BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services

Amir Erfan Eshratifar, Amirhossein Esmaili, Massoud Pedram
Department of Electrical and Computer Engineering, University of Southern California
Los Angeles, California
Email: eshratif@usc.edu, esmailid@usc.edu, pedram@usc.edu

***Abstract*—Recent studies have shown the latency and energy consumption of deep neural networks can be significantly improved by splitting the network between the mobile device and cloud. This paper introduces a new deep learning architecture, called BottleNet, for reducing the feature size needed to be sent to the cloud. Furthermore, we propose a training method for compensating for the potential accuracy loss due to the lossy compression of features before transmitting them to the cloud. BottleNet achieves on average 5.1× improvement in end-to-end latency and 6.9× improvement in mobile energy consumption compared to the cloud-only approach with no accuracy loss.**

***Index Terms*—deep learning, collaborative intelligence, mobile computing, cloud computing, feature compression**

I. Introduction

Mobile and Internet of Things (IoT) devices are increasingly relying on deep neural networks (DNNs), as these networks provide state-of-the-art performance in various intelligent applications [1]–[5]. Due to limited computational and storage resources of mobile devices, which prohibits full deployment of advanced deep models on these devices (the *mobile-only* method), the most common deployment approach of most of the DNN-based applications on mobile devices relies on using the cloud. In this approach, which is referred to as the *cloud-only* approach, the deep network is fully placed on the cloud, and thus the input is sent from the mobile to cloud for performing the computations associated with the inference network, and the output is sent back to the mobile device.

The cloud-only approach requires mobile devices to send vast amounts of data (e.g. images, audios and videos) over the wireless network to the cloud. This can give rise to considerable energy and latency overheads on the mobile device. Furthermore, pushing all computations toward the cloud can lead to congestion in a scenario where a large number of mobile devices simultaneously send data to the cloud. As a compromise between the mobile-only and the cloud-only approach, recently, a body of research work has been investigating the idea of splitting a deep inference network between the mobile and cloud [6]–[12]. In this approach, which is referred to as *collaborative intelligence*, the computations associated with initial layers of the inference network are performed on the mobile device, and the feature tensor (activations) of the last computed layer is sent to the cloud for the remainder of computations. The main motivation for collaborative intelligence is the fact that in many applications which are based on deep convolutional neural networks (CNNs), the feature volume of layers will shrink in size as we go deeper in the model [6], [8], [12]. Therefore, computing a few layers on the mobile and then sending the last computed feature tensor to the cloud can reduce the latency and energy overheads of wireless transfer of the data to the cloud compared to sending the input of the model directly to the cloud. Furthermore, pushing a portion of computations onto the mobile devices can reduce the congestion on the cloud and hence increase its throughput.

978-1-7281-2954-9/19/$31.00 ©2019 IEEE

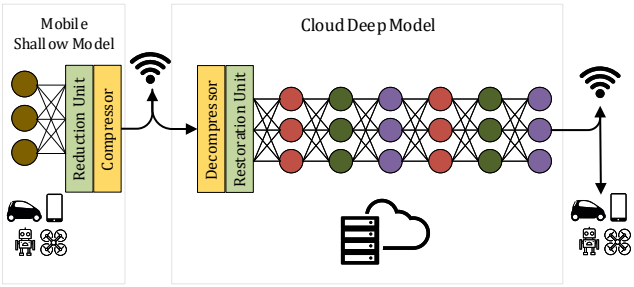

Fig. 1. Overview of the proposed method.

Based on a recent study done for various hardware and wireless connectivity configurations [8], the optimal operating point for the inference network in terms of latency and/or mobile energy consumption is associated with dividing the network between the mobile and cloud, and not the common cloud-only approach, or the mobile-only approach (in case the deep model could be deployed fully on the mobile device). The optimal point of split depends on the computational and data characterization of DNN layers and is usually at a point deep in the inference network.

In research studies investigating collaborative intelligence, a given deep network is split between the mobile device and the cloud without any modification to the network architecture itself [6], [8]–[12]. In this paper, we investigate altering the underlying deep model architecture to make it collaborative intelligence friendly. For this purpose, we mainly focus on altering the underlying deep model in a way that the feature data size needed to be transmitted to the cloud is reduced. This is because in the studies investigating collaborative intelligence, the latency and energy overheads of the wireless data transfer to the cloud yet play a major role in the total mobile energy consumption and the end-to-end latency [6]. Therefore, reducing the transmitted data size to cloud is generally beneficial. For this purpose, we add a non-expensive learnable *reduction unit* after the layers assigned to be computed on the mobile device, and the output of this unit is then compressed using conventional compressors (e.g., JPEG) and sent to the cloud. Correspondingly, a decompressor and a learnable *restoration unit* is added before the layers assigned on the cloud. The main components of reduction and restoration units are convolutional layers which their dimensions are determined in a way that the input of the reduction unit and the output of the restoration unit have the same dimensionality.

An overview of the proposed method is shown in Fig. 1. The

insertion location and size of the the reduction and restoration units in the underlying DNN are determined as explained in Section II-C. Since by inserting the reduction unit, a data bottleneck is created in the model, the combination of the learnable reduction unit, compressor and decompressor, and the learnable restoration unit is referred to as the *bottleneck unit*, and the new network architecture including the bottleneck unit is referred to as *BottleNet*, which is trained end-to-end. For the reduction unit, we evaluate and compare dimension reductions along both the channel and spatial dimensions of intermediate feature tensors as explained in Section II-A.

As we see in Section III, an obvious benefit of using the proposed bottleneck unit is in deep models where feature tensor sizes are relatively high, such as ResNet [2] and VGG [13]. In such networks, the layer in which the feature size is less than the input size is either not present or lies very deep inside in the network. Therefore, if we want to merely split the network and send the intermediate feature tensor to the cloud as in [6], [8], we need to compute considerable number of layers on the mobile. This will push a major workload to the mobile which will likely result in higher latency and energy consumption compared to the cloud-only approach. This could be the main reason that previous work on collaborative intelligence usually has focused on deep architectures where the intermediate feature size is relatively small compared to the input size after computing only a few layers, such as AlexNet [1] and DeepFace [14].

Furthermore, since features of an intermediate layer in a deep model tend to exhibit statistical characteristics such as data redundancy and lower entropy compared to the input of the model, compressing the feature tensor before sending it to the cloud can potentially achieve considerable reductions in the data size needed to be sent over the wireless network. Therefore a major portion of the works studying collaborative intelligence also consider feature compression instead of direct transfer of the feature tensor to the cloud [6], [10]–[12]. In this work, we consider lossy compression of the feature tensor. Lossy compression methods can lead to higher bit savings compared to the lossless compression approaches. However, they may adversely affect the achieved accuracy and thus lossy compression methods are less studied in the works that use feature compression in collaborative intelligence framework, as they mostly use lossless compression techniques on the features before transmitting them to the cloud. In order to compensate for the reduced accuracy due to the lossy compressor, we propose a novel training method for the network which approximates the behavior of the lossy compressor as an identity function in backpropagation. The proposed training method is explained in detail in Section II-B.

In summary, the contributions of this paper are as follows:

- We propose the bottleneck unit, in which by using a learnable reduction unit followed by a lossy compressor, the feature tensor size required to be transmitted to the cloud is significantly reduced.
- For training our model, we propose a lossy compression-aware training method in order to compensate for the accuracy loss.
- Using the proposed BottleNet architecture, compared to the cloud-only approach, we achieve on average 5.1× improvement in end-to-end latency and 6.9× improvement in mobile energy consumption with no accuracy loss.

The remainder of the paper is structured as follows: Section II provides a more detailed explanation of the bottleneck unit and training method. Section III provides the energy and latency improvements of the proposed bottleneck unit and discusses the efficacy of our training approach, and also elaborates on the flexibility of changing the partition point depending on the cloud server congestion and wireless network conditions. Finally, Section IV concludes the paper.

II. Proposed Method

In this section, first, we describe details of the bottleneck unit. Then, we explain our proposed training method when a non-differentiable lossy compression is applied to the intermediate feature tensor before transmitting it to the cloud. Finally, we explain our approach for finding the best insertion location and size of the bottleneck unit in the underlying deep model to achieve the lowest end-to-end latency and/or mobile energy consumption in different wireless settings.

A. Bottleneck Unit

For dimension reduction in the feature tensor in the bottleneck unit, we evaluate and compare dimension reductions along either channel or spatial dimensions. The bottleneck unit, referred to as autoencoder in deep learning context, is responsible for learning a dense representation of the features in an intermediate layer. As depicted in Fig. 2, channel-wise reduction, shrinks the number of channels of the features, and spatial reduction shrinks the spatial dimensions (width and height) of the features. More specifically, channel-wise reduction unit takes a tensor of size ($batch\_size$, w, h, c) as input, and outputs a tensor of size ($batch\_size$, w, h, c') by applying a non-expensive convolution filter of size (1, 1, c, c') followed by normalization and non-linearity layers. The output tensor of the channel-wise reduction unit is the reduced-order representation of its input in terms of channel size ($c' \ll c$). Spatial reduction unit takes a tensor of size ($batch\_size$, w, h, c) as input, and outputs a tensor of size ($batch\_size$, w', h', c) by applying a convolution filter of size (w_f, h_f, c, c) followed by normalization and non-linearity layers. The output tensor of the Spatial reduction unit is the reduced-order representation of its input in terms of width and height ($w' < w$, and $h' < h$). In both channel-wise and spatial reduction units, we use ReLU as the non-linearity function. For reduction in the spatial dimension, the stride step of the convolution should be more than one. It should be noted that to cover each neuron during the convolution at least once, the size of this filter should be more than the stride step size, i.e., $w_f > \frac{w}{w'}$, and $h_f > \frac{h}{h'}$. In this paper, we use the same reduction factor for both width and height, referred to as the spatial reduction factor of s, i.e., $\frac{w}{w'} = \frac{h}{h'} = s$.

The bottleneck unit architecture uses both spatial and channel-wise reduction units followed by a compressor unit on the mobile device to create a compressed representation of the feature tensor which is the tensor transmitted to the cloud. On the cloud, the bottleneck unit uses a decompressor followed by channel-wise and spatial restoration units to restore the dimension of the feature tensor. The detailed architecture of the bottleneck unit is depicted in Fig. 3. The choice of ReLU in reduction units can potentially lead to higher zero rates resulting in higher compression ratios. The bottleneck unit is inserted between two selected consecutive layers of the underlying deep model, where these two layers are selected by an algorithm as explained in Section II-C.

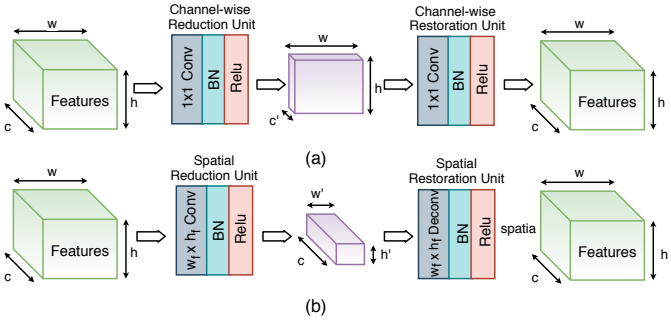

Fig. 2. Learnable dimension reduction and restoration units along the (a) channel and (b) spatial dimension of features.

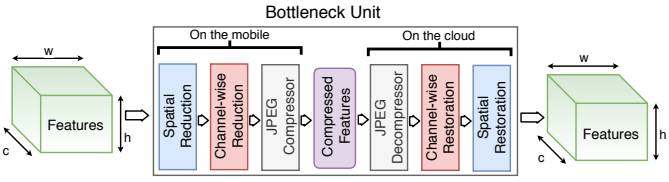

Fig. 3. The bottleneck unit embedded with a non-differentiable lossy compression (e.g., JPEG).

B. Non-differentiable Lossy Compression Aware Training

Lossy compression is inherently a non-differentiable function due to the presence of quantization as an integral part of the compression. Introducing non-differentiable functions in a neural network disables the back-propagation because the gradients are not propagated to the layers before the non-differentiable function, resulting in the model not end-to-end trainable. To solve this issue, we introduce a new training method to enable the model to be end-to-end differentiable by defining a gradient for the pair of compressor and decompressor during the backpropagation. In other words, as depicted in Fig. 4, the pair of lossy compressor and decompressor is used as is during the forward propagation, while we treat this pair as an identity function during backpropagation (i.e., gradients passed without any change to the layers before the compressor). Therefore, the whole model will become end-to-end differentiable.

The intuition behind approximating the pair of compressor and decompressor with identity function during backpropagation is the fact that the input to the compressor is equal to the output of the decompressor in a lossless scenario. Because we are incorporating a lossy compression method, assuming this equality during backpropagation serves as an approximation. However, by still considering lossy compression during the forward propagation, the introduced error is given a chance to be compensated for by the next layers. The effectiveness of using this training method instead of simply training of the model without considering the compression will be explained in Section III-C.

The input to the compressors are typically quantized to unsigned n-bit numbers by an uniform quantizer. Similar to [11], to quantize features, F, we use the following quantizer:

$$\tilde{F} = round(\frac{F - min(F)}{max(F) - min(F)} * (2^n - 1)) \qquad (1)$$

In addition, the input to the compressors should be reshaped into 2-D tensors. The features, F, with C channels, are rearranged in a tiled image with the width of $2^{ceil(\frac{1}{2}log_2(C))}$ and the height of $2^{floor(\frac{1}{2}log_2(C))}$ to keep the aspect ratio as square as possible to achieve the maximum compression ratio.

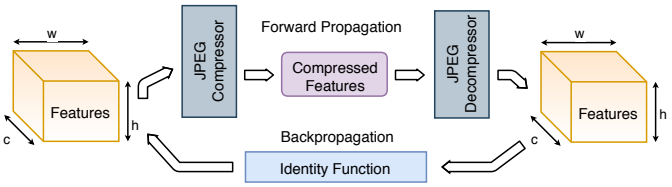

Fig. 4. Embedding non-differentiable compression (e.g., JPEG) in DNN architecture. We approximate the pair of JPEG compressor and decompressor units by identity function to make the model differentiable in backpropagation.

C. Architecture Selection

The proposed algorithm, for choosing the location of the bottleneck unit and the proper value of c' (reflecting the degree of reduction along the channel dimension), and s (reflecting the degree of reduction along the spatial dimension) comprises of three main steps: 1) Training, 2) Profiling, and 3) Selection. We consider placing the bottleneck unit each time after an arbitrary selected layer of the underlying network, for total of M different locations in the network, where M is less than or equal to total N layers of the network. In each of M locations, we train different architectures associated with different degrees of dimension reduction along channel or spatial dimensions, and among those which result in acceptable accuracy levels, we select the one with the minimum bit requirement. We repeat this process for all of M locations. At the end, among M selected models associated with M different partitioning solutions of the network, depending on our optimization target, we choose the best partitioning in terms of minimizing mobile energy consumption and/or end-to-end latency. We measure the latency and mobile energy consumption of computations assigned to the mobile (including reduction and compressor units), wireless transfer of dense compressed feature tensor to the cloud, and computations assigned to the cloud (including decompressor and restoration units). The detailed algorithm for choosing the location of the bottleneck unit and the proper amount of reductions in channel and spatial dimensions is presented in Algorithm 1.

III. Evaluation

A. Experimental Setup

For evaluating our proposed method, we use NVIDIA Jetson TX2 board [15] equipped with NVIDIA Pascal™ GPU with 256 CUDA cores as our mobile platform, while our server platform is equipped with a NVIDIA Geforce® GTX 1080 Ti GPU, which has almost 30x more computing power compared to our mobile platform. We measure the GPU power consumption on our mobile platform using INA226 power monitoring sensor with the sampling rate of 500 KHz [16]. For the wireless network settings, the average upload speed of different wireless networks, 3G, 4G, and Wi-Fi, in the U.S. are used in our experiments [17], [18]. We use the transmission power models of [19] for wireless networks with estimation error rate of less than 6%. The power level for up-link is estimated by $P_u = \alpha_u t_u + \beta$, where t_u is the up-link throughput, and α_u and β are regression coefficients of power models. The values for our power model parameters are presented in Table I.

We prototype the proposed method by implementing the inference networks for both the mobile device and cloud server using NVIDIA TensorRT™ [20], which is the state-of-the-art platform for high-performance deep learning inference. It includes a deep learning inference optimizer and run-time that delivers low latency and high-throughput for

Algorithm 1: The partitioning algorithm for BottleNet

Inputs:
N: number of layers in the DNN
M: number of partitioning points in the DNN ($M \leq N$)
$bottleneck(s, c')$: A bottleneck unit with the spatial reduction factor of s and reduced channel size of c'
S_{max}: maximum allowable spatial reduction factor
C'_{max}: maximum allowable reduced channel size
K_{mobile}: current load level of mobile
K_{cloud}: current load level of cloud
$t_{mobile}, p_{mobile}(j, K_{mobile})|j = 1..M$: latency and power on the mobile corresponding to partition j and load K_{mobile}
$t_{cloud}(j, K_{cloud})|j = 1..M$: latency on the cloud corresponding to partition j and load K_{cloud}
NB: wireless network bandwidth
PU: wireless network up-link power consumption
Outputs:
Best partitioned model
Variables:
$\{D_j|j = 1..M\}$: compressed feature size in each of M partitioning solutions

// Training phase
for $j = 1;\ j \leq M;\ j = j + 1$ **do**
  for $c' = 1;\ c' \leq C'_{max};\ c' = c' + 1$ **do**
    for $s = 1;\ s \leq S_{max};\ s = s + 1$ **do**
      Place $bottleneck(s, c')$ after j-th layer
      Train()
      Store the corresponding model and its accuracy
    end
  end
end
for $j = 1;\ j \leq M;\ j = j + 1$ **do**
  For those models that the bottleneck unit is placed after j-th layer, among ones with acceptable accuracy, store the one with minimum compressed feature size and store its compressed feature size as D_j
end

// Profiling phase
for $j = 1;\ j \leq M;\ j = j + 1$ **do**
  TM_j = $t_{mobile}(j, K_{mobile})$
  PM_j = $p_{mobile}(j, K_{mobile})$
  TC_j = $t_{cloud}(j, K_{cloud})$
  TU_j = D_j/NB // time to upload data to the cloud
end

// Selection phase
if *target is min latency* **then**
  return $argmin_{j=1..M}(TM_j + TU_j + TC_j)$
end
if *target is min energy* **then**
  return $argmin_{j=1..M}(TM_j \times PM_j + TU_j \times PU)$
end

TABLE I
WIRELESS NETWORKS PARAMETERS

Param.	3G	4G	Wi-Fi
t_u (Mbps)	1.1	5.85	18.88
α_u (mW/Mbps)	868.98	438.39	283.17
β (mW)	817.88	1288.04	132.86

deep learning inference applications. TensorRT is equipped with cuDNN [21], a GPU-accelerated library of primitives for deep neural networks. TensorRT supports three precision modes for creating the inference graph, namely FP32 (single precision), FP16 (half precision), and INT8 (8-bit integer). However, our mobile device does not support INT8 operations on its GPU for inference. Therefore, we use FP16 mode for creating the inference graph from the trained model graph, where for the training itself single precision mode is used. As demonstrated in [22], 8-bit quantization would be enough for even challenging tasks like ImageNet [23] classification. Therefore, we apply 8-bit quantization on FP16 data types, using the uniform quantizer presented in Section II-B, before applying the lossy compression on them. Given a partition decision, execution begins on the mobile device and cascades through the layers of the DNN leading up to that partition point. Upon completion of that layer and the reduction unit and lossy compressor, mobile sends the reduced dense feature tensor from the mobile device to the cloud. Cloud server then executes the computations associated with the decompressor, restoration unit, and the remaining DNN layers. Upon the completion of the the execution of last DNN layer on the cloud, the inference result is sent back to the mobile device. For the choice of our lossy compressor, here, we use JPEG compression.

For evaluating our proposed method, we use ResNet-50 and VGG-19 as our underlying deep models, which are two of the widely used networks. ResNet architecture comes with flexible number of layers such as 34, 50, 101. There are 16 residual blocks (RBs) in ResNet-50. We also use the 19-layer version of VGG, VGG-19, in which i-th convolutional layer is referred to as Ci in this paper. In our experiments, we place the bottleneck unit after each residual block in ResNet-50 and after each convolutional layer in VGG-19. Using algorithm 1 explained in Section II-C, we obtain different models where each model is associated with placing the bottleneck unit after a selected layer for both ResNet-50 and VGG-19. As presented in algorithm 1, the obtained c' and s of the bottleneck unit, when it is placed after each of the selected layers could be different, where c' and s were associated with different degrees of reductions in channel and spatial dimensions, respectively.

The input image size of the models and the size of output feature tensor of each layer are presented in Fig. 5. As indicated in Fig. 5, the size of intermediate feature tensors in ResNet-50 are larger than the input size up until RB14, which is relatively deep in the model. Therefore, merely splitting this network between the mobile and cloud for collaborative intelligence may not perform better than the cloud-only approach in terms of latency and mobile energy consumption, since a large portion of the workload is pushed toward the mobile. This is also the same for VGG-19, where the layer in which the feature size is less than input size is 12nd convolutional layer which is quite deep in the model.

We use miniImageNet [24], a subset of ImageNet, as the dataset which includes 100 classes and 600 examples per each class. 85% of whole dataset examples are used as the training set, and the rest as the test set. We randomly crop a 224×224 region from each sample for data augmentation. We train the

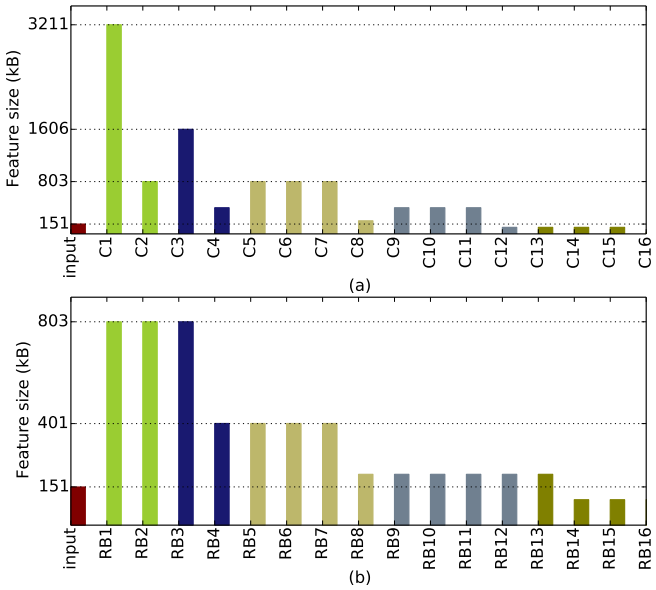

Fig. 5. Input image size and the size of output feature tensor of the convolutional layers in (a) VGG-19 and (b) ResNet-50 models.

models for 90 epochs.

B. Latency and Energy Improvements

The accuracy of ResNet-50 and VGG-19 models for mini-ImageNet dataset without the bottleneck unit are 76.1% and 68.3%, respectively, which serve as our acceptable accuracy values in Algorithm 1. Table II compares the selected partitions with the mobile-only and cloud-only approaches in terms of latency and energy. For the cloud-only approach, before transmitting the input to the cloud, we apply JPEG compression on the input images which are stored in 8-bit RGB format. Note that the best partitioning for the goal of minimum end-to-end latency is the same as the best partitioning for the goal of minimum mobile energy consumption in each wireless network settings. This is mainly due to the fact that end-to-end latency and mobile energy consumption are proportional to each other since the dominant portion of both of them are associated with the wireless transfer overheads of the intermediate feature tensor. Our proposed method can achieve significant dimension reductions and thus bit savings. For instance, when the bottleneck unit is placed after RB1 in ResNet-50, feature tensor of size (56, 56, 256) is reduced to (28, 28, 5) using the channel-wise and spatial reduction units.

Latency Improvement - As demonstrated in Table II, using our proposed method, the end-to-end latency achieves on average 10.6×, 3.4×, 1.3× improvements over the cloud-only approach in 3G, 4G, and Wi-Fi networks, respectively.

Energy Improvement - As demonstrated in Table II, using our proposed method, the mobile energy consumption achieves on average 8.2×, 6.5×, and 5.9× improvements over the cloud-only approach in 3G, 4G, and Wi-Fi networks, respectively.

As observed in Table II, the best partition across all wireless network settings is associated with placing the bottleneck unit after RB1 and C2 for ResNet-50 and VGG-19, respectively. This is an important result since as explained before and according to Fig. 5, due to the relatively large sizes of intermediate feature tensors compared to the input image size for these models, merely splitting the network between the mobile and cloud and transmitting the intermediate feature tensor to the cloud may not perform better than the cloud-only approach in terms of latency and mobile energy consumption. However, using our proposed method, mobile device only computes the first few layers and send the reduced dense output feature tensor to the cloud, achieving minimum latency and mobile energy consumption among all possible partitions and significant improvements compared to the cloud-only approach, while there is no accuracy loss.

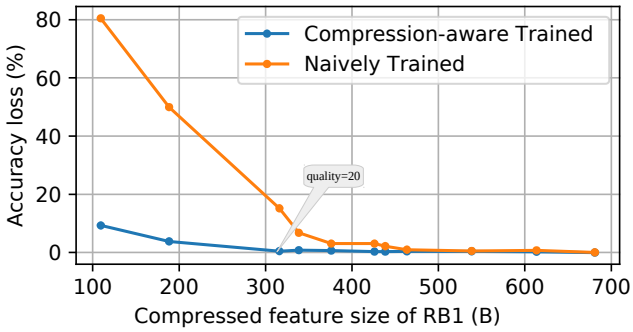

Fig. 6. The comparison between the accuracy loss of the proposed compression-aware training and a naively trained model for different compressed feature size values, when the bottleneck unit is placed after RB1 in ResNet-50.

It is worthwhile to mention that the evaluations are done in a no-congestion setting (K_{mobile} and K_{cloud} in Algorithm 1 are zero). In this scenario, Algorithm 1 is expected to select a layer for the best insertion location the bottleneck unit which is corresponding to an early split in the deep network (RB1 in ResNet-50 and C2 in VGG-19 are selected using Algorithm 1 in no-congestion settings). However, in a scenario where there is a high server utilization on the cloud server, as Algorithm 1 takes the effect of K_{cloud} into consideration during its profiling phase, the best insertion location of the bottleneck unit given by Algorithm 1 is expected to be deeper in the network (bigger portion of computations are pushed toward the mobile device). The effect of the server load is discussed in our prior work [25].

C. The Efficacy of Compression-aware Training

Incorporating the pair of JPEG compressor and decompressor as a new computational unit in a neural network can be performed in two ways: 1) Placing the compression unit in a given trained model (Naive), 2) Training the model from scratch using the proposed compression-aware training method as explained in Section II-B. In JPEG compressor, which is the choice of lossy compressor for our experiments, changing a parameter named *quality* affects the amount of output bits of the compressor. Fig. 6 presents the accuracy loss obtained for ResNet-50 when the bottleneck unit is placed after RB1, versus different number of output bits of the compressor (corresponding to different values of the JPEG quality parameter, ranging from 1 to 100). As depicted in Fig. 6, the accuracy loss of the compression-aware training becomes almost zero by having a JPEG quality level of higher than 20, while the accuracy loss of the naive approach is close to 18% using the quality level of 20. In our experiments, we use the quality level of 20 for JPEG compression in order to achieve maximum bit savings while there is no accuracy loss.

D. Comparison to Other Feature Compression Techniques

In comparison with other works in collaborative intelligence framework which have considered the compression of

TABLE II
COMPARISON OF MOBILE-ONLY AND CLOUD-ONLY APPROACHES WITH THE PROPOSED METHOD (BOTTLENET)

Setup		Latency (ms)		Energy (mJ)		Bottleneck Location		Offloaded Data (B)	
Approach	Network	ResNet-50	VGG-19	ResNet-50	VGG-19	ResNet-50	VGG-19	ResNet-50	VGG-19
Mobile-only	-	15.7	45.3	20.5	59.6	-	-	0	0
Cloud-only	3G	196.2	198.7	310.1	311.9	-	-	26766	26766
	4G	37.9	39.6	168.3	169.7	-	-	26766	26766
	Wi-Fi	13.1	14.9	110.7	112.1	-	-	26766	26766
BottleNet	3G	15.5	23	33.0	44.5	RB1	C2	1580	1720
	4G	9.0	15.5	20.5	35.5	RB1	C2	1580	1720
	Wi-Fi	8.0	14.5	17.5	20.5	RB1	C2	1580	1720

intermediate features before uploading them to the cloud, our proposed method can achieve significantly higher bit savings compared to the cloud-only approach. For instance, as reported in [12], which is one of the few works in collaborative intelligence literature which consider lossy compression of features before transmitting them to cloud, communication overhead in their work can be reduced up to 70% compared to the cloud-only approach. However, in our work, with the proposed trainable reduction unit for spatial and channel dimensions, and the proposed lossy compression-aware training method, as presented in Table II, we can achieve on average 16.3× bit savings compared to the cloud-only approach with no accuracy loss. This shows that in collaborative intelligence framework, adding a learnable reduction in channel and spatial dimension alongside with a compression-aware training method can significantly perform better than merely splitting a network with fixed weights and compressing the intermediate feature tensor before uploading to the cloud.

IV. CONCLUSION

Recent studies have shown that the latency and energy consumption of deep neural networks in mobile applications can be reduced by splitting the network between the mobile and cloud in a collaborative intelligence framework. In this work, we develop a new partitioning scheme that creates a bottleneck in a neural network using the proposed bottleneck unit, which considerably reduces the communication costs of feature transfer between the mobile and cloud. Our proposed method can adapt to any DNN architecture, hardware platform, wireless network settings, and mobile and server load levels, and selects the best partition point for the minimum end-to-end latency and/or mobile energy consumption at run-time. The new network architecture is trained end-to-end using our proposed compression-aware training method which allows significant bit savings. Our proposed method, across different wireless network settings, achieves on average 5.1× improvements for end-to-end latency and 6.9× improvements for mobile energy consumption compared to the cloud-only approach, while there is no accuracy loss.

REFERENCES

[1] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] K. He *et al.*, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[3] R. Girshick *et al.*, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 1, pp. 142–158, 2016.

[4] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[5] T. Mikolov *et al.*, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[6] A. E. Eshratifar *et al.*, "Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services," *arXiv preprint arXiv:1801.08618*, 2018.

[7] A. E. Eshratifar *et al.*, "Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '18. New York, NY, USA: ACM, 2018, pp. 111–116.

[8] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 615–629, 2017.

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

[10] Z. Chen *et al.*, "Intermediate deep feature compression: the next battlefield of intelligent sensing," *arXiv preprint arXiv:1809.06196*, 2018.

[11] H. Choi *et al.*, "Near-lossless deep feature compression for collaborative intelligence," in *2018 IEEE 20th International Workshop on Multimedia Signal Processing (MMSP)*, Aug 2018, pp. 1–6.

[12] H. Choi *et al.*, "Deep feature compression for collaborative object detection," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, Oct 2018, pp. 3743–3747.

[13] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[14] Y. Taigman *et al.*, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1701–1708.

[15] "Jetson TX2 Module," https://developer.nvidia.com/embedded/buy/jetson-tx2, 2018.

[16] "INA Current/Power Monitor," http://www.ti.com/product/INA226.

[17] "State of Mobile Networks in USA," https://opensignal.com/reports/2017/08/usa/state-of-the-mobile-network, 2017.

[18] "United States Speedtest Market Report," http://www.speedtest.net/reports/united-states/, 2017.

[19] J. Huang *et al.*, "A close examination of performance and power characteristics of 4g lte networks," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 225–238.

[20] "NVIDIA TensorRT," https://docs.nvidia.com/deeplearning/sdk/tensorrt-api/index.html, 2018.

[21] S. Chetlur *et al.*, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[22] S. Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[23] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. Ieee, 2009, pp. 248–255.

[24] O. Vinyals *et al.*, "Matching networks for one shot learning," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. USA: Curran Associates Inc., 2016, pp. 3637–3645.

[25] A. E. Eshratifar *et al.*, "Towards collaborative intelligence friendly architectures for deep learning," in *20th International Symposium on Quality Electronic Design (ISQED)*, March 2019, pp. 14–19.