2020 25th International Conference on Pattern Recognition (ICPR)
Milan, Italy, Jan 10-15, 2021

Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks

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

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

***Abstract*—The edge computing paradigm places compute-capable devices - edge servers - at the network edge to assist mobile devices in executing data analysis tasks. Intuitively, offloading compute-intense tasks to edge servers can reduce their execution time. However, poor conditions of the wireless channel connecting the mobile devices to the edge servers may degrade the overall capture-to-output delay achieved by edge offloading. Herein, we focus on edge computing supporting remote object detection by means of Deep Neural Networks (DNNs), and develop a framework to reduce the amount of data transmitted over the wireless link. The core idea we propose builds on recent approaches splitting DNNs into sections - namely head and tail models - executed by the mobile device and edge server, respectively. The wireless link, then, is used to transport the output of the last layer of the head model to the edge server, instead of the DNN input. Most prior work focuses on classification tasks and leaves the DNN structure unaltered. Herein, our focus is on DNNs for three different object detection tasks, which present a much more convoluted structure, and modify the architecture of the network to: (i) achieve in-network compression by introducing a bottleneck layer in the early layers on the head model, and (ii) prefilter pictures that do not contain objects of interest using a convolutional neural network. Results show that the proposed technique represents an effective intermediate option between local and edge computing in a parameter region where these extreme point solutions fail to provide satisfactory performance. The code and trained models are available at https://github.com/yoshitomo-matsubara/hnd-ghnd-object-detectors.**

***Index Terms*—object detection, head network distillation, neural filter, split computing**

I. Introduction

The real-time execution of modern data analysis algorithms requires powerful computing platforms. For instance, accurate Deep Neural Networks (DNNs) for vision tasks, *e.g.*, image classification and object detection, have an extremely large number of layers and parameters. Due to weight, cost or size constraints, mobile devices may have limited computing capacity and energy availability, which makes the execution of such algorithms challenging. The research community is exploring two main approaches to mitigate this problem: simplifying models and computation offloading. The former approach may lead to a degraded performance of the resulting models compared to the full sized ones. The latter approach is often referred to as *mobile edge computing*, which has been proposed primarily as a tool to support time-sensitive mission-critical applications such as autonomous vehicles and augmented reality [1].

However, poor channel conditions increase the time needed to deliver the input data to the edge server, thus making edge computing schemes not effective. Compression, by reducing the channel capacity needed to sustain the data transfer, can mitigate this problem and reduce the total delay. Unfortunately, while lossless compression is either computationally expensive and/or achieves limited compression gain [2], traditional lossy compression mechanisms, such as JPEG, are mainly designed for human perception. As a result, aggressive compression often degrades the final accuracy of analysis [3], [4]. Modern compression strategies, such as autoencoders [5], are computationally expensive, especially when considering information-rich signals such as images and videos.

Recent contributions [6], [7] attempt to distribute the execution of DNN models across the mobile device-edge server system to find delay-optimal "splitting" points. More specifically, the DNN model is *split* into a *head* and *tail* model, which are executed at the mobile device and edge server, respectively. Instead of the overall model input, then, the mobile device transmits over the channel the output of the head layer (a tensor). Unfortunately, most of the models, and especially those for vision tasks, tend to amplify the input in the first layers, meaning that positioning the splitting point in the first part of the DNN would result in a larger amount of data transmitted over the wireless link compared to "pure" offloading [2]. The use of later, smaller, layers as splitting point would reduce the amount of transferred data, but would also result in a large portion of the overall computing load being allocated to the weaker device in the system – the mobile device. In fact, results in [6] show that in most of the analyzed models and channel/computing settings the optimal splitting point is either at the very beginning or end of the model, that is, pure local computing or edge computing have better

This work was partially supported by DARPA under grant HR00111910001, NSF under Grant IIS-1724331 and MLWiNS-2003237, and Google Cloud Platform research credits.

978-1-7281-8808-9/20/$31.00 ©2020 IEEE

2020 25th International Conference on Pattern Recognition (ICPR) | 978-1-7281-8808-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICPR48806.2021.9412388

performance compared to naive splitting.

In this paper, we explore strategies to modify the structure of Convolutional Neural Network (CNN) models for object detection to maximize the efficiency of model splitting. The core idea is to use network distillation to introduce a bottleneck layer, that is, a layer with a small number of nodes, in the early stages of the object detection model. The model is then split at the bottleneck, thus achieving in-network compression, as a smaller tensor needs to be transmitted to the edge server.

We show that the distillation technique we propose allows the injection of extremely effective bottlenecks while modifying only the parameters of the head portion of the model, which is trained to mimic the output of the original head. We build on the results presented in [2], where we applied a similar concept to image classification tasks. The more convoluted structure of object detectors, where the output of intermediate layers is propagated to a final detection module, introduces further challenges discussed in [8] that we address by proposing a generalized loss function guiding the distillation process and applying a bottleneck quantization technique.

Additionally, we observe that while in image classification tasks all images are classified, in object detection tasks only a fraction of the images may contain objects of interest. To take advantage of this feature of the problem, we embed in the head model a small CNN whose binary output indicates whether or not the image contains objects to be analyzed by the detector. The network acts as a filter, blocking empty images while promptly returning an empty detection output, thus also reducing channel usage and server load. We demonstrate that this classifier can be efficiently attached to the first layers of the head model minimizing additional complexity by reusing a portion of the detector.

Results show that our distillation based technique achieves detection performance, measured in terms of mean Average Precision (mAP), comparable to that of state-of-the-art models. In the evaluation, we use established datasets associated with complex detection tasks *e.g.,* person keypoints detection. We demonstrate that we can significantly reduce the total time from image capture to the availability of the detector's output compared with local and edge computing in a region of parameters where these extreme-point options struggle to provide satisfactory performance. To ensure reproducibility of the experimental results and facilitate research in this important research problem, we publish all the code and model weights.[1]

II. Edge-Assisted Object Detection

We consider a scenario where a mobile device acquires images to be analyzed in real-time to support delay-sensitive vision-based applications. An edge server interconnected to the mobile device through a capacity-limited wireless channel (*e.g.*, data rate $\leq$ 10Mbps) assists the execution of the analysis algorithm. As stated earlier, in this paper we focus on object detection tasks. An example of relevant application and framework is presented in [1], where the authors focus on an edge-assisted implementation of augmented reality.

Our main objective is to reduce the time from image capture to the availability of the object detection output (in this case bounding boxes and associated labels) at the mobile device. We refer to this time as *capture-to-output delay* and denote it with T. Intuitively, in settings such as that proposed in [1], a smaller capture-to-output delay would improve tracking performance. We define in the following the composition of T in three settings: (*i*) local computing at the mobile device; (*ii*) offloading of the complete model execution to the edge server (referred to as pure offloading); and (*iii*) split computing, where the execution of the model is divided between the mobile device and the edge server.

(i) Local Computing: The total time T is the time needed to execute the entire object detection model at the mobile device. Thus, $T{=}T_{\mathrm{L}}$, where T_{L} is determined by the model complexity and computing power of the mobile device.

(ii) Pure Offloading: The total time T is the sum of two terms: T_{i} is the time needed to transfer the image to the edge server, and T_{E} is the execution time of the entire model at the edge server.

(iii) Split Computing: The total time T is the sum of three terms: T_{H} is the time needed to execute the head model at the mobile device, T_{o} is the time needed to transfer the output of the head model to the edge server, and T_{T} is the execution time of the tail model at the edge server.

Rather intuitively, the absolute and relative computing capacity of the mobile device and edge server and the channel capacity determine the value of the delay components listed above, and thus, which option is the most advantageous in terms of inference time. Clearly, a large channel capacity would decrease communication-related delay components, eventually making pure offloading a preferable option, and increasingly reducing the need for compression. A small gap in terms of computing capacity between the mobile device and the edge server would make offloading and network splitting effective only in scenarios with high channel capacity.

We emphasize that none of the above options can be declared best for all parameters. Instead, in different parameter regions different options will lead to the smallest overall capture-to-output delay. The technique proposed here realizes an intermediate option between local and edge computing where a small portion of the computing load is allocated to the local device to reduce the amount of transferred data. Clearly, this option is functional in the lower-intermediate range of channel capacity, and when the gap between the mobile device and edge server is not making the allocation of even small computing loads to the mobile device undesirable.

We remark that the latter characteristics is compatible with the scenarios created by the recent advances in miniaturization, which are bringing considerable computing power within the reach of mobile devices. Examples include the NVIDIA Jetson Nano and TX2 embedded computers. However, as illustrated in our results, even relatively low-end full-sized servers can

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

significantly reduce execution time, thus making offloading meaningful.

III. CNN-Based Detection Models

In order to better illustrate our proposed approach and the associated challenges, we first discuss the structure of state-of-the-art object detectors based on deep neural networks. CNN-based models have become the mainstream option for object detection tasks [9]–[11]. These complex object detection models are categorized into two main classes: single-stage or two-stage object detectors. Single-stage models [11], [12] are designed to directly predict bounding boxes and the corresponding object classes given an image. Conversely, two-stage models [13], [14] generate region proposals as output of the first stage, which are then classified in the second stage. In general, single-stage detection models have lower overall complexity, and thus execution time, compared to two-stage models. However, two-stage models outperforms single-stage ones in terms of detection performance.

A. Benchmark Models

In this study, we focus our attention on state-of-the-art two-stage models in resource-constrained edge computing systems. Specifically, we consider Faster R-CNN, Mask R-CNN, and Keypoint R-CNN (from torchvision) with ResNet-50 FPN [13]–[15] pretrained on the COCO 2017 datasets. Due to the limited space, more details of the models and tasks are provided in supplementary material.

B. Motivations

Clearly, mobile devices are often unable to execute these strong, but rather complex and convoluted, detectors. In order to obtain models within the reach of weak computing platforms, the research community developed an approach known as *knowledge distillation* [16]. The core idea of knowledge distillation is to build a lower complexity, "student" model, trained to mimic the output of a pretrained, higher-complexity "teacher" model. The key assumption is that large – teacher – models are often overparameterized, and can be reduced without a significant performance loss. Interestingly, it is widely known that student models trained to mimic the behavior of their teacher models significantly outperform those trained on the original training dataset [17].

Distillation has been applied to detection models [18]–[20]. For instance, Chen *et al.* [21] propose a hierarchical knowledge distillation technique to train a lightweight pedestrian detector, and distill a student ResNet-18 based R-CNN model using a ResNet-50 based R-CNN model as teacher. However, as shown in Table I, the reduction in inference time granted by smaller models is limited when using relatively capable platforms such as the NVIDIA Jetson TX2, which embeds a GPU. Importantly, we remark that smaller models achieve degraded detection performance compared to bigger models due to the complexity of the detection task.

Table II shows the total time T achieved by pure offloading when using the same models. In these results, the execution time is computed using a high-end desktop computer with Intel Core i7-6700K CPU (4.00GHz), 32GB RAM, and a NVIDIA GeForce RTX 2080 Ti as edge server, and the channel provides the relatively low, data rate of 5Mbps to the image stream. It can be seen how, in this setting, reducing the model size by distilling the whole detector [18]–[20] does not lead to substantial total delay savings, while offloading is generally advantageous compared to local computing.

TABLE I: Local computing time [sec] of detection models with different ResNet backbones on NVIDIA Jetson TX2.

Model \Backbone	RN-18	RN-34	RN-50	RN-101
Faster R-CNN	0.617	0.743	0.958	1.26
Mask R-CNN	0.645	0.784	0.956	1.27
Keypoint R-CNN	1.93	2.09	2.25	2.59

TABLE II: Pure offloading time [sec] (data rate: 5Mbps) of detection models with different ResNet backbones on a high-end machine with a NVIDIA GeForce RTX 2080 Ti.

Model \Backbone	RN-18	RN-34	RN-50	RN-101
Faster R-CNN	0.456	0.462	0.472	0.489
Mask R-CNN	0.458	0.4832	0.4904	0.4897
Keypoint R-CNN	0.469	0.473	0.481	0.498

Splitting the model while leaving its structure unaltered, as proposed by Kang *et al.* [6], is technically challenging and intuitively not advantageous due to the data amplification effect of early layers, and the need to forward the output of intermediate backbone layers (see Fig. 1) when positioning the splitting point later in the model.

IV. In-Network Neural Compression

A. Background

As discussed earlier, the weak point of pure edge computing is the communication delay: when the capacity of the channel interconnecting the mobile device and edge server is degraded by a poor propagation environment, mobility, interference and/or traffic load, transferring the model input to the edge server may result in a large overall capture-to-output delay T. Thus, in challenged channel conditions, making edge computing effective necessitates strategies to reduce the amount of data to be transported over the channel.

The approach we take herein is to modify the structure of the model to obtain in-network compression and improve the efficiency of network splitting. We remind that in network splitting, the output of the last layer of the head model is transferred to the edge, instead of the model input. Compression, then, corresponds to splitting the model at layers with a small number of nodes which generate small outputs to be transmitted over the channel. In our approach, the splitting point coincides with the layer where compression is achieved.

Unfortunately, layers with a small number of nodes appear only in advanced portions of object detectors, while early layers amplify the input to extract features. However, splitting at late layers would position most of the computational

complexity at the weaker mobile device. This issue was recently discussed in [2], [8] for image classification and object detection models, reinforcing the results obtained in [6] on traditional network splitting.

In [2], we proposed to inject *bottleneck layers*, that is, layers with a small number of nodes, in the early stages of image classification models. To reduce accuracy loss as well as computational load at the mobile device, the whole head section of the model is reduced using distillation. The resulting small model contains a bottleneck layer followed by a few layers that translate the bottleneck layer's output to the output of the original head model. Note that the layers following the bottleneck layers are then attached to the original tail model and executed at the edge server.

The distillation process attempts to make the output of the new head model as close as possible to the original head. At an intuitive level, when introducing bottleneck layers this approach is roughly equivalent to train a small asymmetric autoencoder-decoder pipeline whose boundary layer produces a compressed version of the input image used by the decoder to reconstruct the output of the head section, rather than the image. Interestingly, it is shown that the distillation approach can achieve high compression rates while preserving classification performance even is complex classification tasks.

This paper builds on this approach [2], [8] to obtain in-network compression with further improved detection performance in object detection tasks. Specifically, we generalize the head network distillation (HND) technique, and apply it to the state of the art detection models described in the previous section (Faster R-CNN, Mask R-CNN, and Keypoint R-CNN).

The key challenge originates from the structural differences between the models for these two classes of vision tasks. As discussed in the previous section, although image classification models are used as backbones of detection models, there is a trend of using outputs of multiple intermediate layers as features fed to modules designed for detection such as the FPN (Feature Pyramid Network) [22]. This makes the distillation of head models difficult, as they would need to embed multiple bottleneck layers at the points in the head network whose output is forwarded to the detectors. Clearly, the amount of data transmitted over the network would be inevitably larger as multiple outputs would need to be transferred. Additionally, we empirically show that injecting smaller bottlenecks resulted in degraded detection performance [8].

To overcome this issue, we redefine here the head distillation technique to (*i*) introduce the bottleneck at the very early layers of the network, and (*ii*) refine the loss function used to distill the mimicking head model to account for the loss induced on forwarded intermediate layers' outputs. We remark that in network distillation (see Fig. 1) applied to head-tail split models, the head portion of the model (red) is distilled introducing a bottleneck layer, and the original teacher's architecture and parameters for the tail section (green) are reused without modification. We note that this allows fast training, as only a small portion of the whole model is retrained.

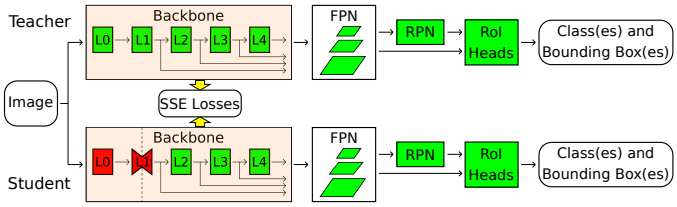


Fig. 1: Generalized head network distillation for R-CNN object detectors. Green modules correspond to frozen blocks of individual layers of/from the teacher model, and red modules correspond to blocks we design and train for the student model. L0-4 indicate high-level layers in the backbone. In this work, only backbone modules (orange) are used for training.

B. Bottleneck Positioning and Head Structure

As shown in Fig. 1, the output of early blocks of the backbone are forwarded to the detectors. In order to avoid the need to introduce the bottlenecks in multiple sections and transmit their output, we introduce the bottleneck within the L1 module of the model, whose output is the first to be forwarded to FPN. Compared to the framework developed in [2], this has two main implications. Firstly, the aggregate complexity of the head model is fairly small, and we do not need to significantly reduce its size to minimize computing load at the mobile device. Secondly, in these first layers the extracted features are not yet well defined, and devising an effective structure for the bottleneck is challenging.

Figure 1 summarizes the architecture. The difference between the overall teacher and student models are the high-level layers 0 and 1 (L0 and L1), while the rest of the architecture and their parameters is left unaltered. The architecture of L0 in the student models is also identical to that in the teacher models, but their parameters are retrained during the distillation process. The L1 in student models is designed to have the same output shape as the L1 in teacher models, while we introduce a bottleneck layer within the module.

The architectures of layer 1 in teacher and student models are summarized in our supplementary material. The architecture will be used in all the considered object detection models: Faster, Mask, and Keypoint R-CNNs. Our introduced bottleneck point is designed to output a tensor whose size is approximately $6-7\%$ of the input one. Specifically, we introduce the bottleneck point by using an aggressively small number of output channels in the convolution layer and amplifying the output with the following layers. As we design the student's layers, tuning the number of channels in the convolution layer is a key for our bottleneck injection.

The main reason we consider the number of channels as a key hyperparameter is that different from CNNs for image classification, the input and output tensor shapes of the detection models, including their intermediate layers, are not fixed [13]–[15]. Thus, it would be difficult to have the output shapes of student model match those of teacher model, that must be met for computing loss values in distillation process described later. For such models, other hyperparameters such

as kernel size k, padding p, and stride s cannot be changed aggressively while keeping comparable detection performance since they change the output patch size in each channel, and some input elements may be ignored depending on their hyperparameter values. More details about the background of bottleneck positioning and size are given in [8].

C. Loss Function

In head network distillation initially applied to image classification [2], the loss function used to train the student model is defined as

$$Loss_{org}(X) = \sum_{x \in X} ||t(x) - s(x)||^2, \quad (1)$$

where $t(x)$ and $s(x)$ are teacher and student functions of input data x in a batch X. The loss function, thus, is simply the sum of squared errors (SSE) between the outputs of last student and teacher layers, and the student model is trained to minimize the loss. This simple approach produced good results in image classification models.

Due to the convoluted structure of object detection models, the design of the loss function needs to be revisited in order to build effective head models. As described earlier, the output of multiple intermediate layers in the backbone are used as features to detect objects. As a consequence, the "mimicking loss" at the end of L1 in the student model will be inevitably propagated as tensors are fed forward, and the accumulated loss may degrade the overall detection performance for compressed data size [8].

For this reason, we reformulate the loss function as follows:

$$Loss(X) = \sum_{x \in X} \sum_{j \in J} \lambda_j \cdot L_j(x, t_j, s_j), \quad (2)$$

where j is loss index, λ_j is a scale factor (hyperparameter) associated with loss L_j, and t_j and s_j indicate the corresponding subset of teacher and student models (functions of input data x) respectively. The total loss, then, is the sum of $|J|$ weighted losses. Following Eq. (2), the previously proposed head network distillation technique [2] can be seen as a special case of our proposed technique.

D. Detection Performance Evaluation

As we modify the structure and parameters of state-of-the-art models to achieve an effective splitting, we need to evaluate the resulting object detection performance. In the following experiments, we use the same distillation configurations for both the original and our generalized head network distillation techniques. Distillations are performed using the COCO 2017 training datasets and the following hyperparameters. Student models are trained for 20 epochs, and batch size is 4. The models' parameters are optimized using Adam [23] with an initial learning rate of 10^{-3}, which is decreased by a factor 0.1 at the 5th and 15th epochs for Faster and Mask R-CNNs. The number of training samples in the person keypoint dataset is smaller than that in object detection dataset, thus we train Keypoint R-CNN student models for 35 epochs and decrease the learning rate by a factor of 0.1 at the 9th and 27th epochs.

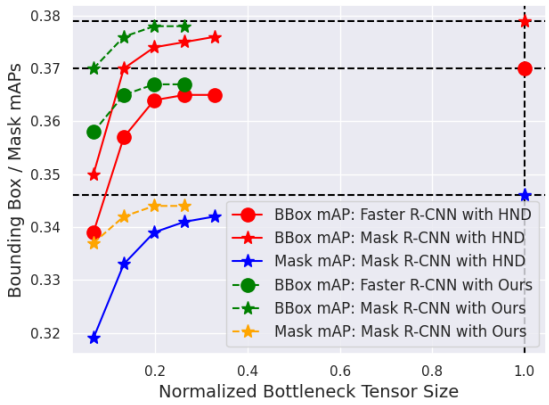


Fig. 2: Normalized bottleneck tensor size vs. mean average precision of Faster and Mask R-CNNs with FPN.

TABLE III: Performance of pretrained and head-distilled (3ch) models on COCO 2017 validation datasets* for different tasks.

R-CNN with FPN	Faster R-CNN	Mask R-CNN		Keypoint R-CNN	
Approach \ Metrics	BBox	BBox	Mask	BBox	Keypoints
Pretrained (Teacher) †	0.370	0.379	0.346	0.546	0.650
HND [2], [8]	0.339	0.350	0.319	0.488	0.579
Ours	**0.358**	**0.370**	**0.337**	**0.532**	**0.634**
Ours + BQ (16 bits)	**0.358**	**0.370**	**0.336**	**0.532**	**0.634**
Ours + BQ (8 bits)	**0.355**	**0.369**	**0.336**	**0.530**	**0.628**

* Test datasets for these detection tasks are not publicly available.
† https://github.com/pytorch/vision/releases/tag/v0.3.0

When using the original head network distillation proposed in [2], the sum of squared error loss is minimized (Eq. (1)) using the outputs of the high-level layer 1 (L1) of the teacher and student models. In the head network distillation for object detection we propose, we minimize the sum of squared error losses in Eq. (2) using the output of the high-level layers 1–4 (L1–4) with scale factors $\lambda_* = 1$. Note that in both the cases, we update only the parameters of the layers 0 and 1, and those of the layers 2, 3 and 4 are fixed. Quite interestingly, the detection performance degraded when we attempted to update the parameters of layers 1 to 4 in our preliminary experiments. As performance metric, we use mAP (mean average precision) that is averaged over IoU (Intersection-over-Union) thresholds in object detection boxes (BBox), instance segmentation (Mask) and keypoint detection tasks.

Figure 2 reports the detection performance of teacher models, and models with different bottleneck sizes trained by the original and our generalized head network distillation techniques. For the bottleneck-injected Faster and Mask R-CNNs, the use of our proposed loss function significantly improves mAP compared to models distilled using the original head network distillation (HND) [8]. Due to limited space, we show the detection performance of Keypoint R-CNN with an injected bottleneck (3ch) in Table III. Clearly, the introduction of the bottleneck, and corresponding compression of the output of that section of the network, induces some performance degradation with respect to the original teacher model.

TABLE IV: Ratios of bottleneck (3ch) data size and tensor shape produced by head portion to input data.

	Input (JPEG)	Bottleneck 32 bits	Quantized Bottleneck 16 bits	8 bits
Data size	1.00	2.56	1.28	**0.643**
Tensor shape	1.00	**0.0657**	**0.0657**	**0.0657**

E. Bottleneck Quantization (BQ)

Using our generalized head network distillation technique, we introduce small bottlenecks within the student R-CNN models. Remarkably, the bottlenecks save up to approximately 94% of tensor size to be offloaded, compared to input tensor. However, compared to the input JPEG, rather than its tensor representation [8], the compression gain is still not satisfactory (see Table IV). To achieve more aggressive compression gains, we quantize the bottleneck output. Quantization techniques for deep learning [7], [24] have been recently proposed to *compress* models, reducing the amount of memory used to store them. Here, we instead use quantization to compress the bottleneck output specifically, by representing 32-bit floating-point tensors with 16- or 8-bit.

We can simply cast bottleneck tensors (32-bit by default) to 16-bit, but the data size ratio is still above 1 in Table IV, that means there would be no gain of inference time as it take longer to deliver the data to the edge server compared to pure offloading. Thus, we apply the quantization technique [24] to represent tensors with 8-bit integers and one 32-bit floating-point value. Note that quantization is applied after distillation to simplify training. Inevitably, quantization will result in some information loss, which may affect the detection performance of the distilled models. Quite interestingly, our results indicate that there is no significant detection performance loss for most of the configurations in Table III, while achieving a considerable reduction in terms of data size as shown in Table IV. In Section VI we report results using 8-bit quantization.

V. Neural Image Prefiltering

In this section, we exploit a semantic difference between image classification and object detection tasks to reduce resource usage. While every image is used for inference, only a subset of images produced by the mobile device contain objects within the overall set of detected classes. Intuitively, the execution of the object detection module is useful only if at least one object of interest appear in the vision range. Figures 4c and 4d are examples of pictures without objects of interest for Keypoint R-CNN, as this model is trained to detect people and simultaneously locate their keypoints. We attempt then, to *filter out* the *empty images* before they are transmitted over the channel and processed by the edge server.

To this aim, we embed in the early layers of the overall object detection model a classifier whose output indicates whether or not the picture is empty. We refer to this classifier as *neural filter*. Importantly, this additional capability impacts several metrics: (*i*) reduced total inference time, as the early decision as an empty image is equivalent to the detector's output; (*ii*) reduced channel usage, as empty images are eliminated in the head model; (*iii*) reduced server load, as the tail model is not executed when the image is filtered out.

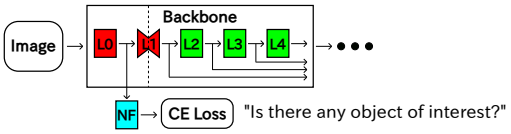


Fig. 3: Neural filter (blue) to filter images with no object of interest. Only neural filter's parameters can be updated.

(a) 2 persons (b) 1 person (c) No person (d) No person

Fig. 4: Sample images in COCO 2017 training dataset.

Clearly, the challenge is developing a low-complexity, but accurate classifier. In fact, a complex classifier would increase the execution time of the head portion at the mobile device, possibly offsetting the benefit of producing early *empty* detection. On the other hand, an inaccurate classifier would either decrease the overall detection performance filtering out non-empty pictures, or failing to provide its full potential benefit by propagating empty pictures.

In the structure we developed in the previous section, we have the additional challenge that the neural filter will need to be attached to the head model, which only contains early layers of the overall detection model (see Fig. 3). Note that parameters of the distilled model are fixed, and only the neural filter (blue module in Fig. 3) is trained. Specifically, as input to the neural filter we use the output of layer L0, the first section of the backbone network. This allows us to reuse layers that are executed in case the picture contains objects to support detection. Importantly, the L0 layer performs an amplification of the input data [25]. Therefore, using L0 for both of the head model and the neural filter is efficient and effective.

In this study, we introduce a neural filter to a distilled Keypoint R-CNN model as illustrated in Fig. 3. Approximately 46% of images in the COCO 2017 person keypoint dataset have no object of interest, and Figures 4c and 4d are sample images we would like to filter out. The design of the neural filter is reported in our supplementary material, and we train the model for 30 epochs. Each image is labeled as "positive" if it contains at least one valid object, and as "negative" otherwise. We use cross entropy loss to optimize model's parameters by SGD with an initial learning rate 10^{-3}, momentum 0.9, weight decay 10^{-4}, and batch size of 2. The learning rate is decreased by a factor of 0.1 at the 15th and 25th epochs. Our neural filter achieved 0.919 ROC-AUC on the validation dataset.

The output values of the neural filter are softmaxed *i.e.*, $[0, 1]$. In order to preserve the performance of the distilled Keypoint R-CNN model when using the neural filter, we set

a small threshold for prefiltering to obtain a high recall, while images without objects of interest are prefiltered only when the neural filter is negatively confident. Specifically, we filter out images with prediction score smaller than 0.1. The BBox and Keypoint mAPs of distilled Keypoint R-CNN with BQ (8-bit) and neural filter are 0.513 and 0.619 respectively. As shown in the next section, the detection performance slightly degraded by the neural filter results in a perceivable reduction of the total inference time in the considered datasets.

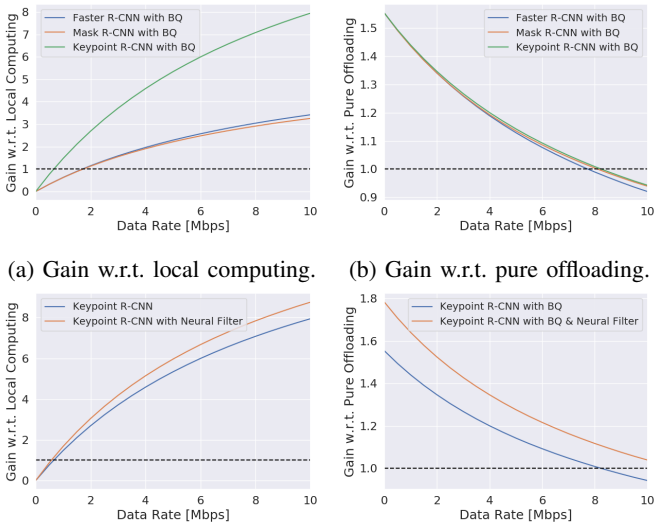
VI. Latency Evaluation

In this section, we evaluate the total time T of capture-to-output pipelines. Following Section III-B, we use the NVIDIA Jetson TX2 as mobile device, and the high-end desktop computer with a NVIDIA GeForce RTX 2080 Ti as edge server. Clearly, scenarios with weaker mobile devices and edge servers will see a reduced relative weight of the communication component of the total delay, thus possibly advantaging our technique compared to pure offloading. On the other hand, a strongly asymmetric system, where the mobile device has a considerably smaller computing capacity compared to the edge server will penalize the execution of even small computing tasks at the mobile device, as prescribed by our approach.

We compare three different configurations: local computing, pure offloading, and split computing using network distillation. Here, we do not consider naive splitting approaches such as Neurosurgeon [6] as the original R-CNN models used in this study do not have any small bottleneck point [8], and the best splitting point would result in either input or output layers *i.e.*, pure offloading or local computing. However, we consider the same data rates for vision task as in [6], and focus thus on rates below 10Mbps. Note that all the R-CNN models are designed to have an input image whose shorter side has 800 pixels. In pure offloading, we compute the file size in bytes of the resized JPEG images to be transmitted to the edge server. The average size of resized images in COCO 2017 validation dataset is 874×1044. In the split configuration, we compute data size of quantized output of the bottleneck layer, and the communication delay is computed dividing the data size by the available data rate.

Figures 5a and 5b show the gain of the proposed technique with respect to local computing and pure offloading respectively as a function of the available data rate. The gain is defined as the total delay of local computing/pure offloading divided by that of the split computing configuration. As expected, local computing is the best option (gain smaller than one) when the available data rate is small. Depending on the specific model, the threshold is hit in the range $0.5 - 2$Mbps. The gain then grows up to 3 (Faster and Mask R-CNNs) and 8 (Keypoint R-CNN) when the data rate is equal to 10Mbps.

The gain with respect to pure offloading has the opposite trend. For extremely poor channels, the gain reaches 1.5, and decreases as the data rate increases until the threshold 1 is hit at about 8Mbps. As we stated in Section II, the technique we developed provides an effective intermediate option between local computing and pure offloading, where our objective is



(a) Gain w.r.t. local computing. (b) Gain w.r.t. pure offloading.

(c) Gain with a neural filter w.r.t. local computing. (d) Gain with a neural filter w.r.t. pure offloading.

Fig. 5: Ratio of the total capture-to-output time T of local computing and pure offloading to that of the proposed technique without (top)/with (bottom) a neural filter.

to make the tradeoff between computation load at the mobile device and transmitted data as efficient as possible. Intuitively, in this context, naive splitting is suboptimal in any parameter configuration, as the original R-CNN models have no effective bottlenecks for reducing the capture-to-output delay [8]. Our technique is a useful tool in challenged networks where many devices contend for the channel resource, or the characteristics of the environment reduce the overall capacity, *e.g.*, non-line of sight propagation, extreme mobility, long-range links, and low-power/low complexity radio transceivers.

Figures 5c and 5d show the same metric when the neural filter is introduced. In this case, when the neural filter predicts that the input pictures do not contain any object of interest, the tail model on an edge server are not executed. *i.e.*, the system does not offload the rest of computing for such inputs, thus experiencing a lower delay. The effect is an extension of the data rate ranges in which the proposed technique is the best option, as well as a larger gain for some models. We remark that the results are computed using a specific dataset. Clearly, in this case the inference time is influenced by the ratio of empty pictures. The extreme point where all pictures contain objects collapses the gain to a slightly degraded - due to the larger computing load at the mobile device - version of the configuration without classifier. As the ratio of empty pictures increases, the classifier will provide increasingly larger gains.

We report and analyze the absolute value of the capture-to-output delay for different configurations. Figure 6 shows the components of the delay T as a function of the data rate when Keypoint R-CNN is the underlying detector. It can be seen how the communication delays T_{i} (JPEG image) and T_{o} tend to dominate with respect to computing components in the range where our technique is advantageous. The split approach

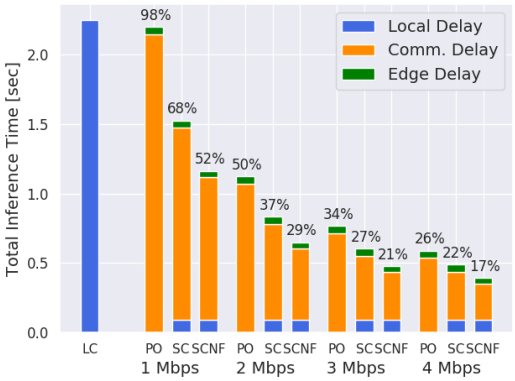


Fig. 6: Component-wise delays of original and our Keypoint R-CNNs in different data rates. LC: Local Comp., PO: Pure Offload., SC: Split Comp., SCNF: Split Comp. w/ Neural Filter

introduces the local computing component T_{H} associated with the execution of the head portion. Note that the difference between the execution of the tail portion (T_{T}) and the execution of the full model at the edge server (T_{E}) is negligible, due to the small size of the head model and the large computing capacity of the edge server. The figure also shows, while the reduction in the total capture-to-output delay is perceivable, the extra classifier imposes a small additional computing load compared to the head model with bottleneck.

VII. Conclusions

Building on recent contributions proposing to split DNNs in head and tail sections [2] executed at the mobile device and edge server respectively, this paper presents a technique to efficiently split deep neural networks for object detection. The core idea is to achieve in-network compression introducing a bottleneck layer in the early stages of the backbone network. The output of the bottleneck is quantized and then sent to the edge server, which executes some layers to reconstruct the original output of head model and the tail portion. Additionally, we embed in the head model a low-complexity classifier which acts as a filter to eliminate pictures that do not contain objects of interest, further improving efficiency. We demonstrate that our generalized head network distillation can lead to models achieving state-of-the-art performance while reducing total inference time in parameter regions where local and edge computing provide unsatisfactory performance.

References

[1] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 25:1–25:16.

[2] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "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*, 2019, pp. 21–26.

[3] P. M. Grulich and F. Nawab, "Collaborative edge and cloud neural networks for real-time video processing," in *Proceedings of the VLDB Endowment*, vol. 11, no. 12, 2018, pp. 2046–2049.

[4] X. Xie and K.-H. Kim, "Source Compression with Bounded DNN Perception Loss for IoT Edge Computer Vision," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[5] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th International Conference on Machine Learning*, 2008, pp. 1096–1103.

[6] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "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*, 2017, pp. 615–629.

[7] G. Li, L. Liu, X. Wang, X. Dong, P. Zhao, and X. Feng, "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *International Conference on Artificial Neural Networks*, 2018, pp. 402–411.

[8] Y. Matsubara and M. Levorato, "Split computing for complex object detectors: Challenges and preliminary results," in *Proceedings of the 4th International Workshop on Embedded and Mobile Deep Learning*, 2020, pp. 7–12.

[9] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587.

[10] R. Girshick, "Fast R-CNN," in *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 2015, pp. 1440–1448.

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[12] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European conference on computer vision*, 2016, pp. 21–37.

[13] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[14] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2961–2969.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[16] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *Deep Learning and Representation Learning Workshop: NIPS 2014*, 2014.

[17] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *NIPS 2014*, 2014, pp. 2654–2662.

[18] Q. Li, S. Jin, and J. Yan, "Mimicking very efficient network for object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6356–6364.

[19] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, "Learning efficient object detection models with knowledge distillation," in *Advances in Neural Information Processing Systems*, 2017, pp. 742–751.

[20] T. Wang, L. Yuan, X. Zhang, and J. Feng, "Distilling object detectors with fine-grained feature imitation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4933–4942.

[21] R. Chen, H. Ai, C. Shang, L. Chen, and Z. Zhuang, "Learning lightweight pedestrian detector with hierarchical knowledge distillation," in *2019 IEEE International Conference on Image Processing (ICIP)*, 2019, pp. 1645–1649.

[22] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2117–2125.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Third International Conference on Learning Representations*, 2015.

[24] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.