JMSNAS: Joint Model Split and Neural Architecture Search for Learning over Mobile Edge Networks

Yuqing Tian, Zhaoyang Zhang†, Zhaohui Yang, and Qianqian Yang
College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China
Zhejiang Provincial Key Laboratory of Info. Proc., Commun. & Netw. (IPCAN), Hangzhou, China
E-mail: {tianyq, ning_ming†}@zju.edu.cn, zhaohui.yang@ucl.ac.uk, qianqianyang20@zju.edu.cn

2022 IEEE International Conference on Communications Workshops (ICC Workshops) | 978-1-6654-2671-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCWORKSHOPS53468.2022.9814499

***Abstract*—Deep neural networks (DNNs) enable the booming intelligent mobile applications owing to the effectiveness and reliability. With the rapid development of fifth generation (5G) communication networks, edge devices can be deployed with computation capability, which makes the distributed implementation of DNN over a wireless network possible. However, the simplified model splitting method cannot guarantee the accuracy and latency performance. They call for the iterative design of model splitting and parameter updating. In this paper, a joint model split and neural architecture search (JMSNAS) framework is proposed to deploy the generated DNN model over mobile edge networks. Considering both computing and communication resource constraints, we reformulates the multi-split problem to a computational graph search problem and optimizes the objective function to realize the trade-off between model accuracy and completion latency. The experiment results reveal the superiority of the proposed JMSNAS over state-of-the-art split machine learning model designs.**

I. Introduction

The fifth-generation (5G) mobile networks are envisioned to support the booming mobile intelligent applications. The advancements on machine learning (ML) provide a powerful tool for stable and reliable applications. As the backbone of intelligent applications, machine learning (ML) technique can guarantee highly accurate and reliable applications. However, deep neural network (DNN) inference is computation-intensive, making it difficult for mobile devices with limited resources to finish the execution process within an acceptable latency.

For example, in the campus access control face recognition scenario, the face data is collected by a terminal device, and a classification result needs to be obtained through network inference. This process is a computationally expensive task depending on the pixel of the captured images and the size of the network. Reducing the waiting time required for each recognition while maintaining the accuracy of the network becomes an challenge.

Thus, it is important to investigate the ML model splitting technology, which can split an ML model and deploy splitted computation tasks onto multiple edge devices with high efficiency and low latency. Model splitting framework can partition a DNN model into several parts. Each part is calculated by one device and the calculation result is passed to the corresponding device based on the model splitting framework via wireless or wired links as shown in Fig 2. The cellular network is a native mobile edge computing structure suitable for ML model splitting. All device interconnections within a cellular network can be abstracted into two types: chain and mesh topology, involving cellular user equipment (UE), small base stations (SBSs), macro base stations (MBSs) equipped with mobile edge computing (MEC) servers, and cloud servers.

Recently, there are some works focusing on applying the ML model splitting on the mobile edge devices in a cellular network, which allows each edge device to compute part of the ML model [1]–[5]. The authors in [1] proposed a single-split method, which can partition a DNN model into two parts for two-device system. Furthermore, Hu et al. [2] utilized directed acyclic graph theory in the single-split method. The works in [1] and [2] are limited to two-device system. Considering the general multi-split problem, the work in [3] constructed a min-cost graph search problem. However, the search algorithm in [3] is limited to the linear network, which cannot be applied to the complex mesh cellular network. Besides, Teerapittayanonet al. [4] proposed to distribute the given DNN model across computing hierarchies for the purpose of reducing the communication data size. Moreover, Li et al. [5] utilized an early-exit mechanism to adjust the splitting model size to accelerate the model inference. However, the existing works [1]–[5] did not consider customizing the neural network according to edge nodes' computation and communication abilities, which indicates that the results obtained in [1]–[5] cannot guarantee the performance including accuracy and completion latency requirements over a mobile edge network.

To realize the purpose of deploying split learning over a wireless MEC system, the DNN model splitting should be adapt to computation and communication abilities of edge nodes. However, the simplified model splitting method cannot guarantee the convergence and latency performance of DNN. As a result, deploying split learning over a wireless network calls for the iterative design of model generating and model splitting, where model generating means that the DNN model should be properly designed and chosen based on the given training accuracy and latency requirements. To solve the model generating problem, neural architecture search (NAS) can be used to automatically design the artificial neural networks. The previous work [6]–[8] utilized NAS for model generating in a centralized manner, which cannot be directly applied to the distributed mobile edge networks. This motivates us to utilize the NAS method in solving the split learning problem over a wireless MEC system.

*This work was supported in part by National Key R&D Program of China under Grant 2020YFB1807101, and National Natural Science Foundation of China under Grant U20A20158 and 61725104.

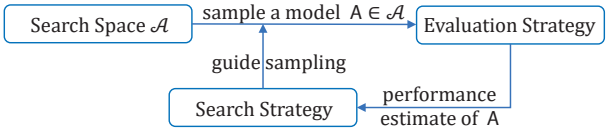


Fig. 1: Three major components of NAS models.

In this paper, based on the emerging technique of NAS, we propose the so-called joint model split and neural architecture search (JMSNAS) framework for deploying an ML model over the MEC. We formulate the multi-split problem by searching for a network achieving the accuracy and latency trade-off on a cellular network with given multi-split points. To our best knowledge, JMSNAS is the first practical framework that splits the ML model for MEC systems by using the NAS method. Our main contributions are list as follows:

- We propose a multi-split algorithm, which can assign the DNN sub-model for each computation node in a given network topology.
- By using multi-objective regularizations, the multi-objective mechanism is integrated in the loss function to various practical requirements.
- The proposed JMSNAS makes full use of all the devices in the cellular network, which can be applied to different MEC topology networks including both chain and mesh.
- Experiment results indicate that the proposed JMSNAS outperforms the state-of-the-art splitting method in terms of accuracy and latency.

II. Preliminaries

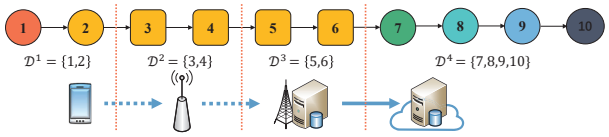
A. Neural Architecture Search

NAS models have outperformed manually designed architectures in many tasks, such as semantic segmentation [6] and model compression [8]. The NAS process involves three major components: *search space*, *evaluation strategy*, and *search strategy* (Fig. 1). Search space defines a family of candidate operations and the way operations connect. Evaluation strategy determines the quality metric of the candidate models to provide feedback that guides the search strategy. Search strategy is the method to explore the search space and generate high-quality model architectures.

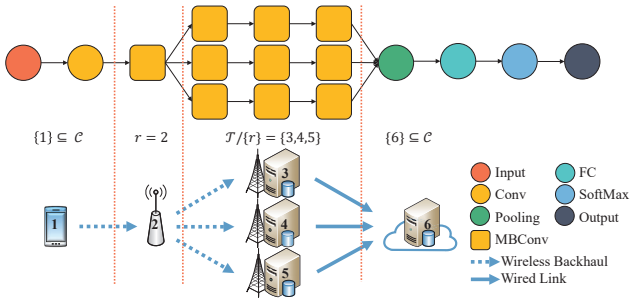
B. ML Model Splitting over MEC Systems

ML model splitting partitions the DNN computing load over MEC infrastructures, to meet specific requirements such as low inference latency. The typical scenarios of ML model splitting are shown in Fig. 2.

Typically, split learning in mobile edge networks generally involves three unique properties: *(1) Multi-split*: The existence of multiple devices in the cellular network requires the multi-split of the DNN model so as to each device compute part of the model. The number of feasible split solution increases exponentially with the number of DNN layers and edge devices. Thus, the linear exhaustive search algorithm used in single-split problem is not practical to solve the multi-split problem. *(2) Multi-object*: The solution of ML model splitting needs to satisfy multiple optimization goals, such as model accuracy, completion latency, and



(a) The chain MEC network



(b) The mesh MEC network

Fig. 2: The typical scenarios of ML model splitting.

data privacy. *(3) Complicated topology*: According to the practical scenario of cellular networks, the edge network topology can be summarized into two forms: chain and mesh. Correspondingly, the DNN includes linear and non-linear forms.

Our NAS approach for ML model splitting considers both communication and computation costs of different devices in a cellular MEC network, and explores the comprehensive search space to obtain a DNN model with low latency and high accuracy on the given network. Besides, our approach can naturally form a splitting scheme for the obtained DNN model that deploys different parts of the DNN to the devices in the target network. Finally, our approach can be applied on general structures, including chain and mesh networks.

III. System Model and Problem Formulation

Consider a deep neural network, which can be described by a directed computational graph $G(e_1, \ldots, e_N)$ as shown in Fig. 2. The nodes in Fig. 2 stand for the operations, while directed edges stand for data stream cross layers. We use $[N]$ to represent set $\{1, \cdots, N\}$. Denote N and M as the number of layers and available devices, respectively. Layer $n \in [N]$ represents the nth node in the computational graph, and device $m \in [M]$ stands for the mth node in the MEC network. Let τ_n^m be the execution latency of layer n on device m and ε_n^m be the communication latency of transmitting the output of layer n between device m and the following device. Specifically, we have $\varepsilon_n^M = 0$ since device M is the last device. All the MEC structures can be abstracted as chain and mesh topologies, which are considered in our model.

A. Chain Network

As shown in Fig. 2(a), consider a network with one UE, $M-2$ edge devices, and a cloud server to form a M-node cellular chain. Since there are M computation nodes, we need to split the original DNN network into M parts and each node can compute one part of them. Let $\mathcal{D}^m$ stand for the layers deployed on device m, where $\mathcal{D}^1 = \{1, 2\}$ means that both layers 1 and 2 are assigned to device 1.

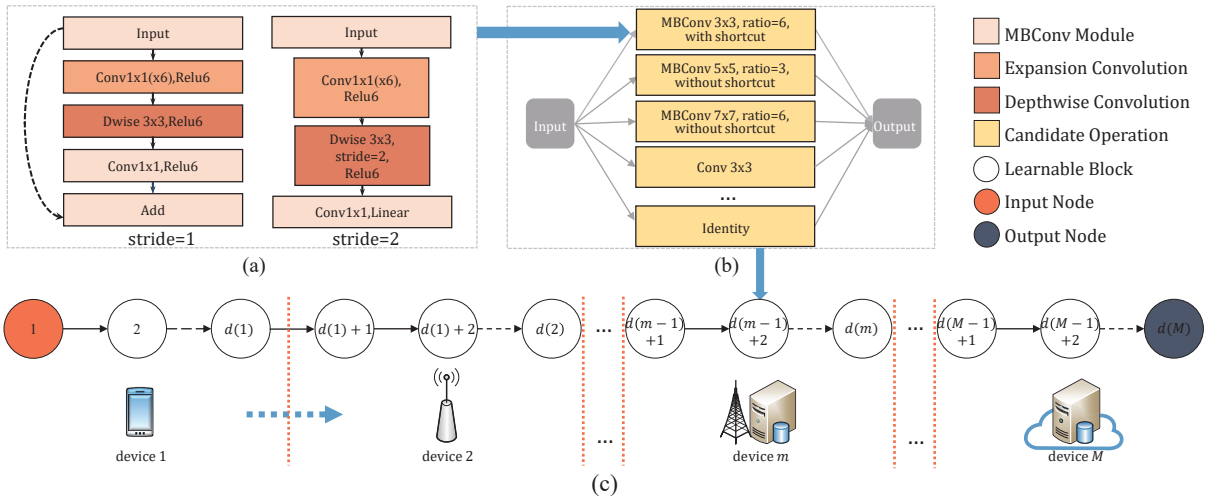


Fig. 3: The implementation procedures. A Leader node collects the MEC network information and constructs the search space by selecting the modules in (a) and designing the candidate operations in (b). After running JMSNAS, the leader node sends the parts of the generated model to the corresponding devices in (c).

With the above notation, the completion latency [9] on chain network can be described as

$$T = \sum_{m\in[M]} \Big(\sum_{n\in\mathcal{D}^m} \tau_n^m + \varepsilon_{d(m)}^m \Big), \tag{1}$$

where $d(m) = \max \mathcal{D}^m$ denotes the index of the last layer executed on device m.

B. Mesh Network

Consider a mesh network where a UE first accesses an SBS through the wireless backhaul, then the SBS broadcasts the results to three MBSs, and finally the outputs of MBSs are aggregated in a cloud server, as shown in Fig. 2(b). Let $\mathcal{C}$ denote the set of devices connected in the chain and $\mathcal{T}$ denote devices connected in the tree form. For example, in Fig. 2(b), $\mathcal{C} = \{1, 6\}$, $\mathcal{T} = \{2, 3, 4, 5\}$. There is a root node r in tree nodes set such as device 2 in Fig. 2(b), which forwards the output data of layer $d(2)$ to devices in set $\mathcal{T}\backslash\{r\}$ for executing the following layers in parallel. Thus, the completion latency [9] on mesh network can be described as

$$\begin{aligned} T = &\sum_{m\in\mathcal{C}} \Big(\sum_{n\in\mathcal{D}^m} \tau_n^m + \varepsilon_{d(m)}^m \Big) \\ &+ \max_{m\in\mathcal{T}\backslash\{r\}} \Big(\sum_{n\in\mathcal{D}^r} \tau_n^r + \varepsilon_{d(r)}^r + \sum_{n\in\mathcal{D}^m} \tau_n^m + \varepsilon_{d(m)}^m \Big). \end{aligned} \tag{2}$$

C. Implementation Model

The detailed implementation procedures of JMSNAS mainly include three steps. In the first step, a leader node collects the device information in the cellular edge network, including the computing capabilities of each device, the communication capabilities between devices, and the device connection topology. Having obtained the device information, the leader node in the second step completes the initialization of the NAS search space, i.e., the connection mode of the DNN layers and the maximum number of DNN layers which can be executed by each device. In other words, the structure of computational graph (Fig. 3(c)) and parameters D^m, $d(m)$ are determined for each m in the second step. According to the characteristics and complexity of the DNN task, the candidate operations in the search space are artificially set. In the third step, the leader node runs JMSNAS to determine the DNN used for the specific task. Additionally, the network naturally has a split matching scheme deployed to each device to meet the constraints set by the task, which can include limited latency, limited power consumption, and so on. The leader node sends the parts of searched network to the corresponding device, and each device in the MEC network executes the computation and communication assignment in a distributed manner.

D. Problem Formulation

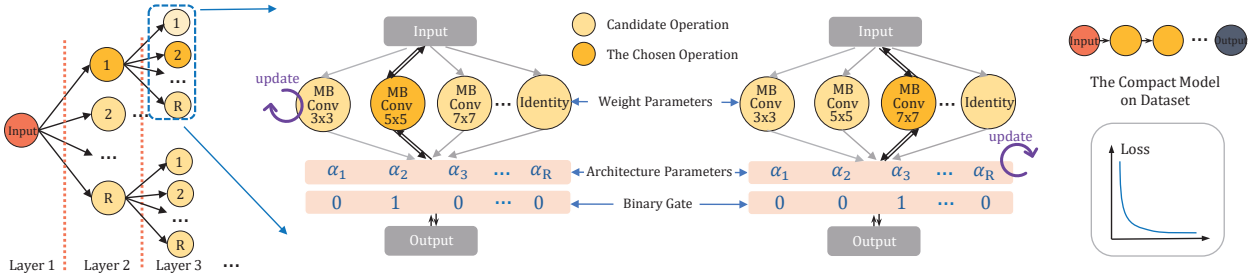
Our goal is to optimize the neural network weight parameter $\boldsymbol{\theta}$ and architecture parameter $\boldsymbol{\alpha}$ so as to minimize the loss function of the DNN model under given latency constraint. Mathematically, the considered optimization problem can be formulated as:

$$\begin{aligned} \min\ & L = \sum_{k=1}^{K} \ell(\boldsymbol{\alpha}, \boldsymbol{\theta}; (\boldsymbol{x}_k, y_k)), \\ s.t.\ & T \le T_{Const}, \end{aligned} \tag{3}$$

where $\boldsymbol{\alpha} = [\alpha_1^1, \alpha_2^1, \cdots, \alpha_i^n, \cdots, \alpha_R^N]$ and T_{Const} is the maximum allowed latency. The variable $\alpha_i^n, n \in [N], i \in [R]$ indicates the weight of choosing the ith candidate as the operation of layer n. In (3), $\boldsymbol{x_k}$ and y_k respectively indicate the kth sampled input image and the corresponding label, S is the number of sampled images. $\ell(\cdot)$ represents the cross entropy loss function. Due to complicated non-convex loss function and latency constraint, it is generally difficulty to solve problem (3) with the conventional convex optimization theory. To solve it, we use the advancement of ML in the following section.

IV. Methodology

In this section, we first introduce the JMSNAS framework, which includes search spaces, evaluation strategies, and search strategies for optimized neural networks [7]. Then, we present a gradient-based algorithm to deal with the non-differential device metric: latency.



(a) Search Space (b) Search Strategy (c) Evaluation Strategy

Fig. 4: The overview of the NAS method. Search space is a full-tree structure (a) making up of all the possible DNN models. The relative parameters are updated in (b), and evaluated in (c).

A. Neural Architecture Search

Through NAS techniques, we can automatically develop a model that outperforms previous designs deployed on cellular MEC networks.

Search Space: Let $\mathcal{V}_n$ be the set of R candidate operations available for layer n. In the initialized computational graph with fixed topology, the candidate operations of each layer together constitute a full-tree search space, as shown in Fig. 4(a). Our search space involves all combinations of R operations for N layers, which is sufficient to specific task with adjustable parameters N and R.

As an example to construct search space $\mathcal{V}_n$, the MobileNetV2 is a lightweight and highly efficient model [10], which performs well on large-scale image classification tasks. It is constructed by module mobile inverted bottleneck convolution (MBConv), which can be used as the backbone to build the candidate operations. The module MBConv takes low-dimension vectors as input, expands to high dimension, and is then filtered with a depthwise convolution. With changeable parameters about expansion ratio and convolution kernel size, $\mathcal{V}_n$ includes the following types:

- 3×3 MBConv with expansion ratio 3
- 3×3 MBConv with expansion ratio 6
- 5×5 MBConv with expansion ratio 3
- 5×5 MBConv with expansion ratio 6
- 7×7 MBConv with expansion ratio 3
- 7×7 MBConv with expansion ratio 6
- Identity

In addition, each operation can choose whether to have a shortcut connection or not. Therefore, there are up to 13 optional operations. The network length can be shortened by selecting identity to skip blocks. To make the model more accurate, the framework tends to choose a large kernel and a high expansion ratio with a large amount of computation, which leads to a larger network. On the contrary, to save execution latency, the framework tends to choose a small kernel and a low expansion ratio, which makes the network thinner. As a result, in order to balance the accuracy and latency, the width and length of the model should be well designed.

Evaluation Strategy: Before demonstrating the evaluation strategy, we need to clarify how to represent the forward propagation result of the full-tree structure as shown in Fig. 4(a). To construct the full-tree structure that includes all the combinations in the search space, we denote the operation of each layer in the computational graph by $\boldsymbol{v} = [v_1, \cdots, v_R]^T$, which is a mixed operation vector with R elements. The output of $\boldsymbol{v}$ is designed based on the output of its R paths.

To simplify the description, we use a certain DNN layer to illustrate our design for mixed operations. Without loss of generality, we replace α_i^n with α_i in the real-valued architecture parameters. Moreover, we introduce the one-hot binary gate $\boldsymbol{g} = [g_1, \cdots, g_R]^T$, where $g_i = 1$ with probability p_i, and $p_i = \exp(\alpha_i)/\sum_j \exp(\alpha_j)$ indicates the probability of choosing operation i. With input $\boldsymbol{x}_k$, the output of $\boldsymbol{v}$ can be formulated as $\sum_{i=1}^{R} g_i v_i(\boldsymbol{x}_k) = \boldsymbol{g}^T \boldsymbol{v}(\boldsymbol{x}_k)$, where $v(\boldsymbol{x}_k) = [v_1(\boldsymbol{x}_k), \cdots, v_R(\boldsymbol{x}_k)]^T$.

To solve problem (3), we modify the objective function as

$$\sum_{k=1}^{K} \ell(\boldsymbol{\alpha}, \boldsymbol{\theta}; (\boldsymbol{x}_k, y_k)) + \lambda_1 ||\boldsymbol{\theta}||_2^2 + \lambda_2 (T - T_{Const})^2, \quad (4)$$

where λ_1, λ_2 are hyper-parameters to adjust the learning process. Note that the latency constraint is formulated as a penalty in (4), which forces that the optimal solution satisfying $T = T_{const}$. This is because the optimal solution of (3) is always achieved at $T = T_{const}$ as deep and time-consuming network can lead to small loss value.

Search Strategy: There are two types of parameters in our framework, i.e., weight parameter $\boldsymbol{\theta}$ and architecture parameter $\boldsymbol{\alpha}$. We train weight and architecture parameters in an alternating manner, as shown in Fig. 4(b). When training weight parameters, the architecture parameters are fixed and the binary gates $\boldsymbol{g}$ are sampled to identify the current DNN model. Then, the sampled model is trained with forward and backward propagation. When updating architecture parameters, the weight parameters are given in the previous step and the binary gates are reset. To update the architecture parameter, the partial derivative $\partial L/\partial \alpha_i$ with respect to discrete operation choosing needs to be calculated, which is provided by the following lemma.

Lemma 1: The partial derivative $\partial L/\partial \alpha_i$ can be approximately presented by

$$\sum_{j=1}^{R} \frac{\partial L}{\partial g_j} p_j (\delta_{ij} - p_i), \quad (5)$$

where $\delta_{ii} = 1$, $\delta_{ij} = 0$ if $i \neq j$, and $\frac{\partial L}{\partial g_j}$ can be obtained from the following equation (8).

Proof: The partial derivative of L with respect to α_i is:

$$\begin{aligned}\frac{\partial L}{\partial \alpha_i} &= \sum_{j=1}^{R} \frac{\partial L}{\partial p_j}\frac{\partial p_j}{\partial \alpha_i} \approx \sum_{j=1}^{R} \frac{\partial L}{\partial g_j}\frac{\partial p_j}{\partial \alpha_i} \\ &= \sum_{j=1}^{R} \frac{\partial L}{\partial g_j}\frac{\partial \left(\frac{\exp(\alpha_j)}{\sum_k \exp(\alpha_k)}\right)}{\partial \alpha_i} = \sum_{j=1}^{R} \frac{\partial L}{\partial g_j} p_j\left(\delta_{ij} - p_i\right),\end{aligned} \tag{6}$$

where the first equality follows from the chain theory and the approximation holds based on the definition of g_j. The derivative $\partial L/\partial g_j$ can be calculated by substituting the expression of g_j into the function (4). In particular, if we consider the cross-entropy loss function, equation (4) can be further rewritten as

$$\begin{aligned}L_{\text{CE}} = -\frac{1}{K}\sum_{k=1}^{K}(y_k \log h_{\boldsymbol{g}}\boldsymbol{v}(\boldsymbol{x}_k) \\ + (1-y_k)\log(1-h_{\boldsymbol{g}}\boldsymbol{v}(\boldsymbol{x}_k))),\end{aligned} \tag{7}$$

where $h_{\boldsymbol{g}}\boldsymbol{v}(\boldsymbol{x}_k) = 1/(1+e^{-\boldsymbol{g}^T\boldsymbol{v}(\boldsymbol{x}_k)})$ indicates the predicted probability. Thus, we can obtain

$$\frac{\partial L_{\text{CE}}}{\partial \boldsymbol{g}} = \frac{1}{K}\sum_{k=1}^{K}\left(h_{\boldsymbol{g}}\boldsymbol{v}(\boldsymbol{x}_k) - y_k\right)\boldsymbol{v}(\boldsymbol{x}_k). \tag{8}$$

This completes the proof. □

Based on Lemma 1, we can update the architecture parameter through backpropagation. The architecture parameter updating procedure involves calculating and storing $v_j(\boldsymbol{x})$ for every j, which costs R times memory. To address this issue, we mask all the paths except for the sampled two in every training process so that we can reduce the memory cost from R times to 2 times.

B. Derivative of Latency Function

Since our training network dynamically chooses operations according to a probability distribution, the latency in loss function (4) is not differentiable with respect to the architecture parameter. To handle this issue, we reformulate the latency of a network to the average latency, which is a continuous function. A mixed operation $\boldsymbol{v}$ includes candidate set $\{v_1, v_2, \cdots, v_R\}$ and each operation v_i corresponds to a selection probability p_i. We build a regression model $U(\cdot)$ to estimate the operation latency. For example, when layer n adopts operation v_i executed on device m, we have $U_n^m(v_i)$ as the execution latency. In such a full-tree structure, the execution latency of layer n on device m, τ_n^m in (1) and (2) should be reformulated as $\mathbb{E}(\tau_n^m) = \sum_i p_i U_n^m(v_i)$. Thus, the gradient of $\mathbb{E}(\tau_n^m)$ with respect to architecture parameter can be given by: $\partial\mathbb{E}(\tau_n^m)/\partial p_i$. Furthermore, we represent the overall latency $\mathbb{E}_{\boldsymbol{\alpha}}(T)$ by replacing τ_n^m with $\mathbb{E}(\tau_n^m)$.

In summary, the proposed method provides ample search space and sufficient flexibility to search for proper layer operations and enables high performance as the trade-off between accuracy and latency. The cost during the NAS process is completely undertaken by the leader node, which requires the leader node to have strong computing capabilities.

TABLE I: Link settings

Transmitter	Receiver	Type	Capacity(Mbps)
UE	SBS	Wireless	25
SBS	MBS	Wireless	50
MBS	Cloud	Wired	200

V. Experiment Results

In this section, we first describe our experiment setup, including the cellular network setup, and the configuration for the NAS procedure. We then present our framework evaluation results on ML model splitting compared with the previous methods.

A. Cellular Network

The performance of the proposed JMSNAS framework is evaluated on both chain and mesh networks. Table I shows the communication link settings between devices.

The execution and communication latency profiles of different operations involved in the DNN model are the key metric of the model workload. To accurately measure the operation latency, we adopt a Pytorch [11] package (torchprof) to track latency on different devices for each involved operation and build an estimator $U(\cdot)$ to predict operation-wise latency during model inference.

We measure the latency profiles on four types of machines to represent the UE, SBS, MBS with MEC server and cloud server, respectively:
(1) Raspberry Pi 4 Model with 4 Cortex-A72 1.5GHz CPUs,
(2) XPS15 Laptop with Intel i7-11800H CPU, 16GB DDR 4 RAM, and Nvidia RTX3050 GPU,
(3) NVIDIA Jetson AGX Xavier with 64 Tensor Core GPUs and 8-core ARM CPUs,
(4) A server with two Intel Platinum 8280 CPUs, and Tesla V100 GPU.

B. Neural Architecture Search

We demonstrate the effectiveness of JMSNAS on the ImageNet dataset [12]. The training set includes 1231167 images of 1000 classes, each with dimension of $224\times224\times3$, while the validation set includes 50000 images.

In order to obtain a model that performs well on the given cellular network, the NAS process consists of two stages. In the first stage, we search the full-tree structure on the training split for 20 warm-up epochs and 60 training epochs, using Adam optimizer with initial learning rate of 0.002 and batch size of 512. Warm-up training is a technique widely used in deep learning. It helps to alleviate overfitting on the mini-batchs, and to maintain the stability of the DNN model. At the end of every training epoch, we evaluate the performance of the current compact network on the validation set. We set up three levels of completion latency constraints on the chain network and mesh network, respectively.

In the second stage, after the architecture parameters of the full-tree structure converge, the compact model architecture is fixed. Then, we further train the model on the training set for 200 epochs, with the weight parameters in the first stage as the pretrained parameters. In this way, the derived compact model will get higher accuracy without compromising its efficiency.

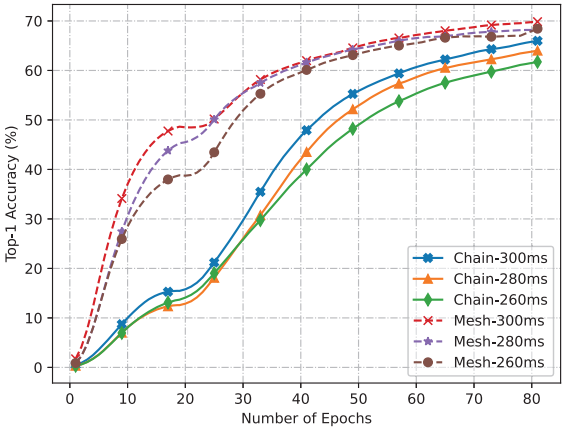


Fig. 5: In the first NAS stage, JMSNAS on the chain and mesh networks with different T_{Const} latency constraints.

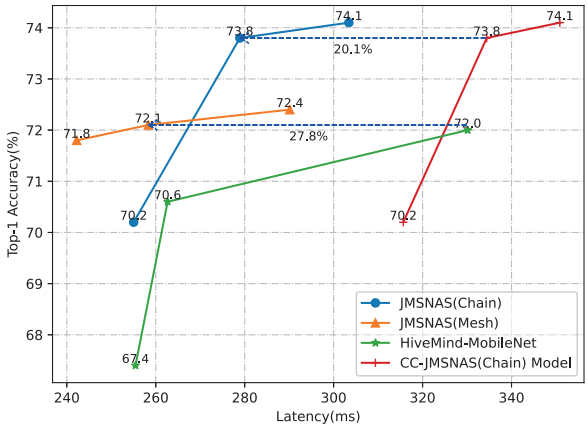


Fig. 6: Final accuracy vs. latency after the second NAS stage.

We adopt two performance metrics of our framework, i.e., top-1 accuracy and latency. Fig. 5 shows the top-1 accuracy corresponding to different latency constraints in the first NAS stage. Here, we find that the models did not reach the highest accuracy during the first stage, since the model architecture is still alternating. And when the maximum allowed latency is larger, the model accuracy tends to be higher. Compared with models on the chain network, models on the mesh network shows faster convergence and higher accuracy with the same number of training epochs.

Since the operations in our search space are mainly MBConvs, modules in MobileNetV2, we also compare the accuracy and latency of the JMSNAS-crafted models with MobileNet (V1[13], V2[10], V3[14]). Fig. 6 shows that our framework achieves higher accuracy, which confirms the effectiveness of JMSNAS. As for the latency, we compare JMSNAS-crafted model performance with two baselines, cloud computing and HiveMind [3] multi-split framework. We apply the cloud computing to the chain model obtained by JMSNAS, by uploading the input data to the cloud center and completing the model inference on the cloud. The latency is mainly determined by the communication link conditions. Fig. 6 shows that JMSNAS can reduce the average latency by up to 20.1% compared with cloud computing. For DNN models with a chain structure such as MobileNet series, we can adopt linear search method like HiveMind to determine the best splitting points. Fig. 6 shows that our framework outperforms MobileNet with linear search method.

VI. Conclusion

In this paper, we have proposed a NAS-based multi-split framework, deploying the generated DNN model to a cellular mobile edge network to meet the accuracy and latency requirements. The proposed JMSNAS works well on large-scale image classification tasks with an ample search space dependent on the mobile edge network conditions. The automatically generated models with native split scheme outperform the previous model split method.

The proposed JMSNAS is a general framework, and it can be applied in any practical scenario to dynamically determine a customized DNN.

References

[1] Y. Kang, J. Hauswald, C. Gao *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[2] C. Hu, W. Bao, D. Wang *et al.*, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM IEEE Conference on Computer Communications*, 2019, pp. 1423–1431.

[3] S. Wang, X. Zhang, H. Uchiyama *et al.*, "Hivemind: Towards cellular native machine learning model splitting," *IEEE Journal on Selected Areas in Communications*, 2021.

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

[5] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*, 2018, pp. 31–36.

[6] L.-C. Chen, M. D. Collins, Y. Zhu *et al.*, "Searching for efficient multi-scale architectures for dense image prediction," *arXiv preprint arXiv:1809.04184*, 2018.

[7] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[8] Y. He, J. Lin, Z. Liu, H. Wang *et al.*, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 784–800.

[9] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," *IEEE Transactions on Mobile Computing*, vol. 16, no. 11, pp. 3056–3069, 2017.

[10] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[11] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.

[12] O. R., J. D., H. S. *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[13] A. G. Howard, M. Zhu, B. Chen *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[14] A. Howard, M. Sandler, G. Chu *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1314–1324.