Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Head Network Distillation: Splitting Distilled Deep Neural Networks for Resource-constrained Edge Computing Systems

YOSHITOMO MATSUBARA[1], (Member, IEEE), DAVIDE CALLEGARO[1], (Member, IEEE), SABUR BAIDYA[2], (Member, IEEE), MARCO LEVORATO[1], (Member, IEEE), and SAMEER SINGH.[1]

[1]Department of Computer Science, University of California, Irvine, CA, 92697 USA (e-mail: {yoshitom, dcallega, levorato, sameer}@uci.edu)
[2]Electrical and Computer Engineering Department at University of California, San Diego, CA 92093 USA (e-mail: sbaidya@eng.ucsd.edu)

Corresponding author: Yoshitomo Matsubara (e-mail: yoshitom@uci.edu).

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

ABSTRACT As the complexity of Deep Neural Network (DNN) models increases, their deployment on mobile devices becomes increasingly challenging, especially in complex vision tasks such as image classification. Many of recent contributions aim either to produce compact models matching the limited computing capabilities of mobile devices or to offload the execution of such burdensome models to a compute-capable device at the network edge – the edge servers. In this paper, we propose to modify the structure and training process of DNN models for complex image classification tasks to achieve in-network compression in the early network layers. Our training process stems from knowledge distillation, a technique that has been traditionally used to build small – student – models mimicking the output of larger – teacher – models. Here, we adopt this idea to obtain aggressive compression while preserving accuracy. Our results demonstrate that our approach is effective for state-of-the-art models trained over complex datasets, and can extend the parameter region in which edge computing is a viable and advantageous option. Additionally, we demonstrate that in many settings of practical interest we reduce the inference time with respect to specialized models such as MobileNet v2 executed at the mobile device, while improving accuracy.

INDEX TERMS Deep neural networks, Edge computing, Head network distillation, Knowledge distillation

I. INTRODUCTION

DEEP Neural Networks (DNNs) achieve state of the art performance in a broad range of classification, prediction and control problems. Hence, the computational complexity of DNN models has been growing together with the complexity of the problems they solve. For instance, within the image classification domain, LeNet5, proposed in 1998 [1], consists of 7 layers only, whereas DenseNet, proposed in 2017 [2], has 713 low-level layers. Emerging connected and autonomous vehicles (CAV) [3], [4] applications increasingly employ these DNNs for prediction and classification tasks supporting driving assistance, navigation, and safety. Some of these applications involve rich data sources, such as high resolution images.

Despite the advances in embedded systems of the recent years, the execution of DNN models over the on-board computing platforms is becoming increasingly problematic, especially for mission critical or time sensitive applications, *e.g.*, CAVs, where the limited on-board processing capacity may degrade the response time of the system or accuracy of the algorithm.

To mitigate the aforementioned problem, vehicular edge computing [5], [6] is becoming an effective solution, where the computation can be offloaded to edge servers mounted within road side units (RSU) connected to a small cell for communications with the vehicles. While offloading data processing to a more powerful server can reduce the computation time, the end-to-end latency can be largely

impacted by wireless communication between the vehicle and the small cell. Traditional edge computing, then, paradigms is a viable – or advantageous – solution only in a limited region of parameters defining the computing capacity of the devices, and the capacity of the communication channel connecting them. Traditionally the small cell offers a subset of bandwidth compared to macro cells, and in high-density scenarios may serve a multitude of vehicles. Moreover, high speed mobility and the complexity of urban propagation scenarios can have a impact impact on the stability of the data rate perceived over time by individual vehicles. Note that even millimiter wave links were demonstrated to have intermittent patterns where high-capacity periods alternates with low-capacity periods [7], [8]. The objective of this paper is to develop techniques to make edge computing effective in the unstable and challenged network conditions affecting mobile, and especially vehicular, DNN-empowered applications.

Recently proposed frameworks [9]–[12] attempt to address this issue by *splitting* DNN models into head and tail sections, deployed at the mobile device and edge server, respectively, to optimize processing load distribution. However, due to structural properties of DNNs for image processing, a straightforward splitting approach may lead to a large portion of the processing load to be pushed to the mobile device, while also resulting in a larger amount of data to be transferred on the network. The outcome is an increase in the overall time needed to complete the model execution. Other approaches provide more advanced solutions, such as retraining bottleneck-injected models with small accuracy loss, and introducing bottlenecks to models is a recent trend in split computing studies [13]–[16].

However, prior studies on bottleneck injection for split DNNs lack a comprehensive discussion on training methodologies for bottleneck-injected models. Existing contributions show reasonable accuracy of the models, using simple and small image classification tasks, such as CIFAR datasets, whose input resolution is 32×32. However, such small input does not motivate offloading in real-world applications. A complete discussion of recent studies on split computing and comparison with the present contribution is provided in Section VI.

The contributions of this paper are summarized as follows:

- **An advanced method to split DNN models and distribute the computation load for real-time image analysis applications:** As illustrated in Fig. 1, the core idea is to alter the structure of the early layers of the network to introduce a "bottleneck" layer, that is, a layer with a small number of nodes. The network is split at that layer, and the – compact – binary representation of the output tensor, then, is propagated to the edge server to complete the classification task.
- **A new training method, head network distillation, inspired by sophisticated training approaches, namely Knowledge Distillation (KD) [17]–[21]:** While distillation has been traditionally used to train small versions of large models, here we propose to use it to train the modified section of the network (containing the bottleneck) to *mimic* the output of the same section in the original model.
- **Comprehensive experiments to discuss accuracy of bottleneck-injected models and training speed:** Our results show that this approach maximizes the compression rate while preserving accuracy, and outperforms knowledge distillation and vanilla training methods used in prior studies. We remark that our results are based on state of the art and complex datasets. Furthermore, we quantize the output of the bottleneck layer, and show that we can further reduce the amount of data transferred over the channel while maintaining the same accuracy.
- **Extensive inference time evaluation based on widely diffused hardware configurations:** Notably, our architectural modifications often lead to a decreased complexity of the section of the model deployed at the mobile device, thus reducing the penalty resulting from allocating computing load to the weaker device in the system. From a high level perspective, our approach can be interpreted as special case of autoencoder-decoder architecture transforming the input image into the input of a later layer through a bottleneck, rather then the input into itself. We conduct an extensive evaluation based on widely diffused hardware configurations. Our results show that the proposed approach extends the region of parameters in which edge computing is advantageous even compared to executing at the mobile device specialized models such as MobileNet v2 [22] and MnasNet [23].

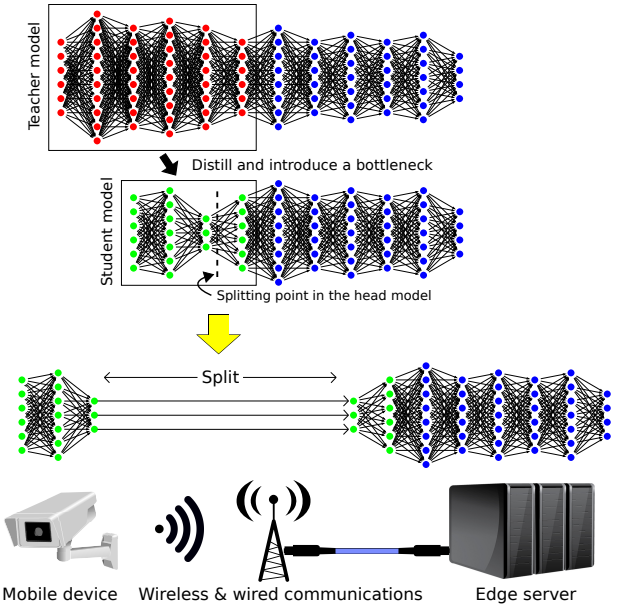


FIGURE 1: The head section of the network, executed at the mobile device-side, is shrunk – distilled – to reduce the computation complexity of that section, and a bottleneck is introduced at the splitting point to decrease the amount of data transferred to the edge server.

We also evaluate the total inference time over real-world networks (Wi-Fi and LTE) and demonstrate that the proposed technique stabilizes and improves performance with respect too edge computing and naive splitting.

Part of this study, including the idea of bottleneck introduction, was first presented in the workshop paper [14]. With respect to that contribution, herein we (*i*) consider a more complex image classification task, corresponding to the ImageNet dataset, (*ii*) provide a comprehensive discussion on the design of student model architectures, (*iii*) thoroughly discuss training methodologies, and (*iv*) present a comprehensive set of results comparing our approach with state of the art models for mobile computing.

The rest of the paper is organized as follows. Section II provides an overview of the problem setting and vision models considered in this work, and a preliminary analysis of traditional local, edge and split computing approaches. Section III describes the architecture modification we introduce in the network to inject the bottleneck while preserving the overall accuracy. In Section IV, we describe the head network distillation technique we propose and discuss its performance compared to baseline training methods. Section V evaluates the performance of our methodology in terms of total inference time in various hardware configurations using and for different DNN models, and discuss inference time in a real-world network setting as well. In Section VI, we discuss related work, and Section VII concludes this paper.

II. PROBLEM OVERVIEW AND PRELIMINARY DISCUSSION

Neural networks are structures organized into layers to approximate complex functions, for instance mapping complex inputs to a label in a finite set. The layers transform the output of other layers through different operations, and the overall network can be represented as a nested function, *e.g.*, $f(x) = l_5(l_4(l_3(l_2(l_1(x)))))$ for a model f consisting of 5 different layers (functions) l_i given the input x. DNNs are neural networks with a deep structure, that is, a large number of layers. Researchers empirically showed that DNNs can achieve better accuracy compared to "shallow" neural networks. For instance, Dauphin and Bengio [24] demonstrated that shallow neural network models trained on large-scale image datasets cannot achieve high accuracy, and Seide *et al.* and Geras *et al.* [25], [26] demonstrated that deeper models achieve higher accuracy than shallow models in speech processing. However, the "deeper" and more complex the model, the larger the computation load, and thus the execution time for a given platform.

DNN models achieving state of the art accuracy in complex tasks are computationally expensive, often too complex for mobile devices. Even when using advanced techniques, reducing the model complexity often results in a lower accuracy. For instance, MnasNet [23], one of the baseline models in this work, is designed in a platform-aware way, and the base model MnasNet (1.0x) outperforms another baseline model, MobileNet v2 [22] in terms of accuracy (73.45% vs. 71.87%) while sacrificing inference time. The smaller version of the model (*i.e.*, more mobile-device-friendly one), MnasNet (0.5x), can run as fast as MobileNet v2, but suffers from degraded accuracy (67.73%). Edge computing, then, is a desirable solution to make such models accessible to mobile devices with constrained resources. However, the performance of edge computing is a function of many parameters, and its use in real-time applications should be evaluated carefully.

We consider a resource-constrained mobile device-edge server system, that is an edge server interconnected to the mobile device through a wireless channel. We focus our investigation on challenged communication links – that is, links experiencing degraded performance due to congestion or propagation environments, that are expected in dense urban environments. We restrict our study to one of the most relevant and complex family of tasks in modern systems: image classification, which in the considered setting is possibly performed collaboratively by the two interconnected devices. The mobile device, which acquires the image to be processed, is assumed to have a weaker computing capacity compared to the edge server. Intuitively, there are two extreme points in terms of computing allocation: (*i*) *Local Computing* - the mobile device executes the DNN model on the acquired image, and propagates the classification output to the edge server if the result is needed system-wide; (*ii*) *Edge Computing* - the mobile device transmits the image to the edge server, which executes the DNN and reports back the outcome to the mobile device if the result is needed locally. On the one hand, due to hardware limitations, local computing is an often prohibitive option. On the other hand, the performance of edge computing is heavily influenced by the - volatile - capacity of the wireless channel interconnecting the sensor to the edge server.

As introduced in the previous section, an intermediate option between those two extreme points is to establish a distributed computing pipeline, where a first section of the model – *head model* – is assigned to the mobile device, and the remaining section – *tail model* – to the edge server. This strategy goes under the name of *Split DNN models* [10], [11], [27]–[29]. Specifically, the mobile device executes the head portion of the model, propagates the output to the edge server, which produces the final classification output by executing the tail portion of the model. From the standpoint of total inference time, whether this approach is advantageous or not again depends on the computing capacity of the mobile device and edge server, the capacity of the wireless channel, the input size, as well as the splitting point and the characteristics of the DNN model itself.

We assume that the pretrained models are available, that is, trained offline prior to their use and loaded at the mobile device and edge server. In this setting, we focus our attention on the total inference time, defined as the time from the acquisition of a data sample at the mobile device to the availability of the model output. In split DNN strategies, the overall inference time T is the sum of three components: the time needed to execute the *head* and *tail* portions of the model – τ_{head} and τ_{tail} respectively – and the time to wirelessly

transfer the output of the head model's last layer to the edge server τ_{comm}. Pure edge computing can be interpreted as an extreme point of splitting, where the head portion is composed of 0 layers, and τ_{comm} is the time to transfer the input image. Conversely, in local computing the head model contains the whole DNN model, and τ_{comm} is either zero if the output label is not propagated to the edge server, and the time to deliver the model's output otherwise, which can be assumed negligible in the parameter region of interest to our reasoning.

In the following, we introduce the DNN models considered in this paper and analyze the performance of available solutions. In our evaluations, we use the following computing platforms either as mobile device or edge server: (*i*) Raspberry Pi 3B+ (mobile device only), (*ii*) NVIDIA Jetson TX2 (mobile device or edge server), and (*iii*) a desktop computer with NVIDIA GeForce RTX 2080 Ti (edge server only). See Table 3 for a complete description. Notice that the NVIDIA Jetson TX2 has a 256-core NVIDIA Pascal™ GPU architecture with 256 NVIDIA CUDA cores, and has a much larger computing capacity compared to the Raspberry Pi 3B+.

A. VISION MODELS

Vision problems are an ideal family of tasks to assess our methodology and compare it with other options. In fact, while these tasks are extremely relevant and pervasive in a broad range of applications and systems, they are also inherently problematic as they require a relatively large input – RGB(-D) image files – which may affect performance of traditional edge computing due to high bandwidth demands, as well as complex DNN models – such as DenseNet [2] and RetinaNet [30] for image classification and object detection, respectively – which may be out of reach of mobile devices. Alternative solutions, then, are of interest to improve performance in parameter regions where the two extreme point solutions provide unsatisfactory performance.

In this study, we focus on image classification tasks, and we consider accurate, but complex, as well as compact, but less accurate, DNN models as performance reference to study effective distributed execution strategies. Specifically, we focus on the following set of models: DenseNet-169, -201 [2], ResNet-152 [31], Inception-v3 [32]. We also consider accurate models designed for mobile execution, such as MobileNet v2 [22] and MnasNet [23], as a competitor to our scheme. We remark that some compact models within the DenseNet and ResNet families – such as DenseNet-121 and ResNet-18 – are available. However, those models are still too complex to be executed on mobile devices, and have a worse computational complexity/performance tradeoff compared to MobileNet v2 and MnasNet.

B. LOCAL AND EDGE COMPUTING

First, we analyze the two most popular computing options, that is, local computing and edge computing. Table 1 reports the accuracy of the models pretrained on the ImageNet (ILSVRC 2012) dataset, and their average inference time on the considered hardware.

TABLE 1: Inference time of original models on the considered computing platforms.

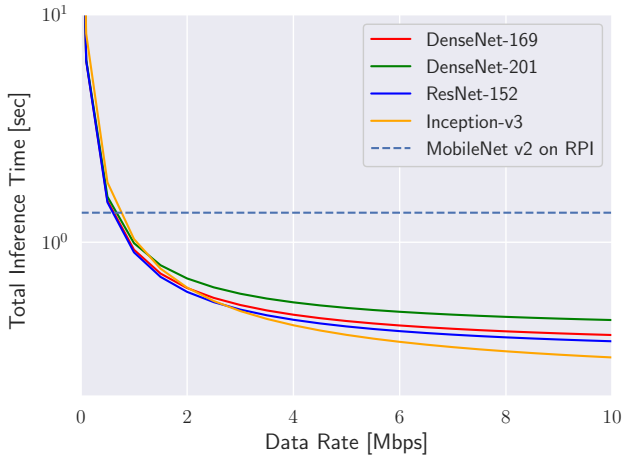
Models (version)	MobileNet v2	MnasNets (0.5x)	MnasNets (1.0x)	DenseNets 169	DenseNets 201	ResNet 152	Incept. v3
Accuracy* [%]	71.87	67.73	73.45	75.60	76.89	**78.31**	77.58
T on RPI [s]	**1.348**	**1.332**	2.332	7.752	9.720	19.64	10.58
T on TX2 [s]	**0.112**	**0.096**	**0.096**	0.332	0.396	0.308	0.232
T on Desktop [s]	**0.0049**	**0.0047**	**0.0047**	0.02	0.028	0.016	0.012

* ILSVRC 2012 validation dataset (test dataset is not available)

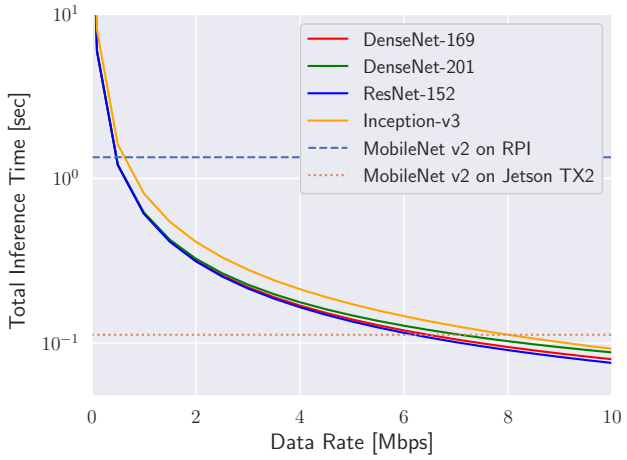
In the weakest platform, the Raspberry Pi 3B+, the execution of large – accurate – models such as Densenet-169 and -201, ResNet-152 and Inception-v3 is feasible, but requires an extremely large amount of time. As expected, local computing is not a viable option in this setting. The stronger Jetson TX2, equipped with a GPU, reduces these times by one or two orders of magnitude. However, the execution still takes between 0.2 and 0.4 seconds, a range which may be incompatible with some time sensitive applications, such as real-time control in robotic systems. The desktop computer, equipped with a strong GPU, reduces the inference time by another order of magnitude. However, granting such computing capacity to a mobile device would likely clash with weight, cost or energy constraints. As a consequence, in our study this hardware configuration is only used as an edge server.

As expected, the execution time of MobileNet v2, MnasNets (0.5x) and (1.0x) is much smaller compared to other models, with the difference in absolute value becoming more noticeable when weak platforms are used. However, the decrease in execution time corresponds to a perceivable decrease in classification accuracy. As indicated in Table 1, however, MnasNets' performance degrades when the model is executed on a device without GPU. On Raspberry Pi 3B+ (RPI), MnasNet's (0.5x) execution time is analogous to that of MobileNet v2, but the former model incurs about 4% accuracy loss compared to the latter. Similarly, MnasNet (1.0x) outperforms MobileNet v2 in terms of accuracy, but has an inference time 73% larger on the mobile device. For these reasons, we adopt MobileNet v2 as a baseline model in the our experiments and evaluations.

We now analyze an edge computing approach, where the Jetson TX2 (Figure 2a) or the desktop computer (Figure 2b) operate as edge server. The two figures report the total inference time as a function of the data rate between the mobile device and the edge server, which has an obvious impact on τ_{comm}. We remark that the data rate indicated in plots and tables is the "effective" data rate perceived by the data stream, that is, the actual transfer rate at which the image data are moved from the mobile device to the edge server. For instance, retransmissions to recover from packet failures would reduce the perceived rate. In the plots, we consider large accurate models executed at the edge server, and compare them with the local execution on the Raspberry Pi 3B+ or Jetson TX2 of – the compact – MobileNet v2 (dashed and dot



(a) NVIDIA Jetson TX2 as edge server.



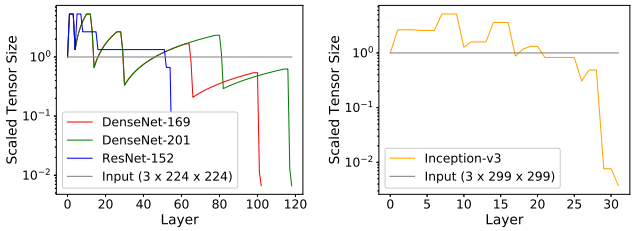
(b) High-end desktop as edge server.

FIGURE 2: Edge computing performances comparing to MobileNet v2 in local computing.

lines).

As the data rate increases, the total inference time achieved by edge computing decreases. When the Jetson TX2 is used as edge server, edge computing is advantageous in terms of inference time (and accuracy) unless the data rate offered by the wireless channel is smaller than $\sim$ 1Mbps – that is, the channel is rather weak. When the desktop computer is used as edge server, the inference time of pure edge computing intersects that of local computing at about 0.7Mbps (Raspberry PI 3B+) and 7Mbps (Jetson TX2). Thus, if a stronger platform is used as mobile device, then local computing is the best choice even in moderately good channels.

However, we can observe that whether or not local and edge computing are viable depends on the computing capacity of the devices and the channel capacity. In general, these two extreme point strategies are effective either if the mobile device is sufficiently strong, or if the channel capacity and edge server computing capacity are sufficiently large. There



(a) (b)

FIGURE 3: Scaled output tensor sizes of splittable layers in original DNN models with respect to input tensor sizes $3 \times 224 \times 224$ and $3 \times 299 \times 299$ for (a) DenseNets-169, -201, ResNet-152 and (b) Inception-v3, respectively.

are hardware-channel parameter regions in which neither of the two provide satisfactory performance from the standpoint of accuracy and inference time. Herein, we seek to develop a tool to reduce the inference time and make accurate image classification possible in those regions.

C. SPLIT DNN MODELS

Split DNN models are a promising class of solutions to mitigate the issues incurred by local and pure edge computing. To the best of our knowledge, Kang *et al.* [10] proposed the first DNN splitting framework – Neurosurgeon. Unfortunately, as also pointed out in [14], this naive splitting approach often ends up merging the weak points of local and edge computing. In many models, the optimal inference time is achieved either by fully deploying the model on the mobile device or edge server, that is, splitting is ineffective in all configurations compared to the extreme points (local and edge computing). This applies to all the models but those presenting at least one "bottleneck" layer, that is, a layer whose output size is smaller than than the model's input size at an early layer. In fact, in the latter case some compression can be achieved with a small amount of computation at the weaker device in the system – the mobile device. As a result, in [10] it is shown that the optimal splitting point is inside the network only for the AlexNet [33] and VGG [34] models (both of image classification DNN models). However, even in the latter models the inference time reduction is limited, and only applies to specific configurations.

We now analyze the structural reasons behind the ineffectiveness of traditional Split DNN models. Figure 3 shows the size of tensors outputted from splittable layers normalized by tensor size of the specific models' input layer. Note that only Inception-v3 has a different input tensor size ($3 \times 299 \times 299$) whereas other models use $3 \times 224 \times 224$ as input tensor size. As shown in the figure, DenseNets-169 and -201 have bottleneck layers (below a gray line) at their early stages, and their total inference time could be improved by the naive splitting approach. However, even if some compression is achieved, the reduced data transfer time is likely offset by the increase in the total computing time unless the mobile device and edge server have similar computing capacity. In fact, in such case

the increased execution time of the head model is limited. Thus, splitting the model is advantageous with respect to edge computing only in those settings where local computing is either optimal or near-optimal solution. For ResNet-152 and Inception-v3, however, the first bottleneck layer resides in the late layers, and in most parameter regions using the Neurosurgeon approach would not grant any improvement in the overall inference time.

D. OBJECTIVES

The goal of this paper is to identify modifications to the architecture and training of accurate image classification models to reduce the total inference time in parameter regions where pure edge computing fails to provide satisfactory performance. Thus, we focus our attention on impaired channel conditions, that is, scenarios where the link between the mobile device and the edge server offers low data rates (*e.g.*, $\leq$ 10Mbps). Clearly, extremely low data rates would force the mobile device to execute the model locally.

Furthermore, we assume the system is asymmetrical, that is, the edge server has a much larger computing capacity compared to the mobile device. As discussed earlier, if the two devices have similar capabilities, then edge computing is the best option only for high channel capacities, and the gain would be anyway minimal.

III. IN-NETWORK COMPRESSION & DESIGNING STUDENT MODELS

The simple results presented in the previous section indicate how the time needed to transfer the image (or the tensor in the split DNN case) plays a central role in determining the performance of pure edge offloading. Intuitively, compression could reduce this delay component. However, traditional compression techniques such as JPEG degrade classification performance [35], whereas in the target family of problems splitting the network would increase inference time in most settings.

Herein, we seek methodologies to achieve compression within the DNN model itself. The core idea is to modify the model architecture to introduce a bottleneck, that is, a layer with a small number of nodes, in the early stages of the network regardless of whether or not the network has some bottlenecks. The network is then split at the bottleneck, thus reducing the size of the tensor that needs to be transferred with respect to traditional split DNNs. Intuitively, the gain (or loss) on the total inference time with respect to edge computing is determined by the balance between (*i*) the (possible) increase in the time to execute the first part of the network, which occurs on a weaker device compared to edge computing, and (*ii*) the reduction in the time needed to transfer the tensor with respect to that needed to transfer the image.

Introducing an effective bottleneck is challenging. In this section, we explain the structural modifications we make to maximize compression while minimizing accuracy loss. Interestingly, in some models these modifications also result in a reduction of the complexity of the head model, which

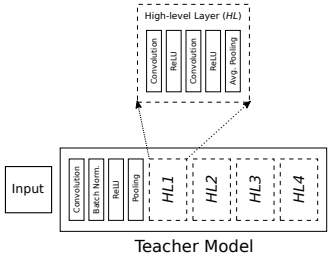


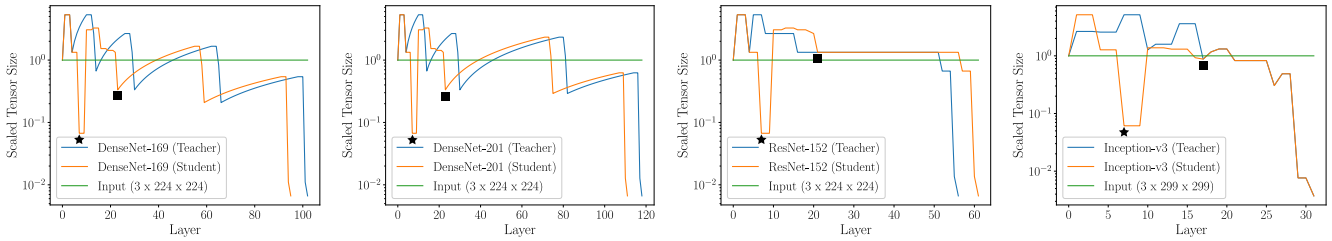
FIGURE 4: Example of high-level layer *HL* consisting of low-level layers and teacher model including 4 high-level layers.

may compensate for the lower computing capacity of the mobile device with respect to the edge server. We note that the splitting point, at the bottleneck, is within the student model, as illustrated in Figure 1, which summarizes the network structure. We emphasize that in the proposed approach, only the head portion of the model is modified, and the tail portion of the model remains identical. In other words, the combination of teacher head and tail models is equivalent to the original (teacher) model, and the architectural difference between the teacher and student models lies in their head models.

We remark that in addition to the structural modifications, one of the key novelties in the proposed technique is the use of a specific form of training, *knowledge distillation* applied to the head network sections. The head model, then, is trained to *mimic* the output of the original head network, and this approach also will save training time. We discuss and evaluate the training aspect of the contribution in the next section.

When identifying the candidate layer to inject the bottleneck layer and modifying the architecture, we aim at the following design objectives: (*i*) minimize the size of the bottleneck layer's output; (*ii*) minimize the aggregate complexity of the model prior to the bottleneck; and (*iii*) minimize the accuracy loss of the overall model's output. Note that (*ii*) is necessary to minimize the load imposed to the mobile device (the weakest device in the system), and is achieved both by placing the bottleneck layer as early as possible in the model, and attempting to "shrink" that portion of the model while inserting the bottleneck.

Our approach stems from the observation that the original, pretrained, (teacher) models and their head portions are often overparameterized. Therefore, we can use a subset of the layers in the original models to design the student head models, modifying the layers' hyperparameters to adjust their input and output shapes. In essence, when designing the student head models, we essentially reuse the architecture of the teacher head models, decomposing their high-level layers into low-level layers and skipping some of them. Figure 4 illustrates an example of a high-level layer and a teacher model. In the example, the high-level layer consists of 5 low-level layers: Convolution, ReLU, Convolution, ReLU, and Average Pooling layers. In the student models, we adjust the



(a) DenseNet-169 (b) DenseNet-201 (c) ResNet-152 (d) Inception-v3

FIGURE 5: Splittable layer-wise scaled output tensor sizes for original DNN models (blue) and their mimic models (orange). Star and square marks indicate the introduced bottleneck and the end of the student head model respectively.

input and output sizes and shape of the layers by changing some layer parameters such as kernel size.

For instance, we can reuse the 4 high-level layers in Figure 4, but each of the high-level layers only contains one Convolution, one ReLU and one Average Pooling layers (i.e., 3 low-level layers, decomposing the original high-level layer and skipping one Convolution and one ReLU layers used in the original one). Another option is to skip 2 high-level layers (e.g., *HL*s 2 and 3) and change some parameters in the rest of the high-level layers in order to adjust their input and output shapes. Note that the output shape of a student head model must match that of the teacher head model to ensure compatibility with the tail model.

Figure 5 compares splittable layer-wise output tensor sizes of original models to those of their mimic models whose architectures are described Tables 6, 7, and 8 in Appendix. In this figure, the output data size is scaled by the input image size in tensor representation ($3 \times 224 \times 224$ for DenseNet-169, -201 and ResNet-152, and $3 \times 299 \times 299$ for Inception-v3, respectively). As shown in the figure, aggressively small bottlenecks (layers with star marks) are introduced as we design student head models whose last layers are indicated with square markers. Note that the splittable layers in the figure not only include low-level layers, but also high-level layers such as dense and inception layers that consist of multiple low-level layers.

IV. MODEL TRAINING

We now describe in detail the procedure we use to train the models we developed. Clearly, this is an extremely important aspect of the technique that we propose, which greatly influences accuracy. Herein, we propose to use Knowledge Distillation (KD) applied to the head network section, and show that this approach allows the introduction of aggressive bottlenecks while preserving accuracy. We compare our proposal (HND) with two baseline approaches, and demonstrate the performance improvement of the training process.

We note that training is a one-time process, and the models will not be trained once they are split and deployed on the mobile device and edge server.

A. NAIVE TRAINING

The most basic approach is to train both the student head and tail models simultaneously [13], [15], [16], and we train the whole DNN model by minimizing the cross entropy loss:

$$L_{CE}(X,Y) = \frac{1}{N} \sum_{x,y \in (X,Y)} -\log \left(\frac{\exp\left(S(x,y)\right)}{\sum_{j \in C} \exp\left(S(x,j)\right)} \right), \tag{1}$$

where X and Y are a minibatch (batch size N) of resized input images and the corresponding class labels, respectively. C is a set of class labels ($|C| = 1,000$ in this study), and $S(x,j)$ indicates a student model's predicted value for a class j given a resized input image x. The parameters of the whole student model are optimized by stochastic gradient descent (SGD) with momentum 0.9 and weight decay 10^{-4}.

B. KNOWLEDGE DISTILLATION

Recent contributions propose Knowledge Distillation (KD) techniques [17]–[21] to train small "student" models that approximate (mimic) the outputs of larger "teacher" models. Complex DNN models are usually trained to learn parameters for discriminating between a large number of classes (*e.g.*, $1,000$ object categories in this study). In standard training – that is, using "hard targets" – we face a side-effect that the trained models assign probabilities to all of the incorrect classes. From the relative probabilities of incorrect classes, we can see how large models tend to generalize. By distilling the knowledge from a large, cumbersome (teacher) model into a smaller (student) one, the smaller model can be trained to generalize, using the outputs of the teacher model as "soft targets" [19].

It was empirically shown that student models trained to mimic the behavior of their teacher models (*soft target*) outperforms those trained on the original training dataset (*hard target*) in terms of prediction performance. Based on KD, in our case the output of the teacher model are used to train both the student head and tail models simultaneously, minimizing the knowledge distillation loss proposed by Hinton *et al.* [19]:

$$L_{KD}(X,Y) = \alpha L_{CE}(X,Y) + (1-\alpha)\left(\frac{t^2}{N}\sum_{x,y\in(X,Y)} KL\left(p(x),q(x)\right)\right) \quad (2)$$

where α is a hyperparameter whose value is used to balance the *hard target* (left term) and *soft target* (right term) losses. $KL(p(x),q(x))$ is the Kullback-Leibler divergence loss, where $p(x)$ and $q(x)$ are softed-probability distributions (similar form to softmax function) of student and teacher models for an input x, that is, $p(x) = [p_1(x), \cdots, p_{|C|}(x)]$ and $q(x) = [q_1(x), \cdots, q_{|C|}(x)]$:

$$p_c(x) = \frac{\exp\left(\frac{S(x,c)}{t}\right)}{\sum_{j\in C}\exp\left(\frac{S(x,j)}{t}\right)}, q_c(x) = \frac{\exp\left(\frac{T(x,c)}{t}\right)}{\sum_{j\in C}\exp\left(\frac{T(x,j)}{t}\right)}, \quad (3)$$

where $T(x,j)$ indicates a teacher model's predicted value for a class j given a resized input image x, and t is a "temperature" (hyperparameter) for knowledge distillation that is normally set to 1.

Following Hinton *et al.*'s work [19], we set α and t to 0.5 and 1 respectively, and use SGD with momentum 0.9 and weight decay 10^{-4} for optimizing the parameters of the whole student model.

C. HEAD NETWORK DISTILLATION (HND)

KD-based approaches have been successfully used to train small models, where the outputs from pretrained large models are used as soft targets. However, training whole DNN models is time-consuming, as the student model is designed to mimic the behavior of the teacher model, which in this case is literally the whole DNN model. Moreover, aggressively downsizing DNN models to achieve small execution time in mobile devices would lead to an accuracy loss. This is shown by the performance of models such as MobileNet v2 and MnasNet (0.5x), relatively large models which already incur a perceivable classification accuracy degradation.

Figure 6 shows that distilling only the head portion of the model, as proposed in this paper, significantly reduces training time. This in addition to allow in-network compression while preserving overall accuracy. Moreover, class labels (*hard targets*) are not required for head network distillation since the student models learns how to represent teacher's features rather than how to classify input images. This is an additional advantage of using the head network distillation technique, as labeled datasets are not always available and/or the amount of labeled samples is often limited. By distilling a teacher head model, the corresponding student head model can produce outputs similar to those of the teacher head model with fewer parameters and computational complexity.

Herein, we train the student head models with Adam [36] for optimizing the parameters of the student head model by minimizing the sum of squared error between outputs of teacher and student head models:

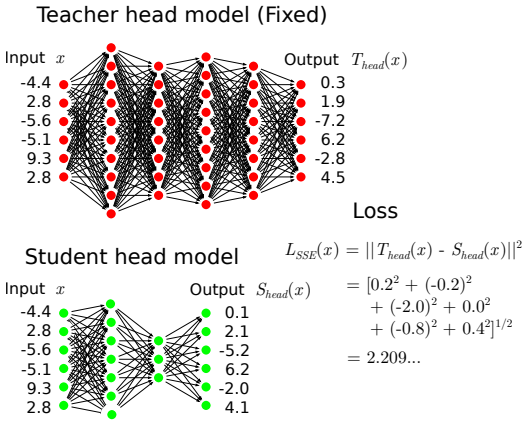


FIGURE 6: Training student head model with teacher head model by minimizing sum of squared error (SSE).

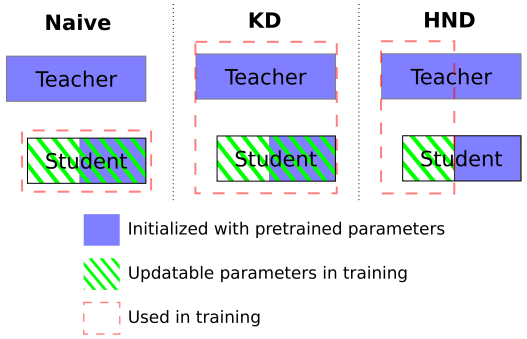


FIGURE 7: Illustrations of three different training methods. Naive: Naive training, KD: Knowledge Distillation, HND: Head Network Distillation.

$$L_{SSE}(X) = \sum_{x\in X} ||T_{head}(x) - S_{head}(x)||^2, \quad (4)$$

where $T_{head}(x)$ and $S_{head}(x)$ are outputs of teacher and student head models, respectively. Since the outputs of teacher and student head models are not class-wise probability distributions like those in Eqs. (1) and (2), and the output is often 3D shaped, we cannot use the cross-entropy based loss in the same way as previous studies commonly do when distilling a whole DNN model [19], [21].

D. EXPERIMENTS

Given the (head-)modified models we developed in the previous section, we now show that the head network distillation technique outperforms the two baseline training approaches. We emphasize that all the following training configurations are applied to all the three training methods. As described above, the tail architecture of a student model is identical to that of the teacher model, thus we first initialize the parameters of a student tail model with the parameters from its pretrained teacher tail model as shown in Figure 7.

We train the student model for 20 epochs with an initial learning rate 0.001 that is reduced by an order of magnitude

every 5 epochs. The batch size is set to 32. In training, we apply two data augmentation techniques [33], which allow to increase the size of training dataset and reduce the risk of overfitting. The idea is to randomly crop fixed-size patches (input patch size: 224×224 or 299×299) from the approximately 1.15 times larger resized images (shorter edge is 256 or 327, respectively). The cropped images are flipped horizontally with probability 0.5.

We remark that in our previous study [14], we demonstrated the potential of using head network distillation technique for Caltech 101 dataset [37], but the models used in this study are specifically designed for a more difficult image classification task - the ImageNet (ILSVRC 2012) dataset [38]. Thus, it is possible that the teacher models in [14] are overparameterized, which could have enabled small bottlenecks while preserving a comparable accuracy. Herein, we face a much harder challenge when introducing the bottlenecks. The trained models and code to reproduce the results are publicly available.[1]

1) Training speed

Given the set of the student models and the training configurations described in the previous sections, we individually train models using a high-end computer with three GPUs.

Naive training vs. Knowledge distillation - First of all, we compare the training performance of naive training and knowledge distillation methods to reproduce the trend in the study of Ba and Caruana [18], which shows that – whole network – student models trained by knowledge distillation outperform those naively trained on the original dataset. Figure 8 depicts the training time and validation accuracy at the end of each epoch (thus 20 data points) for each pair of student models and training methods. It can be observed how in the knowledge distillation method the student models achieve higher accuracy compared to the naive training method used in [13], [15], [16]. This comes at the cost of a longer training time.

Knowledge distillation vs. Head network distillation - We showed that knowledge distillation enables the student models achieve better accuracy compared to naive training. However, in some cases the models did not reach the accuracy of MobileNet v2 (71.87%), a small model for mobile devices, and the training process is time-consuming. As illustrated in Figure 9, the head network distillation approach consistently helps the student models not only converge significantly faster, but also achieve even better accuracy compared to the knowledge distillation method. Recall that we trained exactly the same student models with the common training configuration described in Section IV-D, but in three different ways as illustrated in Figure 7. Therefore, we can conclude that these performance improvements are due to the head network distillation technique we propose.

Summary - Table 2 summarizes the best validation accuracy for each of the student models, and confirms that there is a consistent trend: the knowledge distillation method provides a better accuracy compared to the naive training method, and the head network distillation technique consistently outperforms knowledge distillation applied to the full model. Additionally, the head network distillation technique performs best in terms of training time, as shown in Figures 8 and 9. As described in Section IV-D, we applied the same training configurations to compare the performance of the three different training methods. The accuracy of the head network distillation approach could potentially be further improved by elongating its training to match the training time (that is, number of

[1]https://github.com/yoshitomo-matsubara/head-network-distillation

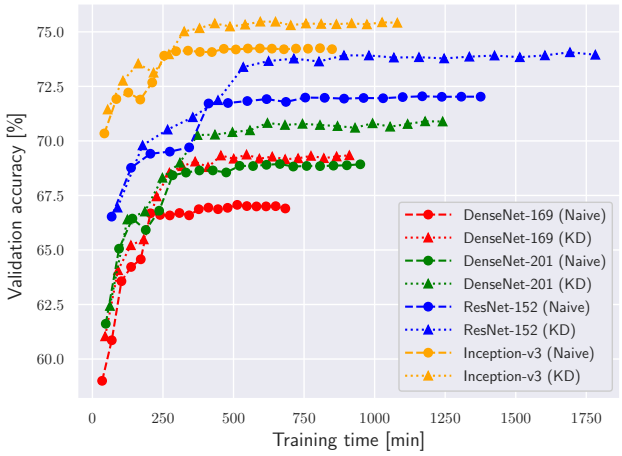


FIGURE 8: Training speed: Naive training (Naive) versus Knowledge Distillation (KD).

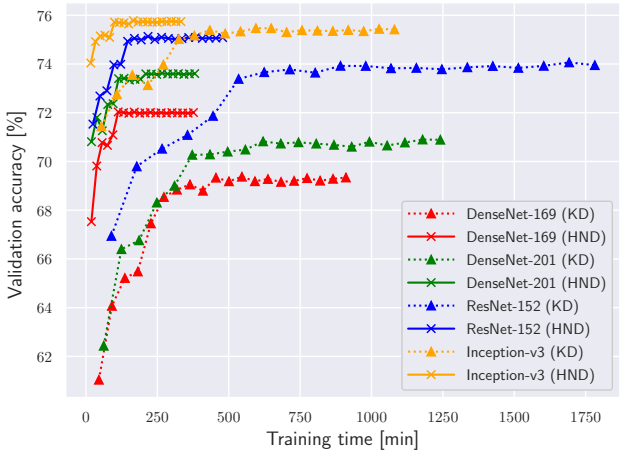


FIGURE 9: Training speed: Knowledge Distillation (KD) versus Head Network Distillation (HND).

TABLE 2: Validation accuracy* [%] of student models trained with three different training methods.

Method	DenseNet-169	DenseNet-201	ResNet-152	Inception-v3
Naive	66.90 (-4.970)	68.92 (-2.950)	72.02 (+0.149)	74.20 (+2.330)
KD	69.37 (-2.500)	70.89 (-0.980)	74.06 (+2.190)	75.46 (+3.589)
HND	**72.03 (+0.159)**	**73.62 (+1.750)**	**75.13 (+3.259)**	**75.78 (+3.910)**

* ILSVRC 2012 validation dataset (test dataset is not available)
** Numbers in brackets indicate differences from MobileNet v2.

epochs) used in the naive training or knowledge distillation.

2) Bottleneck channel

In this set of experiments, we discuss the relationship between the file size of the bottleneck tensor and the accuracy of student models trained using head network distillation. Specifically, we tune the number of output channels (or filters) N_{ch} for the convolution layer at the bottleneck in the student models, and apply head network distillation for the student models using the same training configuration described in Section IV-D.

As shown in Tables 6, 7 and 8, all the student models used in the previous experiments have 12 output channels ($N_{ch} = 12$) in the convolution layer at the bottleneck. Changing the number of output channels N_{ch} and the number of input channels in the following convolution layer, we can adjust the bottleneck size - a parameter which has considerable impact on the overall inference time, especially when the communication channel between mobile device and edge server is weak. For instance, if we set N_{ch} to 6, the output file size will be approximately half of that obtained with $N_{ch} = 12$.

Figure 10 shows the accuracy obtained using $N_{ch} = 3, 6, 9$ and 12. As expected, aggressively reducing the bottleneck size consistently degrades accuracy. Thus, in order to further reduce the amount of the data transferred from the mobile device to the edge server, we adopt the quantization technique proposed in [39] to the output of the bottleneck layer. Specifically, we represent floating-point tensors with 8-bit integers and one parameter (32-bit floating-point). The quantization is applied only in testing time *i.e.*, after the head network distillation process is completed. As shown in Figure 10, bottleneck quantization significantly reduces the bottleneck file size, as much as 75% compression with respect to bottleneck output tensor and 86% compression with respect to resized input JPEG files, without impacting the accuracy.

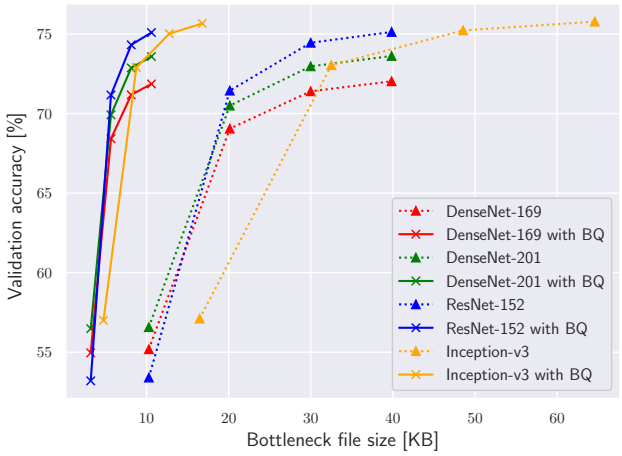


FIGURE 10: Relationship between bottleneck file size and validation accuracy with/without bottleneck quantization (BQ).

TABLE 3: Hardware specifications.

Computer	Processor	Freq.	RAM
Raspberry Pi 3B+	ARM Cortex A53 (quad-core) ARM Cortex-A57 (quad-core)	1.2 GHz	1 GB
Jetson TX2	+ NVIDIA Denver2 (dual-core) + 256-core NVIDIA Pascal™ GPU	2.0 GHz	8 GB
Desktop	Intel i7-6700K CPU (quad-core) + NVIDIA GeForce RTX 2080 Ti	4.0 GHz	32 GB

V. INFERENCE TIME EVALUATION

In the previous section, we showed that it is possible to significantly reduce the amount of data transferred from the mobile device to the edge server without compromising accuracy. In this section, we provide an exhaustive evaluation of the proposed technique in terms of total inference time (capture-to-output delay) with respect to local computing based on full models and mobile-specific models (MobileNet v2) and pure edge computing. We note that naive splitting approaches *e.g.*, Neurosurgeon [10] are not used in the following evaluations as the original benchmark models do not have any small bottleneck point at their early stage, where their best splitting point would result in either input or output layers. *i.e.*, pure offloading or local computing. We remark that the focus is to provide solutions to improve edge computing performance over challenged wireless links, which may present relatively low or intermittent capacity due to congestion or impaired signal propagation.

Note that we assume a channel where all the transmitted packets are eventually delivered. This is a common setting in edge computing, and generally in machine learning, frameworks, and can be realized, for instance, using TCP at the transport layer. Clearly, retransmissions to resolve packet failures will reduce the perceived data rate, and that shown in the figures is the resulting effective rate. Herein, as most literature in this area [10], [13], [14], [29], we also assume that the mobile device is connected to one server at any given time. For instance, the mobile device could simply use the edge server connected through the best channel.

Table 3 summarizes the specifications of the three different computers used as either a mobile device (MD) or an edge server (ES), and we evaluate the overall inference time in the three different mobile device-edge server configurations: (*i*) Raspberry Pi 3B+ – Jetson TX2, (*ii*) Raspberry Pi 3B+ – Desktop, and (*iii*) Jetson TX2 – Desktop.

A. GAIN WITH RESPECT TO LOCAL AND EDGE COMPUTING

First, we discuss the gain (defined as the ratio of the capture-to-output delay T of a traditional setting to that provided by our technique) with respect to local and edge computing with their original (teacher) models, that are defined as $T_{\mathrm{LC}}/T_{\mathrm{O}urs}$ and $T_{\mathrm{EC}}/T_{\mathrm{O}urs}$, respectively. For edge computing, we compute the communication delay based on the JPEG file size after re-

TABLE 4: List of average file sizes [KB] for resized JPEG files and quantized bottlenecks of our student models.

	DenseNet-169	DenseNet-201	ResNet-152	Inception-v3
Resized JPEG	74.34	74.34	74.34	100.1
BQ	10.58	10.58	10.58	16.77

shaping sampled input images in the ILSVRC 2012 validation dataset ($3 \times 299 \times 299$ for Inception-v3 and $3 \times 224 \times 224$ for other models). To compute the communication delay for our split student models, we use the file size of the quantized tensor at bottleneck shown in Table 4, and the file sizes (BQ) match those of the last data points (*i.e.*, $N_{ch} = 12$) with BQ in Figure 10. Table 4 summarizes the average file sizes used to compute communication delay in the following evaluations.

Figure 11 indicates that splitting the computing load using our student models provides significant gains compared to locally execution the original (teacher) models on the mobile devices unless the channel is extremely weak. As for comparison with edge computing, the right-side plots of Figure 12 implies that the smaller the difference of computing power between the mobile device and the edge server the configuration has, the more beneficial the splitting approach is. From the results, it is also shown that for large enough data rates, then edge computing is the best option due to the penalty associated with allocated computing to the weaker device in the system, which overcomes the reduced data transmission time. Intuitively, such penalty is emphasized in very asymmetric configurations. In less asymmetric configurations, our technique outperforms edge computing in an extensive range of data rates.

B. GAIN WITH RESPECT TO LOCAL COMPUTING WITH MOBILENET V2

In Section IV-D, we showed that our student models trained by head network distillation outperform MobileNet v2 in terms of accuracy. Here, we demonstrate that splitting student models can also lead to improved total inference times compared to executing MobileNet v2 locally. Similar to the previous evaluation, we compute the gains, but the denominator is the inference time of MobileNet v2, rather than that associated with their teacher models, on the mobile device.

In the set of plots in Figure 13, we can observe that the gain, although reduced compared to the previous comparison, is still above 1 for most data rates of interest. Note that a stronger mobile device, reduces the gap between the two options. We remark that in addition to a reduced inference time, our methodology also provides an improved accuracy compared to local computing with MobileNet v2.

C. DELAY COMPONENTS ANALYSIS

We now analyze the delay components to obtain further insights on the advantages and disadvantages introduced by the computing configuration we propose. We, first, focus on the execution time at the mobile device and edge server. Figure 14 shows these components for the split head and tail models on different platforms. The differences in computing capacity are apparent, and in many settings the local processing delay is larger than the edge processing delay, despite the much smaller computing task assigned to the mobile device. From the figure, we can also confirm that the computationally weakest and strongest configurations are pairs of Raspberry Pi 3B+ and Jetson TX2, and Jetson TX2 and a desktop computer, respectively.

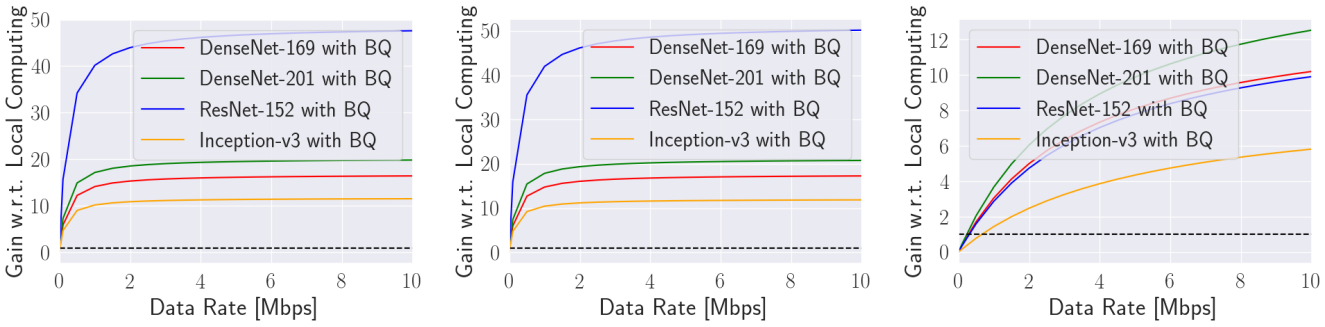
Now, we analyze the communication delay as well. Figure 15 shows the subsampled component-wise capture-to-end delays used to compute the gains in the previous section. As we described in Section V-A, we measured the inference time for local computing (*LC*) and edge computing (*EC*) using the original (teacher) models, and those models are fully deployed on our mobile devices and edge servers. For our student models with bottleneck quantization, we split computing (*SC*), and measured local processing and edge processing time for their split head and tail models (including bottleneck quantization and de-quantization), respectively.

Figure 15a focuses on a setting with a weak edge (Jetson TX2). In this configuration, delay components associated with processing are – comparably – larger than the communication delay. Compared to edge computing, the reduced communication delay offered by the head network distillation technique leads to a smaller capture-to-output delay in impaired channel conditions. The traditional splitting approach suffers either due to high processing load at a weaker platform, or higher communication delay due to the need to transport a larger amount of data to the edge server.

In Figure 15b, a Raspberry Pi 3B+ and high-end desktop computer are used as mobile device and edge server, respectively. Note that this is the most asymmetric configuration we can produce with the considered spectrum of hardware. It can be observed that a weak mobile device strongly penalizes portions of processing executed locally, whereas a limited channel capacity penalizes the data transfer. The split computing approach we propose (*SC*), by reducing the processing load at the mobile device, and reducing the amount of data transferred largely outperforms the best alternative – edge computing (*ES*) – when the network capacity is limited. It can be seen how in this configuration the main issue of the latter option is a large communication delay component, which is only partially offset by the time needed to execute the split head network at the mobile device. Importantly, both distillation and quantization are critical to achieve such results, as they allow a considerable reduction in communication delay. In fact, any other modification of the original model either does not sufficiently reduce the data to be transferred or places an excessive computing load at the weak mobile device unless a degraded accuracy is tolerated.

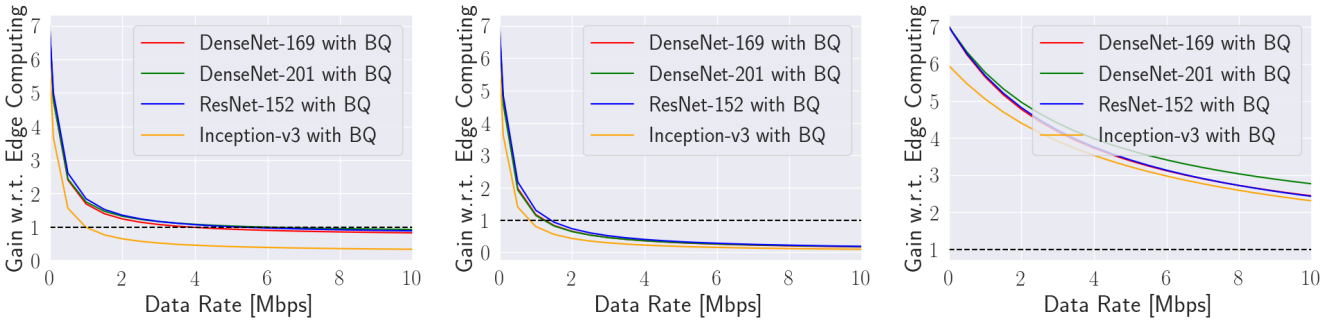
D. INFERENCE TIME OVER REAL-WORLD WIRELESS LINKS

We now analyze the capture-to-output time and its components using emulated networks to provide an evaluation in data rate



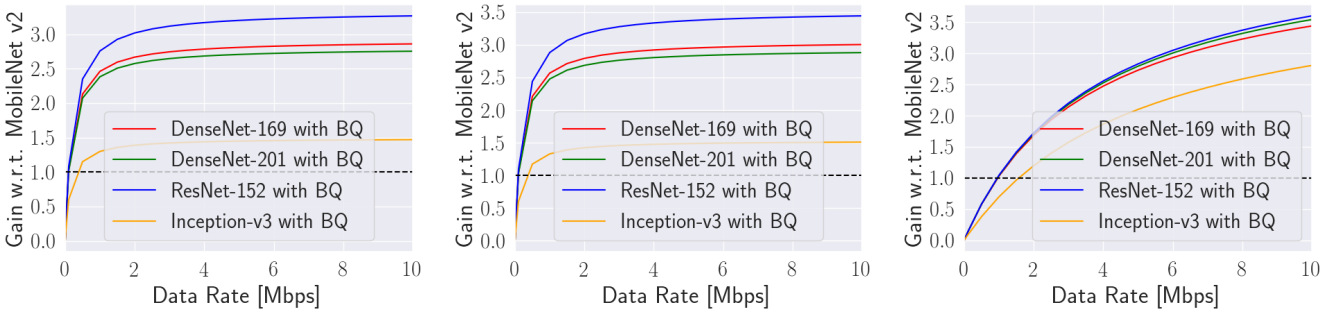
(a) RPI 3B+ (MD) and Jetson TX2 (ES) (b) RPI 3B+ (MD) and desktop (ES) (c) Jetson TX2 (MD) and desktop (ES)

FIGURE 11: Gains with respect to local computing. MD: mobile device, ES: edge server.



(a) RPI 3B+ (MD) and Jetson TX2 (ES) (b) RPI 3B+ (MD) and desktop (ES) (c) Jetson TX2 (MD) and desktop (ES)

FIGURE 12: Gains with respect to edge computing. MD: mobile device, ES: edge server.



(a) RPI 3B+ (MD) and TX2 (ES). (b) RPI 3B+ (MD) and Desktop (ES). (c) Jetson TX2 (MD) and Desktop (ES).

FIGURE 13: Gains with respect to local computing with MobileNet v2 in three different configurations.

ranges achieved in real-world conditions. Specifically, we consider both emulated LTE and WiFi communications and use split models obtained from DenseNet-201 presented in our paper [14] as the overall trend is similar to those for the models presented herein.

LTE Emulation - The full LTE stack is emulated using the opensource software srsLTE [40]. We use Ettus Research's USRP B210 and B200Mini as radio frontend, and run the LTE UE on an UP Board (mobile device) and the eNodeB on a laptop computer with Intel i7-6700HQ, 16GB RAM and NVIDIA Quadro P500. Another UE is connected to the same eNodeB to generate external traffic.

WiFi Emulation - We create an Access Point (AP) using the "hostapd" [41] software on the same laptop which runs the edge server and connect the mobile device. An external node, connected to the same AP, generates traffic over the same wireless channel sharing bandwidth which is set to 54 Mbps as maximum.

In both the networks, we use TCP for the mobile device to edge server data stream and UDP for the external traffic, respectively. The system is deployed in open-field, where the devices are in line-of-sight of each other. To remark how the different parts of the distillation procedure contributes to the final result, we show in the following results for (i) Original model split in the 2^{nd} split point (Org. 2^{nd} SP), (ii) a distilled model without bottleneck (Mimic 2^{nd} SP), (iii) the distilled model with bottleneck (Mimic w/B).

TABLE 5: Delay components and variances for DenseNet-201 in different network conditions.

Model \ Config.	Processing delay*		Delay w/ low traffic (5 Mbps)		Delay w/ high traffic (20 Mbps)	
	Mobile Device [sec]	Edge Server [sec]	Communication [sec]	**Total** [sec]	Communication [sec]	**Total** [sec]
Mobile Device only	1.520 ± 0.007	-	-	$\mathbf{1.520 \pm 0.007}$	-	$\mathbf{1.520 \pm 0.007}$
Edge Server only	-	0.034 ± 0.005	0.091 ± 0.027	$\mathbf{0.125 \pm 0.032}$	0.25 ± 0.11	$\mathbf{0.284 \pm 0.115}$
Org. 2^{nd} SP	0.52 ± 0.005	0.029 ± 0.001	0.096 ± 0.008	$\mathbf{0.715 \pm 0.014}$	0.27 ± 0.05	$\mathbf{0.819 \pm 0.056}$
Mimic 2^{nd} SP	0.0856 ± 0.004	0.0309 ± 0.002	0.0968 ± 0.008	$\mathbf{0.2133 \pm 0.014}$	0.3 ± 0.2	$\mathbf{0.4165 \pm 0.206}$
Mimic w/B	0.062 ± 0.003	0.0225 ± 0.0005	0.008 ± 0.001	$\mathbf{0.0925 \pm 0.0045}$	0.011 ± 0.0007	$\mathbf{0.0955 \pm 0.0042}$

*Processing delay on MD and ES is independent of network conditions and used across the table.

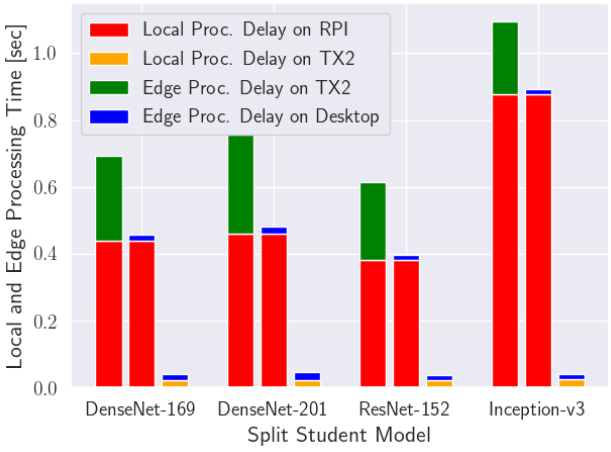


FIGURE 14: Local and edge computing delays for our split student head and tail models in different configurations.

Figure 16 shows the capture-to-output time and its components obtained using the WiFi network as a function of the external traffic load. With increasing external load, the communication delay increases whereas the processing time remains the same. Thus the resulting absolute total delay increases, which is more apparent in configurations where a substantial amount of data is transferred. By minimizing data transfer, our proposed approach is virtually insensitive to channel degradation in the achievable data rate range.

The advantage of our approach is more evident in Figure 17, where we show the mean and variance of the capture-to-output delay. Table 5 reports the value of the various delay components and their variance. We also report the total delay of *Mimic* 2^{nd} *SP*, where a portion of the DNN model is run at the mobile device. *Mimic* 2^{nd} *SP* has a capture-to-output delay larger than that of *Org. ES*, as a portion of processing is executed on a slower device and the amount of data transferred is larger than the actual input. Importantly, the variance of the proposed *Mimic w/B* model is smaller compared to that of both *Org. ES* and *Mimic* 2^{nd} *SP*, mostly due to the fact that a smaller amount of traffic transported over the network reduces protocol interactions such as backoff at the MAC layer and TCP window changes. This observation is confirmed in Figure 18, which shows time series of capture-to-output delay in different traffic conditions. The capture-to-output delay offered by the proposed splitting technique is not only smaller, but is also extremely stable, thus making offloading suitable to mission critical applications.

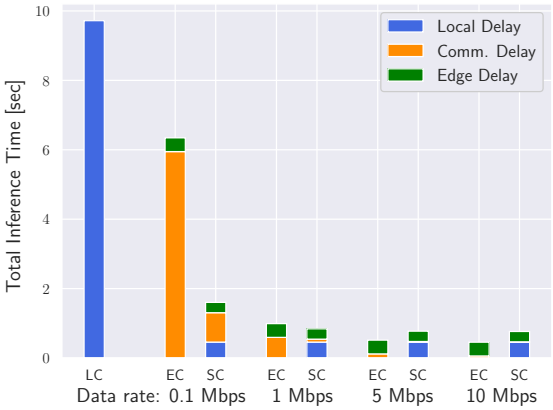
We performed an analogous set of experiments using the LTE network. Note that here we use an UP Board, which is capable of supporting srsLTE. Note that due to the limited processing speed of the UP Board, which is also executing the data processing task, the maximum sampling rate, and thus the data rate, is smaller compared to that achievable by commercial devices. In the considered setup, also due to the power constraints imposed by the programmable radios, the maximum data rate is 3 Mbps. In Figure 19, we observe similar trends to those reported when using high-throughput WiFi-based communications. As the channel degrades, the communication component of the capture-to-output time increases, and the increase is more noticeable if larger amounts of data are transported over the link. Note that LTE mitigates the delay increase when the channel is close to saturation due to the fair scheduling of resource blocks to the two connected devices.

VI. RELATED WORK

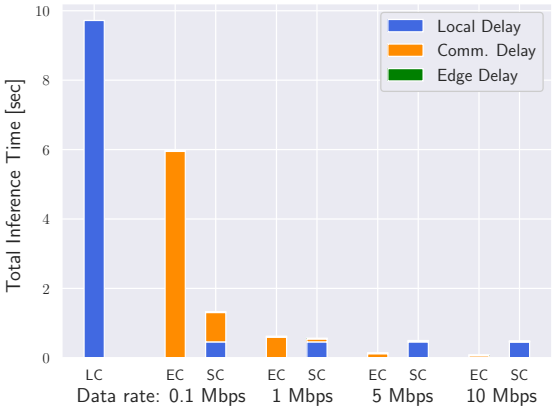
In this section we discuss some of the recent contributions most related to our technique. We note how this area is still rather fragmented, with few clear trends mostly focusing on model reduction and quantization techniques. Recent work investigates techniques to simplify DNN models and make them deployable on mobile devices. Examples include DeepX [9] and distillation frameworks [17]–[21] already mentioned earlier in the paper. However, such family of approaches applied to large DNN models would result either in a loss of accuracy or in unmanageable complexity for weak mobile devices.

Several recent studies investigate techniques to partition DNN models and distribute the processing load. As for distributed computing, Jeong *et al.* [42] and Cha *et al.* [43] discuss distributed training methodologies called federated distillation. Different from these studies, our focus is on split computing for efficient inference rather than training, and our proposed method, head network distillation, can be executed on a single machine. Thus, we describe the related studies on distributed computing for inference.

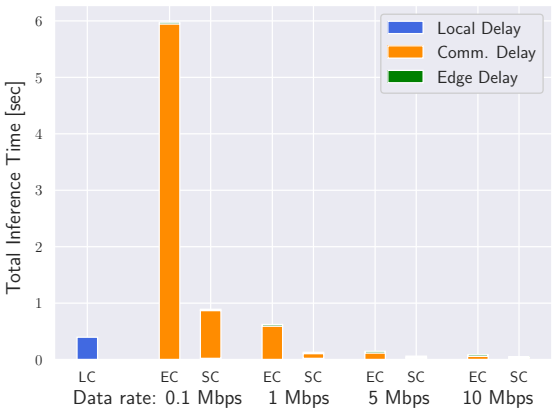
Kang *et al.* [10] and Jeong *et al.* [11] propose partitioning strategies in an edge computing scenario similar to that considered in this paper. However, these approaches do not



(a) RPI 3B+ (MD) and Jetson TX2 (ES).



(b) RPI 3B+ (MD) and Desktop (ES).



(c) Jetson TX2 (MD) and Desktop (ES).

FIGURE 15: Capture-to-output delay analysis for teacher and student models of DenseNet-201. LC: Local Computing, EC: Edge Computing, SC: Split Computing.

consider communication delay as a design principle, and only split the original – unaltered – DNN model. As our results show, such approaches would provide a performance gain only in a very limited parameter region, and is applicable only to DNN models with structural bottlenecks. Moreover, Eshratifar *et al.* [12] demonstrate that from aspects of inference time and energy consumption, it is not an optimal solution to use

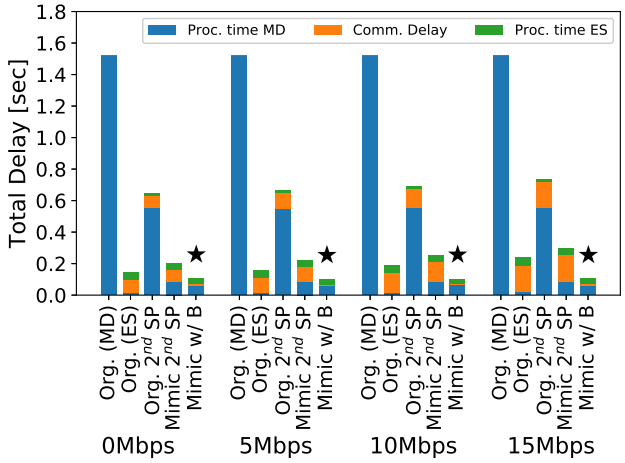


FIGURE 16: Capture-to-output delay and its components for different DNN configurations as a function of the external traffic load.

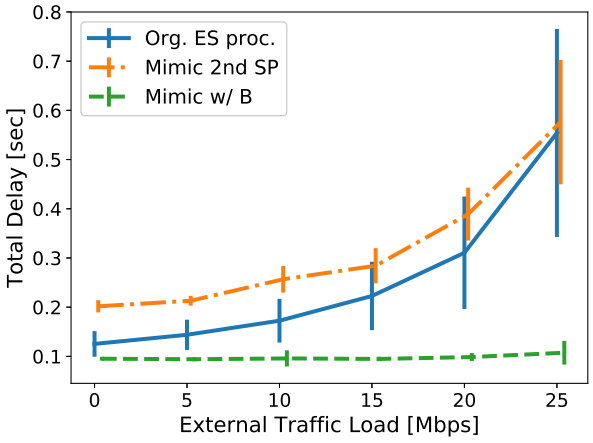


FIGURE 17: Average capture-to-output delay over WiFi as a function of the external traffic load.

either mobile computing only or cloud computing. Similarly, Emmons *et al.* [29] discuss an approach combining DNN splitting and compression methods in vision tasks, and their proposed approach applies transformations (*e.g.*, discrete cosine transform), quantization, and entropy coding to the output from the head portion of DNN models. However, the accuracy degradation of such approach is not evaluated, while insufficient details are provided in the paper to replicate the framework. Li *et al.* [28] proposes a framework using a partitioning and quantization strategy applied to a neural network to reduce the total delay. However, the compression gain is limited, as aggressive quantization will inevitably degrade accuracy. Therefore the technique is limited to relatively good channel conditions.

Teerapittayanon *et al.* [27] proposes Distributed Deep Neural Networks (DDNNs) for cloud and edge servers, and mobile devices. Their approach maps sections of DNNs onto a distributed computing hierarchy and uses a quantization ap-

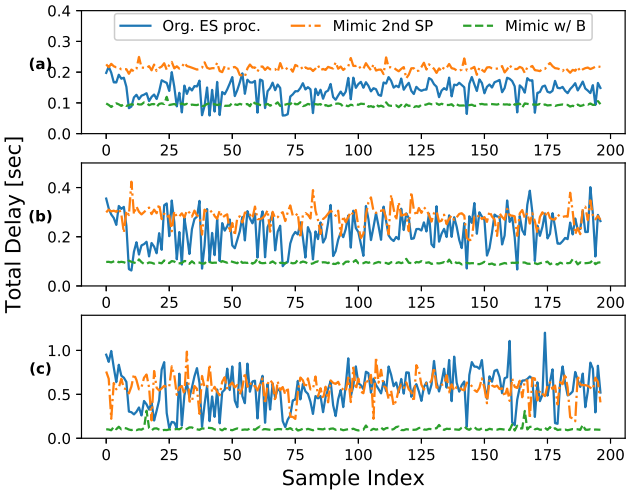


FIGURE 18: Temporal series of capture-to-output per-frame delay over WiFi for (a) low, (b) medium, and (c) high external traffic load.

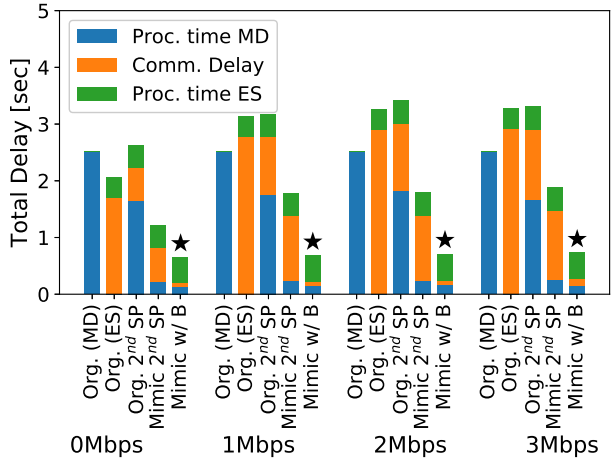


FIGURE 19: Capture-to-output delay and its components over emulated LTE network for different DNN configurations as a function of external traffic load.

proach to reduce communication cost to achieve fast inference. It is reported that DDNN can reduce the communication cost by a factor of over 20 while achieving high accuracy. However, the evaluation is performed on a small dataset, which consists of 851 cropped RGB images (32×32 pixels) with 4 class labels. Also, a comprehensive discussion of the computation burden resulting from the allocation is not provided and the DNN architecture is left unaltered.

In [44], Zhao *et al.* propose DeepThings, a framework for locally distributed and adaptive CNN inference in resource-constrained IoT devices. The proposed framework fuses layers and divides them vertically in a grid fashion instead of horizontally partitioning by layers. Their experiments they show improved memory footprint and inference time using a gateway and up to six edge node devices (Raspberry Pi 3B+). Connected to the considered spectrum of application scenarios, Liu *et al.* [45] propose an edge assisted real-time object detection system for commodity augmented reality system. The framework uses a fast object tracking approach to reducing delay while preserving detection accuracy. We contend that the technique we propose in this paper could be applied to the framework [45] to improve its performance.

Eshratifar *et al.* [13] and we [14] introduced the idea of injecting bottlenecks in vision models to facilitate offloading. Building on this idea, very recent contributions inject a bottleneck in MobileNet v2, VGG-16, and ResNet-50 and use a naive training process [15], [16]. Different from our work, the authors focus on much simpler classification tasks on CIFAR-10 and/or -100 datasets that have 10 and 100 different classes. Similar to [27], the RGB images in both the datasets are fixed as 32×32 pixels, that may be not big enough to validate their offloading techniques, whereas we used RGB input images with $3\times224\times224$ and $3\times299\times299$ pixels in this study. In Table 2, Figures 8 and 9 we demonstrate that their method (*Naive*) is more time-consuming than the method we propose (HND), and that the models trained using our method consistently outperform those trained using their method when considering a complex classification task with the ImageNet dataset. We observe that the methodology in [15] introduce bottlenecks to MobileNet v2 in the same manner as we did in [14], and train the bottleneck-injected MobileNet v2 with the cross entropy loss in an end-to-end manner (naive training), that induces significant accuracy loss on complex tasks as shown in Figure 8.

There are also studies on model compression and pruning [46], [47]. Such network pruning techniques remove redundant neurons with some thresholds to control model size in compression, but require many iterations of complex operations such as pruning and retraining until it converges. Recently proposed neural architecture search (NAS) approaches such as NAS [48] and AutoML [49] design a network using a reinforcement learning algorithm. One of the baseline models used in this study, MnasNet, is designed by an automated platform-aware NAS [23], and we used published, pretrained MnasNet models. As shown earlier, the powerful version, MnasNet (1.0x) is not small enough to run on a weak device, and the smaller version, MnasNet (0.5x) can run as fast as MobileNet v2 on such a device, but sacrifice the accuracy. Comparing to training methodologies used in this study, such neural architecture search methods require a lot of GPU memories and time to learn, but it would be interesting to investigate the approaches to designing a model with bottleneck.

VII. CONCLUSIONS

In this work – which significantly extends our the preliminary study we presented in [14] – we propose head network distillation in conjunction with bottleneck injection to improve the performance of edge computing schemes in some parameter regions. We discussed in detail the structure of the student models we develop to achieve in-network compression while

placing limited amount of computing load to mobile devices and preserve accuracy. It is demonstrated how bottlenecks with a quantization approach can aggressively reduce the communication delay when the capacity of the wireless channel between the mobile device and edge server is limited. We remark that our results are obtained starting from state-of-the-art models and using a complex dataset. In this study, we consider resource-constrained edge computing systems in challenged networks where the data rates are limited ($\leq$ 10Mbps). The study of specific application scenarios with multiple mobile devices and edge servers and operational environments are left for future work.

.

APPENDIX A ARCHITECTURES OF TEACHER AND STUDENT MODELS

Here, we provide architectures of teacher and student head models for DenseNets-169 and -201, ResNet-152 and Inception-v3 in Tables 6, 7, and 8 respectively. Splitting points in our student models are indicated by boldface with an asterisk mark. Note that the tables do not include their tail architectures since architectures of student tail models are identical to those of their teacher tail models.

TABLE 6: Head architectures for DenseNets-169 and -201.

Teacher	Student
Input($3 \times 224 \times 224$)	Input($3 \times 224 \times 224$)
Conv(o=64, k=7x7, s=2x2, p=3)	Conv(o=64, k=7x7, s=2x2, p=3)
BatchNorm	BatchNorm
ReLU	ReLU
MaxP(k=3x3, s=2x2, p=1, d=0)	MaxP(k=3x3, s=2x2, p=1, d=0)
Dense Block(1) [2]	BatchNorm
Transition Layer(1) [2]	ReLU
Dense Block (2) [2]	***Conv(o=12, k=2x2, s=2, p=1)**
Transition Layer(2) [2]	BatchNorm
	ReLU
	Conv(o=512, k=2x2, s=1, p=1)
	BatchNorm
	ReLU
	Conv(o=512, k=2x2, s=1, p=1)
	BatchNorm
	ReLU
	Conv(o=256, k=2x2, s=1, p=0)
	BatchNorm
	ReLU
	Conv(o=256, k=2x2, s=1, p=0)
	BatchNorm
	ReLU
	Conv(o=256, k=2x2, s=1, p=0)
	AvgP(k=2x2, s=2, p=0)

o: output channel, k: kernel size, s: stride, p: padding, d: dilation. Layers with *Italic* font indicate high-level layers which are defined in related studies, and include multiple low-level layers. A bold layer with an asterisk indicates our introduced bottleneck point.

TABLE 7: Head architectures for ResNet-152.

Teacher	Student
Input($3 \times 224 \times 224$)	Input($3 \times 224 \times 224$)
Conv(o=64, k=7x7, s=2x2, p=3)	Conv(o=64, k=7x7, s=2x2, p=3)
BatchNorm	BatchNorm
ReLU	ReLU
MaxP(k=3x3, s=2x2, p=1, d=0)	MaxP(k=3x3, s=2x2, p=1, d=0)
Bottleneck [31]	BatchNorm
Bottleneck [31]	ReLU
Bottleneck [31]	***Conv(o=12, k=2x2, s=2, p=1)**
Bottleneck [31]	BatchNorm
Bottleneck [31]	ReLU
Bottleneck [31]	Conv(o=512, k=2x2, s=1, p=1)
Bottleneck [31]	BatchNorm
Bottleneck [31]	ReLU
Bottleneck [31]	Conv(o=512, k=2x2, s=1, p=1)
Bottleneck [31]	BatchNorm
Bottleneck [31]	ReLU
Bottleneck [31]	Conv(o=512, k=2x2, s=1, p=0)
Bottleneck [31]	BatchNorm
Bottleneck [31]	ReLU
Bottleneck [31]	Conv(o=512, k=2x2, s=1, p=0)
	AvgP(k=2x2, s=1, p=0)

TABLE 8: Head architectures for Inception-v3.

Teacher	Student
Input($3 \times 299 \times 299$)	Input($3 \times 299 \times 299$)
Conv(o=32, k=3x3, s=2x2, p=0)	Conv(o=64, k=7x7, s=2x2, p=0)
BatchNorm	BatchNorm
ReLU	ReLU
Conv(o=32, k=3x3, s=1)	MaxP(k=3x3, s=2x2, p=0, d=0)
BatchNorm	BatchNorm
ReLU	ReLU
Conv(o=64, k=3x3, s=1, p=1)	***Conv(o=12, k=2x2, s=2, p=1)**
BatchNorm	BatchNorm
ReLU	ReLU
MaxP(k=3x3, s=2, p=0, d=1)	Conv(o=256, k=2x2, s=1, p=1)
Conv(o=80, k=1x1, s=1)	BatchNorm
BatchNorm	ReLU
ReLU	Conv(o=256, k=2x2, s=1, p=0)
Conv(o=192, k=3x3, s=1)	BatchNorm
BatchNorm	ReLU
ReLU	Conv(o=192, k=2x2, s=1, p=0)
MaxPool2d(k=3x3, s=2, p=0, d=1)	AvgP(k=2x2, s=1, p=0)

REFERENCES

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[3] N. Lu, N. Cheng, N. Zhang, X. Shen, and J. W. Mark, "Connected vehicles: Solutions and challenges," *IEEE internet of things journal*, vol. 1, no. 4, pp. 289–299, 2014.

[4] M. Di Vaio, P. Falcone, R. Hult, A. Petrillo, A. Salvi, and S. Santini, "Design and experimental validation of a distributed interaction protocol for connected autonomous vehicles at a road intersection," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 10, pp. 9451–9465, 2019.

[5] Y. Dai, D. Xu, S. Maharjan, and Y. Zhang, "Joint load balancing and offloading in vehicular edge computing and networks," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4377–4387, 2018.

[6] Y.-J. Ku and S. Dey, "Sustainable vehicular edge computing using local and solar-powered roadside unit resources," in *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*. IEEE, 2019, pp. 1–7.

[7] M. Zhang, M. Polese, M. Mezzavilla, J. Zhu, S. Rangan, S. Panwar, and M. Zorzi, "Will tcp work in mmwave 5g cellular networks?" *IEEE Communications Magazine*, vol. 57, no. 1, pp. 65–71, 2019.

[8] P. J. Mateo, C. Fiandrino, and J. Widmer, "Analysis of tcp performance in 5g mm-wave mobile networks," in *2019 IEEE International Conference on Communications (IEEE ICC)*. IEEE, 2019, pp. 1–7.

[9] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, ser. IPSN '16.

IEEE Press, 2016, pp. 23:1–23:12.

[10] 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*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 615–629.

[11] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1492–1499.

[12] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: an efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, 2019.

[13] A. E. Eshratifar, A. Esmaili, and M. Pedram, "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, 2019, pp. 1–6.

[14] 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.

[15] D. Hu and B. Krishnamachari, "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, 2020, pp. 157–163.

[16] J. Shao and J. Zhang, "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, 2020, pp. 1–6.

[17] J. Li, R. Zhao, J.-T. Huang, and Y. Gong, "Learning small-size dnn with output-distribution-based criteria," in *Fifteenth annual conference of the international speech communication association*, 2014.

[18] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in neural information processing systems*, 2014, pp. 2654–2662.

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

[20] G. Urban, K. J. Geras, S. E. Kahou, O. Aslan, S. Wang, R. Caruana, A. Mohamed, M. Philipose, and M. Richardson, "Do deep convolutional nets really need to be deep and convolutional?" in *Fifth International Conference on Learning Representations*, 2017.

[21] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton, "Large scale distributed neural network training through online distillation," in *Sixth International Conference on Learning Representations*, 2018.

[22] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[23] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[24] Y. Dauphin and Y. Bengio, "Big neural networks waste capacity," in *ICLR 2013 Workshop*, 2013.

[25] F. Seide, G. Li, and D. Yu, "Conversational speech transcription using context-dependent deep neural networks," in *Twelfth annual conference of the international speech communication association*, 2011.

[26] K. J. Geras, A.-r. Mohamed, R. Caruana, G. Urban, S. Wang, O. Aslan, M. Philipose, M. Richardson, and C. Sutton, "Blending lstms into cnns," in *ICLR 2015 Workshop*, 2015.

[27] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 328–339.

[28] 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*. Springer, 2018, pp. 402–411.

[29] J. Emmons, S. Fouladi, G. Ananthanarayanan, S. Venkataraman, S. Savarese, and K. Winstein, "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*, 2019, pp. 27–32.

[30] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[31] 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.

[32] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Third International Conference on Learning Representations*, 2015.

[35] 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.

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

[37] L. Fei-Fei, R. Fergus, and P. Perona, "One-shot learning of object categories," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 4, pp. 594–611, 2006.

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[39] 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.

[40] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, "srslte: an open-source platform for lte evolution and experimentation," in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*. ACM, 2016, pp. 25–32.

[41] J. Malinen, "Host ap driver for intersil prism2/2.5/3, hostapd, and wpa supplicant," *http://hostap. epitest. fi/*, 2005.

[42] E. Jeong, S. Oh, H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data," *arXiv preprint arXiv:1811.11479*, 2018.

[43] H. Cha, J. Park, H. Kim, M. Bennis, and S.-L. Kim, "Proxy experience replay: Federated distillation for distributed reinforcement learning," *IEEE Intelligent Systems*, 2020.

[44] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[45] 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. 1–16.

[46] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Fourth International Conference on Learning Representations*, 2016.

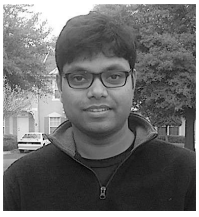
[47] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *Fourth International Conference on Learning Representations*, 2016.

[48] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Fifth International Conference on Learning Representations*, 2017.

[49] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

YOSHITOMO MATSUBARA is a Ph.D. candidate in Computer Science at University of California, Irvine (UCI), working on deep learning for resource-constrained edge computing systems with Profs. Marco Levorato and Sameer Singh. Before UCI, he obtained his Master and Bachelor degrees in University of Hyogo and National Institute of Technology, Akashi College, Japan, respectively. His Master and Bachelor thesis topics were on behavioral biometrics such as keystroke dynamics and flick authentication. His main interests are in machine learning, natural language processing, computer vision and information retrieval.

DAVIDE CALLEGARO obtained both his BE (Information Engineering) and ME (Computer Engineering) from University of Padova, Italy, in 2013 and 2016, respectively. In 2015 he visited Polytechnic University of Catalonia for six months. In Fall 2016 he started his Ph.D in Computer Science, where he currently works as a Ph.D Candidate, under the supervision of his advisor Prof. Marco Levorato. He is interested in task allocation in real-time heterogeneous distributed systems, with focus on performance and reliability trade-offs using reinforcement learning techniques.



SABUR BAIDYA is a Postdoctoral Scholar in the Electrical and Computer Engineering department at the University of California San Diego. His research includes the areas of Internet of Things, wireless networks, intelligent and autonomous systems, and edge-cloud computing. He received his PhD in Computer Science from University of California, Irvine in 2019 and M.S. in Computer Science from University of Texas at Dallas in 2013. In Summer 2012, he was a visiting researcher in the WINLAB at Rutgers University. He has had prior working experience with IBM and Cisco Systems, and internship experience with Blackberry, inc., Huawei Research Lab and Nokia Bell Labs. He is a member of the ACM, IEEE and IEEE Communication society.

MARCO LEVORATO joined the Computer Science department at University of California, Irvine in August 2013. Between 2010 and 2012, He was a post-doctoral researcher with a joint affiliation at Stanford and the University of Southern California. From January to August 2013, he was an Access post-doctoral affiliate at the Access center, Royal Institute of Technology, Stockholm. He is a member of the ACM, IEEE and IEEE Comsoc society. His research interests are focused on next-generation wireless networks, autonomous systems, Internet of Things, and e-health. He has co-authored over 100 technical articles on these topics, including the paper that has received the best paper award at IEEE GLOBECOM (2012). He completed the PhD in Electrical Engineering at the University of Padova, Italy, in 2009. He obtained the B.S. and M.S. in Electrical Engineering summa cum laude at the University of Ferrara, Italy in 2005 and 2003, respectively. In 2016, he received the UC Hellman Foundation Award for his research on Smart City IoT infrastructures and the Dean Research award in 2019.

SAMEER SINGH is an Assistant Professor of Computer Science at the University of California, Irvine (UCI). He is working primarily on robustness and interpretability of machine learning algorithms, along with models that reason with text and structure for natural language processing. Sameer was a postdoctoral researcher at the University of Washington and received his PhD from the University of Massachusetts, Amherst, during which he also worked at Microsoft Research, Google Research, and Yahoo! Labs. He was selected as a DARPA Riser, and has been awarded the grand prize in the Yelp dataset challenge, the Yahoo! Key Scientific Challenges, UCI Mid-Career Excellence in research award, and recently received the Hellman Fellowship. His group has received funding from Allen Institute for AI, Amazon, NSF, DARPA, Adobe Research, Base 11, and FICO. Sameer has published extensively at machine learning and natural language processing venues, including paper awards at KDD 2016, ACL 2018, EMNLP 2019, AKBC 2020, and ACL 2020.