# Split Computing for Complex Object Detectors: Challenges and Preliminary Results

Yoshitomo Matsubara
University of California, Irvine
yoshitom@uci.edu

Marco Levorato
University of California, Irvine
levorato@uci.edu

## ABSTRACT

Following the trends of mobile and edge computing for DNN models, an intermediate option, split computing, has been attracting attentions from the research community. Previous studies empirically showed that while mobile and edge computing often would be the best options in terms of total inference time, there are some scenarios where split computing methods can achieve shorter inference time. All the proposed split computing approaches, however, focus on image classification tasks, and most are assessed with small datasets that are far from the practical scenarios. In this paper, we discuss the challenges in developing split computing methods for powerful R-CNN object detectors trained on a large dataset, COCO 2017. We extensively analyze the object detectors in terms of layer-wise tensor size and model size, and show that naive split computing methods would not reduce inference time. To the best of our knowledge, this is the first study to inject small bottlenecks to such object detectors and unveil the potential of a split computing approach.

## CCS CONCEPTS

• **Computing methodologies** → **Computer vision**; • **Information systems** → **Computing platforms**.

## KEYWORDS

Object detection, Split computing, Head network distillation

**ACM Reference Format:**
Yoshitomo Matsubara and Marco Levorato. 2020. Split Computing for Complex Object Detectors: Challenges and Preliminary Results. In *4th International Workshop on Embedded and Mobile Deep Learning (EMDL '20), September 21, 2020, London, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3410338.3412338



*EMDL '20, September 21, 2020, London, United Kingdom*

ACM ISBN 978-1-4503-8073-7/20/09.
https://doi.org/10.1145/3410338.3412338

## 1 INTRODUCTION

Along with the rapid evolution of computing devices, deep learning approaches have been widely studied to train powerful machine learning models, and many of such models achieve state-of-the-art performance in various tasks such as vision and natural language processing. Such models, however, are often too complex to be executed on mobile devices, which have a severely constrained computational capacity. To address this critical problem, the research community proposed two key strategies: model compression and edge computing. In the former approach, complex models are simplified, *e.g.*, by knowledge distillation [6] and model pruning and quantization [20, 27]. In the latter approach [1, 13, 15], mobile devices offload the execution of computationally expensive models to powerful (edge) computers located at the network edge. Clearly, edge computing requires to wirelessly transport the input data and model outcome on the link connecting the mobile device to the edge computer.

Recently, an intermediate option, namely *split Deep Neural Network* (DNN) or *split computing*, has been attracting a considerable interest [2, 3, 8, 10, 11, 16, 24, 26]. Many of such methods literally split DNN models into head and tail portions, which are executed by the mobile device and edge computer, respectively. Note that in this case, instead of the input data, the tensor produced by the head model should be transported to the edge computer. More sophisticated approaches [3, 8, 16, 24] modify the architecture of the model itself to (a) reduce the size of the tensor to be transferred, and (b) reduce the computing load assigned to the mobile device. Intuitively, the two features mentioned above would prove essential in enabling effective task offloading in challenged settings (*e.g.*, low capacity of the wireless channel and low computing capacity at the mobile device).

Although promising, as discussed in detail in Section 3.2, most of the existing split DNN approaches are either not evaluated [2] or evaluated only in simple classification tasks [3, 8, 16, 24, 26], such as on miniImageNet, Caltech 101, CIFAR -10, and -100 datasets. The goal of this paper is to discuss the several technical challenges in achieving effective DNN splitting for edge computing in one of the most difficult computing tasks, that is, object detection. We specifically consider Faster and Mask R-CNNs [4, 22].

Our extensive module-wise model analysis indicates that naive splitting strategies would fail to provide any improvements, where performance is measured in total inference time, compared to mobile and edge computing. Then, we propose to redesign the object detectors to introduce small bottlenecks whose output tensor sizes are significantly smaller than that of the input layer. To the best of our knowledge, this is the first work that discusses split DNN approaches for such powerful object detectors providing 1) benchmark results on a well-known object detection dataset, COCO 2017, and 2) a thorough illustration of the tradeoff between bottleneck tensor size and detection performance.

As discussed in detail in Section 3.2, the complex structure of CNN-based object detectors poses unique challenges in designing effective splitting approaches. For instance, as the models branch to provide intermediate outputs to later modules, splitting in later layers would require to send multiple tensors, which results in an increased amount of data to be transferred. Our design, then, places the bottleneck early in the model, before the first branching. However, this strategy requires an effective re-design of the network, as the first layers tend to amplify the input, rather than compressing it. The bottlenecks we designed reduced the tensor size by about 80.3 – 93.4% compared to pure edge computing (where the input tensor is transmitted), and the aggressively small bottlenecks resulted in significant loss of the detection performance. Based on our study, it is apparent that there is a strong need for efficient training and compression strategies (*e.g.*, quantization) to achieve effective splitting in a wide range of settings and conditions. The source code and trained models' weights used in this study are released [1] to enable such further studies and developments.

## 2 CNN-BASED OBJECT DETECTORS

In this section, we discuss the architecture of recent object detectors based on Convolutional Neural Network (CNN) that achieve state-of-the-art detection performance. These CNN-based object detectors are often categorized into either single-stage or two-stage models. Single-stage models, such as YOLO and SSD series [14, 21], are designed to directly predict bounding boxes and classify the contained objects. Conversely, two-stage models [4, 22] generate region proposals as output of the first stage, and classify the objects in the proposed regions in the second stage. In general, single-stage models have smaller execution time due to their lower overall complexity compared to two-stage models, that are superior to single-stage ones in terms of detection performance.

Recent object detectors, *e.g.*, Mask R-CNN and SSD [4, 23], adopt state-of-the-art image classification models, such as ResNet [5] and MobileNet v2 [23], as *backbone*. The main role of backbones in detection models pretrained on large image datasets, such as the ILSVRC dataset, is feature extraction. As illustrated in Figure 1, such features include the outputs of multiple intermediate layers in the backbone. All the features are fed to complex modules specifically designed for detection tasks, *e.g.*, the feature pyramid network [12], to extract further high-level semantic feature maps at different scales. Finally, these features are used for bounding box regression and object class prediction.

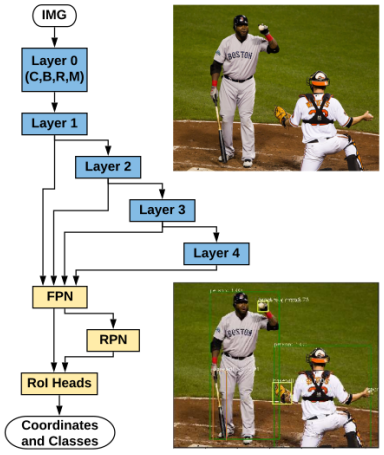


**Figure 1: R-CNN with ResNet-based backbone. Blue modules are from its backbone model, and yellow modules are of object detection. C: Convolution, B: Batch normalization, R: ReLU, M: Max pooling layers.**

In this study, we focus our attention on state-of-the-art two-stage models. Specifically, we consider Faster R-CNN and Mask R-CNN [4, 22] pretrained on the COCO 2017 datasets. Faster R-CNN is the strong basis of several 1st-place entries [5] in ILSVRC and COCO 2015 competitions. The model is extended to Mask R-CNN by adding a branch for predicting an object mask in parallel with the existing branch for bounding box recognition [4]. Mask R-CNN not only is a strong benchmark, but also a well-designed framework, as it easily generalizes to other tasks such as instance segmentation and person keypoint detection.

## 3 CHALLENGES AND APPROACHES

We discuss challenges in deploying CNN-based object detectors in three different scenarios: mobile, edge, and split computing. We use total inference time (including the time needed to transport the data over the wireless channel) and object detection performance as performance metrics.

### 3.1 Mobile and Edge Computing

In mobile computing the mobile device executes the whole model, and the inference time is determined by the complexity of the model and local computing power. Due to

[1] https://github.com/yoshitomo-matsubara/hnd-ghnd-object-detectors

limitations in the latter, in order to have tolerable inference time, the models must be simple and lightweight. To this aim, one of the main approaches is to use human-engineered features instead of those extracted from stacked neural modules. For instance, Mekonnen *et al.* [19] propose an efficient HOG (Histogram Oriented Gradients) based person detection method for mobile robotics. Designing high-level features of human's behavior on touch screen, Matsubara *et al.* [18] propose distance/SVM-based one-class classification approaches to screen unlocking on smart devices in place of password or fingerprint authentications. In recent years, however, deep learning based methods have been outperforming the models with human-engineered features in terms of accuracy. For image classification tasks, MobileNets [7, 23] and MNasNets [25] are examples of models designed to be executed on mobile devices, while providing moderate classification accuracy. Corresponding lightweight object detection models are SSD [14] and SSDLite [23]. Techniques such as model pruning, quantization and knowledge distillation [6, 20, 27] can be used to produce lightweight models from larger ones.

Table 1 summarizes the performance of some models trained on the COCO 2014 minival dataset as reported in the TensorFlow repository. [2] Obviously, the SSD series object detectors with MobileNet backbones outperform the Faster R-CNN with ResNet-50 backbone in terms of inference time, but such lightweight models offer degraded detection performance. Note that in the repository, the COCO 2014 minival split is used for evaluation, and the inference time is measured on a machine with one NVIDIA GeForce GTX TITAN X, which is clearly not suitable to be embedded in a mobile device. Also, the values reported in Table 1 are given for models implemented with the TensorFlow framework with input images resized to $600 \times 600$, though some of the models in the original work such as Faster R-CNN [22] use different resolutions. In general, the classification/detection performance is often compromised in mobile computing due to their limited computing power.

Table 2 highlights that it would be impractical to deploy some of the powerful object detectors on weak devices. Specifically, on Raspberry Pi 4, R-CNN object detectors with even the smallest backbone in the family took 20–30 seconds for prediction per a resized image of which shorter side resolution is 800 pixels, following [4, 22]. For both Faster and Mask R-CNNs, the execution time on NVIDIA Jetson TX2 and a desktop machine with an NVIDIA RTX 2080 Ti is sufficiently small to support real-time applications .

Different from mobile computing, the total inference time in edge computing is the sum of the execution time in Table 2 and the communication time needed to transfer data from the mobile device to the edge computer (*e.g.*, Raspberry Pi 4 and the desktop machine, respectively). If the prediction results are to be sent back to the mobile device, a further communication delay term should be taken into account, although outcomes (*e.g.*, bounding boxes and labels) typically have a much smaller size compared to the input image. As discussed in [10, 16], the delay of the communication from mobile device to edge computer is a critical component of the total inference time, which may become dominant in some network conditions, where the performance of edge computing may suffer from a reduced channel capacity.

**Table 1: Mean average precision (mAP) on COCO 2014 minival dataset and running time on a machine with an NVIDIA GeForce GTX TITAN X. [2]**

| TensorFlow model | mAP | Speed [sec] |
|---|---|---|
| SSDLite with MobileNet v2 | 0.220 | 0.027 |
| SSD with MobileNet v3 (Large) | 0.226 | N/A* |
| Faster R-CNN with ResNet-50 | 0.300 | 0.0890 |

* Reported speed was measured on a different device

**Table 2: Running time [sec/image] of Faster and Mask R-CNNs with different ResNet models.**

| | Backbone with ResNets | -18 | -34 | -50 | -101 |
|---|---|---|---|---|---|
| Faster R-CNN | Raspberry Pi 4 Model B | 27.73 | 23.40 | 26.14 | 35.16 |
| | NVIDIA Jetson TX2 | 0.617 | 0.743 | 0.958 | 1.26 |
| | Desktop + 1 GPU | 0.0274 | 0.033 | 0.0434 | 0.0600 |
| Mask R-CNN | Raspberry Pi 4 Model B | 18.30 | 23.65 | 27.02 | 34.73 |
| | NVIDIA Jetson TX2 | 0.645 | 0.784 | 0.956 | 1.27 |
| | Desktop + 1 GPU | 0.0289 | 0.0541 | 0.0613 | 0.0606 |

## 3.2 Split Computing

Split computing is an intermediate option between mobile and edge computing. The core idea is to split models into head and tail portions, which are deployed at the mobile device and edge computer, respectively. To the best of our knowledge, Kang *et al.* [10] were the first to propose to split deep neural network models. However, the study simply proposed to optimize where to split the model, leaving the architecture unaltered.

In split computing, the total inference time is sum of three components: mobile processing time, communication delay, and edge processing time. To shorten the inference time in split computing compared to those of mobile and edge computing, the core challenge is to significantly reduce communication delay while leaving a small portion of computational load on mobile device for compressing the data to be transferred to edge server. Splitting models in a straightforward way, as suggested in [10], however, does not lead to

[2] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md#coco-trained-models

an improvement in performance in most cases. The tension is between the penalty incurred by assigning a portion of the overall model to a weaker device (compared to the edge computer) and the potential benefit of transmitting a smaller amount of data. However, most models do not present "natural" bottlenecks in their design, that is, layers with a small number of output nodes, corresponding to a small tensor to be propagated to the edge computer. In fact, the *neurosurgeon* framework locates pure mobile or edge computing as the optimal computing strategies in most models.

Building on the work of Kang *et al.* [10], recent contributions propose DNN splitting methods [2, 3, 8, 11, 16, 24, 26]. Most of these studies, however, (I) do not evaluate models using their proposed lossy compression techniques [2], (II) lack of motivation to split the models as the size of the input data is exceedingly small, *e.g.*, $32 \times 32$ pixels RGB images in [8, 24, 26], (III) specifically select models and network conditions in which their proposed method is advantageous [11], and/or (IV) assess proposed models in simple classification tasks such as miniImageNet, Caltech 101, CIFAR -10, and -100 datasets [3, 8, 16, 24].

Similar to CNN-based image classification models, it is not possible to reduce the inference time of CNN-based object detectors by naive splitting methods without altering the models' architecture. This is due to the designs of the early layers of the models, which *amplify* the input data size. It would be worth noting that Matsubara et al. [16] apply a loseless compression technique, a standard Zip compression, to intermediate outputs of all the splittable layers in a CNN model, and show the compression gain is not sufficient to significantly reduce inference time in split computing.

Figure 2 illustrates this effect by showing the amplification of the data at each of core layers in Faster and Mask R-CNNs with ResNet-50, compared to the input tensor size ($3 \times 800 \times 800$). Note that these models are designed for images whose shorter side resolution is 800 pixels [4, 22]. The trends confirm that there is no splitting point (below blue line) in any of the early layers. Therefore, naive splitting does not result in any gain in terms of communication delay. We note that different from the Faster R-CNN model, the output tensor of the RoI Heads in the Mask R-CNN model is significantly larger than the input tensor. As the model emits not only bounding boxes and object classes, but also pixel-level masks for segmentation, the last tensor size surges in Figure 2 (green dotted line), but the general trend looks the same with Faster R-CNN model when using bounding boxes and object classes only.

A promising, but challenging, solution to reduce the inference time in challenged networks is to introduce a small *bottleneck* within the model, and split the model at that layer [16]. In the following section, we discuss bottleneck injection for CNN-based object detectors, specifically Faster and Mask R-CNNs, and present preliminary experimental results supporting this strategy.

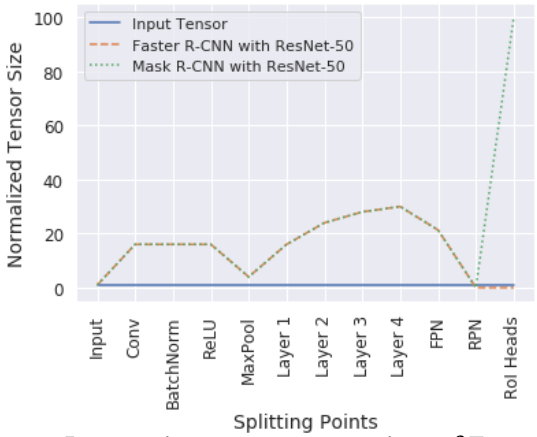


**Figure 2: Layer-wise output tensor sizes of Faster and Mask R-CNNs scaled by input tensor size ($3 \times 800 \times 800$).**

## 4 EXPERIMENTS

One of the core challenges of bottleneck injection in R-CNN object detectors is that the bottleneck needs to be introduced in early stages of the detector compared to image classification models. As illustrated in Figure 1, the first *branch* of the network is after Layer 1. As a result, the bottleneck needs to be injected before the layer to avoid the need to forward multiple tensors produced by the branches (Figs. 1 and 2).

The amount of computational load assigned to the mobile device should be considered as well when determining the bottleneck placement. In fact, the execution time of the *head* model, which will be deployed on the mobile device, is a critical component to minimize the total inference time. Figures 3 and 4 depict the number of parameters of each model used for partial inference on the mobile device when splitting the model at specific modules. The reported values provide a rough estimate of the head model's complexity as a function of the splitting point.

Recall that feature pyramid network (FPN), region proposal network (RPN), and region of interest (RoI) Heads in the R-CNN models are designed specifically for object detection tasks, and all the modules before them are originally from an image classification model (ResNet models [5] in this study). Because of not only the models' branching, but the trends in these figures, it is clear that the bottleneck, and thus the splitting point, should be placed before "Layer 1".

Matsubara *et al.* [16] attempted to introduce a bottleneck in the first convolution layer of DenseNet-169 [9]. The bottleneck uses 4 output channels in place of 64 channels, so that the output tensor of the layer is smaller than the input tensor to the model. Using Caltech 101 dataset, they naively trained

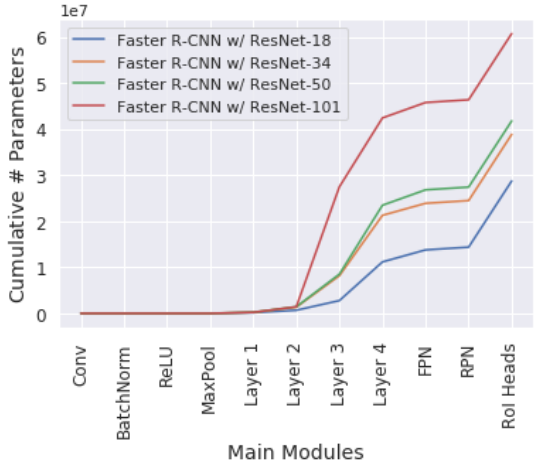


**Figure 3: Cumulative number of parameters in core modules of Faster R-CNN.**

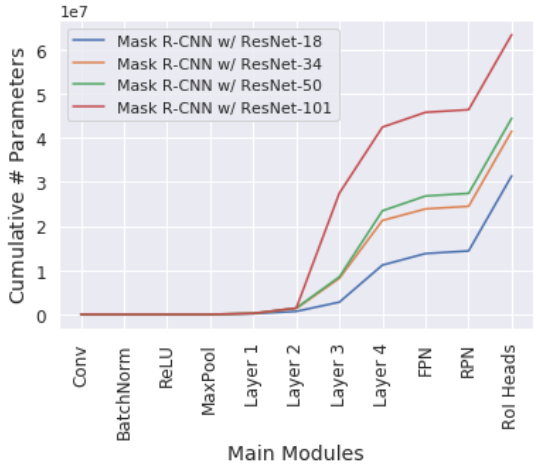


**Figure 4: Cumulative number of parameters in core modules of Mask R-CNN.**

the redesigned model, that significantly degraded classification accuracy even despite the relative low complexity of the dataset compared to the ILSVRC dataset.

Based on the analysis and results, we attempt to introduce a bottleneck to "Layer 1", that consists of multiple low-level modules such as convolution layers. Here, we redesign the layer 1 by pruning a couple of layers and adjust hyperparameters to make its output shape match that of the layer 1 in the original model. The redesigned layer 1 has a small bottleneck with $C$ output channels, a key parameter to control the balance between detection performance and bottleneck size.

Assuming that the original models are overparameterized, we *distill* the head models while injecting bottlenecks. Specifically, we use head network distillation [16], a teacher-student training scheme, only applied to the head portion to reduce training time. We treat the first layers until the layer 1 in the original detector as a teacher model, and those in

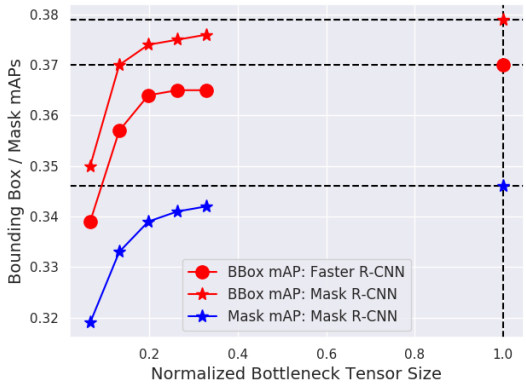


**Figure 5: Average bottleneck tensor size vs. BBox and Mask mAPs on COCO 2017 validation dataset (test split is not publicly available). Bottleneck tensor size is scaled by average input tensor size ($3 \times 874 \times 1044$).**

the redesigned detector as a student model. Note that the redesigned detector reuses all the modules and learnt parameters of the original detector except the modules until the end of the layer 1. The exact network architectures are not described in this paper due to limited space, but the code and trained model weights are released to ensure reproducible results. [1] In this study, we use pretrained Faster and Mask R-CNNs with ResNet-50 as teacher models. To the best of our knowledge, this is the first study that discusses introducing bottlenecks to CNN-based object detectors for split computing and provide experimental results.

Figure 5 illustrates the relationship between the output tensor size of the introduced bottleneck with the number of output channels $C = 3, 6, 9, 12, 15$ and bounding box / mask mAPs of the modified Faster and Mask R-CNNs. The rightmost data points on the dashed line in the figure correspond to the detection performance (mAP) of the original R-CNN models in edge computing, that is, the splitting points are at their input layers. Since in this study we do not alter the tail portion of the models, but modify and train the head portion only, we take the detection performance of the original models (on dashed line) as the upper bound performance of our modified models. It can be observed how we successfully reduce the size of the tensors to be transferred to the edge computer compared to the input tensors incurring some mAP degradation. Recall that there are no effective splitting points in the original models, as shown in Figure 2 (*i.e.*, most of the normalized tensor sizes are above 1), and our introduced bottlenecks save approximately 80.3 – 93.4% of tensor size for offloading compared to edge computing.

## 5 CONCLUSIONS AND FUTURE WORK

In this study, we discussed the challenges in deploying CNN-based object detectors in mobile devices using three key

strategies: mobile computing, edge computing, and the recently proposed split computing. We focused our discussion on two different state-of-the-art two-stage object detectors, which have no suitable "natural" splitting points, and injected small bottlenecks into the models based on the analysis we provided. While the introduced bottlenecks are smaller than the input tensors, the detection performance is sometimes significantly degraded specifically when introducing aggressively small bottlenecks. In addition to improving the detection performance, it would be necessary to assess the inference time using further compression techniques such as quantization, which we further discuss in [17].

## ACKNOWLEDGMENTS

This work was supported by the NSF grant IIS-1724331 and MLWiNS-2003237, and DARPA grant HR00111910001.

## REFERENCES

[1] Marco V Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. 2013. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *Proceedings of IEEE INFOCOM 2013*. 1285–1293.

[2] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. 2019. Cracking open the DNN black-box: Video Analytics with DNNs across the Camera-Cloud Boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*. 27–32.

[3] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. 2019. BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[4] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*. 2961–2969.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[6] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2014. Distilling the Knowledge in a Neural Network. In *Deep Learning and Representation Learning Workshop: NIPS 2014*.

[7] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE International Conference on Computer Vision*. 1314–1324.

[8] Diyi Hu and Bhaskar Krishnamachari. 2020. Fast and Accurate Streaming CNN Inference via Communication Compression on the Edge. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 157–163.

[9] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[10] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China). 615–629. https://doi.org/10.1145/3037697.3037698

[11] Guangli Li, Lei Liu, Xueying Wang, Xiao Dong, Peng Zhao, and Xiaobing Feng. 2018. Auto-tuning Neural Network Quantization Framework for Collaborative Inference Between the Cloud and Edge. In *International Conference on Artificial Neural Networks*. 402–411.

[12] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. 2017. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2117–2125.

[13] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[14] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. SSD: Single shot multibox detector. In *European conference on computer vision*. 21–37.

[15] Pavel Mach and Zdenek Becvar. 2017. Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE Communications Surveys & Tutorials* 19 (2017), 1628–1656.

[16] Yoshitomo Matsubara, Sabur Baidya, Davide Callegaro, Marco Levorato, and Sameer Singh. 2019. Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems. In *Proceedings of the 2019 MobiCom Workshop on Hot Topics in Video Analytics and Intelligent Edges*. 21–26.

[17] Yoshitomo Matsubara and Marco Levorato. 2021. Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks. In *Proceedings of the 25th International Conference on Pattern Recognition (To appear)*.

[18] Yoshitomo Matsubara, Haruhiko Nishimura, Toshiharu Samura, Hiroyuki Yoshimoto, and Ryohei Tanimoto. 2016. Screen Unlocking by Spontaneous Flick Reactions with One-Class Classification Approaches. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 752–757.

[19] Alhayat Ali Mekonnen, Cyril Briand, Frédéric Lerasle, and Ariane Herbulot. 2013. Fast HOG based person detection devoted to a mobile robot with a spherical camera. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 631–637.

[20] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. In *Sixth International Conference on Learning Representations*.

[21] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conf. on computer vision and pattern recognition*. 779–788.

[22] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in neural information processing systems*. 91–99.

[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[24] Jiawei Shao and Jun Zhang. 2020. Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.

[25] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[26] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 328–339.

[27] Yi Wei, Xinyu Pan, Hongwei Qin, Wanli Ouyang, and Junjie Yan. 2018. Quantization mimic: Towards very tiny cnn for object detection. In *Proceedings of the European Conference on Computer Vision*. 267–283.