# BottleFit: Learning Compressed Representations in Deep Neural Networks for Effective and Efficient Split Computing

Yoshitomo Matsubara†, Davide Callegaro†, Sameer Singh†, Marco Levorato†, and Francesco Restuccia*
†Department of Computer Science, University of California, Irvine, United States
*Institute for the Wireless Internet of Things, Northeastern University, United States
E-mails: {yoshitom, dcallega, sameer, levorato}@uci.edu, frestuc@northeastern.edu

***Abstract*—Although mission-critical applications require the use of deep neural networks (DNNs), their continuous execution at mobile devices results in a significant increase in energy consumption. While edge offloading can decrease energy consumption, erratic patterns in channel quality, network and edge server load can lead to severe disruption of the system's key operations. An alternative approach, called *split computing*, generates compressed representations within the model (called "bottlenecks"), to reduce bandwidth usage and energy consumption. Prior work has proposed approaches that introduce additional layers, to the detriment of energy consumption and latency. For this reason, we propose a new framework called *BottleFit*, which, in addition to targeted DNN architecture modifications, includes a novel training strategy to achieve high accuracy even with strong compression rates. We apply BottleFit on cutting-edge DNN models in image classification, and show that BottleFit achieves 77.1% data compression with up to 0.6% accuracy loss on ImageNet dataset, while state of the art such as SPINN loses up to 6% in accuracy. We experimentally measure the power consumption and latency of an image classification application running on an NVIDIA Jetson Nano board (GPU-based) and a Raspberry PI board (GPU-less). We show that BottleFit decreases power consumption and latency respectively by up to 49% and 89% with respect to (w.r.t.) local computing and by 37% and 55% w.r.t. edge offloading. We also compare BottleFit with state-of-the-art autoencoders-based approaches, and show that (i) BottleFit reduces power consumption and execution time respectively by up to 54% and 44% on the Jetson and 40% and 62% on Raspberry PI; (ii) the size of the head model executed on the mobile device is 83 times smaller. We publish the code repository for reproducibility of the results in this study.**

***Keywords*-Split Computing, Split Deep Neural Networks, Edge Computing**

## I. Introduction

Emerging mobile applications, such as real-time navigation and obstacle avoidance on drones [1]–[3], increasingly require the execution of complex inference tasks. The key challenge in supporting these applications is that state-of-the-art deep neural network (DNN) models are characterized by computational requirements that go far beyond what the vast majority of mobile devices can offer today. Recent trends approach this problem by (*i*) reducing model complexity using knowledge distillation (KD) [4] and model pruning/quantization [5], and (*ii*) designing lightweight models such as MobileNet [6] and MnasNet [7], at the expense of significant accuracy loss, *e.g.*, up to 6.4% compared to the large convolutional neural network (CNN) model ResNet-152 [8].

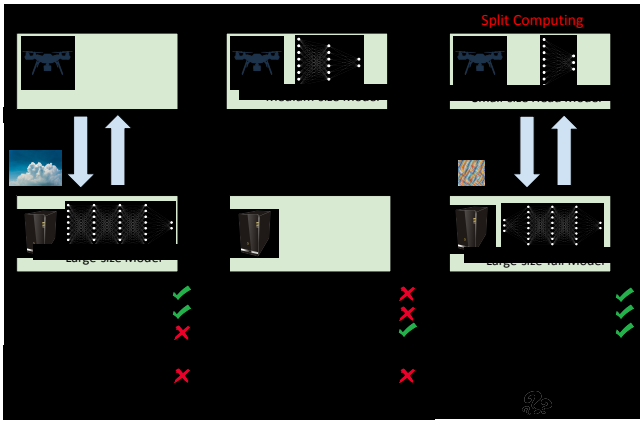


Fig. 1: Pros and cons of edge, local and split computing.

An alternative approach is edge computing [9]–[11], where the execution is completely offloaded to edge servers. While in some scenarios high-throughput links are possible – *e.g.*, a 5G link in Line of Sight (LoS) – the channel quality may suddenly fluctuate, thus impairing performance. For instance, protocol interactions triggered by mobility and impaired propagation have been proven to induce erratic capacity patterns even in high-capacity 5G links [12]. Moreover, most Internet of Things (IoT) devices do not support high data rates, and instead rely on lower-power technologies such as WiFi and LoRa [13], the latter having maximum data rate of 37.5 kbps due to duty cycle and other limitations [14].

The challenging edge-device dilemma can be tackled with *split computing* [15], which is depicted in Fig. 1. The idea is to split a larger DNN into head and tail models executed at the mobile device and edge server, respectively. In the most advanced splitting strategies, the original model is modified by introducing a "bottleneck" layer [16]–[22]. The core intuition behind this strategy is to let the part of the model before the bottleneck learn a *compressed* feature representation. This way, the output tensor of the head model – which is designed to be smaller than the input size – is transmitted over the

This work was supported by the Intel Corporation and the NSF grant MLWiNS-2003237, CNS-2134973 and CNS-2120447.

channel to the edge server instead of the input data. The compressed representation is then used by the tail network to produce the final prediction output (*e.g.*, classification), which is sent back to the mobile device. Notice that different from federated learning [23], training is performed offline (*e.g.*, at edge/cloud servers), while split computing occurs at runtime.

**Challenges.** The success of split computing critically hinges on its capability to substantially decrease energy and latency with respect to traditional computational paradigms. *In other words, the key issue is to design bottleneck architectures that can achieve the best trade-off between head model compression and tail model accuracy.* This problem is extremely challenging, since the introduction of a bottleneck with high compression rate usually comes to the cost of severe detriment in accuracy. The key innovation, therefore, would be to design custom-tailored training strategies to maintain high accuracy despite the bottleneck compression.

We summarize below the core contributions of this paper.

• We present `BottleFit`, a novel framework for split computing. The key innovation of `BottleFit` is a novel multi-stage training strategy to achieve high accuracy even with strong compression rates. In short, in the first stage we pretrain *encoder* and *decoder* structures built around the bottleneck to mimic the intermediate representations in the original pretrained model (without bottlenecks), while freezing parameters in the subsequent layers. Then, we perform a refinement stage where we adapt the compressed representation learnt in the first stage to a target task;

• We apply `BottleFit` on cutting-edge CNNs such as DenseNet-169, DenseNet-201 and ResNet-152 on the ImageNet dataset, and compare the accuracy obtained by `BottleFit` with state-of-the-art local computing [6] and split computing approaches [16]–[19], [21], [24]. Our training campaign concludes that `BottleFit` achieves up to 77.1% data compression (with respect to JPEG) with only up to 0.6% loss in accuracy, while existing mobile and split computing approaches incur considerable accuracy loss of up to 6% and 3.6%, respectively. For the sake of completeness, we compare `BottleFit` with an autoencoder-based approach, which loses up to 16% in accuracy with respect to `BottleFit`.

• We experimentally measure the power consumption and latency of an image classification application. We consider two configurations (i) a Jetson Nano board (GPU-based) communicating with the edge server (a high performance Laptop) using WiFi and Long-Term Evolution, and (ii) a Raspberry PI board (GPU-less) communicating using LoRa. We show that `BottleFit` decreases power consumption and latency respectively by up to 49% and 89% with respect to (w.r.t.) local computing and by 37% and 55% w.r.t. edge offloading. Notably, our head model is 83 times smaller compared to state-of-the-art autoencoder-based approaches.

• Finally, we release the code repository of `BottleFit` and trained model weights produced in this study to ensure reproducibility.[1]

[1] https://github.com/yoshitomo-matsubara/bottlefit-split_computing

## II. Related Work

Many different aspects of mobile edge computing have been extensively investigated [25], [26]. The key assumption of edge-based computing is that wireless links are stable and high capacity, which is seldom the case in many practical scenarios. For this reason, split computing was introduced to divide the computation between the mobile device and the edge [15]. Early contributions, such as [15], split the neural model while leaving unaltered its architecture. This approach has been shown to grant limited improvement over traditional local or edge computing in most application scenarios. Building on this framework, other contributions introduce compression at the splitting point [27] and consider autoencoders and generative models to provide computing options alternative to local and edge computing [28]. SPINN [29] improves the performance Neurosurgeon by introducing "early exits" in existing models, but also incurs a significant deradation of the original accuracy in complex tasks.

In this paper, we focus on bottleneck-injected models to realize effective and efficient split computing. Most of the existing studies train the altered models from scratch [16], [18], [19]. Others reuse pretrained parameters in available architectures for the tail model, while re-designing and re-training the head portion to introduce a bottleneck [17], [21], [22]. These latter contributions introduce the notion of Head Network Distillation (HND) and Generalized HND (GHND), that use knowledge distillation in the training process. None of these studies provide a comprehensive discussion on training methodologies, and the bottleneck-injected models are either not assessed [30] or assessed in relatively simple classification tasks [16]–[20] such as miniImageNet, Caltech 101, CIFAR-10 and -100 datasets. For example, CIFAR datasets present an input RGB image of $32 \times 32$ pixels, that is, transferring the image would require a very short time and the compression granted by split computing strategies is likely unnecessary. Furthermore, many modern applications require higher resolution images and more complex classes. In this paper, we use the large-scale ImageNet dataset [31] for image classification,[2] where the most common input tensor size for CNN models is $3 \times 224 \times 224$, which gives us a strong motivation to compress tensors for split computing.

Some recent architectures make use of autoencoders at the splitting point [32]. We thoroughly compare our approach, where we directly modify the layers of the model to embed a bottlenck, with those frameworks in Section IV-C and show that `BottleFit` greatly improves accuracy. We note that the introduction of autoencoders in the models increases the overall computational complexity. For instance, compared to DeepCOD [32] our head model is 83 times smaller in the worst case. This leads to a reduction in energy consumption at the mobile device of up to 54% (on a Jetson Nano) and 44% (Raspberry Pi 4), as well as lower encoding time – up to 86% less on the Jetson Nano and 82% on the Raspberry Pi 4 – and overall execution time.

[2] https://image-net.org/

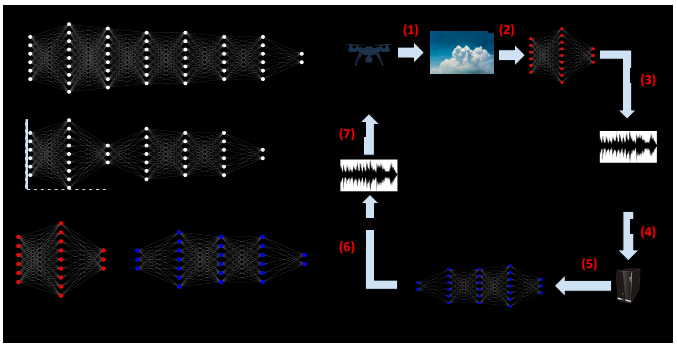



Fig. 2: (left) Split computing: the original model is redesigned with a "bottleneck" and then split into head and tail sections. (right) Example of split computing system. Note that the training process is not split but done offline.

## III. BottleFit: Proposed Framework

We first introduce the system model in Section III-A, describe the bottleneck design in Section III-B, then we present our new training strategy in Section III-C, and finally illustrate the performance tradeoff guiding our system design in Section III-D.

### A. System Model and Preliminaries

The core idea behind *split computing* is to divide a DNN into *head* and *tail* models, which are executed on the mobile device and the edge server at runtime, respectively. The left side of Fig. 2 shows an example of split computing with bottlenecks, while the right side shows a concrete example of a split computing system. When split computing is used in our setting, the mobile device captures high-resolution images (step 1), which are then fed to the head model, ultimately tasked to produce a compressed representation by its output tensor (step 2). The result is transmitted over the wireless channel (step 3) and received by the edge server (step 4). The compressed representation, then, is fed to the tail model to produce the inference output such as predicted class label (step 5), which is sent over the wireless channel (step 6) and received by the mobile device (step 7).

In the following, the notation $\mathcal{R}$ indicates the set of real numbers. Without loss of generality, we consider a deep neural network (DNN) model defined as a mapping $f(\mathbf{x}_i;\theta) : \mathcal{R}^i \rightarrow \mathcal{R}^o$ of an input representation $\mathbf{x}_i \in \mathcal{R}^i$ to an output representation $\mathbf{x}_o \in \mathcal{R}^o$. By defining as $L$ the number of layers, the DNN mapping is computed through $L$ subsequent transformations given a model input $\mathbf{x}$:

$$\mathbf{o}_j = \begin{cases} \mathbf{x} & j = 0 \\ f_j(\mathbf{o}_{j-1}, \theta_j) & 1 \leq j \leq L \end{cases} \tag{1}$$

where $\mathbf{o}_j$ and $\theta = \{\theta_1, \ldots, \theta_L\}$ are the output of the $j$-th layer and the set of parameters of the DNN. Note that $f_j$ can be a low-level layer (*e.g.*, convolution, pooling and fully-connected layers), but also a high-level layer consisting of multiple low-level layers such as residual block or "shortcut" in ResNet models [8], and dense block in DenseNet models [33].

### B. Bottleneck Design

To build and define the bottlenecks, we introduce *encoder* and *decoder* structures within an original pretrained model. Specifically, we replace the first $l_{\text{ed}}$ layers in the original model with an encoder and decoder. The former structure is positioned from the first layer to the bottleneck, and plays a role of "compressor", generating a compact tensor from the input sample. The latter is composed of the layers as part of the tail model that decompress the encoded object, *i.e.*, the bottleneck's output to recreate the output of an intermediate layer. *We point out that while traditional autoencoders compress and reconstruct an input, we modify the layers to operate in an encoder-decoder fashion, which (i) maps the model input to an intermediate output and (ii) is trained to execute a given downstream task without excessive loss in accuracy. We show in Section IV-C that autoencoders lose about 16% in accuracy with respect to our approach.*

Clearly, the design of the encoder/decoder (*e.g.*, position in the model, dimension of the bottleneck) influences the tradeoff between computing load at the mobile device, overall complexity and compression gain, which ultimately determines key performance metrics such as energy consumption, delay and accuracy. Thus, when introducing bottlenecks, we need to carefully (i) design the encoder and decoder; (ii) choose the bottleneck placement in the head model; (iii) preserve the accuracy of the unaltered original model as much as possible. The following sections address all these aspects to build the `BottleFit` framework.

**Bottleneck modeling:** Given an original pretrained model consisting of $N$ layers, we design and introduce bottlenecks to the model, and retrain the bottleneck-injected model to preserve accuracy as much as possible. A bottleneck-injected model is composed of $n < N$ layers and takes as input a tensor whose shape is identical to that for the original model. As illustrated in Fig. 3, the head model $\mathcal{H}$ and tail model $\mathcal{T}$ are built by splitting the model at the bottleneck layer $\mathcal{B}$ (=$f_{k^*}$). The head model $\mathcal{H}$ overlaps with the encoder $f_{\text{enc}}$, composed of $k^*$ layers *i.e.*,

$$\mathcal{H} = f_{\text{enc}} = \begin{cases} \mathbf{h}_0 = \mathbf{o}_0 = x \\ \mathbf{h}_j = f_j(\mathbf{o}_{j-1}, \theta_j) & 1 \leq j \leq k^* - 1, \\ \mathbf{h}_{k^*} = \mathcal{B}(\mathbf{o}_{k^*-1}, \theta_{\mathcal{B}}) \end{cases} \tag{2}$$

where $\theta_{\mathcal{B}}$ is the set of parameters of the bottleneck layer $\mathcal{B}$, and $\mathbf{h}_{k^*}$ indicates the bottleneck representation to be transferred from the mobile device to the edge server in inference session.

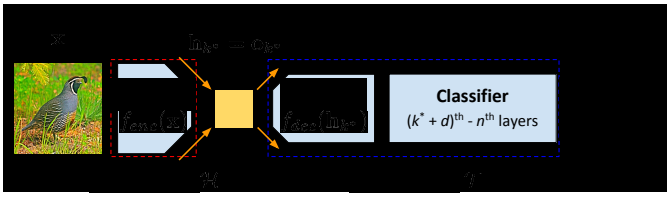



Fig. 3: Model components: encoder, decoder and classifier. Note that the last component, classifier, consists of the last $(n - (k^* + d))$ layers in the original pretrained model.

The bottleneck representation $\mathbf{h}_{k^*}$ is then fed to the tail model $\mathcal{T}$, which consists of the decoder $f_{\text{dec}}$ ($d$ layers where $l_{\text{ed}} = k^* + d$) and the remaining $(n - l_{\text{ed}})$ layers in the original model.

$$\mathcal{T} = \begin{cases} f_{\text{dec}} = \begin{cases} \mathbf{t}_0 = \mathbf{h}_{k^*} \\ \mathbf{t}_{j-k^*+1} = f_j(\mathbf{o}_{j-1}, \theta_j) & k^* < j \leq l_{\text{ed}} \end{cases} \\ \mathbf{t}_{j-k^*+1} = f_j(\mathbf{o}_{j-1}, \theta_j) & l_{\text{ed}} < j \leq n \end{cases} \tag{3}$$

*We remark that different from traditional autoencoders, we do not create additional layers for the bottleneck, thus the overall complexity of a bottleneck-injected model will not increase from that of the original pretrained model.* Instead, to introduce bottlenecks we modify the head portion of the original pretrained models that are often overparameterized.

**Encoder-Decoder Implementation:** To obtain the bottleneck, we then define a new sequence of encoder and decoder layers. Specifically, given a sequence of the first layers in the original pretrained model, we design encoder-decoder architectures using the following steps:

1) Decompose high-level layers (*e.g.*, dense blocks in DenseNet [33] and residual blocks in ResNet [8]) in the sequence into low-level layers such as convolution, batch normalization and ReLU layers;

2) Prune a subset of the layers and define a new sequence by the $l_{\text{ed}}$ remaining layers. If necessary, a set of new layers can be added: deconvolution layers for upsampling after the bottleneck point and/or pooling layers for better convergence;

3) Determine the location of the bottleneck in the new sequence ($k^*$-th layer in the $l_{\text{ed}}$ layers);

4) Adjust layer-specific hyperparameters of layers such as number of output channels, kernel size and kernel-stride to have the sequence's output shape match that expected by the remaining layers in the original pretrained model.

We use convolution layers to create the bottleneck at the $k^*$-th layer ($1 \leq k^* \leq l_{\text{ed}}$) since convolution layers allow us to control their output tensor shape with respect to channel, height and width. Then, we choose 2 consecutive convolution layers in the new sequence to build bottlenecks $\mathbf{o}_{\mathcal{B}} = \mathbf{o}_{k^*}$ at the $k^*$-th layer defined in the previous section, and the following layers gradually decode the compressed representation. To achieve further compression, we also reduce patch size (height and width) of the bottleneck representation. We use a slightly larger kernel-stride size in the early convolution layer(s) to reduce all the output channels, width and height of the output tensor, and introduce a deconvolution layer after the bottleneck layer so that the decoded representation can match the tensor shape expected by the following layers.

### C. Multi-Stage Training Strategy

Our core intuition to preserve accuracy is to maximize the performance of the *encoding* and *decoding* capabilities of the layers neighboring the splitting point to produce an output preserving the overall task performance. Specifically: (*i*) the training strategy should be sophisticated enough to train low-complexity encoders $f_{\text{enc}}$ and (*ii*) the following modules including the decoder should adapt the compressed representations to the downstream tasks.

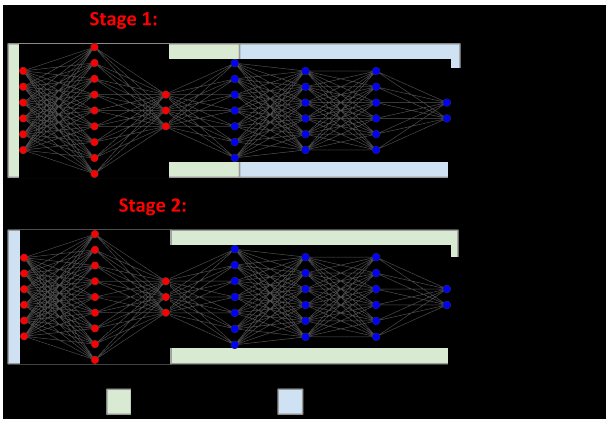



Fig. 4: Proposed multi-stage training method. It is optional to freeze the encoder's parameters in the stage 2.

Figure 4 illustrates our proposed multi-stage training method for bottleneck-injected models. We focus on the training of the compressed bottleneck representations, and adapt the learnt representations to the target task, which in this study is image classification on the ImageNet dataset. In the following, we will refer to the original and modified models as *teacher* and *student* models, respectively.

**Stage 1 – Pretraining Encoder-Decoder:** This stage focuses on training both the encoder $f_{\text{enc}}$ and decoder $f_{\text{dec}}$ in the modified model to reconstruct the representations of the corresponding layer in the original model. At this stage, we use a loss function – Generalized Head Network Distillation (GHND) – which considers the output of multiple layers in the model:

$$\mathcal{L}_{\text{Pre}}(x) = \lambda_{\text{ed}} ||t_{\text{ed}}(x) - f_{\text{dec}}(f_{\text{enc}}(x))||_2^2 + \sum_{j \in \mathcal{J}} \lambda_j ||t_j(x) - s_j(x)||_2^2, \tag{4}$$

where $t_{\text{ed}}$ denotes a function consisting of teacher's layers to be replaced with the encoder $f_{\text{enc}}$ and decoder $f_{\text{dec}}$ in the student model. For example, the function $t_{\text{ed}}$ for DenseNet-169, DenseNet-201 and ResNet-152 consist of layers until (including) the 2nd transition layer in DenseNet-169 and -201 [33], and the 2nd block in ResNet-152 [8], respectively. $j$ is a loss index for a pair of layers that are components of classifier in teacher and student models (*i.e.*, these layers are frozen and in neither encoder nor decoder), and $t_j$ and $s_j$ denote teacher and student model functions of input data $x$ for the loss index $j$ (*i.e*, intermediate outputs of layers in teacher and student models), respectively.[3] $\lambda_*$ is a balancing factor and set to 1 in this study. Importantly, we consider the outputs of frozen layers, in addition to those from the trainable layers during the head network distillation process.

[3]For simplicity, we define $s_j$ as a nested function using the first $K_j$ layers in student model *i.e.*, $s_j(x) = f_{K_j}(f_{K_j-1}(...(f_1(x))))$, and the same applies to teacher model.

**Stage 2 – Adapting Bottleneck to the Target Task:** In this stage, we fine-tune the remaining components of the model to suppress reconstruction errors produced by the encoder-decoder pair while optionally freezing the parameters of the encoder as illustrated in Fig. 4. The classifier learns representations for the downstream task adopting the bottleneck representations learnt to reconstruct the output of the teacher's head model.

We test different approaches to fine-tune the models with encoder-decoder we pretrained in the 1st stage. One could simply use a conventional fine-tuning (FT) method that minimizes a task-specific loss $\mathcal{L}_{\text{task}}$, *e.g.*, a standard cross entropy loss $\mathcal{L}_{\text{CE}}$ defined in Eq. 5 for classification

$$\mathcal{L}_{\text{CE}}(x, y) = -\log\left(\frac{\exp\left(s(x, y)\right)}{\sum_{j\in\mathcal{C}}\exp\left(s(x, j)\right)}\right), \tag{5}$$

where $x$ and $y$ indicate an input image and the associated class label respectively. $\mathcal{C}$ is a set of class labels ($|\mathcal{C}| = 1,000$ for the ImageNet dataset), and $s(x, j)$ is a model's predicted value for a class $j$ given an input image $x$. We can use knowledge distillation (KD) instead of the conventional fine-tuning (FT) approach, then minimize the following loss:

$$\mathcal{L}_{\text{KD}}(x, y) = \alpha\mathcal{L}_{\text{task}}(x, y) + (1-\alpha)\tau^2 KL\left(q(x), p(x)\right), \tag{6}$$

where $\alpha$ is a balancing factor (hyperparameter) between *hard target* (left term) and *soft target* (right term) losses. $\mathcal{L}_{\text{task}}$ is a task-specific loss function, and it is a cross entropy loss in image classification tasks *i.e.*, $\mathcal{L}_{\text{task}} = \mathcal{L}_{\text{CE}}$. $KL$ is the Kullback-Leibler divergence function, where $p(x)$ and $q(x)$ are probability distributions of teacher (*soft-target*) and student models for an input $x$, that is, $p(x) = [p_1(x), \cdots, p_{|\mathcal{C}|}(x)]$ and $q(x) = [q_1(x), \cdots, q_{|\mathcal{C}|}(x)]$:

$$p_c(x) = \frac{\exp\left(\frac{t(x,c)}{\tau}\right)}{\sum_{j\in\mathcal{C}}\exp\left(\frac{t(x,j)}{\tau}\right)}, \quad q_c(x) = \frac{\exp\left(\frac{s(x,c)}{\tau}\right)}{\sum_{j\in\mathcal{C}}\exp\left(\frac{s(x,j)}{\tau}\right)}, \tag{7}$$

where $t(x, j)$ indicates a teacher model's predicted value for a class $j$ given a resized input image $x$, and $\tau$ is a "temperature" (hyperparameter) for knowledge distillation that is normally set to 1. We set $\alpha$ to 0.5 as in [4].

### D. Latency-Power-Accuracy Trade-Off

We break down the end-to-end delay $D$ as the sum of the following random variables: (i) the head model execution time $D_{\mathcal{H}}(k^*, f^*)$, (ii) communication time $D_{\mathcal{N}}(k^*, f^*)$, and (iii) tail model execution time $D_{\mathcal{T}}(k^*, f^*)$. The power consumption at the mobile device is modeled as the sum of the head model execution $P_{\mathcal{H}}(k^*, f^*)$ and communication $P_{\mathcal{N}}(k^*, f^*)$ components. The distributions of these random variables are dependent on the particular mobile device and wireless technology, as well as the model design, and will be computed experimentally. For simplicity, in the following we will drop the $k^*$ and $f^*$ notations. The end-to-end delay then is $D_{\text{e2e}} = D_{\mathcal{H}} + D_{\mathcal{N}} + D_{\mathcal{T}}$, while the power consumption at the mobile device is $P_{\text{md}} = P_{\mathcal{H}} + P_{\mathcal{N}}$. Intuitively, the design of the model also influences the attainable accuracy. The tradeoff between these measures guide our design and can be used to find the best configuration given system-level parameters.

## IV. BottleFit Accuracy Evaluation

In this section, we describe baseline methods and compare their performance with `BottleFit`. Specifically, we will consider ImageNet (ILSVRC 2012) [31], a popular large-scale benchmark dataset for image classification. For ResNet-152, we design two splitting points, called **Splitting Point 1** (SP1) and **Splitting Point 2** (SP2).

**Splitting Point 1 (SP1):** We design an early bottleneck by reducing the output channels of the 2nd convolution layer. This way, we obtain a bottleneck representation whose patch size is $29 \times 29$ for each of the output channels defined in the 2nd convolution layer *e.g.*, $12 \times 29 \times 29$ if the 2nd convolution layer has 12 output channels. Besides the tensor shape, the actual size of the bottleneck representation to be transferred will be explained in Section IV-B.

**Splitting Point 2 (SP2):** SP2 is injected in the third convolution layer. The encoder-decoder architecture we embed (*e.g.*, in ResNet-152) is designed to mimic the output representation shaped $256\times28\times28$. Using a convolution layer, we can reduce the number of channels at bottleneck point from 256 channels to 12 or fewer channels. If we reduce the width and height ($28 \times 28$ in this case) at the bottleneck point (*e.g.*, $14 \times 14$), however, we need then to upsample the compressed tensor with respect to height and width ($14 \times 14$ to $28 \times 28$) by a deconvolution layer so that the decoder output matches the input tensor shape expected by the 2nd block.

We note that later placements in the model would lead to an excessive computing load assigned to the mobile device, which would in turn would result in an increased overall execution time sufficiently large to offset a larger compression gain.

### A. Baseline Evaluation

First, we describe the training strategies used in previous papers that we consider as baseline training methods.

**Conventional Training** In [16], [18], [19], the training strategy applied to bottleneck-injected models is the same conventional strategy used for the training of CNN models. Specifically, training uses cross-entropy (CE) loss defined in Eq. (5) as guiding metric with learning rate adjustment.

**Knowledge Distillation:** We also consider knowledge distillation (KD) [4] as a baseline approach used in the previous study [21]. In KD, the whole model is treated as a "student model" and trained with the original pretrained model (teacher) by minimizing a linear combination of soft-labeled and hard-labeled losses as shown in Eq. (6).

**Head Network Distillation:** In (HND) [17], [21], the the head bottleneck-injected models (student) are trained with the output of the original head model (teacher). This approach does not need human-annotated data (*e.g.*, class label) for training encoder and decoder as the training target is the output from the teacher head model.

TABLE I: Validation accuracy [%]† of models with bottlenecks trained on ImageNet dataset by baseline methods.

| Bottleneck | 12 output channels | | | | | | 3 output channels | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Base Model | DenseNet-169 | | DenseNet-201 | | ResNet-152 | | DenseNet-169 | | DenseNet-201 | | ResNet-152 | |
| Conventional [16], [18], [19] | 66.90 | (-8.700) | 68.92 | (-7.970) | 72.02 | (-11.04) | 60.69 | (-14.91) | 62.85 | (-14.04) | 66.86 | (-11.45) |
| KD [4] | 69.37 | (-6.230) | 70.89 | (-6.000) | 74.06 | (-4.250) | **62.66** | **(-12.94)** | **64.09** | **(-12.80)** | **67.61** | **(-10.71)** |
| HND [21] | **72.03** | **(-3.570)** | **73.62** | **(-3.270)** | **75.13** | **(-3.180)** | 55.18 | (-20.48) | 56.57 | (-20.32) | 53.40 | (-24.91) |

† ImageNet (ILSVRC 2012) test dataset is not publicly available. ‡ Numbers in brackets indicate difference from the original models.

Table I shows the baseline performance on ImageNet dataset [31] by the above three training methods. We empirically find that such bottleneck-injected models trained by the manners used in their studies suffer from degraded accuracy in a complex task, and the accuracy loss is more significant when introducing very small bottlenecks. Furthermore, we confirm that models trained by KD consistently outperform those trained using the conventional training method. Notably, another consistent trend is that models with bigger bottlenecks trained by HND outperform those trained by the other methods, but KD performs best for those with 4 times smaller bottlenecks. We can see that the performance difference between the conventional and KD methods is small compared to that between the HND and KD (or conventional) methods. Interestingly, the performance gap between HND and KD for models with smaller bottlenecks (3 output channels) is, instead, quite perceivable. This implies that compressing bottleneck representations makes it difficult for the models to train with HND, which performs best for those with bigger bottlenecks (12 output channels).

### B. `BottleFit` Evaluation

We train exactly the same models on ImageNet dataset used in Section IV-A, reusing the same training configurations for a fair comparison. Specifically, the number of training epochs is set to 20, with the first and last 10 epochs are dedicated to the 1st and 2nd stages with Adam and SGD optimizers, respectively. The initial learning rates are set to 0.001 in both the optimizers, and decayed by 0.1 after the first 5 epochs in each stage.

**SP1:** We first focus on SP1 and examine the models with the smallest bottleneck (3 output channels) as these models have more room to emphasize the accuracy improvements over the best baseline method for the models (See Table I). We pretrain encoder and decoder at the 1st stage of training, and use both the conventional and KD methods for the 2nd stage with/without freezing encoder pretrained in the 1st stage, thus there are four configurations of the proposed approach. As shown in Table II, all these configurations significantly outperform the best baseline method for models with 3 output channels for bottlenecks, with an accuracy improvement of 4.1 - 5.7 points. Interestingly, the results with the four different configurations are comparable whereas we confirm gaps between the conventional and KD methods in the baseline evaluation (Table I). Given that the proposed multi-stage training methods result in comparable accuracy, we apply the first configuration, Pretraining→FT (FE) to the models with bigger bottlenecks (6, 9 and 12 output channels) as it requires the least training cost among the four configurations.

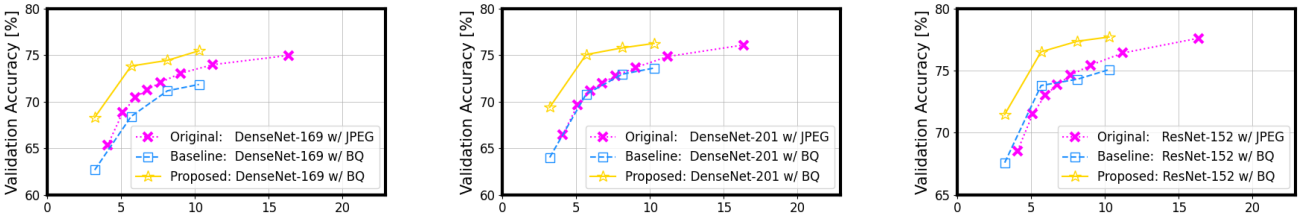
TABLE II: Validation accuracy [%]† of models with bottlenecks (3 output channels) trained by our proposed methods.

| Base Model | DenseNet-169 | DenseNet-201 | ResNet-152 |
|---|---|---|---|
| Best Baseline (KD) | 62.66 | 64.09 | 67.61 |
| Pretraining→FT (FE) | 68.41 | 69.45 | 71.66 |
| Pretraining→KD (FE) | 68.10 | **69.61** | 71.62 |
| Pretraining→FT | **68.43** | 69.41 | 71.50 |
| Pretraining→KD | 68.23 | 69.54 | **71.73** |

FE: Frozen encoder pretrained in the 1st stage

Figure 5 illustrates the trade-off between transferred data size and accuracy. To further compress the size of data to be transferred, we apply a post-training bottleneck quantization (BQ) technique to the output at the bottleneck point. Specifically, the quantization technique proposed in [5] is applied to represent the bottleneck tensor (32-bit floating point) by 8-bit integer tensor and one 32-bit value to dequantize at edge side. Interestingly, as shown in Fig. 5, BQ technique does not degrade the accuracy of the model while reducing the transferred data size by approximately 75%. In Fig. 5, we show the best accuracy of each model in Table I as a baseline. As for the curve with JPEG compression, the transferred data size is average JPEG file size in the ImageNet dataset, resized and center-cropped to $224 \times 224$ pixels for DenseNet-169, DenseNet-201 and ResNet-152. The data size used to compute the total delay for bottleneck-injected models corresponds to the output from the bottleneck (splitting point). *For all the considered models, the approach we propose achieves up to 5.6% improvement in accuracy for each of the models with larger bottlenecks, while the trained models achieve the accuracy of the original pretrained models while reducing 93.3% of tensor elements to be transmitted. Furthermore, this improvement of the trade-off has the bottleneck-injected models significantly outperform the original pretrained models with JPEG compression.* Figure 6 reports the actual data size of model input and bottleneck output used in the inference time evaluation. In addition to about 75% data size reduction of tensor data by bottleneck quantization (32-bit to 8-bit), the quantized bottleneck saves up to 93% compared to the size of JPEG-compressed model input.

**SP2:** We now discuss the trade-off between bottleneck size




(a) DenseNet-169 (b) DenseNet-201 (c) ResNet-152

Fig. 5: Transferred data size vs. accuracy for three different base models. `BottleFit` lifts up the baseline performance of bottleneck-injected models and improves over JPEG compression.

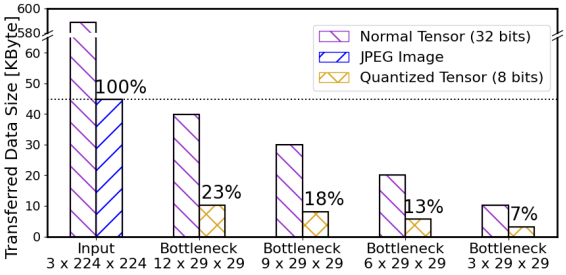



Fig. 6: Tensor size and corresponding transferred data size for our DenseNet-169, DenseNet-201, and ResNet-152.

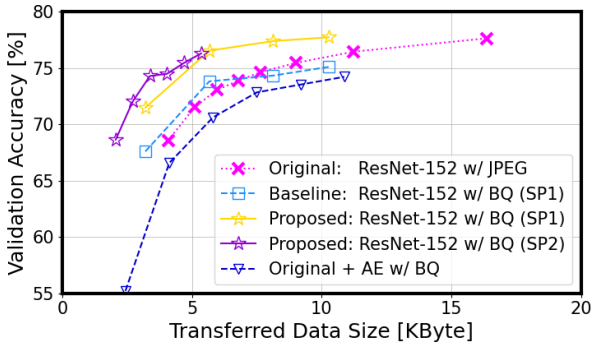



Fig. 7: Comparison of `BottleFit` with (i) our ResNet-152 with bottleneck introduced to its later layer (SP2), and (ii) original ResNet-152 with autoencoder to its 2nd block.

and accuracy in SP2, where the bottleneck reduces by up 98% the amount of transferred data with respect to the input JPEG image. Figure 7 illustrates a trend of trade-off between bottleneck size and accuracy for the models with bottlenecks introduced to their later layers (SP2). As expected, while later bottlenecks impose additional computing load, the accuracy vs compression trend improves, that is, the accuracy degradation is smaller given a compression rate, specifically when further compression (leftward in the figure) is required.

### C. Comparison with Autoencoders

For a comparison purpose, we consider an autoencoder (AE) consisting of $L_{\text{AE}}$ layers (= $L_{\text{AE}}^{\text{Enc}}$ + $L_{\text{AE}}^{\text{Dec}}$ layers for its encoder and decoder) introduced within a pretrained model composed by $L_{\text{M}}$ layers at the $k_{\text{AE}}$-th layer ($1 < k_{\text{AE}} < L_{\text{M}}$). As discussed in Section II, this approach has been widely used in the literature, but has the drawback of increasing models' complexity both of the head and tail. By directly modifying the layers of the models, `BottleFit` creates encoder/decoder structures transforming the output of an intermediate layer into that of a later one through a bottleneck, thus embedding compression in the computing process.

Then, the first $k_{\text{AE}}$ and $L_{\text{AE}}^{\text{Enc}}$ layers in the original model and the encoder of the AE respectively are executed by the mobile device, and the remaining layers ($L_{\text{AE}}^{\text{Dec}}$ and $(L_{\text{M}} - k_{\text{AE}})$ layers) are executed to complete inference. As the additional layers increase computing load, we design lightweight AE using 4 convolution layers (encoder) and 4 deconvolution layers (decoder). We also use batch normalization and ReLU layers between convolution (and deconvolution) layers for better convergence in training. We train AEs on ImageNet for 20 epochs with the following hyperparameters: batch size of 32, initial learning rate of 0.001 decayed by a factor of 10 every 5 epochs. The parameters of AE are learned with Adam optimizer [34] by minimizing a reconstruct loss (sum of squared loss). Due to the limited space, we put our focus on ResNet-152 in this experiment, and inject the AE between the 2nd and 3rd residual blocks. Figure 7 shows the resulting trend compared to the baseline and proposed methods. Interestingly, AEs do not reach even the best baseline methods, while they increase computing load compared to simple and bottleneck-injected model splitting.

## V. Experimental Evaluation

In this section, we extensively evaluate `BottleFit` through a real-world experimental testbed. We first describe our experimental setup in Section V-A, then present the delay and power consumption measurements obtained from Jetson Nano in Section V-B, and finally the delay and power consumption obtained from Raspberry Pi 4 in Section V-C.

### A. Experimental Setup

The experiments were performed indoor in the Donald Bren Hall at University of California, Irvine as shown in Fig. 8. We evaluate `BottleFit` on several configurations of embedded platforms and communication technologies. As mobile device, we use either an NVIDIA Jetson Nano, quad-core ARM 1.9GHz CPU and mounting a 128-core GPU operating at 0.95GHz and 4GB of 64-bit LPDDR4 memory or a Raspberry

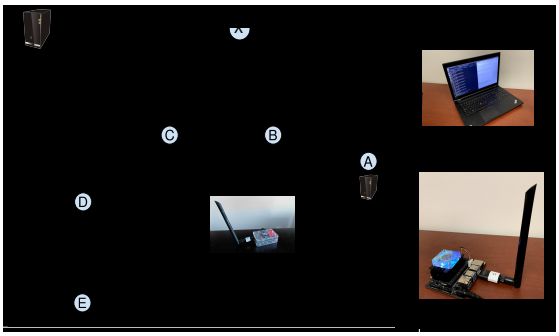



Fig. 8: Experimental setup.

Pi 4, mounting a quad-core ARM 1.5GHz CPU and 2GB of LPDDR4 memory. As edge server, we use a ThinkPad P72 with hexa-core CPU operating up to 4.3GHz, 32GB of memory and NVIDIA GPU Quadro P600 that has 384 cores operating at 1.45GHz as edge server.

In terms of communication technologies we use:

**Wi-Fi:** We use the laptop's Intel Wireless-AC 9560 as hotspot to which the Jetson Nano connects through a Realtek WiFi dongle supporting IEEE 802.11n. The figure shows the location of the edge server, which acts as WiFi hot-spot, to which Jetson Nano board connects as client. We implemented an application generating 300 images in total, at a rate of 5 images/sec. We placed the mobile device in different positions in the building (Fig. 8) to evaluate the impact of link quality on performance. For each of these positions we measure the average quality link – as provided by the `ss` tool in Linux. In the following, we will show the average values as well as 90% confidence intervals.

**LoRa:** the Long Range (LoRa) technology is a widely adopted technology used in Internet of Things (IoT) applications. We extract achievable transmission rates from the networking dataset available in [35], and then, the communication time based on the data size. The total latency is then computed as the sum of the experimental execution time of the models' sections and the communication time.

**Long-Term Evolution:** We use the traces reported in [36] to compute the transmission rate when Long-Term Evolution (LTE) is used by the distributed system. The traces have been collected while driving in densely serviced areas.

### *B. Latency and Power – Jetson Nano/Wi-Fi and LTE*

First, we analyze the end-to-end delay and power consumption in the configuration with Jetson Nano as mobile device and WiFi as a communication technology. Figure 9 shows the end-to-end average delay with 95% confidence interval using edge offloading, and `BottleFit` with $3, 6$ and 12 channels, for each of the four considered models. The results show that in high channel quality regimes, some models work well with edge offloading. However, when the link quality decreases, `BottleFit` offers lower average delay and delay variability. We exclude from these graphs MobileNetV2, whose execution time, independent from channel quality, is $0.17s$ (confidence interval is negligible).

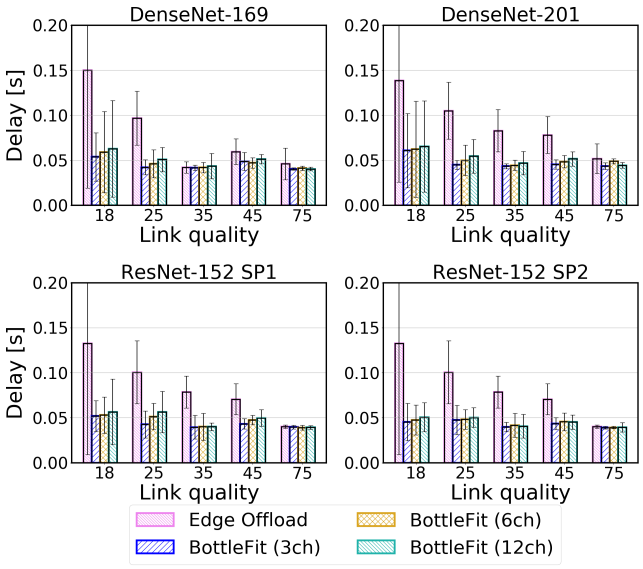



Fig. 9: Delay of the considered models run on **Jetson Nano** in different conditions of the WiFi channel quality. Local processing delay of original models (not displayed): DenseNet-169: $0.27s$, DenseNet-201: $0.42s$, ResNet-152: $0.53s$.

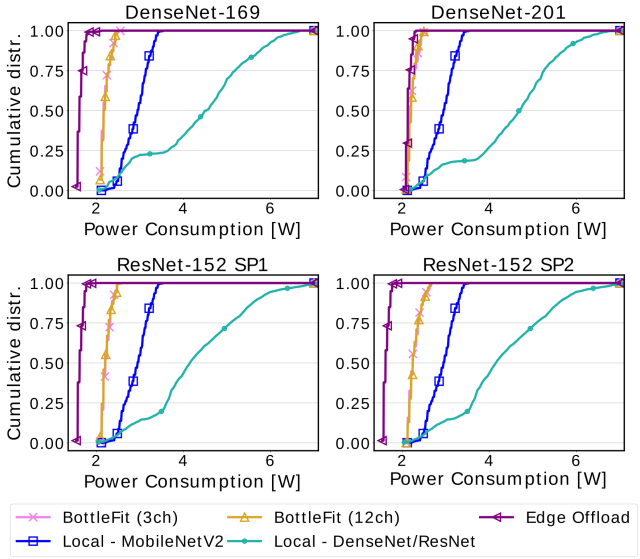



Fig. 10: Cumulative distribution of power consumption for the models considered. We compare edge offloading as well as local processing both using MobileNetV2 and full models (DenseNet/ResNet).

Figure 10 depicts the cumulative distribution of the power consumption – measured at the mobile device – of the considered models. We first observe that the Local DenseNet/ResNet distributions exhibit peak value well over 6W, with average value equal to 4.3W. Being a model designed to be run on mobile devices, MobileNetV2 shows impressive power saving, limiting its range below 4W (and average 2.9W). In line with expectations, we see that in all cases except DenseNet-201, offloading to the edge server is the strategy yielding the lowest power used, which never exceeds 2W (average value 1.6W).

However, as pointed out in Fig. 9, the delay offered by

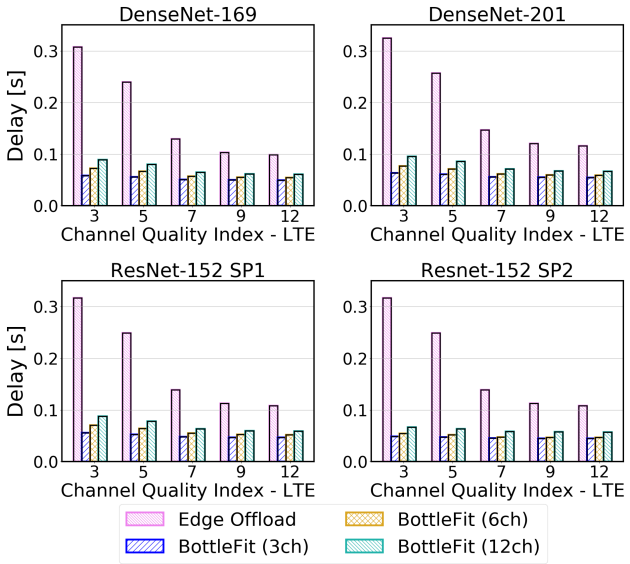



Fig. 11: Delay of the considered models run on **Jetson Nano** in different LTE Channel Quality Index conditions.

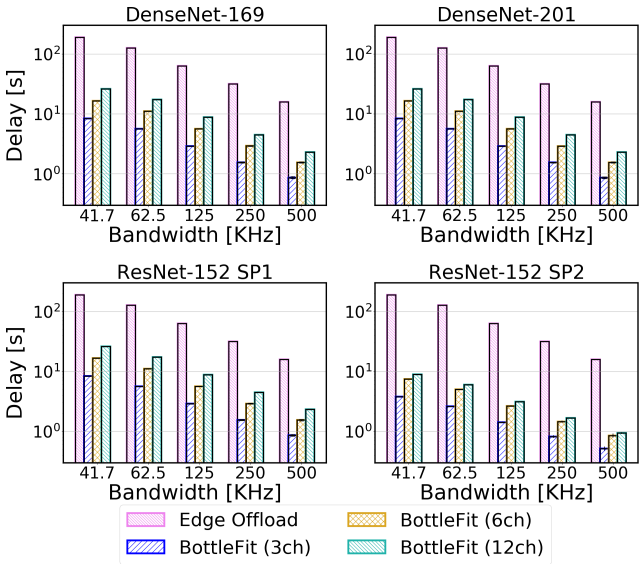



Fig. 12: Delay of the considered models run on **Raspberry Pi 4** with different LoRa bandwidths. Local processing delay of original models (not displayed): DenseNet-169: $2.34s$, DenseNet-201: $2.55s$, ResNet-152: $4.29s$, MobileNetV2: $1.97s$.

edge offloading can exceed the constraint imposed by the application. For this reason, `BottleFit` should be preferred, since with power consumption only $37\%$ higher than edge offloading, and $89\%$ lower than local processing, it offers comparable accuracy when using 12 channels configuration (less than $1\%$ loss in all cases), while delivering the most stable and low delays. Similarly, as shown in Fig.11, when the system employs LTE, we observe a reduction of end-to-end delay in the range $45\% - 70\%$ compared to offloading. Conversely to the WiFi configuration, even for high rates, `BottleFit` always outperforms offloading.

### C. Latency and Power – Raspberry Pi 4/LoRa

We present in Figs. 12 and 13 the delay and power consumption obtained by executing the image classification application on a Raspberry Pi 4 with Long Range (LoRa) connectivity [14], which is not equipped with a GPU. The purpose of these experiments is to evaluate `BottleFit` in the context of a low-power device equipped with low-power, low-rate connectivity. We estimated the network delay by taking the nominal data rates of LoRa with spreading factor 6 and coding rate 4:5 (which yield the highest throughput). As expected, Fig. 12 shows that `BottleFit` outperforms a full edge offloading approach, given the compression provided by the bottleneck. On the other hand, the networking delay given by LoRa makes local computation more delay-effective.

The advantage of local computing with respect to `BottleFit` comes to the detriment of power consumption. Figure 13 shows the average power consumption experienced by the Raspberry Pi 4 as a function of different models. Confidence intervals are not shown as those were below $1\%$. *It indicates that `BottleFit` presents power consumption comparable to edge offloading, while saving up to 50% power consumption with respect to local computing approaches.*

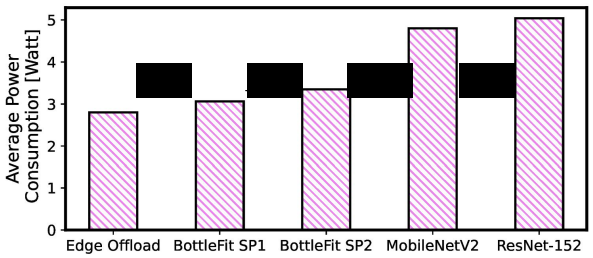



Fig. 13: Average power consumption on Raspberry Pi 4 for the different approaches presented.

## VI. Conclusions

In this paper, we have proposed `BottleFit`, a new framework for split computing. We have applied `BottleFit` on cutting-edge DNN models in image classification, and show that `BottleFit` achieves 77.1% data compression with up to 0.6% accuracy loss on ImageNet dataset. We experimentally measure the power consumption and latency of an image classification application, and shown that `BottleFit` decreases power consumption and latency respectively by up to 49% and 89% with respect to local computing and by 37% and 55% w.r.t. edge offloading. We also compare `BottleFit` with state-of-the-art autoencoders-based approaches, and show that (i) `BottleFit` reduces power consumption and execution time respectively by up to 54% and 62%; (ii) the size of the head model executed on the mobile device is 83 times smaller. To achieve more efficient split computing, it would be essential to further improve the tradeoff between transferred data size and model accuracy while keeping encoder lightweight as discussed in [37], [38].

## REFERENCES


[1] A. Kouris and C.-S. Bouganis, "Learning to Fly by Myself: A Self-supervised CNN-based Approach for Autonomous Navigation," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 1–9.

[2] V. K. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley, "Advanced driver-assistance systems: A path toward autonomous vehicles," *IEEE Consumer Electronics Magazine*, vol. 7, no. 5, pp. 18–25, 2018.

[3] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield, "Toward Low-flying Autonomous MAV Trail Navigation Using Deep Neural Networks for Environmental Awareness," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 4241–4247.

[4] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," in *Deep Learning and Representation Learning Workshop: NIPS 2014*, 2014.

[5] 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.

[6] 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.

[7] 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 Conf. on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[8] 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.

[9] Z. Zhou, H. Liao, B. Gu, K. M. S. Huq, S. Mumtaz, and J. Rodriguez, "Robust mobile crowd sensing: When deep learning meets edge computing," *IEEE Network*, vol. 32, no. 4, pp. 54–60, 2018.

[10] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[11] 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.

[12] 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.

[13] F. Samie, L. Bauer, and J. Henkel, "IoT Technologies for Embedded Computing: A Survey," in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2016, pp. 1–10.

[14] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the Limits of LoRaWAN," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017.

[15] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 615–629.

[16] A. E. Eshratifar, A. Esmaili, and M. Pedram, "BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services," in *2019 IEEE/ACM Int. Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[17] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems," in *Proc. of the 2019 MobiCom Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019, pp. 21–26.

[18] D. Hu and B. Krishnamachari, "Fast and Accurate Streaming CNN Inference via Communication Compression on the Edge," in *2020 IEEE/ACM Fifth Int. Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2020, pp. 157–163.

[19] 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.

[20] M. Jankowski, D. Gündüz, and K. Mikolajczyk, "Joint Device-Edge Inference over Wireless Links with Pruning," in *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE, 2020, pp. 1–5.

[21] Y. Matsubara, D. Callegaro, S. Baidya, M. Levorato, and S. Singh, "Head network distillation: Splitting distilled deep neural networks for resource-constrained edge computing systems," *IEEE Access*, vol. 8, pp. 212 177–212 193, 2020.

[22] Y. Matsubara and M. Levorato, "Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 2272–2279.

[23] J. Konečnỳ, H. B. McMahan, X. Y. Felix, A. T. Suresh, D. Bacon, and P. Richtárik, "Federated learning: Strategies for improving communication efficiency," *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[24] Y. Matsubara and M. Levorato, "Split Computing for Complex Object Detectors: Challenges and Preliminary Results," in *Proceedings of the 4th International Workshop on Embedded and Mobile Deep Learning*, 2020, pp. 7–12.

[25] Z. Zhou, X. Chen, W. Wu, D. Wu, and J. Zhang, "Predictive online server provisioning for cost-efficient iot data streaming across collaborative edges," in *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2019, pp. 321–330.

[26] A. Li, C. Wu, Y. Chen, and B. Ni, "Mvstylizer: An efficient edge-assisted video photorealistic style transfer system for mobile phones," in *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, ser. Mobihoc '20. Association for Computing Machinery, 2020, p. 31–40.

[27] H. Choi and I. V. Bajić, "Deep feature compression for collaborative object detection," in *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2018, pp. 3743–3747.

[28] 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.

[29] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–15.

[30] 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.

[31] 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.

[32] S. Yao, J. Li, D. Liu, T. Wang, S. Liu, H. Shao, and T. Abdelzaher, "Deep compressive offloading: speeding up neural network inference by trading edge computation for network latency," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 476–488.

[33] 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.

[34] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Third International Conference on Learning Representations*, 2015.

[35] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the limits of lorawan," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017.

[36] D. Raca, J. J. Quinlan, A. H. Zahran, and C. J. Sreenan, "Beyond throughput: a 4g lte dataset with channel and context metrics," in *Proceedings of the 9th ACM Multimedia Systems Conference*, 2018, pp. 460–465.

[37] Y. Matsubara, R. Yang, M. Levorato, and S. Mandt, "Supervised Compression for Resource-Constrained Edge Computing Systems," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2022, pp. 2685–2695.

[38] ——, "SC2: Supervised Compression for Split Computing," *arXiv preprint arXiv:2203.08875*, 2022.