Channel-Adaptive Dynamic Neural Networks for Low-Complexity Distributed Signal Awareness

Mohammad Abdi*, Jonathan Ashdown‡, Kurt Turck‡, and Francesco Restuccia*
‡ Air Force Research Laboratory, United States
* Institute for the Wireless Internet of Things, Northeastern University, United States
Corresponding author e-mail: abdi.mo@northeastern.edu

***Abstract*—Deep neural networks (DNNs) are being increasingly used to achieve signal awareness. Usually, DNNs have been entirely executed at the mobile device. However, as DNNs become more complex, edge computing techniques will become necessary for DNN deployment. Notably, it has been shown that the original DNN architectures can be modified by introducing a "bottleneck" producing a compressed representation of the input. The DNN split computation is then orchestrated between the mobile device and the edge, which respectively execute the layers before and after the bottleneck. As yet, the concept of split computing has not been applied for signal awareness. Moreover, in existing work, the dynamic nature of the wireless channel is not considered in the bottleneck design. In this paper, we propose Split Computing Enhancing Signal Awareness (SpliCES), a new approach where the bottleneck is adapted at inference time according to ongoing channel conditions. Moreover, we present a novel training strategy which optimizes the bottleneck-injected ensemble parameters to generalize to all channel conditions in its first stage, and then specializes each bottleneck separately to a certain level of channel impairment in the second stage. We evaluate SpliCES on the well-known RadioML dataset for modulation recognition, and show that on average, our proposed method transmits 12x less data while experiencing only about 3% accuracy drop in high signal-to-noise ratio (SNR) regimes. With low SNR, SpliCES achieves 5% better accuracy thanks to its dynamic structure.**

I. Introduction

Mobile wireless devices are increasingly being used to achieve spectrum awareness across time, space and frequency. Among others, DNNs have been shown to be extremely effective to solve a wide variety of signal awareness problems, for example, detecting spectrum holes and thus achieve better spectrum efficiency [1], radio fingerprinting to identify unauthorized spectrum access [2–4], or modulation classification to decode dynamic waveforms [5–7].

As DNNs increase in size and complexity, so does the computational burden required at the mobile device. Therefore, fog and edge computing techniques [8] will become a compelling necessity to offload DNN-based signal awareness tasks [9]. Traditional edge computing, however, requires frequent spectrum access and bandwidth utilization, which can reveal the presence of mobile devices and exhaust the already limited, congested and contested RF spectrum. **The computational and spectrum constraints of mobile wireless systems require non-traditional and adaptive computational paradigms that can dynamically achieve the right trade-off between effectiveness and efficiency.**

Approved for Public Release; Distribution Unlimited: AFRL-2023-3237.

Among others, split computing (SC) has been proposed as a valid alternative to traditional edge computing [10]. SC is an emerging computational paradigm where the DNN is divided into a *head network* and *tail network*, respectively executed at the mobile device and edge server [11]. To further decrease the communication burden, the head model produces an output that is (much) smaller than the input tensor, with a procedure called synthetic *bottleneck injection* [12–15]. The key intuition is that the representation size decreases only in the late layers of the DNN, providing limited number of splitting point candidates, also known as natural bottlenecks [16]. Overall, SC is an extremely suitable candidate to perform signal awareness tasks in urban scenarios.

The main limitation of existing bottleneck injection techniques is that they completely neglect the dynamic nature of communication environments. Specifically, a fixed-size bottleneck is introduced into the original DNN regardless of the wireless channel condition [17, 18]. However, this is definitely not true in contested and congested spectrum environments, where experiencing a rapidly-changing communication channel is the norm. Such erratic changes may adversely affect the overall signal awareness performance. Moreover, varying communication environments can result in inconsistent task performance. Therefore, we believe taking the dynamic nature of channel into consideration is a necessity. We argue that in a generic SC scenario, one can exploit the signal used for reporting the task result back to mobile device to estimate the channel condition and use an appropriate bottleneck accordingly. **To the best of our knowledge, this is the first work that makes the intermediate feature size in SC adaptive with respect to the channel condition**.

We summarize below the core contributions of this paper:

- We propose a novel two-stage training strategy for dynamic bottleneck design. In the first stage, all the parameters of the bottleneck-injected model ensemble are jointly optimized using the whole dataset in order for the head model to generalize to different channel conditions. In the second stage, the head model is frozen, while different bottleneck-injected models are fine-tuned using channel-conditional data allowing them to specialize to their corresponding channel quality, thus reducing memory footprint on the mobile device;
- We evaluate SpliCES on an Auto Modulation Classification (AMC) task by using the RadioML 2018.a dataset [19]. We were able to achieve high feature compression ratios

while equalizing the performance across different channel conditions, and show that on average, our proposed approach transmits 12x less data while experiencing only about 3% accuracy drop with high SNR. Thanks to its dynamic structure, SpliCES achieves 5% better accuracy than a static approach with low SNR.

II. Related Work

Early work on *local computing* follows two different approaches: (i) some investigate the design of small models meeting the limitations of portable devices [20, 21]; or (ii) adapt state-of-the-art architectures to low-end devices by making them more efficient, using for example, compression techniques [22, 23]. However, local computing suffers from significant performance loss.

At the other end of the spectrum, mobile devices may completely offload the computation to edge computers [24], also called *edge computing*, so as to achieve better task results while relieving the computation burden on mobile devices. However, mobile devices still need to transfer each and every input to the edge, which incurs in excessive communication burden. In addition, these methods fail to strike a balance between exploiting the computational and communication power of mobile devices. As an intermediate option, recent work has proposed to distribute the processing load between the mobile device and the edge server in a way that they can collaboratively execute the model, also known as *split computing* [11]. However, early studies do not alter the original model architecture disregarding communication delay as a design principle. Therefore, such designs depend on the natural bottlenecks as splitting point candidates [25] to reduce the communication cost. Early SC work is typically impractical since representation tensor size often reduces only close to the final output layer [13] which results in an inappropriate distribution of computational burden. Therefore, others suggest methods to compress the intermediate features calculated within the model [26] to reduce the amount of transferred data.

Compression methods include feature compression and supervised compression. Traditional feature compression such as Zip and JPEG [13] or transformations such as quantization and entropy coding may be applied to the output of the head model [27, 28]. The compression gain of such approaches is limited, as aggressive compression will degrade performance significantly [29]. Lightweight autoencoders and generative models for feature compression incur in additional computation and/or performance loss [16, 30]. Task-oriented compression techniques mostly use a procedure called bottleneck injection [12]. [14, 31] use naive training process from scratch, while [13] use Knowledge Distillation (KD) to train the bottleneck-injected model. Building on this idea, very recent work [18] proposes a new training method to improve the performance. **However, the work in [18] does not consider signal awareness and does not propose dynamic bottleneck injection, which is the target of this paper.** In the literature, many algorithms are able to perform signal awareness tasks with large DNN models, for example [19, 32]. Although they proposed various models capable of achieving satisfactory accuracy in performing AMC, most of them cannot be deployed on low-end embedded devices.

III. Overview of Dynamic Bottleneck Injection

A. General Split Computing Scenario

Our intuition is that we can leverage channel estimation to drive the reconfiguration of the bottleneck size. As illustrated in Fig. 1, when the edge server communicates the task result (such as classification label and segmentation mask) back to the mobile device, the mobile device can exploit the received signal to assess the channel condition by finding proper channel impairment factors.

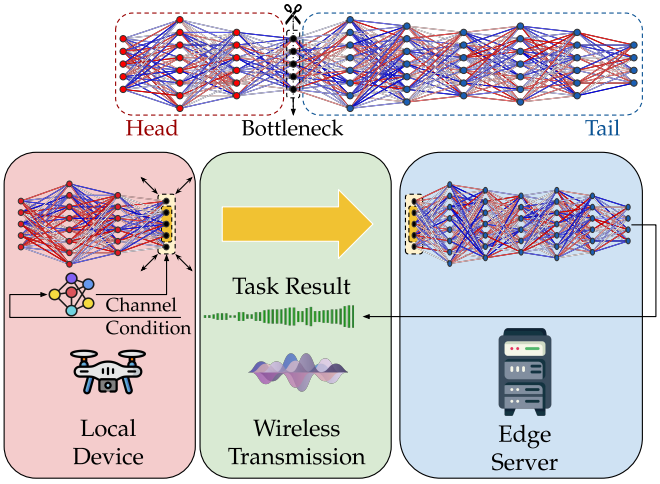

Fig. 1: Channel condition-adaptive bottleneck injection in general split computing scenario.

Estimating the channel condition can be done using machine learning techniques or traditional approaches such as measuring metrics like Error Vector Magnitude (EVM) and Modulation Error Ratio (MER). In the former case, one can use a dedicated simple neural network to compute metrics such as Signal to Interference plus Noise Ratio (SINR) or utilize the already-available primary model's head as a feature extractor and add a separate classifier on top of that to find the desired metrics with a negligible overhead. The problem of channel estimation has been well reported and therefore, we consider it beyond the scope of this paper.

The proposed channel-driven approach provides the mobile device with an opportunity to change the representation size associated with the bottleneck according to the communication environment. For example, when functioning in an environment with a reliable high-data-rate link between the mobile device and edge, the device, being aware of the satisfactory channel condition with negligible performance drop due to its effects, might decide on saving some bandwidth by transferring less data to the edge. In addition to leaving more space for the device to use the available bandwidth for other communication purposes, this reduction in communication burden can save significant energy.

B. Split Computing for Signal Awareness

Signal awareness can be viewed as a special case of the general scenario with an arbitrary task, and it is particularly interesting for us to investigate since there are public datasets available for this task. By comparing our approach across previous studies, we will be able to demonstrate the effectiveness of SpliCES.

In this scenario, we assume that the desired task is a simple signal analysis task, for example AMC. As shown in Figure 2, the mobile device, which could be a strictly hardware-constrained device such as a micro-processor, aims to receive data from the primary flexible transmitter. The flexible transmitter is capable of changing its modulation scheme according to the real-time requirements based on an adaptive algorithm, as practically shown in [5]. **Therefore, the mobile device has to find the modulation scheme being used in real time to decode the waveform appropriately.** Considering the fact that the demodulation process itself consumes significant amounts of computational power, this does not leave much space for the mobile device to execute a large model just as to find the modulation scheme. Utilizing split computing, mobile devices can offload the execution of the major part of the model to compute-capable edge devices to perform the AMC task with high confidence.

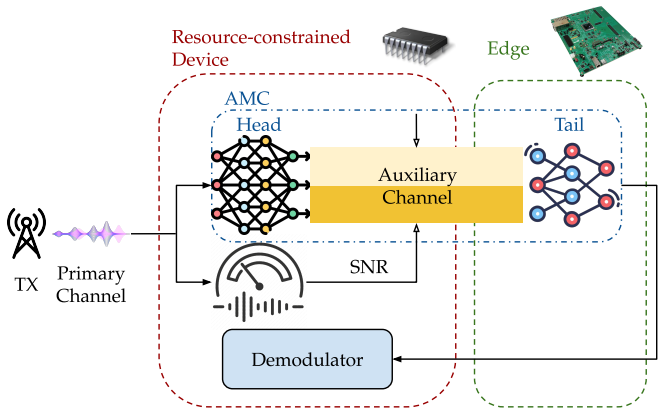

Fig. 2: Automatic modulation classification in resource-constrained devices using state-of-the-art models enabled by split computing with SNR-based dynamic intermediate representation size extraction.

IV. Problem Formulation and Training Approach

We assume a pre-trained DNN $\mathcal{N}^t$ consisting of L layers. We first split this model into two parts, where the split point should be chosen according to the maximum task performance degradation due to channel distortions and the maximum tolerable computational load of the head model on the mobile device. **There is a trade-off between the aforementioned objectives, but since our target is resource-constrained devices, we mostly prefer placing the breaking point in the very first layers.**

The original DNN model defines a mapping $f(\mathbf{x};\Theta^t) : \mathbb{R}^{n_{in}} \rightarrow \mathbb{R}^{n_{out}}$ as subsequent transformations:

$$\mathbf{o}_\ell^t = \begin{cases} \mathbf{x} & \ell = 0 \\ f_\ell(\mathbf{o}_{\ell-1}^t;\theta_\ell^t) & 0 < \ell \leq L \end{cases}$$

where $\mathbf{o}_\ell^t$ and $\Theta^t = [\theta_1^t, ..., \theta_L^t]$ are the output of the ℓ-th layer and the set of all parameters of the DNN. The superscript t denotes teacher since later we will use this model as the teacher to transfer its knowledge to the adaptively bottleneck-injected models.

We create new models by injecting bottlenecks at the layer $\ell^{\mathcal{B}}$. To do so, we prune a subset of kernels in that layer. Since we are using convolutional neural networks (CNNs), the tensor shape of the output of all layers from $\ell^{\mathcal{B}}+1$ to the last layer L remain intact even though we have removed some filters from the bottleneck layer. **In order to have an adaptive bottleneck, we choose K candidate bottleneck sizes, each intended to be used for a certain level of channel impairment**. We perform the training of the new adaptive bottleneck-injected models in two stages. In the first stage, our goal is to train a general head which can work properly for all possible channel conditions. In the second stage, however, we focus on specializing the bottleneck layer parameters to specific channel states.

A. Stage 1: Generalized Merged Ensemble Training

Since we have different bottlenecks, the parameters of the layers immediately before and after the bottleneck layer are going to be different forming an ensemble of sub-models near the bottleneck, each tailored for a given channel condition. Our training algorithm has to train all of their parameters jointly, while keeping the number of parameters in the head model to a minimum. To achieve minimum head parameters, we assume that all the bottleneck-injected models use the same parameters for all the layers preceding the bottleneck layer. **Then, we have an ensemble of networks, each corresponding to a certain candidate bottleneck size.** Each of the resulting bottleneck-injected models can be formulated as follows:

$$\mathbf{o}_{k,\ell}^s = \begin{cases} \mathbf{x} & \ell = 0 \\ f_\ell(\mathbf{o}_{k,\ell-1}^s;\theta_\ell^s) & 0 < \ell \leq \ell^{\mathcal{B}}-1 \\ f_\ell(\mathbf{o}_{k,\ell-1}^s;\theta_{k,\ell}^s) & \ell^{\mathcal{B}}-1 < \ell \leq L \end{cases}$$

where k is the index of the bottleneck-injected model ($1 \leq k \leq K$), and superscript s is added to denote student. We need to emphasize that the parameters of the layers before $\ell^{\mathcal{B}}$ does not depend on k which means that we do not need to store different heads in the mobile device other than the parameters of the very last head layer, reducing the memory footprint. **Notice that we want to train a head which can extract useful features for all channel conditions.** We know that most embedded devices have more stringent constraints when it comes to computational power compared to memory. Moreover, in cases when they do not have specialized DNN accelerators, loading different parameters for the very last layer only does not noticeably change the head execution

time, still being able to meet real-time constraints faced in most time-sensitive signal awareness problems.

In this stage, we merge the bottleneck-injected model ensemble and use the weighted summation of their output tensors of the layer $\ell^{\mathcal{B}}+1$ as the merged model output. Then, we feed all the training dataset regardless of the channel condition to the merged model. We want the merged model to mimic the behavior of the original model to reconstruct the representations of the corresponding layers in the original model. To this end, we use the reconstruction loss for all the succeeding layers to $\ell^{\mathcal{B}}+1$ apart from the final KD loss. The resulting loss function has the following form:

$$\begin{aligned}
\mathcal{L}^{merged} &= \mathcal{L}^{reconst} + \mathcal{L}^{KD} \\
\mathcal{L}^{reconst} &= || \sum_{\ell=\ell^{\mathcal{B}}+1}^{L-1} (\mathbf{o}_{\ell}^{t} - \sum_{k=1}^{K} \lambda_k \mathbf{o}_{k,\ell}^{s})||, \\
\mathcal{L}^{KD} &= \alpha \mathcal{L}^{task} + (1-\alpha)\tau^2 KL(\mathbf{o}_{L}^{t}, \sum_{k=1}^{K} \lambda_k \mathbf{o}_{k,L}^{s})
\end{aligned} \tag{1}$$

where $\Lambda = [\lambda_1, ..., \lambda_K]$ is a balancing weight vector which is set reciprocally proportional to the bottleneck size. This is to penalize larger bottlenecks which correspond to worse channel conditions less than low channel distortion levels which can already attain satisfactory task performance. Therefore, since smaller bottlenecks often have less confident outputs, we need to scale them by multiplying their contribution in the merged ensemble result. α is a balancing factor between hard and soft target losses. $\mathcal{L}^{task}$ is a task-specific loss (hard target), e.g., a standard cross entropy loss for classification. KL is the Kullback-Leibler divergence function (soft target), and we are assuming the last layer outputs of the teacher and student model represent the probability distributions, meaning that the last layer incorporates the softmax layer. τ is a "temperature" hyper-parameter for knowledge distillation normally set to 1.

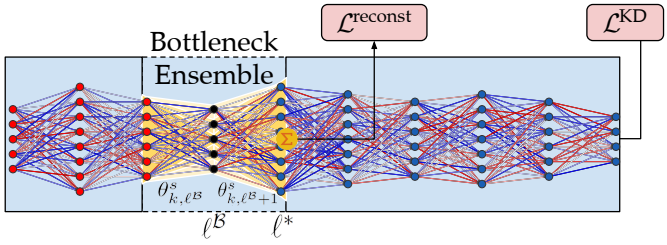

Fig. 3: Jointly training of bottleneck-injected model ensemble.

We initialize the parameters of all layers except for those neighboring the bottleneck with their pretrained values from the original model. Since learning of parameters of earlier layers is slower than that of later layers due to their distance from the final output (i.e., vanishing gradient problem), we first train by freezing all the layers after $\ell^{\mathcal{B}}+1$, and then we start unfreezing the next layers one by one, reducing the learning rate in each step to achieve the fine-tuning goal. Hence, through iteration, the system manages to preserve accuracy and get better convergence time (shorter training time).

B. Stage 2: Specialized Split Ensemble Training

In the second training stage, the objective is to specialize each bottleneck-injected model to a certain channel condition. Thus, we break the training dataset into multiple subsets based on channel condition metrics. Then, we split the sub-model ensemble around the bottleneck and train them using the channel state-conditional data. While doing so, training freezes the layers before $\ell^{\mathcal{B}}$ since this is the part of the head model that should stay fixed. Through fixing such parameters, the only part of the head model whose parameters will diverge is the very last layer of the head model ($\ell^{\mathcal{B}}$ itself). Hence, there is only a need to save different parameters associated with different bottleneck sizes for this layer only. The loss that is used in second stage for each bottleneck candidate is:

$$\mathcal{L}_{k}^{split} = || \sum_{\ell=\ell^{\mathcal{B}}+1}^{L-1} \mathbf{o}_{\ell}^{t} - \lambda_k \mathbf{o}_{k,\ell}^{s}|| + \mathcal{L}^{KD} \tag{2}$$

After this stage of training, the parameters except for the layers before the bottleneck will diverge and become specialized to extract representations which are specifically immune to certain levels of severity in channel condition. That is to say that for more harsh channel states they tend to extract features which are not only larger in size but also more resistant to distortions when passing through the wireless channel. Since we are training the bottleneck-injected models using task-oriented loss, becoming resistant is in the sense that their task result are less prone to change even though the transferred representations get distorted severely.

If we want to reduce the tail model memory footprint on the edge device as well, we may use the same parameters for all the successive layers after $\ell^{\mathcal{B}}+1$, train them in the first stage using the full training set, and freeze them in the second stage of training. This way, the bottleneck-injected models can adapt to varying environments through learning the parameters of layers surrounding the bottleneck only which can significantly decrease the number of total parameters which need to be stored. However, we do not consider any hardware constraints at the edge. Consequently, we prefer using different parameters for the second part of the model which is to be executed on the edge server.

V. Performance Evaluation

We evaluated SpliCES on the RadioML RML2018.01a dataset [19] since it is one of the largest publicly-available datasets for AMC tasks and has been used in most research works as a benchmark dataset. All the existing AMC works use lightweight models either designed from scratch or a watered-down version of the state-of-the-art models. Since SC enables us to use larger models, as for the model, we used the unmodified ResNet model. Since ResNet is designed to work with 2D image data, the only change we have made to ResNet is to swap all layers with their equivalent 1D version without

changing any parameters. To be more specific, we have used ResNet-18 since based on our experiments, its capacity is enough with regard to RML2018.01a dataset to achieve satisfactory classification accuracy. The dataset is randomly split into training, validation and test sets with a ratio of 8:1:2, respectively. We first train the network on training set for 100 epochs using Adam optimizer with a learning rate of 0.001, reducing the learning rate by a factor of 0.5 when the validation loss stops improving after 5 consecutive epochs. We have also used early stopping after 20 epochs to prevent overfitting. The trained model and its parameters are then used as the teacher model to train the bottleneck-injected student models.

As discussed, we injected K different bottleneck candidate sizes for different channel conditions. We used SNR as a measure of channel quality, and since the dataset consists of data collected using various SNR values ranging from -20 to 30 dB, we split the dataset to K equal-sized SNR ranges within this interval representing different levels of channel impairment. In our experiments, we used $K = 5$ and 8, and trained each bottleneck injected model to specialize in a certain SNR range. In testing phase, we calculated MER as an estimate of SNR, and used the appropriate bottleneck accordingly.

As shown in Figure 4, injecting SNR-adaptive bottleneck reduces the accuracy only marginally in some cases, and even improves the performance for some SNRs. **Specializing the parameters for different SNR ranges allows SpliCES to find a better local minimum**. On the other hand, as illustrated in Figure 5, while adaptive bottleneck does not compromise the accuracy, it will reduce the intermediate representation size significantly. As evident in the results, the effect of dynamic bottleneck injection can be interpreted as equalizing the task performance to gain a noticeable compression ratio. For high SNR values, SpliCES saves tremendous bandwidth by significantly reducing the amount of wirelessly transmitted data over the wireless link to the edge server at the cost of a negligible drop in accuracy. On the other hand, when experiencing poor conditions of the communication channel, the algorithm chooses to transfer more data to maintain a better performance. **Overall, on average, SpliCES transmits 12x less data while experiencing only about 3% accuracy drop with high SNR. With low SNR, our approach achieves 5% better accuracy thanks to its dynamic structure.** Finally, Figure 6 shows the computational complexity of head models expressed in floating point (multiply-accumulate) operations (FLOP). We notice that the FLOPs for naive SC is an order of magnitude larger than methods with bottleneck injection.

VI. Conclusions

DNNs will become more and more important for signal awareness in the near future, while edge computing techniques will become necessary to handle the complexity. In this paper, we have improved the state of the art by proposing a new approach for dynamic bottleneck design in split computing. This paper presented a novel training strategy which optimizes the bottleneck-injected ensemble parameters to generalize to all channel conditions in its first stage, and then specializes each bottleneck separately to a certain level of channel impairment in the second stage. We have evaluated SpliCES on the well-known RadioML dataset for modulation recognition, and shown that on average, SpliCES transmits much less data while experiencing limited performance decrease.

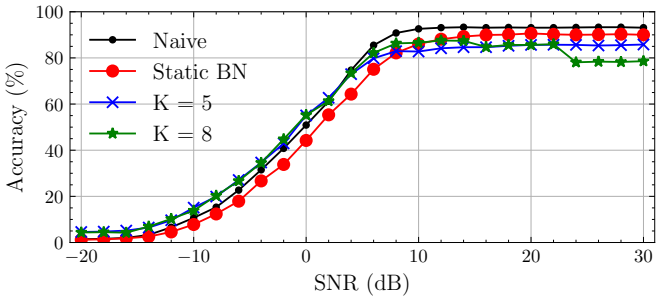

Fig. 4: Classification accuracy comparison of adaptive bottleneck injection using different number of candidate sizes with static case and naive split computing (without bottleneck injection).

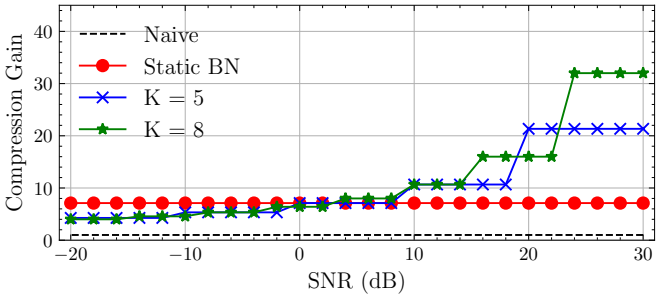

Fig. 5: Compression gain in intermediate representation size which should be transmitted over the wireless channel to edge.

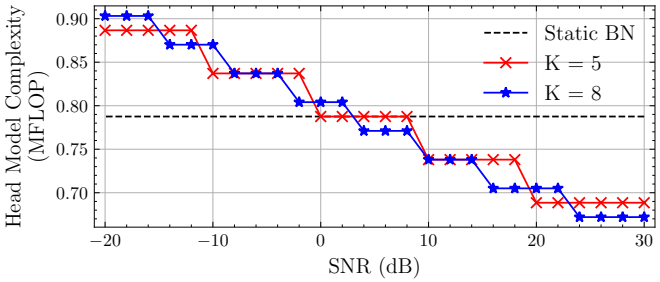

Fig. 6: Comparison of head model computational complexity estimated as the number of floating point operations.

Acknowledgment of Support and Disclaimer

We sincerely thank Dr. Erik Blasch for concept coordination and paper editing. This work is funded in part by the National Science Foundation (NSF) grant CNS-2134973,

CNS-2120447, and ECCS-2229472, by the Air Force Office of Scientific Research under contract number FA9550-23-1-0261, by the Office of Naval Research under award number N00014-23-1-2221, and by an effort sponsored by the U.S. Government under Other Transaction number FA8750-21-9-9000 between SOSSEC, Inc. and the Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views expressed are those of the authors and do not reflect the official guidance or position of the United States Government, the Department of Defense, the United States Air Force or the United States Space Force.

REFERENCES

[1] D. Uvaydov, S. D'Oro, F. Restuccia, and T. Melodia, "DeepSense: Fast wideband spectrum sensing through real-time in-the-loop deep learning," in *Proc. of IEEE Intl. Conf. on Computer Communications (INFOCOM)*, (Vancouver, BC, Canada), May 2021.

[2] F. Restuccia, S. D'Oro, A. Al-Shawabka, M. Belgiovine, L. Angioloni, S. Ioannidis, K. Chowdhury, and T. Melodia, "DeepRadioID: Real-Time Channel-Resilient Optimization of Deep Learning-based Radio Fingerprinting Algorithms," *Proc. of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2019.

[3] F. Meneghello, M. Rossi, and F. Restuccia, "DeepCSI: Rethinking Wi-Fi Radio Fingerprinting Through MU-MIMO CSI Feedback Deep Learning," in *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 1062–1072, IEEE, 2022.

[4] A. Al-Shawabka, F. Restuccia, S. D'Oro, T. Jian, B. C. Rendon, N. Soltani, J. Dy, K. Chowdhury, S. Ioannidis, and T. Melodia, "Exposing the Fingerprint: Dissecting the Impact of the Wireless Channel on Radio Fingerprinting," *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2020.

[5] F. Restuccia and T. Melodia, "PolymoRF: Polymorphic Wireless Receivers Through Physical-Layer Deep Learning," in *Proceedings of the ACM International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc)*, pp. 271–280, 2020.

[6] F. Restuccia and T. Melodia, "Deep Learning at the Physical Layer: System Challenges and Applications to 5G and Beyond," *IEEE Communications Magazine*, vol. 58, no. 10, pp. 58–64, 2020.

[7] F. Restuccia and T. Melodia, "DeepWiERL: Bringing Deep Reinforcement Learning to the Internet of Self-Adaptive Things," *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, 2020.

[8] A. Munir, J. Kwon, J. H. Lee, J. Kong, E. Blasch, A. J. Aved, and K. Muhammad, "Fogsurv: A fog-assisted architecture for urban surveillance using artificial intelligence and data fusion," *IEEE Access*, vol. 9, pp. 111938–111959, 2021.

[9] C. Weinbaum, S. Berner, and B. McClintock, "Sigint for anyone: the growing availability of signals intelligence in the public domain," tech. rep., RAND NATIONAL DEFENSE RESEARCH INST SANTA MONICA CA, 2017.

[10] Y. Matsubara, M. Levorato, and F. Restuccia, "Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges," *ACM Computing Surveys (CSUR)*, 2021.

[11] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

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

[13] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "Distilled split deep neural networks for edge-assisted real-time systems," in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pp. 21–26, 2019.

[14] J. Shao and J. Zhang, "BottleNet++: An End-to-end Approach for Feature Compression in Device-Edge Co-Inference Systems," in *Proceedings of IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, IEEE, 2020.

[15] 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)*, pp. 1–5, IEEE, 2020.

[16] 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*, vol. 20, no. 2, pp. 565–576, 2019.

[17] 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*, pp. 2685–2695, 2022.

[18] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, "Bottlefit: Learning compressed representations in deep neural networks for effective and efficient split computing," in *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 337–346, IEEE, 2022.

[19] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air Deep Learning Based Radio Signal Classification," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, 2018.

[20] 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*, pp. 4510–4520, 2018.

[21] 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/CVF conference on computer vision and pattern recognition*, pp. 2820–2828, 2019.

[22] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[23] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *arXiv preprint arXiv:1802.05668*, 2018.

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

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

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

[27] 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*, pp. 27–32, 2019.

[28] G. Li, L. Liu, X. Wang, X. Dong, P. Zhao, and X. Feng, "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27*, pp. 402–411, Springer, 2018.

[29] E. Blasch, H.-M. Chen, J. M. Irvine, Z. Wang, G. Chen, J. Nagy, and S. Scott, "Prediction of compression-induced image interpretability degradation," *Optical Engineering*, vol. 57, no. 4, pp. 043108–043108, 2018.

[30] 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*, pp. 476–488, 2020.

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

[32] T. J. O'Shea, J. Corgan, and T. C. Clancy, "Convolutional Radio Modulation Recognition Networks," in *International Conference on Engineering Applications of Neural networks*, pp. 213–226, Springer, 2016.