HiveMind: Towards Cellular Native Machine Learning Model Splitting

Song Wang, Xinyu Zhang, *Senior Member, IEEE*, Hiromasa Uchiyama, and Hiroki Matsuda

***Abstract*— The increasing processing load of today's mobile machine learning (ML) application challenges the stringent computation budget of mobile user equipment (UE). With the wide deployment of 5G edge-cloud, a new ML offloading scheme called split ML is provisioned to enable computation-intensive mobile ML applications by splitting an ML model across mobile UE, edge, and cloud. However, the complex split assignment problems pose new challenges for split ML system design. In this paper, we introduce HiveMind, the first practical multi-split ML system tailored for 5G cellular networks. HiveMind reformulates the complicated multi-split problem to a min-cost graph search and optimizes the distributed algorithm to drastically reduce the signaling overhead. Benefit from its low overhead property, HiveMind makes the optimal split decision on multiple computing nodes in real-time and adapts the split decisions to the instantaneous network dynamics. HiveMind also incorporates a multi-objective mechanism that accommodates heterogeneous objectives for a single ML task. HiveMind adapts to a wide range of ML frameworks, including non-linear models like Recurrent Neural Network (RNN), Federated Learning (FL), and Multi-agent Reinforcement Learning (MARL). We evaluate HiveMind on 5G MEC network simulators with realistic traffic patterns and real-life MEC computation/communication profiles. Our experiments demonstrate that HiveMind achieves the optimal efficiency comparing to state-of-art split ML designs.**

***Index Terms*— Edge computing, machine learning (ML), neural networks, 5G mobile communication.**

I. Introduction

THE booming mobile Machine Learning (ML) applications are challenging the current computing and communication network architectures. Amid the rapid maturity of mobile machine learning platforms, *e.g.*, Google ML kit [1], Apple Core ML [2], and Fritz AI [3], more than 10% of mobile apps have incorporated ML models, with use cases ranging from face identification, object detection, to intelligent personal assistants and augmented reality [4]. Recent studies also proposed to integrate ML into 5G networks to optimize network functions such as QoS-aware routing, resource allocation, and slice management [5]–[8]. In addition, the emerging 6G is envisioned to bring human-like intelligence into every aspect of networking systems [9]. However, due to the computation resource constraints on mobile devices, such mobile ML applications typically only use miniature models hundreds of times smaller than standard ML models [4], which hampers model accuracy and limits their use cases. The stringent computation power budget further renders the more computation-heavy ML training tasks impossible. On the other hand, offloading these tasks to the cloud may incur high data transfer overhead and sometimes can be even slower than on-device computing [10].

To enable computation-intensive mobile ML applications, recent work explored edge computing infrastructures [11] that offer cloud-like services within the cellular network. To minimize latency without compromising model accuracy, such edge ML implementations can split a model into multiple parts, and allocate them among different computing nodes, including the user equipment (UE), mobile edge computing (MEC) servers, and cloud servers. For the commonly used deep neural network (DNN) model, for example, each part corresponds to multiple DNN layers. Each node executes the model up to a specific layer, and sends the intermediate data to the next node. With such UE-edge-cloud synergy, a split ML system can dynamically assign parts of a model to the computing nodes based on network conditions and computation resources, to alleviate the pressure of computation on UE devices and potentially optimize end-to-end latency and energy consumption [10], [12]–[15].

Existing research abstracts the ML model splitting as redistributing the computing load across a generic client-server link. As the 5G network infrastructure evolves to embrace built-in computing capabilities, an important question arises: Can the 5G networking and computing stack itself natively support AI/ML through model splitting? Unlike in the abstract model, a single 5G site often consists of many MEC servers distributed across different vantage points in the RAN/core network. Splitting the ML model across such a unique distributed system, potentially involving dynamic links and UEs with different performance objectives, becomes a non-trivial problem.

More specifically, such a cellular-native model splitting needs to address three unique challenges. ***(i)*** *Multi-split ML models over a 5G network.* 5G's native MEC support [11] and flexible traffic steering capabilities can enable a new *multi-split* scheme among the UE, multiple MEC servers, and the cloud server. Fig. 1 shows a typical case of ML inference model

Manuscript received March 1, 2021; revised September 13, 2021; accepted September 22, 2021. Date of publication October 6, 2021; date of current version January 17, 2022. This work was supported by the Faculty Innovation Award of Sony Research Award Program (https://www.sony.com/en/SonyInfo/research-award-program/#FacultyInnovationAward). *(Corresponding author: Song Wang.)*

Song Wang and Xinyu Zhang are with the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: sowang@ucsd.edu; xyzhang@ucsd.edu).

Hiromasa Uchiyama and Hiroki Matsuda are with Sony Group Corporation, Tokyo 141-8610, Japan (e-mail: hiromasa.uchiyama@sony.com; hiroki.matsuda@sony.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JSAC.2021.3118403.

Digital Object Identifier 10.1109/JSAC.2021.3118403

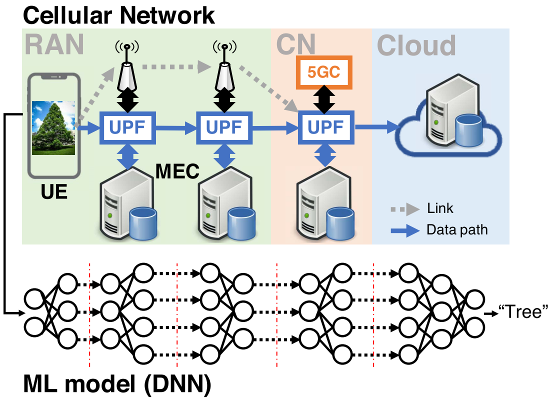


Fig. 1. An example of 5G cellular native ML: An UE, three MEC nodes, and a cloud server form a 5-hop MEC chain. Each device executes a part of the ML model to a certain layer and send the intermediate data to the next device on the chain.

splitting, where an image classification NN is split across a UE, three MEC nodes, and a cloud server. The intermediate output from each partition of the NN is transmitted to the next computing node via wireless backhaul or wired links. The inference result is output at the last partition of the network. In such multi-split scenarios, the number of split options grows exponentially with the number of ML layers and computing nodes, and are often on the order of millions. In addition, the split decision needs to accommodate the computing resources distributed across the network, along with varying network conditions. Due to all such complexities, the linear searching method in existing ML model splitting designs cannot be applied to multi-split systems.

(ii) *Multi-objective split.* Compared with a single atomic model, the split ML decision over 5G networks should be optimized to flexibly accommodate different objectives, *e.g.*, inference/training latency, energy consumption, and privacy preservation. Some of these objectives are based on best efforts, *i.e.* maximization/minimization, whereas some are quality assurances, *i.e.*, ensuring a performance metric does not exceed a predefined threshold. Accommodating such heterogeneous objectives simultaneously poses a new challenge for the splitting decision making.

(iii) *Splitting for non-linear ML models.* Existing single-split approaches are limited to standard *linear* ML models with a chain of layers. Thus, a simple linear search across all inter-layer cuts suffices to identify the optimal split point. However, other commonly used ML paradigms, *e.g.*, Recurrent Neural Network (RNN) and Collaborative Learning, require additional information transfer between the same or different modules. Directly applying split ML on them fails to account for the extra communication overhead and may result in highly suboptimal performance.

In this paper, we propose *HiveMind* , a novel multi-split ML framework that addresses the aforementioned challenges through three design choices. *(i)* A distributed split ML algorithm. We first reformulate the multi-split problem into a min-cost graph search. To avoid the huge communication overhead that renders the existing solutions infeasible, we propose a distributed min-cost graph algorithm tailored for 5G MEC networks. Through graph pruning and information aggregation, our algorithm dramatically cuts down the number of inter-node signaling messages, thus enabling an efficient and practical multi-split without the loss of optimality on split decisions. *(ii)* A mechanism to simultaneously accommodate best efforts objectives (*e.g.*, minimizing energy cost) and quality assurance objectives (*e.g.*, latency threshold) in the splitting decision. The mechanism discriminates the quality assurance metrics with a non-linear mapping function, and enforces the quality assurance objectives without compromising the optimality of the best effort metrics. *(iii)* Splitting non-linear ML models. We further broaden the application domain of our split ML algorithm by extending it to non-linear ML models, including recurrent ML models and collaborative ML models, whereas the latter involves Multi-agent Reinforcement Learning (MARL) and Federated Learning (FL). Our solution takes into account the iterative feedback structures commonly seen in such models while requiring little modifications to the standard algorithm.

We evaluate HiveMind on a 5G network simulation framework, which represents a tree-structured integrated access and backhaul (IAB) network and edge/cloud computing devices co-located on all IAB gNBs, CN, and cloud server, in compliance with 3GPP's provisioning of 5G MEC architecture [16], [17]. The evaluation framework adopts synthetic traffic traces that faithfully reproduce the traffic characteristics of a cellular network. Our experimental results demonstrate that *(i)* HiveMind is able to adapt to a wide range of traffic load. It outperforms cloud and UE-based baselines by up to 89.8% under high traffic load. HiveMind benefits more from MEC capability gains than the baselines by up to 47.2%, especially from the MECs co-located with IAB gNBs. *(ii)* HiveMind can simultaneously accommodate the best effort and quality assurance objectives, and outperforms the heuristic linear multi-objective by 22.9% on the best effort objective. *(iii)* HiveMind reduces the parameter feedback latency on both RNN and collaborative learning models, and outperforms the standard split by up to $2.3\times$ on multiple criteria.

HiveMind, to our knowledge, marks the first practical multi-split ML system tailored for 5G MEC networks. Its contributions can be summarized as follow: (i) A novel cellular native split ML algorithm that enables the practical multi-split ML by distributively optimizing split assignment with negligible overhead. (ii) A multi-objective mechanism that adapts different types of objectives to a single multi-split task. (iii) Extension of the multi-split algorithm to widely adapted non-linear ML models including RNN and collaborative ML. (iv) Validation of HiveMind on a 5G simulator against state-of-art ML splitting designs.

II. Related Work

A. Distributed ML

Recent research explored distributed machine learning to reduce the processing time of mobile ML applications leveraging edge or cloud computing devices. Ho *et al.* [18] proposed to use a centralized parameter server to aggregate the local gradient, and schedule training tasks on the local nodes. Agarwal *et al.* [19] designed AllReduce that further extends this paradigm to a tree structure, by accumulating and passing the local gradient from child nodes to parent nodes. Foster *et al.* [20] introduced a fully distributed paradigm where each node broadcasts the local gradient to all other nodes. In addition to the latency-oriented parallel computing paradigms, Konevcny *et al.* [21]–[26] proposed Federated Learning (FL) framework with an emphasis on preserving data privacy. FL obscures the local update from local nodes so that the parameter server cannot infer sensitive information, but can still keep the training accuracy. The above distributed ML designs assume a client/server architecture, where each computing client trains one instance of the whole model. In contrast, our split ML framework partitions an ML model so that the different parts are executed sequentially on different computing nodes within a cellular network. In addition, distributed ML mainly focuses on ML training. Our split ML framework can be applied to both training and inference.

B. ML Model Splitting

ML model splitting has garnered much interest in the past two years. In particular, the 3GPP standardization group recognized the performance benefits for ML model splitting and has been investigating protocol-level primitives to support ML model splitting within the cellular edge/core networks [17]. Much of the related research verified the advantages of ML model splitting and focused on partitioning the DNN computing load to meet certain optimization objectives. Kang *et al.* [10] proposed to identify a single splitting point to cut a DNN inference model in two parts, executed by a generic client and server respectively, to optimize running latency or energy consumption. Hu *et al.* [27] further extended the single split scheme to DNN models with directed acyclic graph representation. The single-split approach splits between the UE and one server, and the performance improvement is largely limited by the computation resource and the link condition of the server machine. In contrast, our multi-split approach can flexibly assign the ML model to multiple MECs in order to adapt to dynamic link and computation resources. Narayanan *et al.* [28] proposed an optimization-driven split ML framework, PipeDream, that assigns parts of a model to multiple GPUs to reduce training latency on a single machine. Among the existing research in distributed/parallel ML, PipeDream shares the most similarity with our work. However, PipeDream assumes static links between GPUs where the partitioning is done once and for all. The linear programming based optimizer itself incurs around 8 s running latency on a server-level machine [28], which renders it unsuitable for real-time splitting of ML *inference* models in dynamic cellular networks.

C. Tailoring ML Models to Edge Computing Systems

Besides model distribution and partitioning, existing work also explored other mechanisms to customize ML models for edge computing. Teerapittayanon *et al.* [29] proposed an early-exit mechanism, BranchyNet, which adds exit points in the middle of an ML model to cut the inference delay at the cost of lower accuracy. A follow-on distributed ML design, DDNN [30], further proposed that each end device sends the output of its exit point to an edge server which performs aggregation. Since the early exit mechanism skips part of the model structure, the inference accuracy is largely compromised. So far no early exit design achieves more than 80% of inference accuracy from the early exit points comparing to the full model [29]–[34]. Besides, it is feasible to compress the intermediate data transfer between UE and cloud in order to reduce communication latency [35]–[42]. The state-of-art intermediate data compression design [42] achieves up to $5000/1$ compression ratio by training a model-specific NN compressor while suffering a relative small accuracy loss of 8% due to the information loss during compression. Comparing to above two mechanisms, our split ML framework does not modify the model structure or intermediate data and reduces latency without sacrificing the inference accuracy. Note that the early-exit and compressing approaches are not in conflict with our split design. Instead, it is possible to apply them on top of HiveMind, *i.e.* adding early exit points at the split points or compressing the intermediate data out of the split points, to further optimize the processing latency.

III. The Need for Multi-Split in 5G MEC Networks

In this section, we explain the motivation for adopting multi-split ML and demonstrate its advantages with an example scenario. In a mobile ML application, the total overhead is attributed to two factors: the computation overhead in running the neural network layers, and the communication overhead in transmitting data between computing nodes, *e.g.*, UE uploading an input image to the cloud server. Conventional NN model runs on either UE or cloud server. UE-based ML models execution suffers from large computation overhead due to its stringent computation budget and cloud-based ML often incurs large communication overhead due to limited communication link capacity. In comparison, split ML achieves a flexible tradeoff between computation overhead and communication overhead by dynamically splitting a ML model between the UE and cloud [10], [16], [27]. It can achieve low computation overhead when the link capacity is high and avoid high communication overhead otherwise. In addition, comparing to existing single-split schemes [10], [16], [27], multi-split ML further improves the efficiency by leveraging a chain of MEC servers, striking a middle ground between UE and cloud with more computation budget than UE-only and a faster and more stable communication link than cloud-only model execution.

To showcase the advantage of multi-split ML, we consider a simple scenario where a UE, MEC, and cloud server split a 3-layer NN, as shown in Fig. 2. The layers have computation loads of 0.5, 1, and 4 units and communication load of 1.2, 1, and 4 units. The three computation nodes have computation capacity of 1, 3, and 10 units respectively and the link between UE and MEC has a capacity of 2 units. We assume a simple overhead model where the overhead is load divided by capacity, *e.g.*, transmitting input on UE-MEC link takes $\frac{1.2}{2} = 0.6$ unit time. The link capacity between MEC and cloud is set to vary between $\frac{1}{10}$ to 1 to simulate the link dynamics. We compare multi-split ML with two schemes:

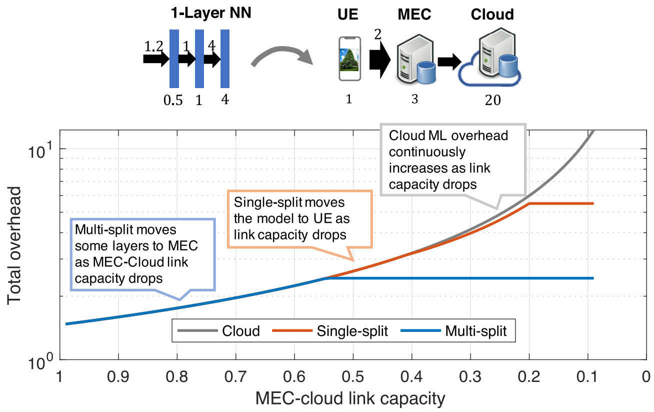


Fig. 2. The latency comparison of 4 split architectures: (1) UE computing, (2) Cloud single split, (3) Edge single split, (4) Multi-split on UE, edge, and Cloud.

cloud ML, where UE always upload the input to the cloud for computation, and single-split ML, where the model is split between UE and the cloud. Fig. 2 shows the total overhead of three schemes as the MEC-cloud link capacity decreases from 1 to $\frac{1}{10}$. We see cloud ML's overhead continuously increases as the communication overhead increases (*i.e.*, link capacity decreases). Single-split ML starts with all layers assigned to the cloud and transfers the layers to UE when link capacity further drops, to avoid increasing communication overhead. Mutli-split ML achieves a even lower total overhead on top of single-split ML by assigning the last two layer to MEC instead of UE. This experiment shows that multi-split avoids the increasingly large communication overhead in cloud ML and achieves a lower computation latency than single-split ML. Note that this experiment only demonstrates an simplified typical scenario. In Sec. VII, we will show that in a more detailed and realistic 5G setting, our multi-split ML reduces total latency by 37 to 90% compared with cloud ML and 32 to 55% compared with single-split ML.

IV. HiveMind Multi-Split Design

In this section, we first provide a primer on the 5G MEC system that enables the multi-split ML framework. Then we introduce HiveMind multi-split design for both ML inference and training, and its runtime optimization under network dynamics.

A. A Primer on 5G MEC for ML

Similar to the legacy 4G architecture, 5G networks separate the Radio Access Network (RAN) and Core Network (CN) functions. Yet Integrated Access and Backhaul (IAB) is introduced in the RAN as a unique feature, where basestations (*i.e.*, gNBs) form a tree-like topology with multi-hop wireless backhaul links [16]. Due to the flexible deployment of data plane, ETSI group [11] provisions a flexible deployment of MECs at different vantage points within the 5G infrastructure, including the gNBs, RAN aggregation point, and the core network site. The flexible MEC deployment provides mobile applications with easier and faster access, especially for the gNB MECs which can be reached by UEs in one hop. The communication overhead is hence expected to be much shorter than remote cloud access [12], rendering it feasible to accelerate ML inference/training. On the other hand, the 5G User Plane Function (UPF) enables the free steering and routing of application traffic among UEs and MECs attached to different network entities [43], allowing the formation of multi-hop MEC chains. All above 5G features jointly enable the multi-split ML paradigm where an ML model is split into multiple parts and assigned to a chain of MEC nodes, as shown in Fig. 1.

B. Problem Formulation

1) The Multi-Split Problem: We now describe the multi-split problem formulation. For simplicity, we assume a linear DNN and a single objective of minimizing inference latency. In later sections, we will extend the design to non-linear ML models and multi-objectives.

Consider a scenario where $P-2$ MEC nodes in 5G system, along with a UE and a cloud server, form a P-node MEC chain where the first node $p = 1$ is the UE and the P-th node $p = P$ is the cloud server. The UE, serving as the *source node*, initiates a split ML task that utilizes the MECs along the route from UE to the cloud. We refer to the cloud server as the *sink node*, as it is the last node along the chain that may undertake part of the ML processing load. The ML model to be split across the network has L layers. We define a split decision as two functions $u(p) = m, w(p) = n$ to represent assigning layers m to n to the node p. Suppose the layer-wise computation latency τ_l^p and communication latency ϵ_l^p of transferring intermediate data are known to all P nodes through profiling [10], the problem of finding the optimal split decision to minimize the total latency can be expressed as follow:

$$\min_{u,w} \sum_{p=1}^{P} \sum_{l=u(p)}^{w(p)} \tau_l^p + \sum_{p=1}^{P} \epsilon_{w(p)}^p \tag{1}$$

$$\text{s.t. } u(p) \le w(p), \quad \forall p \tag{2}$$

$$u(p) = w(p-1) + 1, \quad 2 \le p \le P \tag{3}$$

$$w(P) = L \tag{4}$$

The first term in Eq. (1) sums up the computation latency of all P nodes and the second term sums up the communication latency of transferring intermediate data across adjacent nodes. Eq. (2)-(4) ensure the assignment includes all L layers in the ML model in correct order without overlapping. Note that it is valid to "skip" a node by assigning no layer to it. In such cases, the communication latency over the skipped node still needs to be included since within the IAB network the intermediate data has to travel through the skipped node.

2) Mapping Split Assignments to Graph: The above optimization framework uses functions $u(p), w(p)$ as variables. Although they can be treated as vectors to fit into existing integer programming solutions, the mapping between them and the computation latency, *i.e.* $\tau_{u(p)}^{w(p)}$, is a non-convex function since τ_l^p is arbitrary. Hence, it is hard to solve the problem directly with integer programming. If we take the brute-force approach and examine all possible split options, then it requires calculating the latency for all $\binom{P+L-1}{L}$ split options (split L layers into P sets while allowing 0 layer in a set since it is possible not to assign layers to a node), *e.g.*, a ResNet50 [44] split on 5 nodes has 4.0×10^7 options. Examining such a huge amount of split options requires significant processing time, let alone the overhead caused by gathering the latency profiles and distributing the split decisions to all MECs. Hence a direct search, as done in existing single-split solutions, cannot meet the real-time requirement of cellular native split ML. To overcome these limitations, we modify the variables and reformulate the problem as a classic linear optimization - shortest path problem. We first convert the search space for the split decision into a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. The set of vertices:

$$\mathbf{V} = \{v^p_{m,n} | \forall p, 1 \leq m \leq n \leq L\} \tag{5}$$

embodies all possible assignment decisions on all P nodes, where a single vertex $v^p_{m,n}$ represents the decision of assigning layers m to n to node p. To avoid confusion, we use the term "vertex" to refer to the vertices in the graph and "node" to refer to the computing nodes in the cellular network. Then we can easily connect the vertex following the constraints in Eq. (2)-(4):

$$\begin{aligned}\mathbf{E} = \{(v^p_{m,n}, v^q_{x,y}) | p < q, \quad m \leq n,\ x \leq y,\\ \times x = n+1, \quad y = L \text{ when } q = P\}\end{aligned} \tag{6}$$

where an edge $(v^p_{m,n}, v^q_{x,y})$ represents choosing the assignment $v^q_{x,y}$ after the assignment $v^p_{m,n}$. To be consistent with the objective function in Eq. (1), we set the weight on the edge $(v^p_{m,n}, v^q_{x,y})$ to be the cost of choosing the decision $v^q_{x,y}$ after $v^p_{m,n}$, *i.e.*, the sum of the communication latency in transferring intermediate data from p to q, and the computation latency on the node q:

$$c(v^p_{m,n}, v^q_{x,y}) = \sum_{l=x}^{y} \tau^q_l + \epsilon^p_n \tag{7}$$

Finally, we add a pair of virtual start/end vertices v^s, v^e connecting to vertices corresponding to the first and last node with zero-cost edges, *i.e.*, $\{v^1_{m,n} | 1 \leq m \leq n \leq L\}$ and $\{v^P_{m,n} | 1 \leq m \leq n \leq L\}$ respectively, to serve as source and destination in the graph.

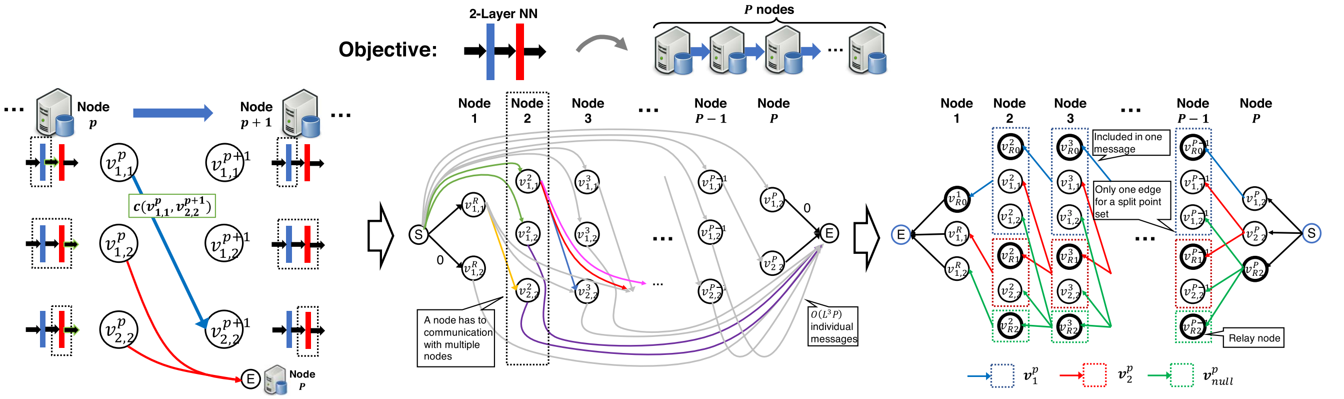
The optimization objective now becomes finding the shortest path from v^s to v^e, and the vertices along this path form the assignment decision functions u, w, *i.e.*, $u(p) = m$, $w(p) = n$ if and only if $v^p_{m,n}$ belongs to the shortest path. By reformulating the optimization, we can derive the optimal $u(p), w(p)$ by finding the shortest path with classic linear programming based solutions and avoid the complicated non-standard optimization problem in the original formulation.

C. Split Cost Information (SCI) Design

At first glance, the problem can be straightforwardly solved by applying the well-known Dijkstra's algorithm. However, Dijkstra's algorithm requires reconstructing the entire graph on a central controller node, and the size of the graph grows exponentially with the number of layers and linearly with the number of nodes. In the previous example of ResNet50 running on 5 nodes, the corresponding graph comprises 4.5×10^5 edges, each requiring the profiling of computation and communication latency to determine the cost. Despite the relatively low computation complexity of Dijkstra's algorithm, gathering such information and feeding it back to the controller incurs significant overhead, especially when the decision needs to be updated frequently under network dynamics.

We now introduce our *Split Cost information (SCI)* design which can efficiently solve the graph representation of the split ML problem. SCI inherits the logic of the distributed Dijkstra's algorithm [45], [46] and is tailored to the split ML graph to tackle the information gathering overhead. In SCI, each vertex calculates its own shortest path by traversing its neighbor vertices' *path cost*, *i.e.* the sum of all edges on the shortest path of a vertex. Specifically, given that a vertex A's neighbor vertices' path costs are known, A selects the neighbor vertex with the minimal sum of shortest path cost and edge cost as its predecessor vertex on the shortest path, and the sum value as its path cost. The shortest path can then be found by iteratively following the predecessor vertex all the way to the destination. Hence, to determine the shortest path, a vertex needs to know the path costs of all its neighboring vertices. However, acquiring such information may induce non-trivial communication overhead. In existing distributed algorithms [45], [46], a vertex needs to send individual messages to all its neighbor vertices with its shortest path cost value. Note that each node has $\binom{L}{2}$ vertices (choosing the start and stop point from L layers for a split assignment) and each vertex has $\frac{(L+1)P}{4}$ outgoing edges on average (a vertex $v^p_{m,n}$ has outgoing links to all the following nodes after p, which is $\frac{P}{2}$ nodes on average, and for each node, the vertex has links to all $L - n + 1$ vertices with starting layer n+1, which is $\frac{L+1}{2}$ vertices on average). Therefore, a node needs to send $\binom{L}{2}\frac{(L+1)P}{4} = O(L^3P)$ messages. In the previous example of ResNet-50 running on 5 nodes, this translates to 3×10^6 messages, each with only one cost value. Sending such a large number of short messages all at once incurs huge overhead and can easily congest the network, rendering it impossible to directly apply the existing algorithm to the split ML problem.

1) Transforming the Split Graph: To reduce the communication overhead for distributed shortest path algorithm, we introduce a *split graph transformation* technique. We observe that for a vertex on computing node p, the neighboring vertices are mostly located on the adjacent node $p-1$, except for those vertices that skip the node $p-1$. For example, edge $(v^1_{1,3}, v^3_{4,6})$ assigns the 6 layers between node 1 and 3 and skips node 2. If we can eliminate such edges, we can limit the communication strictly between two adjacent nodes, and the $O(L^3)$ short messages from a node can be aggregated as one single message. To this end, we transform the graph by introducing *relay vertex* v^P_{Rn} to represent the "null" workload assigned to the skipped node. In the above case, edge $(v^1_{1,3}, v^3_{4,6})$ can be broken down into two edges $(v^1_{1,3}, v^2_{R3})$ and $(v^2_{R3}, v^3_{4,6})$, both connecting the vertices of neighboring nodes. Fig. 3 showcases the transformation graph of splitting a 2-layer model on P nodes with relay vertices. With this measure, we aggregate the $O(L^3)$ path cost values



(1) Map split assignments to the graph

(2) Original split graph

(3) Pruned and transposed split graph

Fig. 3. Graph representation of the HiveMind multi-split problem (From left to right): (1) The mapping from the split assignment to the graph representation. (2) The original split graph representation has numerous edges across multiple MEC nodes, incurring huge communication overhead. (3) The pruned and transposed graph limits the edges strictly between adjacent MEC nodes and reduces the size of the messages carrying the cost information.

into one message, thus saving the overhead of sending $O(L^3)$ individual messages.

2) *Slimming the Inter-Node Messages:* The use of relay vertex in the graph transformation reduces the messaging overhead. But the size of a message increases to $O(L^3)$ times due to the aggregation and may still incur non-negligible transmission latency when the ML model has a large number of layers. For example, for a ResNet-152, each message would contain 4×10^6 path cost values. Suppose each path cost value is stored in float format, the size of the message would be 80 MB, which is too large for real-time signaling. We thus further reduce the size of each message by pruning the number of path cost values in a message. We first transpose the graph, *i.e.*, reverse the direction of all edges and reverse the role of the start/end nodes, which does not change the optimal shortest path. We then group the vertices $v_{m,n}^p$ on a node p by the starting layer m. We denote such a group of vertices as a *split point set* $\mathbf{v}_m^p$. Fig. 3 demonstrates the split point sets on different nodes. We see that in the transposed graph, a vertex's neighbor vertices must belong to the same split point set. This is because according to Eq. (6), a vertex's neighbor vertices all share the same starting layer. Recall that in the shortest path algorithm, a vertex only needs to know the minimal path cost among its neighbor vertices in order to calculate its own path cost. This means that only the cost of the optimal vertex in a split point set, *i.e.* the vertex with the minimal cost, is required by the shortest path calculation, and instead of sending cost values for all vertices, a node can just send one per split point set.

Algorithm 1 Optimal Split Cost Calculation

```
Input: Optimal cost for each split point set on node
       $p+1$: $\{\zeta_0^{p+1}, \zeta_1^{p+1}, \ldots, \zeta_{L+1}^{p+1}\}$, layer-wise
       computation latency: $\{\tau_0^p = 0, \tau_1^p, \ldots, \tau_L^p\}$,
       layer-wise communication latency from node
       $p-1$: $\{\epsilon_0^{p-1}, \epsilon_1^{p-1}, \ldots, \epsilon_L^{p-1}\}$ where $\epsilon_0^{p-1}$
       corresponds to model input
Output: Optimal cost for each split point set (SCI
        message) $\{\zeta_0^p, \zeta_1^p, \ldots, \zeta_{L+1}^p\}$, Optimal split
        points $\{n_0^p, n_1^p, \ldots, n_{L+1}^p\}$
1  $i \leftarrow 0$ ;
2  while $i \leq L$ do
       ;            // Iterate split point sets
3      for $j \leftarrow i$ to $L$ do
           ;     // Iterate vertices in a split
           point set
4          $\theta_j^i \leftarrow \sum_{l=i}^{j} \tau_l^p + \epsilon_i^{p-1} + \zeta_{j+1}^{p+1}$ ;   // Calculate
           the cost of $j$-th vertex in $i$-th
           split point set
5          ;
6      $n_i^p \leftarrow \mathrm{argmin}_j(\theta_j^i)$ ;
7      $\zeta_i^p \leftarrow \min_j(\theta_j^i)$ ;
8      if $n_i^p > i+1$ then
9          for $z \leftarrow i+1$ to $n_i^p$ do
10             $n_z^p \leftarrow n_i^p$ ;
11             $\zeta_z^p \leftarrow \zeta_i^p - \sum_{l=i} z\tau_l^P$ ;
12         $i \leftarrow n_i^p$;
13     $i \leftarrow i+1$
```

Leveraging the above property, we introduce an *optimal split cost algorithm*, as described in Algorithm 1. The algorithm simultaneously reduces the message size and finds the shortest paths for vertices on a node. Line 1 to Line 5 first calculates the path cost of all vertices. Then, based on the aforementioned property, Line 7 finds the optimal path cost ζ_i^p for each split point set. These path cost values are packed into a signaling message called *Split Cost Information (SCI) message*, as shown in Fig. 4, and sent to the adjacent node to serve as the input of the optimal split cost algorithm on that node. In the meantime, Line 6 finds the stop layer index n_i^p of the vertices corresponding to the optimal paths, which are later used as the key information for split assignment decision making. A SCI message contains only $L+1 = O(L)$ shortest path cost values corresponding to L layers in the model plus a relay node layer. For the previous ResNet-152 example, it means a less than 4KB message size, nearly 24000× smaller than the original 80MB message. The optimal split cost algorithm design can thus be safely extended to models with a large number of layers without inducing large overhead. To further improve the efficiency of the algorithm, we observe that if n_i^p for a split

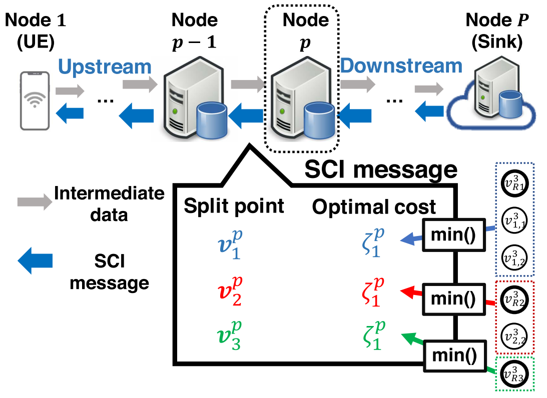


Fig. 4. A showcase of split cost information (SCI) message transmitted from node p to node $p-1$. The message contains a optimal path cost for each split point set.

point set i is greater than i, then $n_f^p = n_i^p$ for all split point sets $f \leq n_i^p$. This is because the costs θ_j^f on split point sets $f \leq n_i^p$ is just θ_j^i minus a constant $\sum_{e=i+1}^{f} \tau_l^p$ and the optimality of n_p^i holds for these split point sets. Hence, we compare n_i^p with i in line 8 and skip the computation for iteration $i+1$ to n_i^p if $n_i^p > i+1$.

3) SCI Protocol in 5G Networks: We now introduce how to execute the above SCI solution framework in 5G MEC networks. As illustrated in Fig. 5, the operation consists of two processes: (1) *SCI update*, (2) *Split ML task*. The SCI update runs along the *upstream* direction, *i.e.*, from sink node to source node, whereas the split ML task runs on the *downstream* direction. During the SCI update process, a node calculates the shortest path costs with Algorithm 1 and sends the SCI message to its upstream node. Since the downstream SCI message is required by the algorithm, the SCI update starts from the sink and moves upstream towards the source node. The split ML task starts immediately after the SCI update is completed. During the split ML run-time, each node receives the ML intermediate data from its upstream node, executes the ML model up to a certain split point, and sends the intermediate data to its downstream node. A node chooses its own split point based on the calculation results from the SCI update and the upstream node's split point. Specifically, recall that Algorithm 1 derives the stop layer index of the optimal vertices n_i^p for each split point set. Since the vertices in a split point set represent split assignments with the same starting layer i, then n_i^p is the optimal split point for node p if its upstream node splits at layer $i-1$. Hence, given the upstream node's split layer index x, a node p can easily identify its optimal split point by finding n_{x+1}^p. Note that since the source node always executes the ML model from the first layer, its optimal split point is always n_1^1. A split ML task process finishes when the entire ML model is executed and the output is sent back to the source or the cloud server for further application-specific processing.

Owing to the split graph transformation and SCI message design, the SCI protocol achieves high efficiency in solving the multi-split assignment problem: A node only sends out one SCI message to its adjacent node per SCI update and the message size is only a few KB. Combined with the low complexity of the optimal split cost calculation algorithm ($O(L^2)$), the SCI update process can be completed in an instant. In our experiment, we observe that the average running time for one SCI update is only 27 ms.

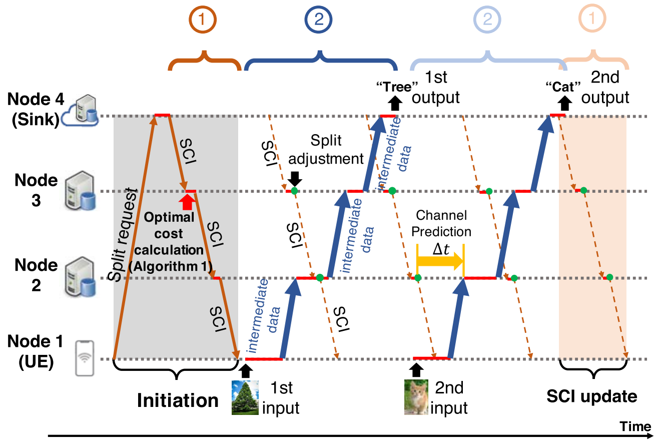


Fig. 5. Split DNN procedure: ① SCI update: each node calculates its shortest path costs and signal its upstream node with SCI message, ② Split ML task: each node chooses its own split point and execute the layers.

The low running time of SCI update is crucial for combating network dynamics. Due to the variation of the wireless channel and background traffic demand, the link throughput between MECs often varies drastically over time. Consequently, the communication cost profiles used by the optimal split cost calculation are likely to expire very quickly. As a result, the optimal split points calculated by an SCI update also expire quickly, leading to a sub-optimal split. A fast SCI update process means the process can update the optimal split points at a fast pace and thus adapt to more severe network dynamics.

D. Cost Analysis

In this section, we provide a simple analysis to examine the advantages of our split ML approach in terms of computation and communication cost. With the assumption that $\tau_l^P \leq \tau_l^p \leq \tau_l^1, \forall l, p$, *i.e.* the cloud server has the minimal layer-wise computation latency and the UE has the maximal, we can easily deduce that the minimal total computation latency is $\sum_{l=1}^{L} \tau_l^P$ and maximal is $\sum_{l=1}^{L} \tau_l^1$. Note that the maximal total computation cost equals the total cost in that case because UE-only model execution does not entail any communication overhead. Since SCI always selects the split decision with the minimal cost, the maximal possible communication cost in a split ML task is $\sum_{l=1}^{L} \tau_l^1 - \sum_{l=1}^{L} \tau_l^P$, *i.e.*, the difference between the maximal and minimal total computation cost. In other words, SCI allows up to $\sum_{l=1}^{L} \tau_l^1 - \sum_{l=1}^{L} \tau_l^P$ of communication before switching from split ML to conventional UE-only ML. This indicates that unlike conventional cloud-based ML whose communication cost may grow unbounded under poor link conditions, SCI is able to gracefully degrade to the UE-only execution when link capacity becomes too low.

E. Extension to Split DNN Training

The above design focuses on splitting a single-pass ML inference model. In contrast, ML training is an iterative bi-directional process: a forward inference pass, same as the ML inference process, is followed by a backward pass, which travels through the layers in reverse order to calculate the parameter updates using intermediate results from the forward pass [28]. The shared intermediate result means *a node needs to have the same set of layers for both the forward and backward passes*. Hence, only one split assignment is needed. Similar to split inference, we can formulate the split assignment as a graph and derive the optimal split assignment by finding the shortest path. However, two additional costs need to be considered in the edge cost of the graph. The first one is the cost of running the backward pass, including both computation and communication cost. The second is the cost of passing the layer parameters when the split assignment changes. To better explain parameter passing, consider a case where two MECs p_1 and p_2 are initially assigned with layer 1-2 and 3-4 respectively. Later the assignment changes to 1-3 for p_1 and 4 for p_2. In this case, the parameters of layer 3 need to be passed from p_2 to p_1. The parameter passing cost only exists in training because in inference the parameters from all layers are fixed and can be loaded to MECs prior to the inference task, while in training the parameters vary rapidly. Assuming the layer-wise training cost $\tau_l'^p$, $\epsilon_l'^p$ and the cost of passing l-th layer's parameter from p to q $\eta_l^{p,q}$ are known, the cost for an edge is the sum of the forward and backward passes:

$$c(v_{m,n}^p, v_{x,y}^q) = \sum_{l=x}^{y}(\tau_l^q + \tau_l'^q) + \epsilon_n^p + \epsilon_n'^p + \sum_{l \in L_T} \eta_l^{p,q} \quad (8)$$

where L_T is a set of the layers required to be transferred from p to q. In the case where the layers needs to be transferred from q to p, we replace $\eta_l^{p,q}$ with $\eta_l^{q,p}$. Therefore, to enable split training, we simply need to modify the corresponding edge cost computation (Line 3) in Algorithm 1:

$$\theta_j \leftarrow \sum_{l=i}^{j}(\tau_l^p + \tau_l'^p) + \epsilon_i^{p-1} + \epsilon_i'^{p-1} + \sum_{l \in L_T} \eta_l^{p,q} + \zeta_{j+1}^{p+1} \quad (9)$$

F. Runtime Optimization Under Network Dynamics

The foregoing discussion assumes that the network dynamics can be counteracted by frequent SCI updates for most of time, which is corroborated by our experiments in Sec. VIIB. However, due to the sparse high variance in 5G links [47], the latency information in the SCI messages may still occasionally expire at the time when the split ML is executed. As a result, the corresponding split decisions become outdated and highly sub-optimal. Furthermore, the latency information in an SCI message is often aggregated across multiple hops before it reaches an upstream node, during which the links may experience larger dynamics. So the optimal decisions made at these upstream MECs are more likely to expire.

To showcase this phenomenon, we create a simplified split DNN scenario with 1 UE and 5 MECs on the route. The link capacities are $\{4000, 2000, 800, 400, 100\}$ Mbps for each hop starting from the UE. We generate dynamic traffic according to the background UE density and demand distributions extracted from a real-world cellular network trace [48]. Fig. 6 shows the time series of total split latency. We see that at $t = 120ms$ and $450ms$, the split latency surges to over 500 ms. As illustrated, this is because the split assignment is made at $t = t_1$ when the access link capacity is $c(t_1)$, while the split task runs at $t = t_2$ when the link capacity drops to $c(t_2)$, making the split assignment outdated.

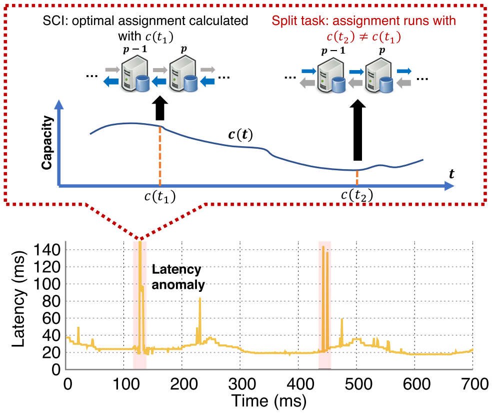


Fig. 6. Link dynamic showcase: the split ML latency surges at $120ms$ and $450ms$ due to high variances in link capacity.

To tackle the link dynamics, we introduce a simple *predictive splitting mechanism* to the SCI design. This mechanism leverages existing cellular link capacity forecast schemes [49] to predict the inference latency. Recall that in Eq. (7), the edge cost involves the communication latency ϵ_n^p, which can be estimated by the link capacity $c^{p,p+1}(t)$ and the intermediate data size of n-th layer s^n:

$$\epsilon_n^p(t) = \frac{s^n}{c^{p,p+1}(t)} + \psi^n \quad (10)$$

where ψ^n is the MEC overhead for transferring n-th layer intermediate data, including the network stack overhead and 5G signaling overhead, which can be profiled in advance. From Fig. 6 we see the latency spikes are caused by the link capacity variation between the SCI message at t_1 and the split task at t_2. We denote such time gap as $\Delta t = t_2 - t_1$. Ideally, if Δt is known in advance, a node p at t_1 can forecast the link capacity at t_2 $c^{p,p+1}(t_2) = c^{p,p+1}(t + \Delta t)$ and then calculate the communication latency $\epsilon_n^p(t_2)$ at t_2 from Eq. (10), thus eliminating the effect of link dynamics. However, Δt is determined by the dynamics of upstream links and the split assignment of upstream nodes, both of which are not known to node p. Nonetheless, since the upstream nodes only run a part of an ML model, the variation in Δt is largely limited, often smaller than the link coherent time at node p. Hence, we estimate Δt on a node by averaging the time gaps from previous g split tasks on this node and calculate the communication latency accordingly. Note that since procedure 1 and 2 are mirrored, different nodes have different Δt, *e.g.* node 2 in Fig. 5 has smaller Δt than node 4. In the case where Δt is smaller than link coherent time, we simply disable predictive split as it is no longer necessary. Experiments in Sec. VII shows the predictive split can eliminate over 95% of latency spikes under realistic cellular link dynamics when the link capacity prediction error is less than 13%.

V. HiveMind Multi-Objective Split

It is straightforward to apply the HiveMind multi-split to metrics other than latency. One can simply replace the layer-wise latency profiles τ_l^p and ϵ_l^p with the corresponding cost profiles, *e.g.*, the energy consumed for running layer l. However, a 5G MEC application often needs to account for a mix of metrics simultaneously [11], [50]. Depending on the specific application, these metrics may require different objectives. There are two categories of performance objectives defined in 5G [50]: *(i)* Best effort, where a metric needs to be optimized to the best effort, *e.g.*, reducing the energy cost as much as possible, *(ii)* Quality assurance, where a metric needs to be limited by a certain threshold, *e.g.*, making sure the latency is below 100ms. It is non-trivial to apply multiple metrics with different objectives to the shortest path solutions.

The canonical multi-metric shortest path solution linearly combines the metrics as the edge cost in the graph [51]:

$$c(v_{m,n}^p, v_{x,y}^q) = \sum_{j=1}^{M} w_j (\sum_{l=x}^{y} (\tau_j)_l^q + (\epsilon_j)_n^p) \tag{11}$$

where $(\tau_j)_l^p$ and $(\mu_j)_l^p$ are the computation and communication costs of the j-th metric, and w_j is the corresponding weight. Such the linear combination method does not require any modification to the HiveMind operations other than the edge cost calculation. However, this method does not distinguish the best effort objectives from quality assurance objectives, and cannot represent the threshold of the quality assurance objectives. Since the objectives of different metrics are often conflicting, *e.g.* optimizing latency may lead to increased power consumption, the linear combination method may lead to excessive optimization on the quality assurance objectives and compromise the best effort objectives. For example, in Sec. VIID, we run a multi-objective split task with a best effort objective on energy consumption, and a quality assurance objective on latency whose threshold is 120ms. We observe that comparing to the optimal method, the linear combination unnecessarily optimizes the latency to $75ms$ while increasing energy consumption by 29%.

To address the above problem, we introduce a *non-linear weight function design*. The high-level idea is to restrain the weight of quality assurance objectives in the edge cost when the metric is well below the threshold so that the best effort objectives are not affected, and quickly increase the weight of the quality assurance objectives when the metric approaches the threshold to prevent the quality assurance violations. Doing so requires a non-linear mapping between the metric to its weight in the edge cost. Hence, we use non-linear weight functions to reshape the quality assurance metrics, before linearly combining them with the minimization metrics. Suppose there are B objectives in a split ML task, where the first C objectives are quality assurance and the remaining are best effort. We define $\phi_j = \sum_{l=x}^{y} (\tau_j)_l^q + (\epsilon_j)_n^p$ as the cost of the j-th objective on edge $(v_{m,n}^p, v_{x,y}^q)$. Then the total cost of the edge can be calculated as:

$$c(v_{m,n}^p, v_{x,y}^q) = \sum_{j=1}^{C} W_c(\phi_j) + \sum_{j=C+1}^{B} w_b \phi_j \tag{12}$$

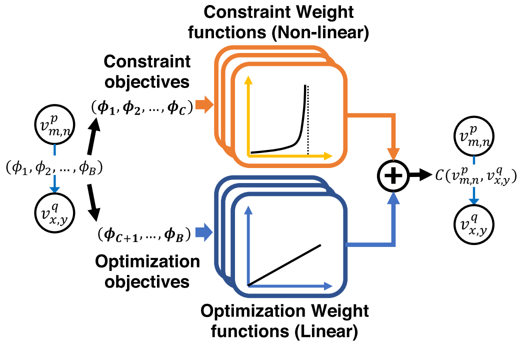

Constraint Weight functions (Non-linear)
Constraint objectives
$(\phi_1, \phi_2, \dots, \phi_C)$
$v_{m,n}^p$
$(\phi_1, \phi_2, \dots, \phi_B)$
$v_{x,y}^q$
$(\phi_{C+1}, \dots, \phi_B)$
Optimization objectives
Optimization Weight functions (Linear)
$C(v_{m,n}^p, v_{x,y}^q)$
$v_{m,n}^p$
$v_{x,y}^q$

Fig. 7. The edge cost calculation in HiveMind multi-objective: the quality assurance metrics are reshaped by non-linear weight functions before linearly combined with the best effort metrics.

where W_c is the non-linear weight function and w_b is the linear weight for best effort objectives.

Fig. 7 shows the calculation of the edge cost. By design, W_c should be a monotonically increasing convex function that asymptotically approximates the constraint $x = \Phi_j$. We empirically find that the inversely proportional function $y = -\frac{1}{x - \Phi_j}, x \in (-\infty, \Phi_j)$ achieves the best optimization performance. Given the updated edge cost definition, we can simply plug it in the optimal latency calculation algorithm (Line 3 in Algorithm 1) to enable the multi-objective split.

VI. Splitting Non-Linear Neural Networks

The HiveMind design we have introduced so far is applicable to DNNs. In this section, we describe how to extend the design to non-linear NN models, including Recurrent Neural Network (RNN) and collaborative learning models.

A. Split RNN

RNN models are widely used for applications with temporal correlated and sequential inputs, such as natural language processing and speech recognition. An RNN model consists of a sequence of identical *recurrent modules*. Each recurrent module contains several linearly-organized layers. Part of the model output called "hidden states" is feed to the next module along with the input sequence. Fig. 8 (a) demonstrate the topology of a typical RNN model. A straightforward way to adapt HiveMind to RNN is to split the recurrent module since it shares a similar linear structure as a standard DNN. This would require running the recurrent module repeatedly on the MEC network. In the common case where the first layer is on the source node and the final output layer is on a MEC node or cloud server, the hidden states generated at the final layer need to be fed back to the source node everytime the recurrent module is repeated. The size of the hidden states is usually in the same order as the intermediate data passed across adjacent layers, if not larger [52]. Hence, transferring the feedback to the source, usually across multiple hops, causes a large overhead.

To address this problem, we propose to *linearize the RNN splitting problem* so it becomes similar to the DNN splitting. Consider an RNN with R recurrent modules, each consisting of L layers. As shown in Fig. 9, we regard the l-th layer

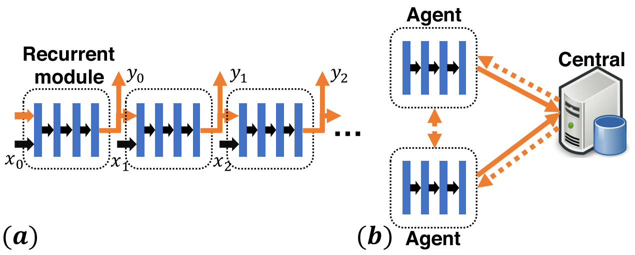


Fig. 8. Non-linear NN showcase: (a) Recurrent neural network (RNN), (b) Collaborative learning.

on the r-th recurrent module as layer $(r-1) \times L + l$. The remaining input sequence $\{x_{r+1}, x_{r+2}, \ldots, x_{P-1}\}$ and the previous output sequence $\{y_1, y_2, \ldots, y_{r-1}\}$ are transferred along with the intermediate output. This way we can reuse the HiveMind mechanism while avoiding sending the hidden state across multiple hops within the MEC network.

B. Split Collaborative Learning

Collaborative learning is a widely adopted distributed ML training paradigm. Common collaborative learning models include Distributed Deep learning (DDL), Multi-agent Reinforcement Learning (MARL), and Federated Learning (FL). In a collaborative learning setup, multiple identical copies of a model are deployed on different nodes called *agents*, each training its models with locally observed environment and states. In order to achieve global optimum, the agents periodically transfer their model parameters to each other or to a central controller to merge the parameters, as shown in Fig. 8(b). The model parameters are usually on the order of 100 MB [53], significantly larger than inter-layer intermediate data which are on the order of 100 KB (Sec. VIIB). As a result, the parameter transfer cost constitutes a significant part of collaborative learning.

To optimize the runtime cost of an agent network, the HiveMind design needs to account for the cost of parameter transfer. The parameter transfer happens after calculating the final layer of an agent model. Hence for split assignments on node p that involve the final layer, *i.e.*, $v^p_{i,L}, 0 \le i \le L-1$, we need to add the parameter transfer cost to the edge costs. Suppose the parameter transfer cost at node p ω^p is known through link capacity profiling. Then, we can simply add ω^p to the standard HiveMind training (Eq. 9) to reflect the parameter transfer cost:

$$\theta_j \leftarrow \sum_{l=i}^{j} (\tau^p_l + \tau'^p_l) + \epsilon^{p-1}_i + \epsilon'^{p-1}_i + \sum_{l \in L_T} \eta^{p,q}_l + \zeta^{p+1}_{L+1} + \omega^p \qquad (13)$$

We can then replace the cost calculation for $v^p_{i,L}, 0 \le i \le L-1$, *i.e.*, the L-th iteration of the inner for-loop in Algorithm 1, with Eq. (13) to enable split collaborative learning. Note that since the split assignments that do not involve the final layer are not affected by the parameter transfer cost, the cost calculations in the first $L-1$ iterations remain unmodified.

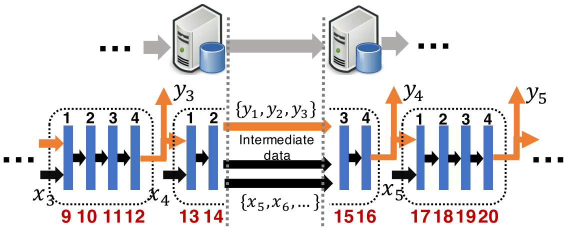


Fig. 9. A showcase of linearized RNN in HiveMind split RNN design.

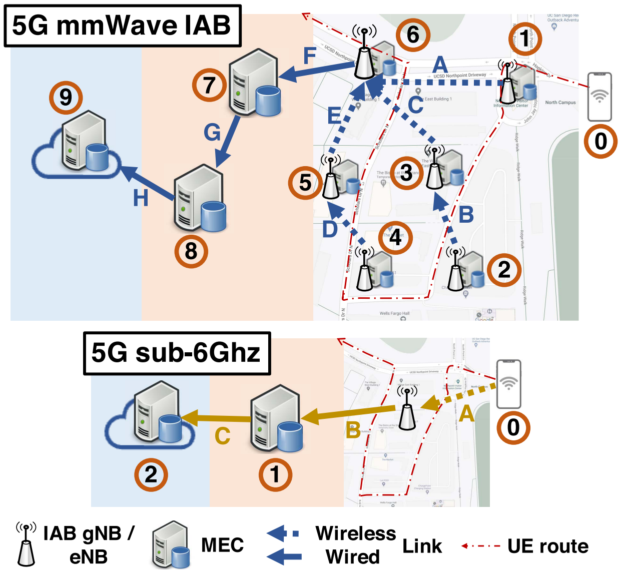


Fig. 10. Simulated 5G network topology and UE trajectory.

VII. Evaluation

A. Simulation Setup

We evaluate the HiveMind framework on a custom built 5G network simulator. Below we introduce the key components of the simulator.

1) Network Topology: We evaluate HiveMind on 2 representative 5G networks: a 5G mmWave IAB network and a 5G sub-6 GHz network. As illustrated in Fig. 10, the 5G mmWave IAB network consists of 6 IAB nodes, each equipped with a mobile MEC. The IAB nodes are deployed in a $1km \times 1km$ region, and form a tree-topology in compliance with the 3GPP Release-15 guidelines [16]. Traffic from the IAB nodes is aggregated and flows to the core network through the donor IAB (node 6) and its wired backhaul link (link F). There are 2 additional MECs (node 7 and 8) in the core network. A cloud server is connected to node 8 through public IP, serving as the sink node. The sub-6 GHz network consists of a single basestation. There is one MEC in the core network and a cloud server in public IP. All wireless link capacities are configured according to the maximum uplink MCS' bitrate specified by 3GPP [54], as shown in Table I.

2) UE Mobility: A target UE within the IAB network's coverage area requests the split ML task at the beginning of each experimental trial. It keeps running the task while moving along a pre-defined trajectory at a speed of $40km/h$. We calculate the UE's access link SNR using the Friis model assuming an EIRP limit of 40dBm as specified by the

FCC [55]. We then convert the SNR to link bitrate by 5G NR CQI to MCS mapping table [56]. Although this channel and link model does not account for sophisticated propagation effects, it should suffice to generate a similar level of dynamics as real-world mobile cellular links, which is critical for testing the model splitting.

3) Background Traffic: We generate the number of UEs and individual UEs' bit-rate distribution following the traffic emulation approach in [48], which has been cross-validated with real-world traffic traces. We then feed the background UEs' bit-rate samples along with the target UE's bit-rate to a widely-adopted Proportional Fair Scheduler (PFS) [57], which allocates the channel resources and determines the link capacity for the target UE. In addition, we scale the UE population and individual UEs' bitrate by $2, 1$, and $\frac{1}{2}$ to create high, mid, and low background traffic scenarios.

4) ML Cost Profiles: The computation and communication cost profiles of an ML model are crucial for a faithful representation of the model workload, and also serve as inputs to the HiveMind framework. To obtain realistic ML cost profiles, we build a Python-based latency/energy profiling tool as a stand-alone package for Pytorch. The profiling tool traverses the ML model object and registers a forward hook and a backward hook for each layer (a `torch.nn.functional` object). To generate latency profiles, the hooks measure the processing time of a layer during inference and training phases using Python's built-in `time.perf_counter()` module with an accuracy of $\pm 1 \mu s$. The per-layer processing time values are organized in the order of the layers as the computation latency profile. The hooks also record the output size of the layer for inference and training, which is used to calculate the communication latency based on Eq. (10). For computing energy profiles, we adopt *PyJoules* [58], a third party energy footprint monitor that measures the computing energy of a Python function. For communication energy profiles, due to the lack of 5G interface on our devices, we adopt the link bitrate/power mapping data from the latest operational 5G measurement [47] to convert the instantaneous link bitrate in our simulation to power consumption. We measure the cost profiles on the following four sets of machines to represent the UE, IAB MEC, core MEC, and cloud server, respectively: *(i)* Raspberry Pi 3 Model B+, *(ii)* MacBook Air 2020 with Apple M1 CPU and 16GB unified memory, *(iii)* A PC with Ryzen 3800X CPU, 64GB DDR 4 RAM, and Nvidia RTX 2080 GPU *(iv)* A server with Intel 9990XE CPU, 128GB DDR4 RAM, and 4 × Nvidia RTX 1080Ti GPUs.

B. Multi-Split Performance Validation

1) Impact of Background Traffic: To evaluate the effectiveness of HiveMind under different background network traffic intensities, we run HiveMind inference and training along with three baseline approaches: *UE Only*, which runs the whole model on the UE; *Cloud Only*, which runs the whole model on the cloud server; *Single Split*, which splits the network only between the UE and a cloud server as in [10]. We use ResNet18, a widely used image classification CNN with a moderate number of layers (52) as the split ML model. For inference tasks, we choose the total processing latency as the optimization objective. For training tasks, we choose the sum energy consumption of IAB MECs and UE as the optimization objective, since the training tasks are not as latency-sensitive as the inference tasks.

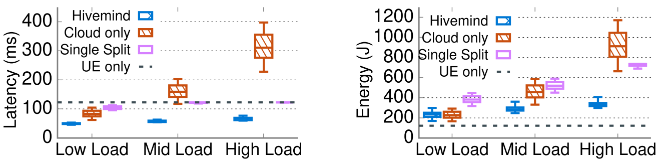
TABLE I
LINK SETTINGS

Index	Type	Capacity (Mbps)	UE bandwidth	
			Mean	Std.
5G mmWave IAB				
A	Wireless	200	66.49	19.47
B	Wireless	200	71.89	19.72
C	Wireless	200	47.06	8.95
D	Wireless	200	54.03	11.61
E	Wireless	200	48.59	9.41
F	Wired	400	54.05	12.94
G	Wired	200	36.03	8.63
H	Wired	50	18.12	4.43
5G sub-6GHz				
A & B	Wireless	50	33.94	8.45
C	Wired	50	18.78	3.66

Fig. 11 and 12 plot the running cost distribution of each method under the mmWave IAB and sub-6GHz networks, respectively. We see that for both mmWave IAB and sub-6GHz networks, HiveMind achieves the minimal running cost for all traffic intensities. For mmWave IAB split inference (Fig. 11(a)), the latency of the cloud-only baseline quickly increases to over 300ms as the network becomes more loaded, while the single split converges to the UE-only baseline due to the increasing communication latency. HiveMind is able to keep a stable running latency of $\leq 79\ ms$ by limiting the usage of the core and cloud servers under high network load. For mmWave IAB split training (Fig. 11(b)), the energy consumption of the cloud-only method reaches up to $890J$ due to the less power efficient low bitrate links under high network load. HiveMind keeps the IAB nodes' energy consumption under $400J$ by adjusting the split to restrain the data transfer on the links. This set of experiments proves that in the mmWave IAB network, HiveMind can adapt the split assignment to the dynamic network load and outperforms all the baselines on both the inference and training tasks.

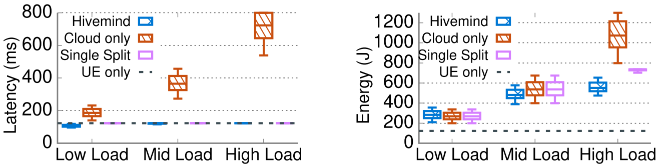
For sub-6 GHz split inference, we observe that the running latency of HiveMind converges to the UE-only approach as the network load increases. The underlying reason is that, due to the relatively lower link capacity (high communication latency) outweighs the low computation latency on the MECs, so that the best split strategy is simply to run the whole model locally on the UE. The result verifies that, even when the network condition provides no benefit for splitting the ML, HiveMind is able to converge to the best non-split strategy and yield the best performance.

2) Impact of ML Models: We repeat the above experiments with three ML models with different computation and communication characteristics: *VGG11*, a 16-layer CNN model with an average layer output size of 356KB; *ResNet18*, a 52-layer CNN model with an average layer output size of 80KB; *R-CNN*, a two-part CNN with the average layer output size of 242KB. We define *relative efficiency* of a strategy as the



(a) (b)

Fig. 11. Efficiency of (a) HiveMind split inference and (b) HiveMind split training, in mmWave IAB network.



(a) (b)

Fig. 12. Efficiency of (a) HiveMind split inference and (b) HiveMind split training, in sub-6GHz network.

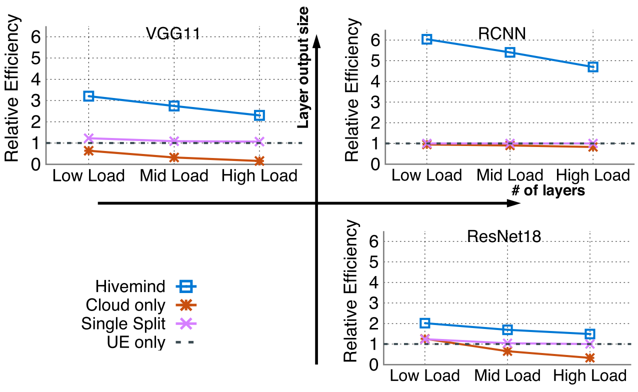


Fig. 13. Impact of ML models on HiveMind.

average ratio between the cost of the strategy and the cost of the UE-only baseline under the same scenario for both training and inference. Fig. 13 shows the results. We see that HiveMind achieves the highest relative efficiency on RCNN, 3× over ResNet18 and 1.8× over VGG11. This is because the large communication overhead corresponding to the larger layer output size provides HiveMind with more improvement margin, while a large number of layers gives HiveMind more flexibility to split the model in more efficient ways. In other words, HiveMind sees more performance gains with a larger layer output size and a larger number of layers.

3) Impact of MEC Capability: In this section we examine the performance of split ML with different MEC capabilites. The capability changes on different MEC nodes may affect the split ML differently depending on the MEC servers' proximity to the UE. Therefore, we isolate the impact of capability change for each type of MEC. Specifically, we repeat the above ResNet18 inference experiment and in each trial, choose one type of MEC nodes and scale their computation latency profiles to $\{\times 1/10, \times 1/2, \times 1, \times 2, \times 10, \times 20\}$, in order to represent the different computational capabilities while keeping the rest of the MEC nodes' latency profiles unchanged. The results are denoted as *improved IAB*, *improved CN*, and *improved cloud* for IAB MEC nodes, core MEC nodes, and

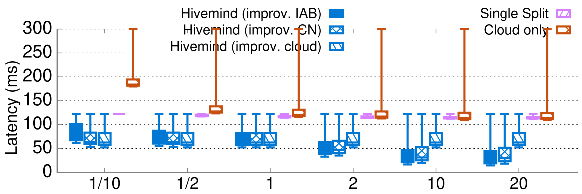


Fig. 14. Impact of computation capability of various MECs on HiveMind split inference.

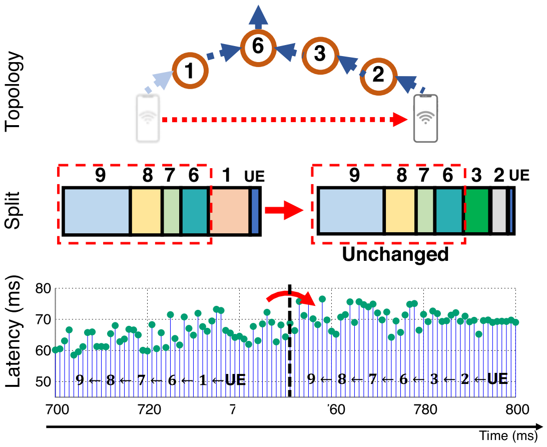


Fig. 15. HiveMind topology adaptation showcase: the split assignment does not change for the unchanged part of the route and the average latency increases less than 5ms after the topology change.

the cloud server, respectively. Note that for cloud only and single split baselines, we only evaluate the performance under cloud server capability change since they do not use IAB or core MEC nodes. Fig. 14 shows the latency under different MEC capabilities. We see that as the scale increase from $1/10$ to 20, the improved IAB MECs result in the most latency reduction, 14.5% lower latency than the CN MECs, and 47.2% than the cloud server. This implies that *the split DNN benefits the most with computational capacity improvement on IAB MECs, which are closest to the UEs*. Meanwhile, the single-split and cloud-only approaches converge to 115 ms and 181 ms regardless of the MEC capabilities, indicating the bottleneck of these two approaches lies in the communication latency. We also observe that the overall latency performance for $\times 10$ and $\times 20$ is very close, which shows that improving MEC capability cannot improve the split ML's performance indefinitely since it is also bounded by the communication links' capacity.

C. Performance Under Network Dynamics

1) Impact of Network Topology Changes: In 5G IAB networks, the network topology may vary over time, as some gNBs may be put into sleep, or additional gNBs are added for load balancing purposes. Meanwhile, the UE mobility and handoff also causes changes in the network path. To showcase the effectiveness of our HiveMind design under such topology dynamics, we randomly select a a period of time in the above ResNet18 experiment where a topology change happens due to UE mobility, and plot its corresponding workload and latency change in Fig. 15. This topology change represents an extreme

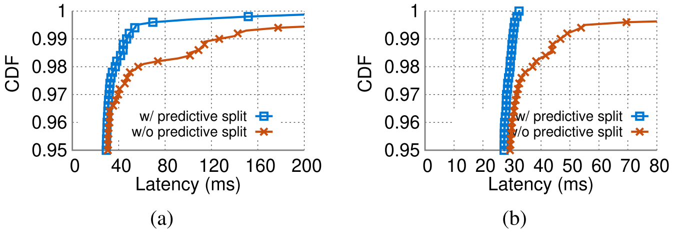


Fig. 16. Predictive splitting performance under (a) 10ms and (b) 50ms link coherent time.

case where not only the access gNB changes from gNB1 to gNB2 due to the UE mobility but also the number of hops on the route increases by 1. After the topology change, we see the workload assignment remains the same for the unchanged part of the route. Such persistence originates from a unique property of our HiveMind design, *i.e.*, *a MEC node's optimal split decision is independent of the upstream MECs* (Sec. IV). Moreover, *HiveMind is stateless and does not require data transfer between the old and new MECs upon a gNB handoff*, which further improves the responsiveness of the system in the presence of UE mobility. We also see a less than 5 ms increase in average latency after the topology change event. This implies the added MEC can compensate for the extra communication latency caused by the extended route and vice versa.

2) Effectiveness of Predictive Split: To investigate the performance of HiveMind's predictive split under different levels of network dynamics, we compare it with the standard split DNN. We use link coherent time, *i.e.*, the time during which the link capacity is stable, to represent varying levels of network dynamics. We choose 10ms coherence time to represent extreme network dynamics and 50ms coherent time to represent typical network dynamics in cellular networks during rush hours [59]. We compare the predictive split with the standard split DNN Fig. 16 shows the result. We see under 10 ms link coherent time, the predictive split still leaves 12% of latency spikes over 100ms. As mentioned in Sec. IVE, this is because the time gap Δt between the SCI message and the split task is much larger than the link coherent time. With the same amount of time gap estimation error, the link capacity estimation is more likely to deviate from the correct value under short link coherent time. In contrast, predictive split eliminates all latency spikes over 100ms when link coherent time is 50 ms, which is similar to the average look-ahead time for predictive split in our setup. This implies *the predictive split mechanism is effective, as long as the link coherent time is no less than the average look-ahead time of the split decision*.

3) Impact of Link Prediction Accuracy: The previous experiment assumes a 100% link prediction accuracy. We further investigate the impact of link prediction errors on HiveMind's performance. Reusing the experiment setup for the 50ms link coherent time, we run the split task with different link prediction accuracy. We model the link prediction error as a normal distribution with the correct link capacity as mean and various relative prediction errors as standard deviation. We generate random samples from these distributions for predictive split. Fig. 17 shows the median latency against relative prediction error. We see the latency performance deteriorates when the prediction error increases and the performance gain turns to loss at 13%. Given that the state-of-art link prediction mechanism can achieve $< 10\%$ prediction error over 91% of time [49], the result implies the *predictive split is able to tolerate typical link prediction errors*. Note that the median latency converges to 130 ms as prediction error increases beyond 17%, because as the prediction error increases, it is more likely that the access link or one of the IAB links is predicted with extremely small capacity and the predictive split assigns the whole model to UE and IAB MECs, which takes around 130 ms to run the model.

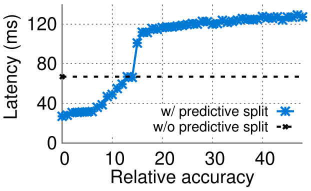


Fig. 17. Impact of link prediction accuracy on the predictive split.

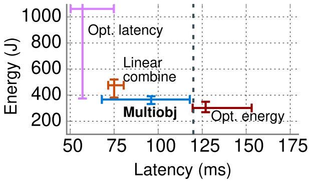


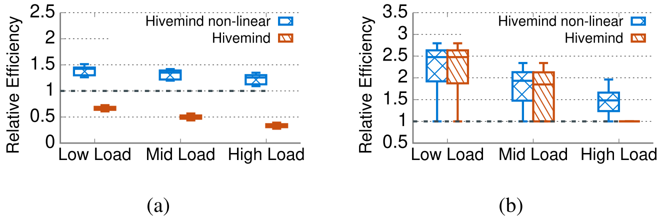
Fig. 18. Energy consumption and running latency comparison of HiveMind multi-objective and baselines.

D. Effectiveness of Multi-Objective Split

To evaluate the effectiveness of the multi-objective split, we repeat the above ResNet18 experiment under intermediate network load. The objective is to minimize energy consumption (best effort) while limiting the latency below $120\ ms$ (quality assurance). We compare HiveMind's multi-objective design with three baselines: *Opt. latency* which only optimizes latency, *Opt. energy* which only optimizes energy, and *linear combine*, which linearly combines the objectives to calculate the edge cost as discussed in Sec. V. Fig. 18 shows the latency and energy consumption distribution. We see that both Opt. latency and Opt. energy achieve the overall minimal for their targeted metric, while sacrificing the other metric, implying that energy and latency are two conflicting objectives in such split tasks. Linear combine balances the two metrics and achieves an average cost of $75ms/476J$, whose latency value is unnecessarily low. In contrast, the HiveMind multi-objective design further reduces the average energy consumption by 22.9% on top of linear combine while keeping the maximal latency under the $120ms$ constraint. The result indicates that HiveMind multi-objective design is able to simultaneously accommodate the best effort and quality assurance objectives.

E. Effectiveness in Splitting Non-Linear ML Models

We investigate the performance of split collaborative learning and split RNN under varying traffic loads in the mmWave IAB network. To evaluate the split RNN model, we build a customized sequence labeling RNN consisting of 200 GRU



(a)

(b)

Fig. 19. Efficiency improvement of HiveMind non-linear on (a) RNN model (GRU) and (b) collaborative learning model (QMIX) over standard HiveMind split.

recurrent module. For the split collaborative learning model, we choose a well known multi-agent reinforcement learning model, QMIX [60], which consists of a central mixing model and N RNN-based agent models. We assume the central model is located at the cloud server (the sink) and the optimization only applies to the agent models. For both experiments, we set the energy consumption as the optimization objective with a latency constraint of $120ms$. We compare split RNN with the standard HiveMind design which repeats a split recurrent module every iteration and split collaborative learning with the standard HiveMind which splits the agent network without considering the parameter transfer cost. Fig. 8 shows the relative efficiency. We see the split RNN outperforms the standard HiveMind by $1.7\times$ under low load and $2.3\times$ under high load, due to the increasing hidden state transfer cost under the high network load. Similarly, split collaborative learning outperforms standard HiveMind by $1.5\times$ under high network load, where the latter fails to account for the large parameter transfer cost. The results prove that *the split non-linear designs can capture the unique cost components in non-linear NNs and outperform the standard HiveMind by a large margin*.

VIII. Conclusion

In this paper, we have explored the multi-split ML as a new paradigm to integrate edge intelligence to 5G systems. Our HiveMind framework distributively finds the optimal multi-split under network dynamics and adapts to multiple optimization objectives and neural network structures. Our experiments demonstrate that HiveMind significantly improves ML running efficiency with various network dynamics, ML models, and MEC capabilities. We believe HiveMind envisions a new direction to harness collaborative edge power to boost ML intelligence in the 5G era.

References

[1] L. Google. (2020). *Ml Kit—Google Developers*. [Online]. Available: https://developers.google.com/ml-kit

[2] A. Inc. (2020). *Core Ml—Apple Developer Documentation*. [Online]. Available: https://developer.apple.com/documentation/coreml

[3] A. kodra. (2020). *Awesome-Mobile-Machine-Learning*. [Online]. Available: https://github.com/fritzlabs/Awesome-Mobile-Machine-Learning

[4] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *Proc. World Wide Web Conf.*, 2019, pp. 2125–2136.

[5] *Y.3172: Architectural Framework for Machine Learning in Future Networks Including IMT-2020*, Int. Telecommun. Union, Geneva, Switzerland, Jun. 2019.

[6] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jun. 2016, pp. 25–33.

[7] Z. Lin and M. van der Schaar, "Autonomic and distributed joint routing and power control for delay-sensitive applications in multi-hop wireless networks," *IEEE Trans. Wireless Commun.*, vol. 10, no. 1, pp. 102–113, Jan. 2011.

[8] K.-L.-A. Yau, J. Qadir, C. Wu, M. A. Imran, and M. H. Ling, "Cognition-inspired 5G cellular networks: A review and the road ahead," *IEEE Access*, vol. 6, pp. 35072–35090, 2018.

[9] Y. Xiao, G. Shi, Y. Li, W. Saad, and H. V. Poor, "Toward self-learning edge intelligence in 6G," *IEEE Commun. Mag.*, vol. 58, no. 12, pp. 34–40, Dec. 2020.

[10] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[11] S. Kekki *et al.*, "Mec in 5G networks," *ETSI White Paper*, vol. 28, pp. 1–28, Jun. 2018.

[12] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.

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

[14] I. Stoica *et al.*, "A Berkeley view of systems challenges for AI," 2017, *arXiv:1712.05855*. [Online]. Available: https://arxiv.org/abs/1712.05855

[15] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun.*, Aug. 2018, pp. 31–36.

[16] *Nr; Study on Integrated Access and Backhaul*, document TR 38.874 V16.0.0, 3GPP, 2019.

[17] *5G System (5GS); Study on Traffic Characteristics and Performance Requirements for AI/ML Model Transfer*, document TR 22.874 V0.0.0, 2020.

[18] Q. Ho *et al.*, "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.

[19] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1111–1133, 2014.

[20] I. Foster and A. Iamnitchi, "On death, taxes, and the convergence of peer-to-peer and grid computing," in *Proc. Int. Workshop Peer-To-Peer Syst.* Berkeley, CA, USA: Springer, 2003, pp. 118–128.

[21] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Trans. Intell. Syst. Technol. (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[22] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*. [Online]. Available: https://arxiv.org/abs/1610.05492

[23] C. Niu *et al.*, "Billion-scale federated learning on mobile clients: A submodel design with tunable privacy," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw.*, Sep. 2020, pp. 1–14.

[24] Q. Chen, Z. Zheng, C. Hu, D. Wang, and F. Liu, "Data-driven task allocation for multi-task transfer learning on the edge," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1040–1050.

[25] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. USA: JMLR, 2017, pp. 1273–1282.

[26] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. 53rd Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2015, pp. 1310–1321.

[27] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2019, pp. 1423–1431.

[28] D. Narayanan *et al.*, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 1–15.

[29] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, Dec. 2016, pp. 2464–2469.

[30] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 328–339.

[31] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," 2017, *arXiv:1702.07811*. [Online]. Available: http://arxiv.org/abs/1702.07811

[32] C. Lo, Y.-Y. Su, C.-Y. Lee, and S.-C. Chang, "A dynamic deep neural network design for efficient workload allocation in edge computing," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Nov. 2017, pp. 273–280.

[33] S. Leroux *et al.*, "The cascading neural network: Building the internet of smart things," *Knowl. Inf. Syst.*, vol. 52, no. 3, pp. 791–814, 2017.

[34] L. Li, K. Ota, and M. Dong, "Deep learning for smart industry: Efficient manufacture inspection system with fog computing," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4665–4673, Oct. 2018.

[35] X. Xie and K.-H. Kim, "Source compression with bounded DNN perception loss for IoT edge computer vision," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2019, pp. 1–16.

[36] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "JALAD: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst.*, Dec. 2018, pp. 671–678.

[37] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," 2017, *arXiv:1712.01887*. [Online]. Available: https://arxiv.org/abs/1712.01887

[38] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified SGD with memory," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 4447–4458.

[39] H. Tang, S. Gan, C. Zhang, T. Zhang, and J. Liu, "Communication compression for decentralized training," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 7652–7662.

[40] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.

[41] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: https://arxiv.org/abs/1510.00149

[42] S. Yao *et al.*, "Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency," in *Proc. 18th Conf. Embedded Networked Sensor Syst.*, 2020, pp. 476–488.

[43] F. Giust *et al.*, "MEC deployments in 4G and evolution towards 5G," *ETSI White Paper*, vol. 24, pp. 1–24, Feb. 2018.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[45] P. A. Humblet, "Another adaptive distributed shortest path algorithm," *IEEE Trans. Commun.*, vol. 39, no. 6, pp. 995–1003, Jun. 1991.

[46] M. S. Corson and A. Ephremides, "A distributed routing algorithm for mobile wireless networks," *Wireless Netw.*, vol. 1, no. 1, pp. 61–81, Mar. 1995.

[47] D. Xu *et al.*, "Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 479–494.

[48] J. Ding, X. Liu, Y. Li, D. Wu, D. Jin, and S. Chen, "Measurement-driven capability modeling for mobile network in large-scale urban environment," in *Proc. IEEE 13th Int. Conf. Mobile Ad Hoc Sensor Syst.*, Oct. 2016, pp. 92–100.

[49] C. Yue, R. Jin, K. Suh, Y. Qin, B. Wang, and W. Wei, "LinkForecast: Cellular link bandwidth prediction in LTE networks," *IEEE Trans. Commun.*, vol. 17, no. 7, pp. 1582–1594, Jul. 2017.

[50] *Multi-Access Edge Computing (MEC); Support for Network Slicing*, document ETSI GR MEC 024 V2.1.1, ETSI, 2019.

[51] P. Manyem and J. Ugon, "Computational complexity, NP completeness and optimization duality: A survey," *Electron. Colloq. Comput. Complex.*, vol. 19, p. 9, Feb. 2012.

[52] Sgrvinod. (2020). *A-Pytorch-Tutorial-to-Sequence-Labeling*. [Online]. Available: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Sequence-Labeling

[53] H. Pan *et al.*, "Dissecting the communication latency in distributed deep sparse learning," in *Proc. ACM Internet Meas. Conf.*, 2020, pp. 528–534.

[54] *Nr; Physical Channels and Modulation*, document TR 38.214 V16.4.0, 3GPP, 2021.

[55] *FCC 15-138A1*, Federal Commun. Commission, Washington, DC, USA, 2015.

[56] *Nr; Physical Layer Procedures for Data*, document TR 38.214 V16.4.0, 3GPP, 2021.

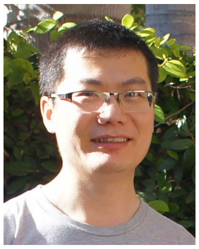
[57] R. Kwan, C. Leung, and J. Zhang, "Proportional fair multiuser scheduling in LTE," *IEEE Signal Process. Lett.*, vol. 16, no. 6, pp. 461–464, Jun. 2009.

[58] R. R. C. Belgaid and A. d'Azémar. (2020). *PyJoules PyPi*. [Online]. Available: https://pypi.org/project/pyJoules/

[59] F. Malandrino, C.-F. Chiasserini, and S. Kirkpatrick, "Cellular network traces towards 5G: Usage, analysis and generation," *IEEE Trans. Mobile Comput.*, vol. 17, no. 3, pp. 529–542, Mar. 2017.

[60] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, and S. Whiteson, "QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning," 2018, *arXiv:1803.11485*. [Online]. Available: https://arxiv.org/abs/1803.11485

Song Wang received the B.S. degree in the Internet of Things engineering from Beijing University of Posts and Telecommunications in 2018. He is currently pursuing the Ph.D. degree in communication system and theory with the Department of Electrical and Computer Engineering (ECE), University of California at San Diego, San Diego, CA, USA. His research interests include wireless networking and communication.



Xinyu Zhang (Senior Member, IEEE) received the Ph.D. degree from the University of Michigan in 2012. Prior to joining the University of California San Diego in 2017, he was an Assistant Professor at the University of Wisconsin–Madison. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of California San Diego. His research interests include wireless systems and ubiquitous computing. He was a recipient of two ACM MobiCom Best Paper Award in 2011 and 2020, the Communications of the ACM Research Highlight in 2018, the ACM SIGMOBILE Research Highlight in 2018, the NSF CAREER Award in 2014, the Google Research Award in 2017, 2018, and 2020, and the Sony Research Award in 2018 and 2020. He served as the TPC Chair for ACM MobiCom 2019, IEEE SECON 2017, a Co-Chair for NSF millimeter-wave research coordination network, and an Associate Editor for IEEE TRANSACTIONS ON MOBILE COMPUTING from 2017 to 2020.

Hiromasa Uchiyama received the B.E. and M.S. degrees in engineering from Tokyo University of Agriculture and Technology, Tokyo, Japan, in 2006 and 2008, respectively. In 2008, he joined Sony Group Corporation. He has been attending 3GPP RAN1 Group and working on standardizations of 4G/5G cellular communications. He is currently a Senior Manager at the Research and Development Center. His research interests include sidelink communication and advanced relay communication.

Hiroki Matsuda received the B.S. and M.S. degrees in communications engineering from Tohoku University, Sendai, Japan, in 2008 and 2010, respectively. He is currently working at Sony Group Corporation, where he researches next generation technologies. He is also delegate of Third Generation Partnership Project (3GPP). His research interests include non-orthogonal multiple access (NOMA) and non-terrestrial network (NTN).