







Faster YOLO-LITE: Faster Object Detection on Robot and Edge Devices

ZhengBai Yao , Will Douglas , Simon O’Keeffe , and Rudi Villing^(✉) 

Department of Electronic Engineering, Maynooth University, Maynooth, Ireland
rudi.villing@mu.ie

Abstract. Mobile robots and many edge AI devices have a need to trade off computational power against power consumption, battery size, and time between charges. Consequently, it is common for such devices to have significantly less computational power than the powerful GPU-based systems typically used to train and evaluate deep neural networks. Object detection is a key aspect of visual perception for robots and edge devices but popular object detection architectures that run fastest on GPU based systems or that are designed to maximize mAP with large input image sizes may not scale well to edge devices. In this work we evaluate the latency and mAP of several model architectures from the YOLO and SSD families on a range of devices representative of robot and edge device capabilities. We also evaluate the effect of runtime framework and show that some unexpected large differences can be found. Based on our evaluations we propose new variations of the YOLO-LITE architecture which we show can provide increased mAP at reduced latency.

Keywords: Deep learning · Object detection · Convolutional neural network · Embedded system · Real-time performance · Edge AI

1 Introduction

Essentially all training and a large majority of the inference and evaluation of deep-learning models reported in the literature is based on powerful GPU-based systems deployed locally or in the cloud. However, mobile robots and many edge AI devices need to trade off computational power against power consumption, battery size, and time between charges. For this reason, such devices are usually much less computationally powerful than those used to train and test deep neural networks and frequently they might not have a GPU. One solution to this problem might be for the edge device to do minimal processing and instead offload data for processing to the cloud [1]. However, this solution has its own issues which include data privacy, the cost of communication, and processing latency, among others. Therefore, there has recently been a trend towards deploying more sophisticated AI functionality in edge devices, including mobile robots.

In this work, we are particularly focused on object detection, a branch of computer vision AI focused on identifying and locating objects within an image. In mobile robotic applications, object detection may be used to identify and locate semantically meaningful

features of the scene including objects to be grasped or manipulated, objects whose location within the map should be remembered, people and obstacles to be approached or avoided, etc. For a review of object detection in general, see [2].

A review of the literature indicates that most new developments in object detection and consequent comparative evaluations focus on increasing the mean average precision (mAP) metric for the COCO [3] or VOC 2007 and 2012 datasets [4, 5]. To achieve small percentage gains the size of neural network models has increased dramatically and real time performance is almost always quoted for GPU based systems only. There has been relatively less attention paid to object detection models that perform well on less powerful hardware using only the CPU. Additionally, most evaluations use general datasets with many object classes, but edge AI applications are often specialized and may only require a small number of classes leading to wasted representational capability (and power consumption) if large models are applied to the problem.

Therefore, this work focuses on deep neural network architectures for object detection using CPU only processing that can potentially achieve real-time video capability for which we have chosen a guideline threshold at 50 ms (or 20 FPS). The main contributions of this paper are as follows:

- We release our SPL Object Detection Dataset V2 as described in Sect. 4.1.
- We evaluate mAP and inference latency for several object detection models on a variety of representative hardware platforms.
- We also examine the effect of inference engine selection on latency and any interactions with the underlying hardware.
- Finally, we propose and evaluate two modified YOLO-LITE architectures [6].

The remainder of this paper is organized as follows. Section 2 presents some related work, particularly focusing on the model architectures. Section 3 introduces the modified YOLO-LITE architectures. Section 4 details the experimental setup used to conduct the evaluations while Sect. 5 presents the results and related discussion. Our conclusions and future work may be found in Sect. 6.

2 Related Work

The two most well known families of object detection models available today are those based on the Single Shot Multi-box Detector (SSD) architecture [7] and those derived from YOLO [8–11]. In their default configurations both these models are rather heavy-weight and real-time performance is usually reported for high-end hardware such as a Titan X GPU.

SSD has been primarily optimized for speed using the MobileNetV1 and MobileNetV2 backbones [12, 13] and generally there are relatively few variations of this model available, though some exist (e.g. [14]). YOLO, on the other hand, has attracted a lot of interest and resulted in many variations (e.g. [6, 15]), perhaps because of its simpler architecture (at least up to YOLOv2). Additionally, a number of studies find that YOLO models outperform SSD in terms of latency and mAP. Such comparisons tend not to use the MobileNet V1 or V2 versions of SSD (giving a latency disadvantage) and such

comparisons normally use GPU based systems. Therefore there is a need to compare these model families in an edge device or embedded system environment. In this work we are particularly interested to evaluate lightweight or faster variants of YOLO and SSD, including YOLO-LITE [6] and YOLO with a MobileNet family backbone (e.g. [16]).

Many embedded systems require an inference framework optimized for CPU based operation and that is our focus here. The most well known inference frameworks include TensorFlow Lite [17], OpenVINO [18], and ONNX runtime. ONNX runtime uses some of the same components as OpenVINO so was not considered for evaluation here. In the RoboCup community there are two other inference frameworks of relevance. CompileNN [19] is a JIT compiler which compiles neural network models at runtime into machine code that performs inference. DCG, is an unreleased code generator from the Nao Devils Dortmund RoboCup SPL team which generates a cpp file from a keras model. The cpp file has no dependencies outside the standard libraries. Both CompileNN and DCG make it particularly easy to integrate neural network implementation with custom embedded applications.

The Coral Edge TPU [20] is a USB based accelerator that has been purpose built to run machine learning models on the edge. It operates as a coprocessor to the system it is connected to and can greatly speed up inference on slower devices.

3 New YOLO-LITE Based Models

YOLO-LITE is a simple architecture (with just seven convolutional layers) that is relatively easy to modify but preliminary results indicated it was not the fastest, despite its small size. We observed that MobileNetV1-YOLOv4 offered the best trade-off between mAP and latency while MobileNetV2-SSD had the highest overall mAP. We identified several design decisions in YOLO-LITE which differed from the two better performing models: (1) the use of standard convolutions throughout; (2) the use of Max Pooling layers only after each convolution has been performed at full resolution; (3) the exclusion of Batch Normalization layers; and (4) use of the Leaky ReLU activation function.

Based on this analysis we developed two scalable modifications of the YOLO-LITE architecture for inclusion in our experimental evaluations. Figure 1 shows the key blocks used in the original YOLO-LITE and our modified versions.

3.1 YOLO-LITE-M1

The first variant of YOLO-LITE modifies the backbone of the model inspired by MobileNetV1. The number of channels per layer is based on YOLO-LITE. All convolution layers are replaced by mobile convolution (MConv) layers in which the convolution is replaced by a depthwise convolution followed by a pointwise 1×1 convolution (see Fig. 1). LeakyReLU activations are replaced by ReLU6. Max pooling layers are removed and instead the relevant convolutions are performed with a stride of 2. Finally, in contrast to YOLO-LITE, Batch Normalization layers are re-introduced.

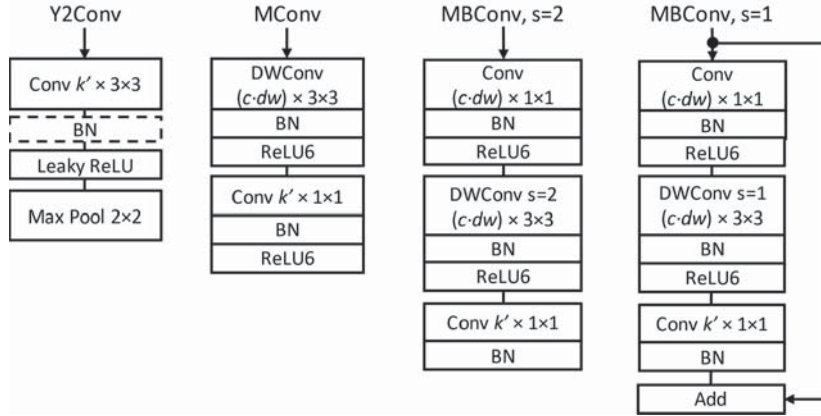


Fig. 1. Main convolutional stages used in TinyYOLOv2 and YOLO-LITE (Y2Conv, no BN in YOLO-LITE), YOLO-LITE-M1 (MConv), and YOLO-LITE-M2 (MBCConv). The number of input channels, c , is expanded by the depthwise multiplier, dw . The number of output channels, k' , is modified by the channel multiplier in accordance with (1).

Like MobileNetV1 we include a channel multiplier (they call it the width multiplier) which scales the number of kernels or output channels generated at every convolutional layer. A factor of 1 has the same number of output channels at each layer as the original YOLO-LITE architecture. Factors smaller than one reduce the number of channels (and consequently the number of parameters and the computation load) while factors larger than 1 increase them. To facilitate efficient SIMD processing, we constrain the channels at each layer to be the nearest multiple of a tiling parameter that matches the requested channel multiplication factor. The tiling parameter defaults to 8 in this work. In other words the actual output channels, k' , is calculated as:

$$k' = \text{round}(k \cdot cm, \text{tile}) \quad (1)$$

MobileNetV2 bottleneck layers introduce the concept of an expansion factor such that the number of channels in an input layer is first expanded before applying a depthwise separable convolution. The MobileNetV2 authors found that this increased the expressiveness of the network. In YOLO-LITE-M1 we use a simplification of this idea, controlled by a depthwise multiplier which determines the number of output channels for each input channel. When the depthwise multiplier is 1, this is a standard depthwise separable convolution. When it is greater than 1, the depthwise output is expanded relative to the input and each input layer is convolved depthwise with multiple independent filter kernels, giving the expanded output more representative capability. In this simplified scheme, each expanded channel is a function of one kernel convolved with one input channel whereas in the MobileNetV2 scheme, each expanded channel is a weighted sum of all the input channels.

As a final modification from the original YOLO-LITE architecture, we modify the overall depth of the network in a rather simple manner depending on the input image size. This modification is inspired both by the need to maintain a certain minimum spatial resolution in the object detection head layers and by the development of EfficientNet,

whose authors found that network depth (that is the number of layers) should be adjusted alongside other factors such as the input image size. Consequently the 5th convolution layer in the model is optional and is included only if the input image size is of size 224×224 or larger.

Table 1 depicts the final model architecture. The MConv block is as shown in Fig. 1. The Conv block is a standard 2D convolutional layer whose number of outputs are scaled by the channel multiplier, cm , in accordance with (1). YOLOChannels is calculated according to:

$$YOLOChannels = anchors \times (classes + 5) \quad (2)$$

Table 1. YOLO-LITE-M1 architecture (for nominal $224 \times 224 \times 3$ input) where s is the stride, cm is the channel multiplier and dw is the depthwise expansion multiplier which is scalable for most stages but fixed at 1 for stage 2 of the architecture. Stage 5 is excluded for inputs smaller than 224×224 .

Stage	Input size (for $cm = 1$)	Operator(s)
1	$224 \times 224 \times 3$	Conv(16, 3×3 , $s = 2$), BN, ReLU6
2	$112 \times 112 \times 16$	MConv(32, $s = 2$, cm , $dw = 1$)
3	$56 \times 56 \times 32$	MConv(64, $s = 2$, cm , dw)
4	$28 \times 28 \times 64$	MConv(128, $s = 2$, cm , dw)
5 (optional)	$14 \times 14 \times 128$	MConv(128, $s = 2$, cm , dw)
6	$7 \times 7 \times 128$	Conv(256, 3×3 , cm), BN, ReLU6
7	$7 \times 7 \times 256$	Conv(YOLOChannels, 1×1)

3.2 YOLO-LITE-M2

The second variant of YOLO-LITE modifies the backbone of the model based on MobileNetV2 instead of MobileNetV1. Unlike YOLO-LITE-M1, the number of output channels per stage is based more closely on the MobileNetV2 architecture than YOLO-LITE. The resulting model is much deeper than YOLO-LITE but with fewer output channels per stage. Our expectation was that a deeper but narrower model, with more expressive bottleneck layers would yield higher mAP, possibly at the expense of increased inference latency.

Similar to MobileNetV2, mobile bottleneck (MBConv) blocks are the principal elements of the design. As shown in Fig. 1, these include an extra 1×1 convolution that implements the expansion of input channels within the block in a more sophisticated manner than the MConv block. Additionally, when MBConv blocks are repeated in a cascade within a stage, the first block in the cascade implements the specified stride. Subsequent blocks in the cascade use a stride of 1 and in this case include a so-called inverted residual connection (between bottlenecks) as shown in Fig. 1.

Table 2 depicts the final model architecture. The MBCConv block is as shown in Fig. 1. Again, the Conv block is a standard 2D convolutional layer whose number of outputs may be scaled by the channel multiplier, cm , in accordance with (1).

Table 2. YOLO-LITE-M2 architecture (for nominal $224 \times 224 \times 3$ input) where s is the stride (only applied to the first block in a cascade), cm is the channel multiplier, dw is the bottleneck expansion multiplier, and n is the number of times the operation is cascaded in that stage. Stage 5 is excluded for inputs smaller than 224×224 .

Stage	Input size (for $cm = 1$)	Operator(s)	n
1	$224 \times 224 \times 3$	Conv(32, 3×3 , $s = 2$), BN, ReLU6	1
2	$112 \times 112 \times 32$	MBCConv(16, $s = 2$, cm , dw)	1
3	$56 \times 56 \times 16$	MBCConv(24, $s = 2$, cm , dw)	2
4	$28 \times 28 \times 24$	MBCConv(32, $s = 2$, cm , dw)	2
5 (optional)	$14 \times 14 \times 32$	MBCConv(64, $s = 2$, cm , dw)	3
6	$7 \times 7 \times 64$	Conv(256, 3×3 , cm), BN, ReLU6	1
7	$7 \times 7 \times 256$	Conv(YOLOChannels, 1×1)	1

4 Experimental Setup

4.1 Dataset and Training

Unlike traditional evaluations which utilize VOC (20 classes) or COCO (80 classes) datasets, the dataset used here was based on RoboCup robot soccer (4 classes) and is available for download¹. This dataset may be more representative of application specific datasets in embedded or edge AI applications which feature relatively few classes. The dataset comprised 4416 training images and 492 test images with 13178 and 1486 object instances respectively divided into four object classes: robot (5412), ball (4452), goal post (2912), and penalty spot (1888). Objects are present in a variety of sizes and can overlap. The goal post and penalty spot classes are relatively challenging for most object detectors.

For reasons not directly related to this evaluation, two different training regimes were used. MobileNetV2-SSD (at full width) and TinyYOLOv3 were both trained on a Dell XPS 8930 with an Intel i7-9700 CPU and an NVIDIA RTX2060 GPU. Both models were trained using weights that had been pre-trained on the COCO dataset. MobileNetV2-SSD was trained in TensorFlow 1.15 for 40000 iterations with a batch size of 6, resulting in 240000 images being processed in total. TinyYOLOv3 was trained in the DarkNet framework using the same batch size and number of iterations.

All other models were trained on a Dell Precision 3630 workstation with an Intel i7-8700K CPU and an NVIDIA P2000 GPU. These models did not use pre-trained weights

¹ <https://roboeireann.maynoothuniversity.ie/research/SPLObjDetectDatasetV2.zip>.

and were trained from scratch on the evaluation data set. Models were trained for up to 200 epochs with a batch size of 32 and 10% of the training data used for validation. Therefore, 793600 images were processed during training.

The dataset images are 640×480 RGB. We standardized on three input sizes for training and subsequent evaluation: 128×128 , 224×224 , and 416×416 . The standard training and data augmentation procedures were followed for SSD and YOLO based models.

4.2 Models

We selected models that were specifically designed to be less computationally intensive for evaluation. Two variants of MobileNetV2-SSD were evaluated. The first (using transfer learning from pre-trained weights) utilised the default MobileNetV2 width (width = 1). Preliminary investigation indicated this might be very slow, so the second variant used width = 0.25 so that it might run with lower latency.

Two variations of TinyYOLO were also included in the evaluation: TinyYOLOv4 and TinyYOLOv3. TinyYOLOv3 used transfer learning from pre-trained weights so its mAP may not be directly comparable to that of TinyYOLOv4. The implementation of TinyYOLOv4 was based on TensorFlow and Keras [16]. The learning rate was set to $1e-4$ and the only modifications involved were setting the number of classes, the class names, and the anchors to use.

MobileNetV1-YOLOv4 used width = 0.25 for the MobileNetV1 backbone. Again, preliminary work indicated that the full width backbone would run too slowly. The implementation is based on [21]. Other than the width, remaining modifications were the same as for TinyYOLOv4.

YOLO LITE was originally developed for the DarkNet/DarkFlow framework. We ported this to Keras using the general structure of the TinyYOLOv4 implementation. Learning rate, classes, and anchors were set to match the other YOLO family models.

YOLO-LITE-M1 was evaluated in two configurations: (1) $cm = 1$, $dw = 1$; (2) $cm = 0.5$, $dw = 4$ which might have higher mAP. Similarly YOLO-LITE-M2 was evaluated using two configurations: (1) $cm = 1$, $dw = 4$ and (2) $cm = 0.5$, $dw = 4$. MobileNetV2 originally set the expansion to 6 but we used 4 to reduce the size and maintain a number of outputs that was a multiple of 4 and should tile well.

4.3 Inference Configurations

We evaluated the models on the following hardware platforms:

- Raspberry Pi 3B with quad core Broadcom BCM2837 CPU @ 1.2 GHz running Raspian Linux 10 (Buster).
- Acer Aspire One netbook with an Intel Atom N270 CPU @ 1.6 GHz running Ubuntu 18.04.5. This is a very similar vintage and capability to the Atom Z530 CPU in the Softbank Nao V5 robot, but more convenient to use for this evaluation. Notably, unlike other systems in this list, this is a 32-bit processor.

- Softbank Nao V6 with a quad core Intel Atom E3485 CPU @ 1.9 GHz
- Latitude 7400 notebook with an Intel i7-8665U CPU @ 1.9 GHz running Ubuntu 18.04.5 (potentially representative of higher end or more modern robotic systems).

Including the i7-8665U gives some insight into differences that may appear due to more modern instruction sets (for example the inclusion of AVX2 and FMA) and also gives insight into the difference between machines used for development (even those without a GPU) and less powerful CPUs often used in edge and mobile robot devices.

Inference was evaluated using (1) TensorFlow Lite version 2.5.0 or built from source (as of 8 April 2021); (2) CompileNN, built from source (as of 6 April 2021); (3) Nao Devils DCG code generator (unreleased version as of 14 April 2021); (4) Intel OpenVINO developer package, version 2021.3.394; and (5) the Coral Edge TPU Accelerator version 1.0. Table 3 lists the configurations evaluated.

Table 3. Hardware and inference framework combinations evaluated. Note: fp = floating point, q8 = quantized 8-bit integer, NT = not tested, NC = not compatible, NR = tested but not reported as times were worse than floating point

Hardware	tf-lite-fp	tf-lite-q8	tf-lite + Coral	OpenVINO	CompileNN	DCG
RPi3B	Y	Y	Y	NT	NC	NC
Atom N270	Y	NR	NC	NC	Y	Y
NaoV6	Y	NR	Y	Y	Y	Y
i7-8665U	Y	NR	Y	Y	Y	Y

Common benchmarking parameters were used on all systems as follows: a batch size of 1; just 1 thread for inference; at least 5 warm-up runs; and 100 inference runs averaged for latency measurement. Each framework was benchmarked using its own benchmark tool, namely `benchmark_model` for TensorFlow Lite, `Benchmark` for CompileNN, `benchmark_app.py` for OpenVino, and the embedded benchmarking code in DCG generated cpp files.

Perhaps surprisingly, when applied to DCG cpp files, we found that clang (6.0.0) produced a binary that ran 20% faster than code compiled with g++ (7.5.0) using the same optimisation settings.

5 Results and Discussion

5.1 Model Architecture and Input Image Size

The effect of model architecture and input image size was evaluated using all models with the one inference framework (TensorFlow Lite) available on all hardware platforms. On the NaoV6, Fig. 2 shows that just two of the models attain our real-time target and both have an mAP which is much lower than the maximum achieved. MobileNetV2-YOLOv4 and YOLO-LITE both demonstrate significant mAP gains for moderate increase in

latency, suggesting that these models are the most promising for further development and tuning. MobileNetV2-SSD-0.25 has the best overall mAP with input size 224 and 416 and is only beaten by MobileNetV2-SSD at size 128.

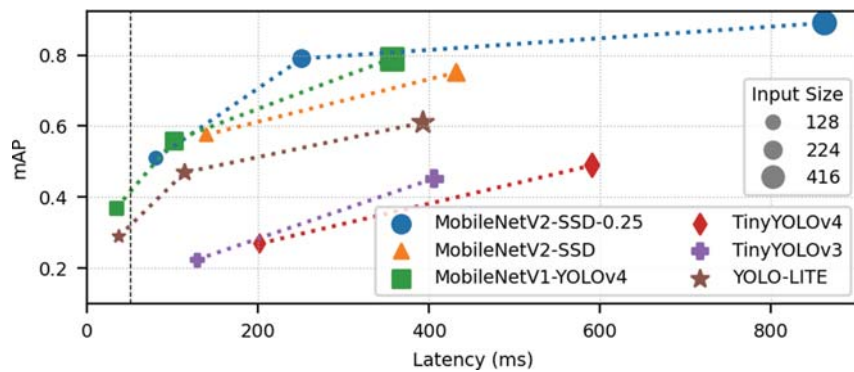


Fig. 2. Latency and mAP in relation to object detection model and input image size evaluated on the NaoV6 (Atom E3845) using the TensorFlow Lite runtime. Very long latencies associated with input size 416 are not shown for presentation reasons.

Latencies for the i7 platform follow a similar pattern to the NaoV6, but are between 6 and 14 times shorter, and are therefore not shown. The situation was broadly similar on the older Atom N270 and the Raspberry Pi, but the latencies are longer as shown in Fig. 3. The latency lead of MobileNetV1-YOLOv4 over YOLO-LITE is more pronounced here. This would suggest that the speedup advantage of the MobileNet family of architectures is even more important on these platforms.

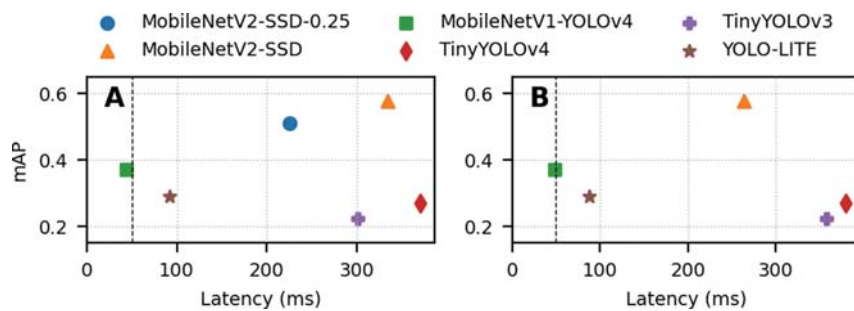


Fig. 3. Object detection latency on Atom N270 and Raspberry Pi 3B with an input image size of 128×128 . Larger input sizes yielded much longer latencies and are consequently not shown.

5.2 Choice of Inference Framework, Quantization, and Accelerator Support

To compare inference frameworks, we selected two unrelated models that were suitable for quantization: an SSD model and a YOLO based model. Figure 4 shows the latency

results for YOLO-LITE (second fastest overall and easy to quantize) and MobileNetV2-SSD (well supported for quantization) across the range of hardware and inference framework combinations presented in Table 3.

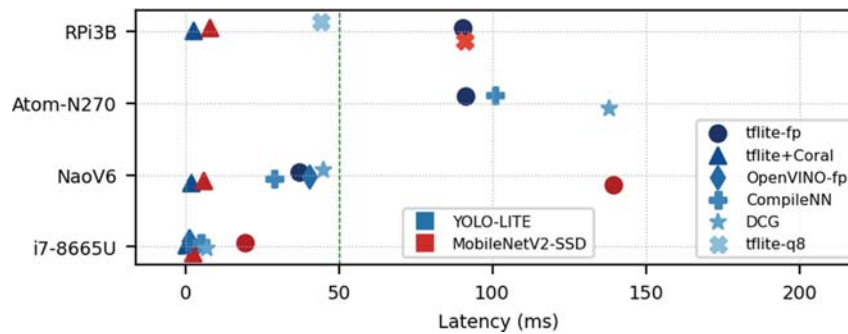


Fig. 4. Object detection latency on multiple hardware platform and inference runtime combinations for two different network architectures. Latencies longer than 200 ms excluded for presentation reasons.

Considering speedup relative to the TensorFlow Lite floating point on each platform we can make some observations. For floating point inference, OpenVINO is fastest on the i7 (1.78 \times), CompileNN is fastest on the NaoV6 (1.28 \times), and TensorFlow Lite is fastest on the Atom N270 and Raspberry Pi. OpenVINO likely makes better use of the more powerful SIMD instructions of the i7. DCG yielded latencies that were longer than TensorFlow Lite on the Atom N270 (0.66 \times) and the NaoV6 (0.83 \times). However it has the advantage of being very easy to integrate with an application having essentially no non-standard dependencies.

On the Raspberry Pi, quantized models demonstrated a useful speed up (2–2.9 \times). However, this was not replicated on Intel CPUs where quantized models had larger latency than floating point models. TensorFlow Lite is better optimized for the ARM NEON instruction set and it appears that the XNNPACK delegate does not (yet) sufficiently optimize quantized models. OpenVINO does provide support for quantized models but the process of creating them is much more involved than with TensorFlow Lite and was not completed for this evaluation.

The Coral Edge TPU accelerator demonstrated dramatic latency improvements on the Raspberry Pi (33 \times), NaoV6 (19–23 \times), and i7 (5–8 \times), but required the replacement of LeakyReLU layers with ReLU in the YOLO-LITE model. The edge TPU could not be used with the Atom N270.

5.3 YOLO LITE-M1 and YOLO-LITE-M2

Figure 5 summarizes the performance of the new models (named as model-channel multiplier-depthwise multiplier). YOLO-LITE-M2-1-4 has the best mAP and is always faster than MobileNetV1-YOLOv4 (for the same image size), while YOLO-LITE-M2-0.5-4 is always fastest overall and may have the best tradeoff between mAP and latency. The results also show that it is easy to tune the new models or speed or mAP.

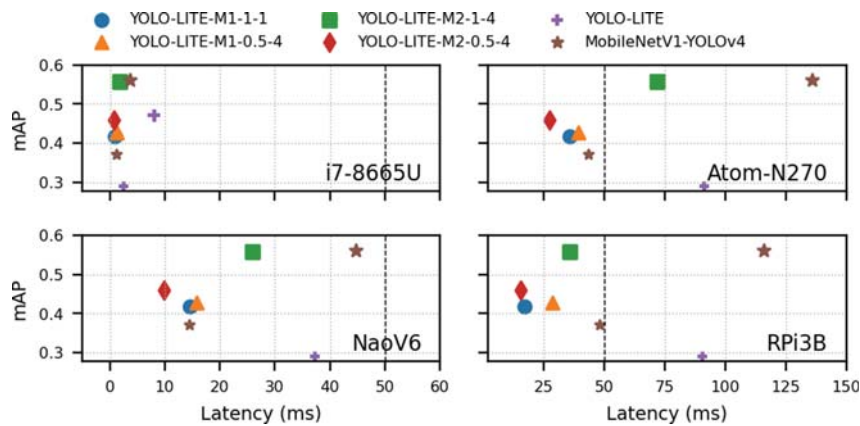


Fig. 5. Latency and mAP for new YOLO-LITE based models and the fastest models identified previously. All models were tested with an input image size of 224, but YOLO-LITE, and MobileNetV1-YOLOv4 were also tested at size 128 (shown with smaller markers).

6 Conclusions

In this work we evaluated several object detection models in the YOLO and SSD families to identify which could provide the best performance (mAP vs latency) on a variety of hardware platforms and inference frameworks. We also proposed two new models based on YOLO-LITE and included these in our evaluations.

Our results showed that our new models offer the best performance amongst the evaluation set and are easy to tune for maximizing speed or maximizing mAP. Our results also suggest that there are additional gains to be made on Intel CPUs if quantization was well supported, but current frameworks all have issues in this regard. We also observe that the Coral Edge TPU facilitates dramatic speed-ups if the model to be used can be easily quantized and only uses operations supported by the TPU. Future work may include evaluation of other accelerators and more recent model architectures.

Finally, to support further building upon this work and development of better object detectors, we have also released the RoboCup SPL object detection data set (V2) used for this evaluation.

Acknowledgements. The authors would like to acknowledge the assistance of Arne Moos and Nao Devils TU Dortmund who kindly generated C++ implementations of object detection networks using their as yet unreleased DCG code generator. We are also grateful for the assistance of Tobias Kalbitz and Nao Team HTWK who provided us with an early build of TensorFlow Lite for the Nao V6 (eventually superseded for the final evaluation).

References

1. Saha, O., Dasgupta, P.: A comprehensive survey of recent trends in cloud robotics architectures and applications. *Robotics* 7, 47 (2018)

2. Zhao, Z., et al.: Object detection with deep learning: a review. *IEEE Trans. Neural Netw. Learn. Syst.* **30**(11), 3212–3232 (2019)
3. Lin, T.-Y., et al.: Microsoft COCO: common objects in context. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) *ECCV 2014. LNCS*, vol. 8693, pp. 740–755. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10602-1_48
4. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL visual object classes challenge 2007 (VOC2007) results (2007)
5. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL visual object classes challenge 2012 (VOC2012) results (2012)
6. Huang, R., Pedoeem, J., Chen, C.: YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers. In: 2018 IEEE International Conference on Big Data (Big Data) (2018)
7. Liu, W., et al.: SSD: single shot multibox detector. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) *ECCV 2016. LNCS*, vol. 9905, pp. 21–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46448-0_2
8. Redmon, J., et al.: You only look once: unified, real-time object detection. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016)
9. Redmon, J., Farhadi, A.: YOLOv3: an incremental improvement. *arXiv e-prints arXiv:1804.02767* (2018)
10. Redmon, J., Farhadi, A.: YOLO9000: better, faster, stronger. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017)
11. Bochkovskiy, A., Wang, C.-Y., Liao, H.-Y.M.: YOLOv4: optimal speed and accuracy of object detection. *arXiv e-prints arXiv:2004.10934* (2020)
12. Howard, A.G., et al.: MobileNets: efficient convolutional neural networks for mobile vision applications. *arXiv e-prints arXiv:1704.04861* (2017)
13. Sandler, M., et al.: MobileNetV2: inverted residuals and linear bottlenecks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (2018)
14. Zhang, X., et al.: A fast SSD model based on parameter reduction and dilated convolution. *J. Real-Time Image Proc.* **18**(6), 2211–2224 (2021). <https://doi.org/10.1007/s11554-021-01108-9>
15. Zhao, H., et al.: Mixed YOLOv3-LITE: a lightweight real-time object detection method. *Sensors (Basel Switz.)* **20**(7), 1861 (2020)
16. Bubbliiiiing: YOLOV4-tiny: the realization of you only look once-tiny target detection model in Keras (2021)
17. Authors, T.: TensorFlow for Mobile & IoT. <https://www.tensorflow.org/lite>. Accessed 2021
18. Authors, O. OpenVINO Toolkit Overview. <https://docs.openvino toolkit.org/latest/index.html>. Accessed 2021
19. Thielke, F., Hasselbring, A.: A JIT compiler for neural network inference. In: Chalup, S., Niemueller, T., Suthakorn, J., Williams, M.A. (eds.) *RoboCup 2019. LNCS*, vol. 11531, pp. 448–456. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-35699-6_36
20. coral.ai. USB Accelerator datasheet. <https://coral.ai/docs/accelerator/datasheet/>. Accessed 2021
21. Bubbliiiiing: YOLOV4: you only look once object detection model - modified mobilenet series backbone network - realization in Keras (2021)