.1	2.59	0.82
YoloV8	27	435	648	2351	37	2.3	1	0.42
YoloV8 [int8]	14	212	241	959	71	4.7	4	1

Since YoloV7's latency was drastically higher than for other models, it is excluded from the graph to improve readability. Note that the horizontal axis has three different scales: one for Raspberry Pi 4B (Fig. 4*a*) and Jetson Nano (Fig. 4*b*), and different for Raspberry Pi 3B (Fig. 4*c*) and PC (Fig. 4*d*).

4. Discussion

To assess accuracy of our models we utilized a test set of 170 images. While using a larger test sample could potentially increase a statistical significance of estimates' quality, we decided to maintain the small test sample size for several reasons. Firstly, given the specific task of detecting three classes of unique household objects, the dataset inherently possesses a limited number of unique objects and scenarios. Increasing the test sample size would not introduce substantially different or novel situations, thereby potentially providing diminishing returns in terms of an added diversity and could lead to a redundancy. Instead, we invested a significant effort in carefully selecting the test set to be diverse and representative of various scenarios, different from those encountered during the model training and validation. The test sample was thoughtfully designed to encompass diverse lighting conditions, angles, and distances from objects, making it more challenging and distinct from the training and validation sets. Furthermore, despite utilizing only three unique objects across all sets, their complex shapes guarantee substantial variations in appearance when viewed from different angles.

Based on the experiments and the results obtained, we made a number of important observations on models and device, which are presented in this section.

4.1. Model comparison

Although the YoloV7 320×320 model has the highest mAP, surpassing even the 512×512 models, it also has the highest latency, making it unsuitable for any real-time object detection on Raspberry Pi 4 and other low-power platforms. On the other hand, the YoloV7 Tiny is the most accurate among models that process at least two frames per second, as shown in Fig. 5 – 7.

Fig. 5 presents the mAP comparison. The model is shown on the horizontal axis, with each bar representing the model's mAP over a range of IoU values from 0.5 to 0.95. The color of each bar indicates the size of the model, as well as whether it uses int8 or float16. The models are arranged in an ascending order based on the average mAP of their float16 performance.

Fig. 6 and 7 demonstrate an expected FPS comparison for Raspberry Pi 4 and NVIDIA Jetson Nano, respective-

ly; each bar of the graphs represents a number of FPS processed by the model.

The standard YoloV5 has a fairly average accuracy and quality. Unfortunately, the PTQ affects the accuracy of YoloV5 much more than other models, which can be seen in Fig. 4 or 5. However, with QAT it was possible to recover most of the accuracy losses while maintaining more than a 2-fold increase in speed.

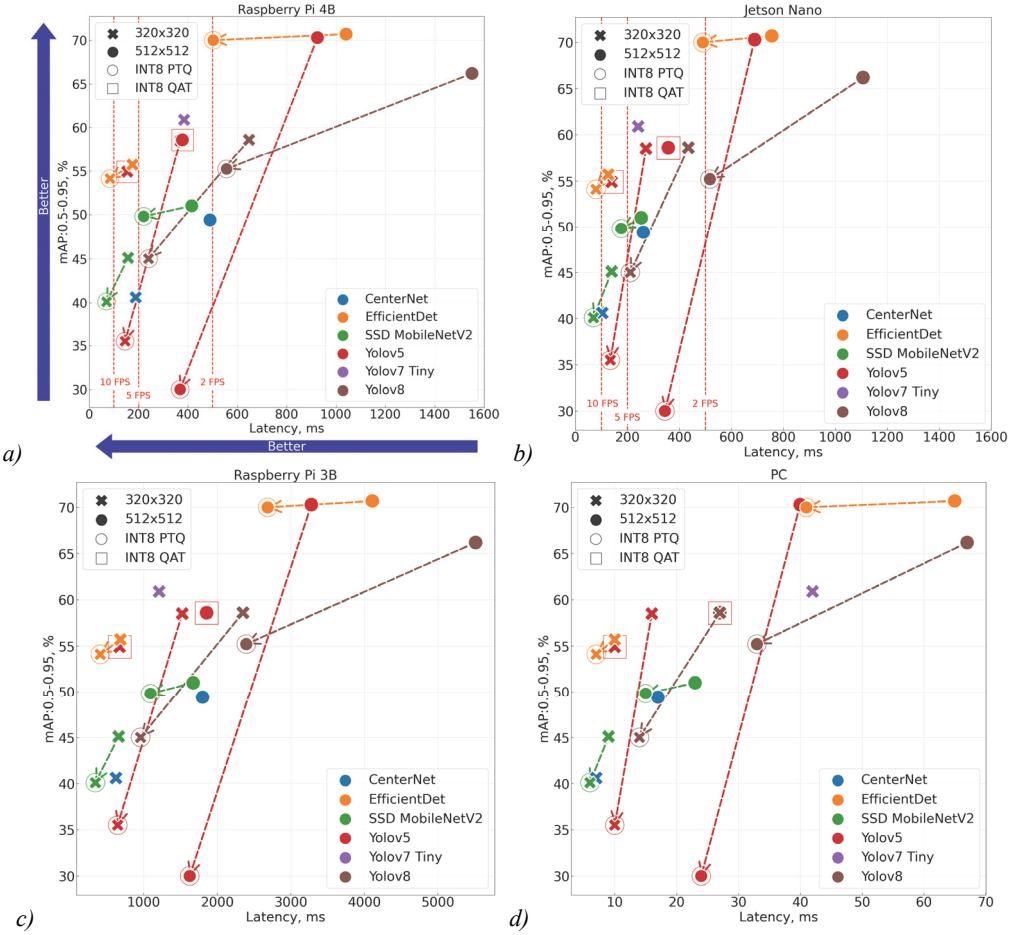

Fig. 4. Scatter plot of latency versus mAP on: a) Raspberry Pi 4B; b) NVIDIA Jetson Nano; c) Raspberry Pi 3B; d) PC

YoloV8 models perform better in terms of accuracy with the post-training int8 quantization than YoloV5 quantized models, but still not as well as YoloV5 models optimized with QAT. It's worth noting that YoloV8 is a recent addition to the YOLO family, and using SparseML QAT for YoloV8, when it becomes available, could further improve its accuracy.

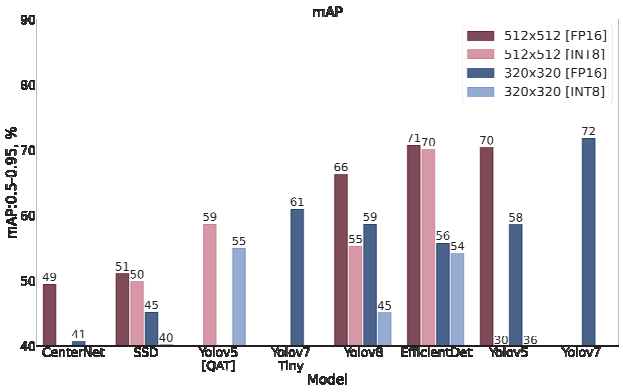

Fig. 5. Mean average precision comparison

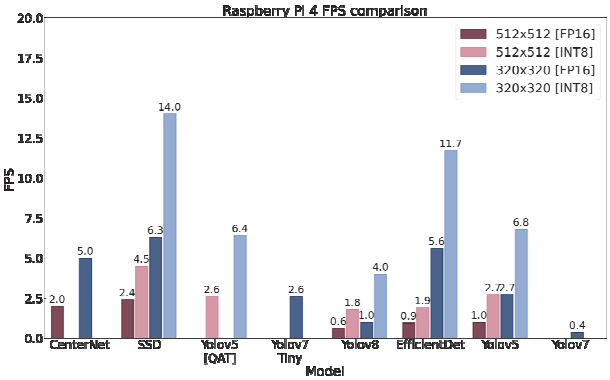

Fig. 6. Raspberry Pi 4 expected FPS comparison

Fig. 8 compares floating-point models (fp16) and quantized models (int8) in terms of average FPS and mAP. The graph illustrates the average speed improvement achieved by quantized models compared to floating-point models, as well as the average decrease in mAP resulting from the quantization, evaluated across all platforms. The X-axis represents the different models tested, the Y-axis represents the average difference in percentage, and the color

represents the metric. The dashed line represents the overall average difference across all models.

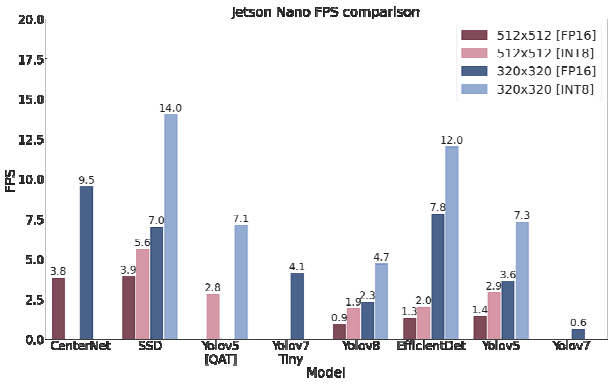

Fig. 7. NVIDIA Jetson Nano expected FPS

Fig. 9 compares 512×512 and 320×320 sized models in terms of average FPS and mAP. The graph illustrates the average speed improvement achieved by reducing the model size from 512×512 to 320×320, as well as the average decrease in mAP resulting from this reduction in size, evaluated across all platforms.

When comparing floating-point models (fp16) to quantized models (int8), the average speed improvement is 96 % with a 19 % average decreased in mAP as demonstrated by dashed lines in Fig. 8. By reducing the model size from 512×512 to 320×320, the average speed gain was approximately 154 % with an average decrease in mAP of 16 % as demonstrated by dashed lines in Fig. 9. The overall average difference for FPS was calculated without including EfficientDet since it had a significantly higher speedup compared to other models.

To assess the statistical significance of the model performance comparison, we performed paired t-tests on the inference times for all models on both Raspberry Pi 4 and PC. The results reveal significant differences ($p \leq 0.05$) in latency among most models. However, some pairs of models did not show statistically significant differences in performance; these include YoloV5 [QAT] vs. SSD MobilenetV2 [FP16], YoloV5 [QAT] 512×512 vs. YoloV7 Tiny 320×320. Notably, there was no significant difference in performance between YoloV5 [int8] vs. YoloV5 [QAT].

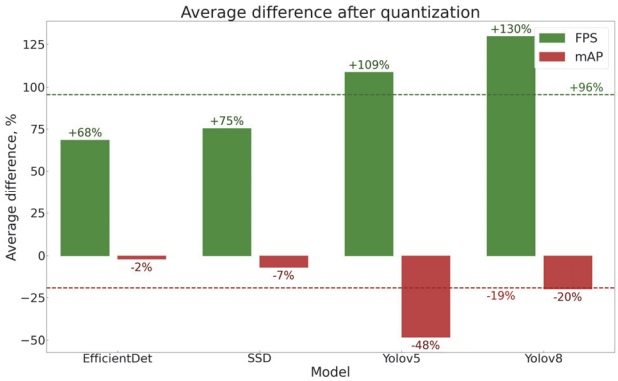

Fig. 8. Comparison of floating-point models (fp16) and quantized models (int8) in terms of average FPS and mAP

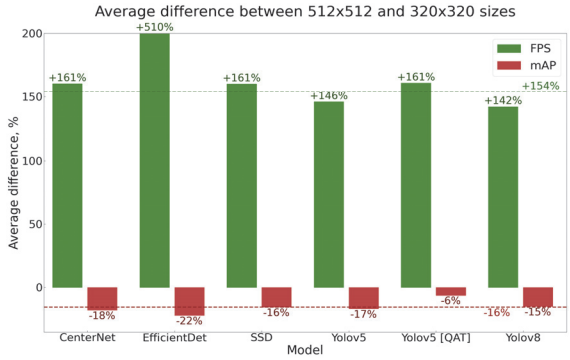

Fig. 9. Comparison of 512×512 and 320×320 sized models in terms of average FPS and mAP

In addition, it was noted that quantized models had a higher rate of repeated detections, which cannot be reliably resolved by applying the Non-Maximum Suppression algorithm, as illustrated by Fig. 10.

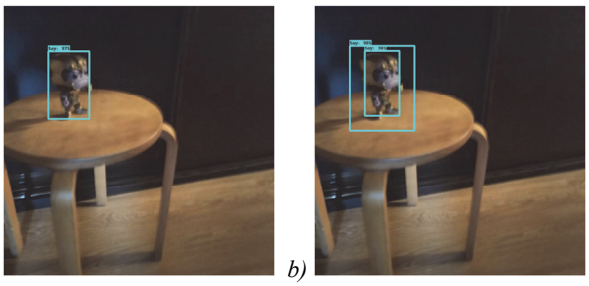

Fig. 10. Example of incorrect detection after quantization: a) YoloV5 320×320 fp16; b) YoloV5 320×320 after int8 PTQ

The most efficient model in terms of accuracy vs. inference speed was the quantized EfficientDet Lite 320×320 on all devices. It's worth noting that both versions of EfficientDet Lite demonstrated a very little loss in accuracy that followed the quantization process.

The comparison of our results with previous works reveals a number of interesting insights. EfficientDet stands out as one of the most efficient detectors for mobile devices and embedded platforms, which is supported by the original study [20] and by other research papers, e.g., [26]. Comparing the models' accuracy to other works is challenging due to typically custom datasets that limit direct comparisons. However, when evaluating a relative performance of the 512×512 models without quantization in terms of mAP:0.5 – 0.95, their order remains consistent with results on the COCO val2017 dataset, with an exception of YoloV8, which demonstrated a slightly lower performance in our experiments.

Comparing a latency across models is also a challenging task, with limited prior studies using the same input sizes and quantization on the same devices. For example, in [42], authors assessed a frame rate of EfficientDet with different input sizes on Raspberry Pi 3, yielding results (1.1 FPS for 320×320 version (D0) and 0.4 FPS for 512×512 version (D3)) consistent with our findings. Another work [43] utilized SSD MobileNet V2 at 320×320 input size on Raspberry Pi 4B and achieved 5.2 FPS, which mostly aligns with our recorded 6.3 FPS.

4.2. Device comparison

After assessing differences between devices by calculating the overall average FPS across all models, as shown in Fig. 11, and also by averaging the differences between individual models, the following data was obtained: the Jetson Nano was approximately 90 % slower than the PC, the Raspberry Pi 4B was approximately 20 % slower than the Jetson Nano, and the Raspberry Pi 3B was approximately 75 % slower than the Raspberry Pi 4B. The data for the PC is not presented in Fig. 11 to improve its readability due to scaling. The red arrows demonstrate the FPS decrease and the numbers below the arrows show the relative decrease percentage of the right (to the arrow) bar with respect to the left bar.

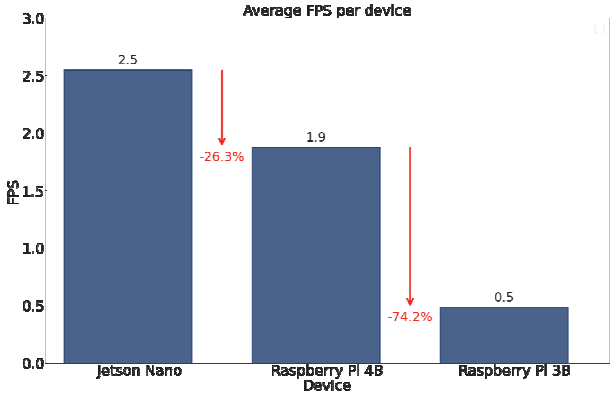

Fig. 11. Average frames per second across all models for each device, except PC

Notably, certain models, including the CenterNet, SSD MobileNet V2 512×512 [FP16], YoloV7 and YoloV7 Tiny, showed a larger performance difference between the Jetson Nano and the Raspberry Pi 4B, with a gap of around 40 %, while the remaining models showed a performance difference of around 15 % between the two devices. Additionally, when performing paired t-tests on the results of the same model on Raspberry Pi 4B and Jetson Nano, we observed significant differences ($p \leq 0.05$) for most models, except for quantized EfficientDet models and quantized SSD MobileNetV2 320×320.

The small difference between the Raspberry Pi 4B and the Jetson Nano was due to their comparable CPU power. As it was mentioned earlier, Tensorflow Lite only uses the CPU for inference. Therefore, a different runtime inference engine, such as TensorRT, may be more appropriate for the Jetson Nano, which has a significantly more powerful GPU compared to the Raspberry Pi 4B.

The Raspberry Pi 3B is generally not suitable for real-time object detection, since only the SSD MobileNet V2 320×320 with int8 quantization and the EfficientDet Lite 320×320 with int8 quantization can deliver around 2 FPS.

Conclusions

In this paper, we have presented a comparative analysis of the fine-tuned MobileNetV2 SSD, CenterNet MobileNetV2, EfficientDet Lite, YoloV5, YoloV7 and YoloV8 models. The experiments carried out on the Raspberry Pi 4B, Raspberry Pi 3B and NVIDIA Jetson Nano.

The experimental results demonstrated that the most appropriate algorithm selection depends on task requirements. For tasks that require around 2 FPS on Raspberry Pi 4B, EfficientDet Lite 512×512 quantized or YoloV7 Tiny are recommended. If the task demands more than 10 FPS, then EfficientDet Lite 320×320 quantized or SSD Mobilenet V2 320×320 should be preferred. Finally, tasks with intermediate FPS requirements were best served by either EfficientDet Lite 320×320 or YoloV5 320×320 with QAT.

The presented comparative analysis of models and optimization methods in terms of accuracy and performance could guide researchers, educators, and engineers in selection of the most suitable approaches and instruments for use on low-performance devices.

Acknowledgements

This paper has been supported by the Kazan Federal University Strategic Academic Leadership Program ("PRIORITY-2030").

References

[1] Javaid M, Haleem A, Singh RP, Rab S, Suman R. Exploring impact and features of machine vision for progressive industry 4.0 culture. Sens Int 2022; 3: 100132.

[2] Nicholson L, Milford M, Sünderhauf N. QuadricSLAM: Dual quadrics from object detections as landmarks in object-oriented SLAM. IEEE Robot Autom Lett 2019; 4(1): 1-8.

[3] Motoda T, Petit D, Nishi T, Nagata K, Wan W, Harada K. Shelf replenishment based on object arrangement detection and collapse prediction for bimanual manipulation. Robotics 2022; 11(5): 104.

[4] Elhassouny A, Smarandache F. Trends in deep convolutional neural Networks architectures: a review. 2019 Int Conf of Computer Science and Renewable Energies (ICCSRE) 2019: 1-8.

[5] Branco S, Ferreira AG, Cabral J. Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey. Electronics 2019; 8(11): 1289.

[6] Howard AG, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreetto M, Adam H. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv Preprint. 2017. Source: <https://arxiv.org/abs/1704.04861>.

[7] Abadi M, et al. TensorFlow: A system for large-scale machine learning. In Book: Keeton K, Roscoe T, eds. Proceedings of the 12th USENIX conference on operating systems design and implementation, Savannah, GA, USA, 2016. Berkeley, CA: USENIX Association; 2016: 265-283.

[8] Tencent/ncnn. 2018. Source: <https://github.com/Tencent/ncnn>.

[9] Jiang X, et al. MNN: A universal and efficient inference engine. Proc 3rd MLSys Conf 2020; 2: 1-13.

[10] Myrzin V, Tsoy T, Bai Y, Svinin M, Magid E. Visual data processing framework for a skin-based human detection. In Book: Ronzhin A, Rigoll G, Meshcheryakov R, eds. Interactive collaborative robotics. 6th International Conference,

ICR 2021. Cham, Switzerland: Springer Nature Switzerland AG; 2021: 138-149.
[11] Buyval A, Gavrilenkov M, Magid E. A multithreaded algorithm of UAV visual localization based on a 3D model of environment: implementation with CUDA technology and CNN filtering of minor importance objects. 2017 Int Conf on Artificial Life and Robotics (ICAROB 2017) 2017; 22: 356-359.
[12] Girshick R, Donahue J, Darrell T, Malik J. Rich feature hierarchies for accurate object detection and semantic segmentation. 2014 IEEE Conf on Computer Vision and Pattern Recognition (CVPR) 2014: 580-587.
[13] Liu W, et al. SSD: Single shot multibox detector. In Book: Leibe B, Matas J, Sebe N, Welling M, eds. Computer Vision – ECCV 2016. Pt I. Cham, Switzerland: Springer International Publishing AG; 2016: 21-37.
[14] Huang J, et al. Speed/accuracy trade-offs for modern convolutional object detectors. Proc IEEE Conf on Computer Vision and Pattern Recognition (CVPR) 2017: 3296-3297.
[15] Li Y, Huang H, Xie Q, Yao L, Chen Q. Research on a surface defect detection algorithm based on MobileNet-SSD. Appl Sci 2018; 8(9): 1678.
[16] Sandler M, Howard A, Zhu M, Zhmoginov A, Chen L-C. MobileNetV2: Inverted residuals and linear bottlenecks. 2018 IEEE/CVF Conf on Computer Vision and Pattern Recognition (CVPR) 2018: 4510-4520.
[17] Zhang F, Li Q, Ren Y, Xu H, Song Y, Liu S. An expression recognition method on robots based on MobileNet V2-SSD. 2019 6th Int Conf on Systems and Informatics (ICSAI) 2019: 118-122.
[18] Ahmed I, Ahmad M, Ahmad A, Jeon G. IoT-based crowd monitoring system: Using SSD with transfer learning. Comput Electr Eng 2021; 93: 107226.
[19] Kamath V, Renuka A. Deep learning based object detection for resource constrained devices: Systematic review, future trends and challenges ahead. Neurocomputing 2023; 531: 34-60.
[20] Tan M, Pang R, Le QV. EfficientDet: Scalable and efficient object detection. Proc IEEE/CVF Conf on Computer Vision and Pattern Recognition (CVPR) 2020: 10778-10787.
[21] Redmon J, Divvala S, Girshick R, Farhadi A. You only look once: Unified, real-time object detection. Proc IEEE Conf on Computer Vision and Pattern Recognition (CVPR) 2016: 779-788.
[22] Zhou X, Wang D, Krähenbühl P. Objects as points. arXiv Preprint. 2019. Source: <http://arxiv.org/abs/1904.07850>.
[23] Tan M, Le Q. EfficientNet: Rethinking model scaling for convolutional neural networks. Int Conf on Machine Learning (ICML) 2019: 6105-6114.
[24] Chollet F. Xception: Deep learning with depthwise separable convolutions. IEEE Conf on Computer Vision and Pattern Recognition (CVPR) 2017: 1251-1258.
[25] Nguyen H-H, Tran DN-N, Jeon JW. Towards real-time vehicle detection on edge devices with Nvidia Jetson TX2. 2020 IEEE Int Conf on Consumer Electronics – Asia (ICCE-Asia) 2020: 1-4.
[26] Song S, Jing J, Huang Y, Shi M. EfficientDet for fabric defect detection based on edge computing. J Eng Fibers Fabr 2021; 16: 1-13.
[27] Abdulganeev R, Lavrenov R, Safin R, Bai Y, Magid E. Door handle detection modelling for Servosila Engineer robot in Gazebo simulator. 2022 Int Siberian Conf on Control and Communications (SIBCON) 2022: 1-4.
[28] Lyu S, Li R, Zhao Y, Li Z, Fan R, Liu S. Green citrus detection and counting in orchards based on YOLOv5-CS and AI edge system. Sensors 2022; 22(2): 576.
[29] ultralytics/yolov5. 2020. Source: <https://github.com/ultralytics/yolov5>.
[30] Wang C-Y, Bochkovskiy A, Liao H-YM. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. arXiv Preprint. 2022. Source: <https://arxiv.org/abs/2207.02696>.
[31] ultralytics/ultralytics. 2023. Source: <https://github.com/ultralytics/ultralytics>.
[32] Gillani IS, et al. Yolov5, Yolo-x, Yolo-r, Yolov7 Performance comparison: A survey. 8th Int Conf on Artificial Intelligence and Fuzzy Logic System (AIFZ 2022) 2022. DOI: 10.5121/csit.2022.121602.
[33] Nguyen H-V, Bae J-H, Lee Y-E, Lee H-S, Kwon K-R. Comparison of pre-trained YOLO models on steel surface defects detector based on transfer learning with GPU-based embedded devices. Sensors 2022; 22(24): 9926.
[34] Xia H, Yang B, Li Y, Wang B. An improved CenterNet model for insulator defect detection using aerial imagery. Sensors 2022; 22(8): 2850.
[35] Jacob B, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. Proc IEEE/CVF Conf on Computer Vision and Pattern Recognition (CVPR) 2018: 2704-2713.
[36] Wu H, Judd P, Zhang X, Isaev M, Micikevicius P. Integer quantization for deep learning inference: Principles and empirical evaluation. arXiv Preprint. 2020. Source: <http://arxiv.org/abs/2004.09602>.
[37] Cantero D, Esnaola-Gonzalez I, Miguel-Alonso J, Jauregi E. Benchmarking object detection deep learning models in embedded devices. Sensors 2022; 22(11): 4205.
[38] Lin T-Y, et al. Microsoft COCO: Common objects in context. In Book: Fleet D, Pajdla T, Schiele B, Tuytelaars T, eds. Computer Vision--ECCV 2014. Pt V. Cham, Switzerland: Springer International Publishing Switzerland; 2014: 740-755.
[39] Paszke A, et al. PyTorch: An imperative style, high-performance deep learning library. NIPS'19: Proc 33rd Int Conf on Neural Information Processing Systems 2019: 8024-8035.
[40] Han H, Siebert J. TinyML: A systematic review and synthesis of existing research. 2022 Int Conf on Artificial Intelligence in Information and Communication (ICAIIC) 2022: 269-274.
[41] Kurtz M, et al. Inducing and exploiting activation sparsity for fast inference on deep neural networks. Int Conf on Machine Learning 2020: 5533-5543.
[42] Kamath V, A R. Performance analysis of the pretrained EfficientDet for real-time object detection on Raspberry Pi. 2021 Int Conf on Circuits, Controls and Communications (CCUBE) 2021: 1-6.
[43] Konaite M, Owolawi PA, Mapayi T, Malele V, Odeyemi K, Aiyetoro G, Ojo JS. Smart hat for the blind with real-time object detection using Raspberry Pi and TensorFlow lite. Proc Int Conf on Artificial Intelligence and Its Applications (icARTi '21) 2021: 6.

Authors' information

Artur Zagitov (b. 2001), currently is an undergraduate student of Institute of Information Technology and Intelligent Systems in Kazan Federal University. Research interests are computer vision and machine learning.
E-mail: *artazagitov@gmail.com*

Elvira Chebotareva (b. 1983), currently an associate professor in Institute of Information Technology and Intelligent Systems at Kazan Federal University, Russia. Research interests are intelligent robotic systems, mobile robotics, collaborative robotics, computer vision applications in robotics. E-mail: *elvira.chebotareva@kpfu.ru*

Alexander Toschev (b. 1989), currently an assistant professor in Institute of Information Technology and Intelligent Systems at Kazan Federal University, Russia. Research interests are artificial intelligence, machine cognition, and machine learning. E-mail: *atoschev@kpfu.ru*

Evgeni Magid (b. 1975), currently a full professor, a Head of Intelligent Robotics department and a Head of Laboratory of Intelligent Robotic Systems (LIRS) at Kazan Federal University, Russia. A full professor at HSE University, Russia. Senior IEEE member. Previously he worked at University of Bristol, UK; Carnegie Mellon University, USA; University of Tsukuba, Japan; National Institute of Advanced Industrial Science and Technology, Japan. He earned his Ph.D. degree from University of Tsukuba, Japan. He authors over 270 publications. Research interests are mobile robotics, path planning, search and rescue robotics, human robot interaction, medical robotics, heterogeneous robotic teams, image processing, and computer vision. E-mail: *magid@it.kfu.ru*

Code of State Categories Scientific and Technical Information (in Russian – GRNTI)): 28.23.15
Received May 10, 2023. The final version – August 3, 2023.