GReMS: Graph-Reinforced Neural Micro-Splitting over Dynamic Heterogeneous IoT Networks

Antonio Di Maio
ETH Zurich
Zurich, Switzerland
dimaio@ethz.ch

Ahmed Ibrahim
ETH Zurich
Zurich, Switzerland
ahibrahim@ethz.ch

***Abstract*—The deployment of Deep Neural Networks at the network edge is constrained by limited compute and communication resources. While Split Learning distributes inference across edge devices to improve resource utilization and scalability, existing methods fix the model partitioning at deployment time and cannot adapt to runtime fluctuations in link bandwidth or compute availability. Moreover, prior work assumes static network topologies, requiring retraining whenever the device mesh changes, which is a critical methodological limitation for heterogeneous and dynamic edge environments. This work proposes Graph-Reinforced Neural Micro-Splitting (GReMS), a Graph Neural Network-based Reinforcement Learning system in which a centralized coordinator observes the network state and continuously readapt the optimal Split Learning configuration. GReMS encodes heterogeneous, time-varying topologies and their device resources as a labeled graph, and a shared Graph Neural Network-based policy enables zero-shot generalization to variable mesh sizes and dynamics without retraining, continuously adapting split configurations under non-stationary channel and compute conditions. Across a broad range of system sizes, link capacities, and compute load fluctuations GReMS reduces inference latency by up to $90\,\%$ over state-of-the-art baselines, and delivers robust, effective micro-split reconfigurations under non-stationary conditions where non-adaptive methods fail catastrophically.**

***Index Terms*—Split Learning; Edge Intelligence; Heterogeneous Internet of Things.**

I. Introduction

Integrating Artificial Intelligence (AI) into everyday infrastructure, from healthcare monitors to autonomous robots, requires executing demanding Deep Neural Networks (DNNs) in near-real time on computation- and communication-constrained Internet of Things (IoT) devices. In modern deployments, the extreme and continuously improving energy efficiency make computation throughput and communication bandwidth the primary bottlenecks determining inference latency [1]. Split Learning (SL) bridges the gap between massive computational demands and the tight resource budgets of edge hardware by partitioning a Neural Network (NN) into distinct segments and balancing the workload between an edge device and a server or peer cluster, offering a hardware efficient execution framework for IoT devices [2], [3]. A representative deployment is a mesh of wireless sensors monitoring an industrial facility, where sensor nodes may relocate during operation and links appear or disappear due to fading, interference, and node failures. In such a deployment, an edge orchestrator can designate any node as the *inference hub*, i.e., the device that ingests raw sensor data and consumes the resulting model prediction while the remaining peers share the intermediate computation, because the optimal hub depends on which node sits closest to the relevant sensor cluster, how well it is connected to the rest of the mesh, and how much compute capacity it currently holds. For example, a smoke-sensing node captures air-quality readings and uses the inferred fire probability to trigger a local sprinkler, acting as both the data source and the consumer of the prediction. For this class of applications, the value of a prediction degrades rapidly with delay, making end to end inference latency the governing performance objective. Sending all raw sensor data to a remote server requires transmitting large volumes of data over the constrained wireless medium, increasing end to end inference latency, network congestion, and limiting scalability, so the Deep Neural Network (DNN) inference pipeline should instead be optimally distributed across the local cooperative mesh to meet the application's latency requirements, despite heterogeneous node hardware and fluctuating link quality.

Deploying DNNs over the network edge is challenging because such environments are highly dynamic and heterogeneous, i.e., compute capabilities fluctuate with competing workloads and thermal constraints, while wireless link quality varies due to channel fading, interference, obstacles, and node mobility. Real IoT deployments are fundamentally *heterogeneous*: node compute capabilities differ across hardware generations and device classes [4], while link capacities vary with distance, fading, and interference, spanning orders of magnitude within the same network [5], [6]. Effective inference orchestration in realistic IoT demands continuously replanning which nodes execute which model segments, since both the mesh topology and the per link and per node resource capacities fluctuate. Most existing solutions, however, commit to static, precalculated execution plans that cannot respond to this variability because they ignore the stochastic nature of edge environments, leading to suboptimal resource utilization and higher end to end latency.

To overcome these limitations, this work develops an adaptive, resource aware framework for SL. We move beyond fixed algorithms by using Reinforcement Learning (RL) to drive the partitioning logic. Existing RL based splitters target fixed topologies with a predetermined number of nodes, requiring

979-8-3195-0167-7/26/$31.00 ©2026 IEEE

full retraining whenever the mesh size or connectivity changes. By learning from real time system states, the agent identifies offloading strategies that adapt to environmental shifts ignored by static models.

This work aims to answer the following research questions:

- How can a DNN partitioning and allocation be continuously adapted across cooperative, resource-constrained, highly-dynamic IoT networks by leveraging system-level state prediction for long-term optimization of end-to-end inference?
- To what extent can a policy learned on one distribution of network topologies generalize to unseen scenarios of different scales and properties?
- Under time-varying network resource conditions, how does an adaptive model splitting and allocation replanning strategy affect inference latency compared to a fixed optimized assignment?

The main contributions of this work are as follows. We propose Graph-Reinforced Neural Micro-Splitting (GReMS), a topology-agnostic Graph Neural Network (GNN)-based RL agent for real time adaptive SL that continuously observes the channel and compute state of an IoT device mesh and reselects the optimal model microsplitting and worker allocation, generalizing across variable size meshes and unseen Random Geometric Graph (RGG) instances without retraining. Under simultaneous link degradation modeled by Gilbert-Elliott (G-E) processes and compute fluctuations modeled by a Markov-Modulated Poisson Process (MMPP), GReMS achieves significant latency reductions over non-adaptive baselines and sustains robustness under non-stationary conditions.

II. Related Works

The shift from centralized cloud processing toward distributed execution has produced diverse partitioning strategies. Early work, like MoDNN, used horizontal splitting to distribute layer-wise workloads across neighboring nodes and aggregate idle processing power [7]. Multi-dimensional frameworks such as DiSNet [8] extended this with vertical partitioning for micro-split deep learning. However, these systems select execution plans from initial system snapshots, so when nodes enter power-saving modes or experience signal degradation, the lack of adaptive scheduling causes synchronization bottlenecks and severe performance loss.

EnSplit [9] applies a Double Deep Q-Network (DDQN) agent to split inference in 6G networks, adding a subscription-level energy credit constraint to regulate UE offloading over time, but its topology is fixed by the architecture and it supports only layer-wise splitting, so it neither targets irregular IoT mesh topologies nor multi-dimensional partitioning.

Beyond partitioning granularity, other work targets communication overhead at the model's bottleneck layer through learned compressed representations [10], multi-task compression models [11], and standardized supervised-compression benchmarks [12]. These methods improve data efficiency but remain passive, leaving the high-level orchestration needed for mesh-wide stability in volatile topologies unaddressed.

Other communication-aware designs add resilience through early-exit strategies [13] or contrastive learning against data distribution shifts and catastrophic forgetting [14], yet a critical gap remains in the autonomous management of volatile mesh networks, as existing methods stay largely reactive or rely on fixed heuristics that cannot cover the vast state space of a constantly changing environment.

GReMS is the first method that combines multi-dimensional splitting, fine-grained partitioning, latency optimization, support for variable resources, a focus on IoT mesh networks, autonomous adaptability, and a heterogeneous network model.

III. System Model

A. Network Model

We consider a scenario containing a set of N heterogeneous devices identified by an index set $V = \{1, \dots, N\}$. Each device $w \in V$ has a static compute capability $\rho_w \in \mathbb{R}_+$ [FLOP/s]. When a pair $(i, j) \in E_t$ of devices can communicate wirelessly at time t we say an *active link* exists between i and j, where $E_t \subseteq V \times V$ is the set of active links at step t. Each link $(i, j) \in E_t$ has a capacity $B_{ij}^{(t)}$[Mbit/s] $\in \mathbb{R}_{>0}$ at time t. Therefore, the scenario can be modeled as a time-varying graph $G_t = (V, E_t)$. A representative instance of such a network is illustrated in Figure 1.

B. Dynamic Network Conditions

Real IoT mesh networks are subject to two forms of temporal variation: fluctuating node compute availability due to background processes, and fluctuating link quality due to wireless fading and interference. We model the time-varying compute capability as a MMPP chain and the time-varying link quality as a G-E chain (Figure 2) as follows.

a) Node Compute-Load Dynamic Model: Background tasks on device w consume a fraction $\ell_w^{(t)} \in [0, 1)$ of its compute capacity, leaving an effective capacity $\rho_w^{(t)} = (1-\ell_w^{(t)})\rho_w$ available for inference.

The fraction $\ell_w^{(t)}$ is drawn from a two-state MMPP [15], [16]: the *Low-Load* state L_ℓ corresponds to $\ell_w^{(t)} \sim \mathcal{U}([\ell_{\min}^{\mathrm{L}}, \ell_{\max}^{\mathrm{L}}])$ and the *High-Load* state H_ℓ corresponds to $\ell_w^{(t)} \sim \mathcal{U}([\ell_{\min}^{\mathrm{H}}, \ell_{\max}^{\mathrm{H}}])$. The probabilities of transition from low to high and high to low load states are p_{LH} and p_{HL}, respectively (Figure 2).

b) Link Capacity Dynamic Model: Each link (i, j) is independently governed by a two-state G-E [15], [16] composed of a *High-Capacity* state and a *Low-Capacity* state. The link capacity at step t is $B_{ij}^{(t)} = \beta_{ij}^{(t)} B_{ij}$, where B_{ij} is the (i, j) link's data rate, i.e. its maximum theoretical throughput, $\beta_{ij}^{(t)} \sim \mathcal{U}([\beta_{\min}^{\mathrm{H}}, \beta_{\max}^{\mathrm{H}}])$ in the High-Capacity state and $\beta_{ij}^{(t)} \sim \mathcal{U}([\beta_{\min}^{\mathrm{L}}, \beta_{\max}^{\mathrm{L}}])$ in the Low-Capacity state. The probabilities of transition from Low to High and High to Low capacity states are, respectively, q_{LH} and q_{HL} (Figure 2).

C. DNN Model

The system device cooperate to perform an inference task by executing a sequential DNN with L layers $\mathcal{L} = \{1, \dots, L\}$

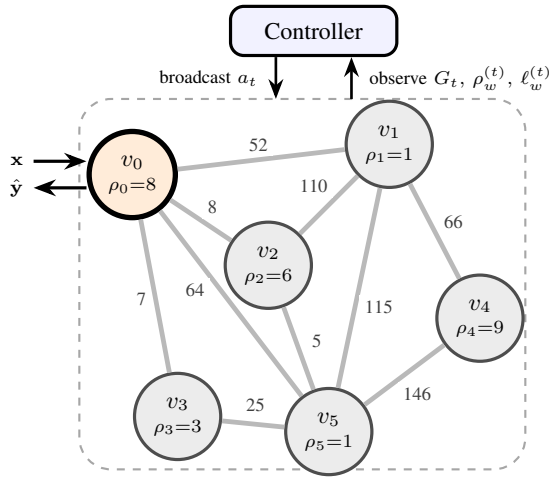


Fig. 1. Representative IoT mesh network with $N = 6$ heterogeneous devices at time t. Node labels show compute capability ρ_i. Edge labels show link capacity $B_{ij}^{(t)}$ [Mbit/s]. The hub node v_0 (highlighted) receives the input x and returns the inference result $\hat{y}$. An out-of-band controller continuously observes the network state $(G_t, \rho_w^{(t)}, \ell_w^{(t)})$, computes the split configuration a_t, and broadcasts it to all participating workers.

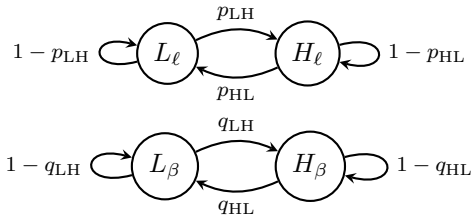


Fig. 2. Two-state Markov chains for the node-load MMPP (left, Low/High compute load with transition probabilities $p_{\mathrm{LH}}, p_{\mathrm{HL}}$) and the link-capacity G-E model (right, Low/High capacity with transition probabilities $q_{\mathrm{LH}}, q_{\mathrm{HL}}$)

over a given input $\mathbf{x}$. The network produces an inference result $\hat{y}$ representing the model's prediction, such as a class label or anomaly score. Layer l has output activation size $D_l[\mathrm{B}] \in \mathbb{R}_{>0}$, a computational requirement $T_l[\mathrm{FLOP}] \in \mathbb{R}_+$, and an execution time $T_{lw}[\mathrm{s}] = T_l[\mathrm{FLOP}]/\rho_w[\mathrm{FLOP/s}]$ on device w.

The DNN comprises two execution categories that behave differently under spatial parallelism. *Spatial layers* such as convolutional and pooling layers, whose index set we denote $\mathcal{L}^S \subseteq \mathcal{L}$, operate on three-dimensional feature maps of shape $H \times W \times C$ and admit row-wise parallelism: the feature map is partitioned into K horizontal row slices, one per worker. At each of the $K - 1$ borders between adjacent workers, each worker exchanges r boundary rows with its neighbour to correctly compute convolution outputs near that shared border, where

$$r = \left\lfloor \frac{1 + \sum_l (k_l - 1)\, j_l}{2} \right\rfloor, \qquad j_l = \prod_{i<l} s_i, \tag{1}$$

k_l and s_l are the kernel size and stride of layer l, and j_l is the cumulative stride of all preceding layers in the group. j_l maps one output-space row back to j_l rows in the group's input space: each subsampling operation (e.g. a pooling layer with stride $s_l > 1$) reduces spatial resolution, so that a single row at the output of that layer corresponds to s_l rows at its input. Multiplying the kernel spread $(k_l - 1)$ by j_l therefore expresses the boundary reach of each layer in terms of rows of the group's original input. Each of the $K - 1$ worker borders therefore involves an exchange of $r \times W \times C$ elements per direction. Because pooling reduces spatial resolution by its stride factor, a layer boundary immediately after a pooling layer carries a smaller activation D_l and a lower transfer cost. *Fully-connected layers* operate on the full flattened feature vector and are also splittable in the same row-wise manner as spatial layers. The activation size D_l varies substantially across layer boundaries: early convolutional layers produce large feature maps, while pooling layers yield considerably smaller ones. These differences directly shape the communication overhead of partitioning at a given layer boundary and inform the agent's layer-assignment decisions.

Algorithm 1: Split Inference

Input: a_t, $G^{(t)}$, $\{\ell_w^{(t)}, \rho_w\}_{w \in V}$, $\mathbf{x}$
Output: $\hat{y}$

// distribute $\alpha_w^{(t)}$-fractions of $\mathbf{x}$ to $\mathcal{W}_1^{(t)}$
for $w \in \mathcal{W}_1^{(t)}$ **do**
  $\mathbf{x}_w^{(1)} \leftarrow \textsc{Slice}(\mathbf{x}, \alpha_w^{(t)}, r_1)$
  SendTo$(h^{(t)}, w, \mathbf{x}_w^{(1)})$
for $g \in \{1, \ldots, G^{(t)}\}$ **do**
  for $w \in \mathcal{W}_g^{(t)}$ **do**
    for $u \in \mathcal{W}_{g-1,w}^{(t)}$ **do**
      ReceiveFrom$(w, u, \mathbf{x}_w^{(g)})$
  if $\mathcal{L}_g \subseteq \mathcal{L}^S$ **then**
    // parallel local execution
    for $w \in \mathcal{W}_g^{(t)}$ **in parallel do**
      $T_{w,g}^{\mathrm{comp}} \leftarrow \alpha_w^{(t)} \sum_{l \in \mathcal{L}_g} T_{lw}$
      $D_w^{(g)} \leftarrow w.\textsc{Exec}(\mathcal{L}_g, \mathbf{x}_w^{(g)})$
    // parallel $\mathcal{W}_g^{(t)} \to \mathcal{W}_{g+1}^{(t)}$ transfer
    if $g < G^{(t)}$ **then**
      for $w \in \mathcal{W}_{g+1}^{(t)}$ **do**
        for $u \in \mathcal{W}_{g,w}^{(t)}$ **do**
          SendTo$(u, w, D_{uw}^{(g)})$
  else
    // FC: gather then execute on full tensor
    $D^{(g)} \leftarrow \textsc{Gather}(\{D_w^{(g-1)} : w \in \mathcal{W}_{g-1}^{(t)}\})$
    $D^{(g)} \leftarrow \mathrm{node}_g.\textsc{Exec}(\mathcal{L}_g, D^{(g)})$
$\hat{y} \leftarrow D^{(G^{(t)})}$
return $\hat{y}$

D. Split Inference Execution Model

A *stage* g is a contiguous subsequence of layers $\mathcal{L}_g \subseteq \mathcal{L}$ in which each worker $w \in \mathcal{W}_g^{(t)} \subseteq V$ independently executes all layers in $\mathcal{L}_g$ on its local spatial slice, with all workers in $\mathcal{W}_g^{(t)}$ running in parallel across the mesh. A centralized partitioning

and allocation agent splits the L layers into $G^{(t)}$ contiguous vertical stages and distributes them across the devices. The vertical assignment $\sigma_{\mathrm{v}}^{(t)} = (L_1^{(t)}, \ldots, L_{G^{(t)}}^{(t)}) \in \mathbb{Z}_{>0}^{G^{(t)}}$ gives the layer count of each stage, with $\sum_{g=1}^{G^{(t)}} L_g^{(t)} = L$. The horizontal assignment $\sigma_{\mathrm{h}}^{(t)} = (K_1^{(t)}, \ldots, K_{G^{(t)}}^{(t)}) \in \{1, \ldots, N\}^{G^{(t)}} = [N]^{G^{(t)}}$ gives the number of workers assigned to each stage. The worker sets $\mathcal{W}^{(t)} = \{\mathcal{W}_g^{(t)}\}_{g=1}^{G^{(t)}}$ assign specific devices to each stage. Each worker $w \in \mathcal{W}_g^{(t)}$ receives a workload fraction $\alpha_w^{(t)} = \rho_w \,/\, \sum_{w' \in \mathcal{W}_g^{(t)}} \rho_{w'}$, proportional to its compute capability relative to its stage peers. The hub node $h^{(t)}$ serves as the pipeline entry and exit point. The inference result $\hat{y}$ is produced at the worker in $\mathcal{W}_{G^{(t)}}^{(t)}$ executing the final stage and returned to the hub $h^{(t)}$. Together they form the split configuration $a_t = (G^{(t)}, \sigma_{\mathrm{v}}^{(t)}, \sigma_{\mathrm{h}}^{(t)}, h^{(t)}, \mathcal{W}^{(t)})$, recomputed at every inference step from the current channel and compute state.

The controller assigns a set of workers $\mathcal{W}_g^{(t)} \subseteq V$ to execute each stage g in parallel. The hub $h^{(t)}$, selected by a centralized controller as part of each configuration, may participate as a regular worker in any stage. We assume a centralized controller observes the full network state and computes the complete split configuration a_t, which is broadcast to all participating nodes before inference begins and ultimately yields the inference output $\hat{y}$, as illustrated in Figure 1.

Inference proceeds as a sequential pipeline over $G^{(t)}$ stages, as detailed in Algorithm 1. We define $\textsc{Slice}(X, \alpha_w^{(t)}, r)$ as the $\alpha_w^{(t)}$-fraction of the rows of tensor X assigned to worker w, extended by r boundary overlap rows on each interior border. The hub $h^{(t)}$ computes $x_w^{(1)} = \textsc{Slice}(x, \alpha_w^{(t)}, r_1)$ for each $w \in \mathcal{W}_1^{(t)}$ and transmits it (Lines 1 – 3). The volume for worker w is $(\alpha_w^{(t)} H + n_w r_1) \times W \times C$ elements, where $n_w \in \{0, 1, 2\}$ is the number of neighboring workers in the partition and r_1 is given by (1) for $\mathcal{L}_1$. At each stage g, every worker $w \in \mathcal{W}_g^{(t)}$ receives its input slice from $\mathcal{W}_{g-1}^{(t)}$, with $\mathcal{W}_0^{(t)} \triangleq \{h^{(t)}\}$ (Lines 4 – 7). If $\mathcal{L}_g \subseteq \mathcal{L}^S$, all workers execute in parallel on their local slices (Lines 8 – 11). Since r_g covers the cumulative receptive field of all layers in $\mathcal{L}_g$ (including convolutional layers as a canonical example), no intermediate activations are exchanged within the stage. The time for worker w to complete its share of stage g is $T_{w,g}^{\mathrm{comp}} = \alpha_w^{(t)} \sum_{l \in \mathcal{L}_g} T_{lw}$. When $\ell_w^{(t)} = 0$ for all workers, $\alpha_w^{(t)}$-proportional row allocation equalizes completion times by construction. Under background load, effective capacities $\rho_w^{(t)}$ differ across workers and the worker with the lowest remaining capacity becomes the stage bottleneck. Let us define the set $\mathcal{W}_{g-1,w}^{(t)}$ of *predecessor splits* for worker w in stage g as the set of workers u that send activation $D_{uw}^{(g-1)}$ to w. At each stage boundary $g < G^{(t)}$, every worker $w \in \mathcal{W}_{g+1}^{(t)}$ receives $D_{uw}^{(g)}$ from the set of predecessor splits $\mathcal{W}_{g,w}$ along the shortest path in G_t, where a direct link is a path of length one and multi-hop routes are used otherwise (Lines 12 – 15). When $\mathcal{L}_g \not\subseteq \mathcal{L}^S$, a single node gathers all partial activations and executes the stage on the full tensor (Lines 17 – 18). All sender-receiver transfers at a stage boundary are modelled as running in parallel, therefore the stage-boundary latency is the maximum over all pairwise path latencies.

E. Latency Model

Each worker $w \in \mathcal{W}_g^{(t)}$ executes its assigned fraction $\alpha_w^{(t)}$ of the input activation tensor, so its computation time scales with its assigned fraction and its effective compute capacity $\delta_w^{\mathrm{C}} = \alpha_w^{(t)} \sum_{l \in \mathcal{L}_g} T_{lw}$.

We assume the hub is a special worker associated with fictitious Stages 0 and $G+1$ with the only task to generate the input, partition it, forward it to Stage 1 workers, and receive the final activations from stage G. In particular, the hub is the only predecessor of any worker in the first stage, i.e., $\mathcal{W}_{0,w}^{(t)} = \{h^{(t)}\}, \forall w \in \mathcal{W}_1^{(t)}$, and all Stage-$G$ workers are predecessors of the hub, belonging to the imaginary Stage $G+1$, i.e., $\mathcal{W}_{G+1,h^{(t)}}^{(t)} = \mathcal{W}_G^{(t)}$. Any device selected as hub at stages 0 and $G+1$ does not perform computation, while the same device can perform computation if selected at any other stage. Let $p(i,j)$ denote a path from i to j in G_t as a sequence of edges and let $\mathcal{P}(i,j)$ be the set of all such paths in G_t. The *latency transfer time* for data volume D routed from i to j is defined as the minimum transmission delay $\delta_{ij}^{\mathrm{T}} = \min_{p \in \mathcal{P}(i,j)} \sum_{(u,v) \in p} \frac{D}{B_{uv}^{(t)}}$, where a direct link $(i,j) \in E_t$ corresponds to a single edge path. This formulation can be extended to generalized path cost from hop count and assumes store-and-forward routing.

We now define the delay from the inference start, when partial inference is complete at worker w in stage g as the sum of the compute delay of worker w to execute the workload share of stage g plus the longest delay needed to receive all the required activation to proceed with computation from all predecessor splits in the previous stage $g-1$, defined recursively $\forall g \in \{1, \ldots, G+1\}$ as

$$\Delta_{g,w} = \delta_w^{\mathrm{C}} + \max_{u \in \mathcal{W}_{g-1,w}^{(t)}} \{\Delta_{g-1,u} + \delta_{uw}(D_{uw}^{(g-1)}\}$$

, where $D_{uw}^{(g-1)}$ is the volume of activation rows that sender worker u in stage $g-1$ must forward to receiver worker w in stage g. The hub does not compute during first input partitioning and transmission to Stage-1 workers, therefore $\Delta_{0,h^{(t)}} = 0$.

F. Problem Formulation

We assume that the centralized controller can assign as low as one layer per stage, i.e., total number of stages is L. The network controller's task is to optimize number of stages, the number of layers and worker per stage, the hub placement, and the DNN partitioning that induces a compute Directed Acyclic Graph determined by the predecessor sets, which minimizes the end-to-end delay $\Delta_{G^{(t)}+1,h^{(t)}}$ as in

$$\underset{G^{(t)},\, \sigma_{\mathrm{v}}^{(t)},\, \sigma_{\mathrm{h}}^{(t)},\, h^{(t)},\, \mathcal{W}^{(t)}}{\text{minimize}} \quad \Delta = \Delta_{G^{(t)}+1,h^{(t)}} \tag{P1}$$

Since the optimization problem P1 is NP-hard [8], exact methods are intractable at IoT scale, motivating the use of a RL agent to learn a near-optimal policy directly from experience.

The optimization problem P1 contains, in addition to the workload sizing problem shown to be NP-hard in [8], a topology aware worker assignment over a general multi-hop mesh. DISNET's hardness proofs target workload sizing under an adjacency constraint and do not directly establish hardness of this placement component. We therefore give a direct reduction that isolates the hardness of the worker assignment alone.

Proposition 1. *The optimization problem P1 is NP-hard.*

Proof Sketch. We prove NP-hardness by reduction from the quadratic assignment problem (QAP) , which is known to be NP-hard [17]. An instance of QAP is given by n facilities, n locations, a flow matrix f_{ij}, and a distance matrix d_{kl}, and asks for a bijection π minimizing $\sum_{i,j} f_{ij}\, d_{\pi(i)\pi(j)}$. Given a QAP instance, we construct an instance of our problem by fixing the topology size $G^{(t)} = n$, the vertical split degree $\sigma_v^{(t)} = 1$, the horizontal split degree $\sigma_h^{(t)} = 1$, and the hub node $h^{(t)}$, itself already fixed rather than optimized in the general problem, so that the worker assignment $\mathcal{W}^{(t)} = \pi$ is the only remaining free variable. We set the data dependency between stages i and j equal to f_{ij}. We set the transmission delay between devices k and l equal to d_{kl}. Under this construction, the restricted problem's objective becomes $\min_{\mathcal{W}^{(t)}} \sum_{i,j} f_{ij}\, \delta^{\mathrm{T}}_{\pi(i)\pi(j)} = \min_{\pi} \sum_{i,j} f_{ij}\, d_{\pi(i)\pi(j)}$, which is the QAP objective. An optimal solution to our problem's restricted instance yields an optimal solution to the QAP instance, and the construction takes polynomial time. Since QAP is NP-hard, the restricted problem is NP-hard. The optimization problem P1 additionally optimizes over $G^{(t)}, \sigma_v^{(t)}, \sigma_h^{(t)}$ and therefore contains this restriction as a special case, so it is at least as hard, that is, NP-hard. □

IV. Methodology

A. Split Inference

GReMS[1] assigns L layers to G contiguous stages (layer sets $\mathcal{L}_g$, determined by $\sigma_{\mathrm{v}}^{(t)}$) and splits feature-map rows across each stage's workers via row-fractions $\alpha_k^{(g)}$ (determined by $\sigma_{\mathrm{h}}^{(t)}$), as illustrated in Figure 3. Hub h slices the input x and transmits each fraction $x_w^{(1)}$ to its assigned Stage-1 workers. Each stage executes in parallel across its workers and transfers activations to the next stage. Figure 4 shows a concrete execution on an 8-node network topology with G stages. Algorithm 1 formalizes this procedure.

B. Reinforcement Learning Framework

The combinatorial nature of jointly selecting hub, stage partition, and worker sets makes exhaustive search intractable even for modest networks, and a fixed heuristic cannot adapt to topologies and device counts unseen at inference time.

[1]Implementation at: https://github.com/ediamondlabs/grems

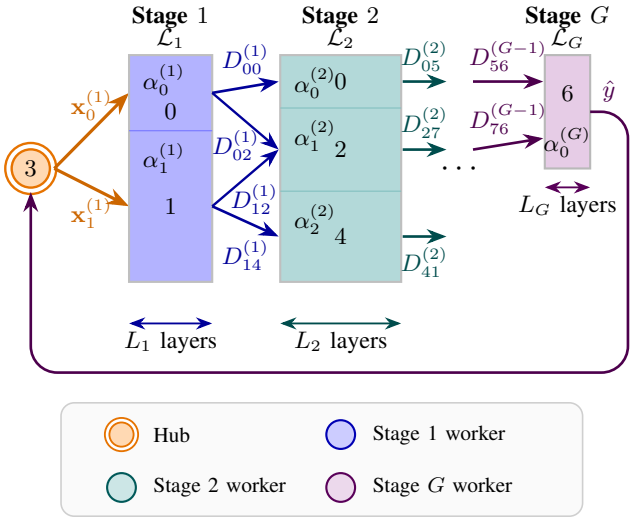


Fig. 3. GReMS DNN Logical Partitioning model, showing split-layer activation dependency. Each split block in each stage contains the index of the worker that executes that split, and its height encodes $\alpha_k^{(g)}$. Stage block width is proportional to layer count L_g and block height to activation-map size. Arrows show logical flow of activations between model splits. Intermediate stages from 3 to $G-1$ are omitted for brevity.

At deployment (Algorithm 2), two processes run concurrently. Incoming inference requests are handled continuously by Algorithm 1. In parallel, the policy controller observes the network state (Line 2), encodes it together with the previous action a_{t-1} (Line 3), selects the split configuration deterministically via argmax (Line 4), and dispatches the worker assignment (Line 5).

During training (Algorithm 3), the policy is trained across topologies drawn from a distribution $p(\mathcal{G})$, promoting generalization to unseen deployments. The agent operates in rollouts of S steps, collecting transitions into a buffer $\mathcal{B}$, with action memory a_0 and reward memory r_0 initialized to zero. At each step t, the system state is observed (Line 5), a split configuration a_t is selected stochastically for exploration and the value baseline V_t is estimated by the critic (Line 7), the configuration is dispatched, the environment returns reward r_t and next state s_{t+1}, and the transition $(s_t, a_{t-1}, a_t, r_t, V_t)$ is added to $\mathcal{B}$ (Line 10). After S steps, discounted returns R_t and per step advantage estimates $\hat{A}_t = R_t - V_t$ are computed (Line 11). For K update epochs, a minibatch $\mathcal{M} \subseteq \mathcal{B}$ is drawn, fresh encodings are produced to discard stale rollout representations (Line 14). The policy gradient and value losses $\mathcal{L}_\pi$ and $\mathcal{L}_V$ are computed over $\mathcal{M}$ (Line 15 – 16), and both actor and critic weights are updated via gradient descent (Line 17 – 18).

C. State space

The state s_t, shown in the state bar of Figure 5, is produced by the following six observation processes, plus the previous action a_{t-1}. **Network monitor** $s_{\mathrm{glb}}^{(t)}$ is a fixed-size vector that summarizes the whole network. Let $d_w^{(t)} = |\mathcal{N}^{(t)}(w)|$ denote the degree of node w at step t. $s_{\mathrm{glb}}^{(t)}$ contains per-node values $(\rho_w^{(t)}, \ell_w^{(t)}, d_w^{(t)})$, per-link bandwidth $B_{ij}^{(t)}$, and three

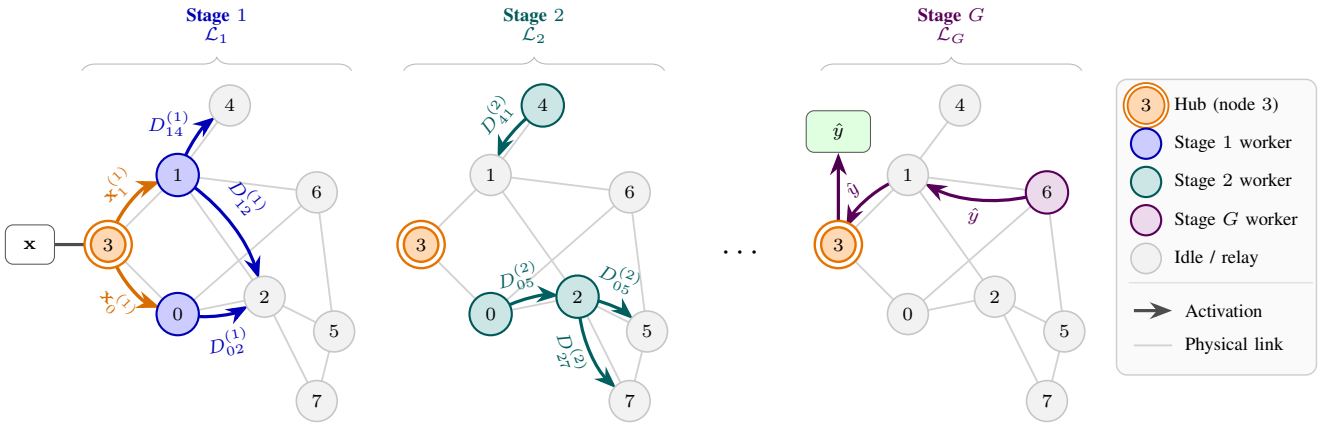


Fig. 4. Physical execution of split inference on an 8-node network across G stages. One snapshot per active stage highlights active workers (colored vertices) and activation transfers over physical links. **Stage 1:** Hub 3 sends input partitions $x_0^{(1)}$ to worker 0 and $x_1^{(1)}$ to worker 1 over direct links. Workers 0 and 1 forward boundary activations to worker 2 ($D_{02}^{(1)}$, $D_{12}^{(1)}$); worker 1 also sends a slice to worker 4 ($D_{14}^{(1)}$), which participates as a Stage 2 worker. **Stage 2:** Workers 0, 2, and 4 forward activations to next-stage workers 1, 5, and 7 over direct and multi-hop paths. Intermediate stages from 3 to $G-1$ omitted for brevity. **Stage G:** After receiving activations from Stage $G-1$, worker 6 produces $\hat{y}$ and relays it to hub 3 via worker 1.

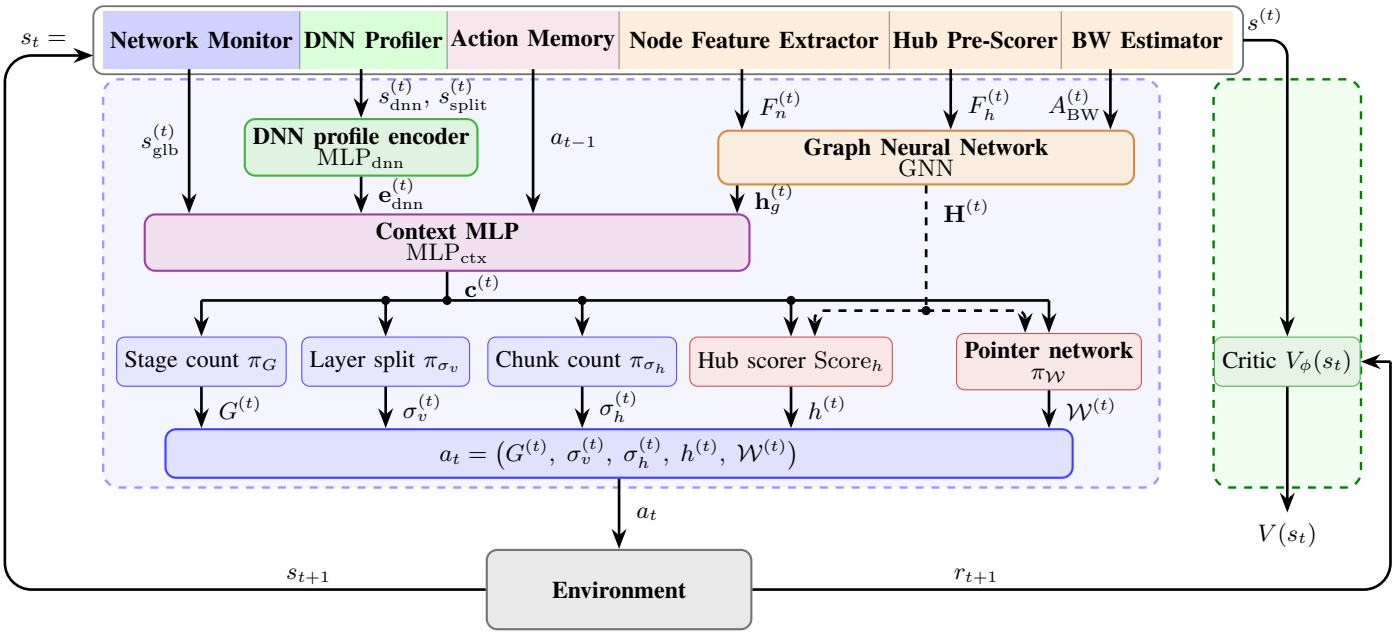


Fig. 5. GNN Actor-Critic architecture of GReMS. The blue dashed box encloses the actor π_θ; the green dashed box encloses the critic V_ϕ. Global actor heads (blue) depend on $\mathbf{c}^{(t)}$ alone. Node-conditioned heads (red) additionally receive $\mathbf{H}^{(t)}$. The critic V_ϕ (green) receives the full state $s^{(t)}$ and reward r_{t+1}, and outputs $V(s_t)$ used only during training.

topology descriptors: clustering coefficient $C_{\text{clust}}^{(t)}$, algebraic connectivity $\lambda_2^{(t)}$, and diameter $\Gamma^{(t)}$. **DNN profiler** $s_{\text{dnn}}^{(t)}$, $s_{\text{split}}^{(t)}$ records at each candidate cut point the cumulative FLOPs and activation size, together with per-layer compute cost T_l and data volume D_l. **Node feature extractor** $F_n^{(t)} \in \mathbb{R}^{N\times 3}$ arranges the three per-node values $(\rho_w^{(t)}, \ell_w^{(t)}, d_w^{(t)})$ as a matrix with one row per node. **BW estimator** $A_{\text{BW}}^{(t)} \in \mathbb{R}^{N\times N}$ is a matrix whose entry (w,u) is the bottleneck capacity along the path $\mathcal{P}(w,u)$ from w to u, $\bar{B}_{wu}^{(t)} = \min_{(i,j)\in\mathcal{P}(w,u)} B_{ij}^{(t)}$. **Hub pre-scorer** $F_h^{(t)} \in \mathbb{R}^{N\times 2}$ adds two scores per node that capture how well suited a node is to act as a hub. The first is a binary flag $c_w^{(t)} = \mathbf{1}[\exists\,\text{shortest path through } w]$ that indicates whether node w lies on any shortest path at step t. The second is a normalised score $\phi_w^{(t)} = \frac{\rho_w^{(t)}\cdot \bar{B}_w^{\text{agg},(t)}}{\max_{w'}\left(\rho_{w'}^{(t)}\cdot \bar{B}_{w'}^{\text{agg},(t)}\right)}$, where $\bar{B}_w^{\text{agg},(t)} = \frac{1}{N-1}\sum_{u\neq w} \bar{B}_{wu}^{(t)}$ is the mean bottleneck bandwidth from w to all other nodes. This ranks nodes by the product of their effective compute and their mean bottleneck bandwidth. Stacking $F_n^{(t)}$ and $F_h^{(t)}$ gives $X_n^{(t)} = [F_n^{(t)} \,\|\, F_h^{(t)}] \in \mathbb{R}^{N\times 5}$.

D. Reward and Action Space

Upon committing a_t, the environment evaluates the resulting latency Δ and returns $r_t = -\kappa \cdot \log(\Delta)$, where κ is a scaling factor. The logarithmic transformation prevents large latency values from dominating the gradient signal, stabilizing training across topology sizes. The agent's action is the split configuration a_t.

Algorithm 2: GReMS Execution Phase

Input: Trained actor π_θ (weights θ)
Output: Model splitting and allocation a_t at every step

1 **for** $t \in \mathbb{N}$ **do**
2 $s_t \leftarrow$ `ObserveSystem()`
3 $(\mathbf{c}^{(t)}, \mathbf{H}^{(t)}) \leftarrow$ `EncodeState`(s_t, a_{t-1})
// arg max replaces $\sim$ Categorical at each head
4 $a_t \leftarrow$ `DecodeConfig`$(\mathbf{c}^{(t)}, \mathbf{H}^{(t)})$
// communicate split configuration to workers
5 `Dispatch`(a_t)

6 **Function** `EncodeState`(s_t, a_{t-1})**:**
// $X_n^{(t)}$ convolved with capacity-weighted adjacency $A_{\mathrm{BW}}^{(t)}$
7 $(\mathbf{H}^{(t)}, \mathbf{h}_g^{(t)}) \leftarrow \mathrm{GNN}(X_n^{(t)}, A_{\mathrm{BW}}^{(t)})$
8 $\mathbf{e}_{\mathrm{dnn}}^{(t)} \leftarrow \mathrm{MLP}_{\mathrm{dnn}}(s_{\mathrm{dnn}}^{(t)}, s_{\mathrm{split}}^{(t)})$
// fuse per-node values, mean-pooled graph embedding, DNN profile, and action memory into $\mathbf{c}^{(t)}$
9 $\mathbf{c}^{(t)} \leftarrow \mathrm{MLP}_{\mathrm{ctx}}([\, s_{\mathrm{glb}}^{(t)}, \mathbf{h}_g^{(t)}, \mathbf{e}_{\mathrm{dnn}}^{(t)}, a_{t-1}\,])$
10 **return** $(\mathbf{c}^{(t)}, \mathbf{H}^{(t)})$

11 **Function** `DecodeConfig`$(\mathbf{c}, \mathbf{H})$**:**
12 $G^{(t)} \sim \mathrm{Categorical}(\mathrm{softmax}(\mathrm{Lin}_G(\mathbf{c})))$
// PropRound enforces $\sum_g L_g^{(t)} = L$
13 $\sigma_v^{(t)} \leftarrow \mathrm{PropRound}(\mathrm{softmax}(\mathrm{Lin}_v(\mathbf{c})), G^{(t)})$
14 $\sigma_h^{(t)} \sim \mathrm{Categorical}(\mathrm{softmax}(\mathrm{Lin}_h(\mathbf{c})))$
// linear score over $\mathbf{H}^{(t)}$ selects hub $h^{(t)}$
15 $h^{(t)} \sim \mathrm{Categorical}(\mathrm{softmax}(\mathbf{w}^\top \mathbf{H} + b))$
16 $\mathcal{W}_g^{(t)} \leftarrow \emptyset \; \forall g$
// one worker per slot, $|\mathcal{W}_g^{(t)}|$ slots per stage from $\sigma_h^{(t)}$
17 **for** $g = 1$ **to** $G^{(t)}$, $k = 1$ **to** $|\mathcal{W}_g^{(t)}|$ **do**
// $\mathbf{r}_{g,k}$: learned slot embedding
18 $\mathbf{q}_{g,k}^{(t)} \leftarrow \mathbf{r}_{g,k} + \mathrm{Lin}(\mathbf{c})$
19 $w_{g,k}^{(t)} \sim \mathrm{Categorical}(\mathrm{softmax}((\mathbf{q}_{g,k}^{(t)})^\top \mathbf{H}^\top))$, $\mathcal{W}_g^{(t)} \leftarrow \mathcal{W}_g^{(t)} \cup \{w_{g,k}^{(t)}\}$
// layer sets follow from $\sigma_v^{(t)}$
20 $\mathcal{L}_g^{(t)} \leftarrow \{l \in \mathcal{L} : \sum_{g'<g} L_{g'}^{(t)} < l \le \sum_{g' \le g} L_{g'}^{(t)}\} \; \forall g$
21 **return** $(G^{(t)}, \sigma_v^{(t)}, \sigma_h^{(t)}, h^{(t)}, \{\mathcal{W}_g^{(t)}\}_{g=1}^{G^{(t)}})$

E. Policy Architecture and Split Configuration Selection

The actor π_θ, comprising the GNN encoder, context MLP, and all action heads, forms the learnable component of the Actor-Critic system (Figure 5). All actor modules are jointly parameterized by θ. The critic V_ϕ, parameterized by weights ϕ, estimates the expected return as $V_\phi^{(t)} = \mathrm{MLP}_{\mathrm{critic}}(\mathbf{c}^{(t)}, r_{t-1})$. The actor maps s_t to a_t through the EncodeState and DecodeConfig functions of Algorithm 2.

The central design choice is a conditioning structure at two levels. A GNN operating on node features $X_n^{(t)}$ with $A_{\mathrm{BW}}^{(t)}$ as capacity weighted edge weights produces per node embeddings $\mathbf{H}^{(t)} \in \mathbb{R}^{N \times d_h}$, capturing individual node capabilities and local link bottlenecks, and a mean pooled graph embedding $\mathbf{h}_g^{(t)} \in \mathbb{R}^{d_h}$ summarizing the whole topology. The context MLP then fuses these with the DNN profile, the per-node values $s_{\mathrm{glb}}^{(t)}$ and the previous action a_{t-1} into a shared context vector $\mathbf{c}^{(t)} \in \mathbb{R}^{d_c}$, giving every head a globally consistent view so that all decisions are jointly conditioned and mutually compatible.

Decisions that depend only on global topology statistics are taken from $\mathbf{c}^{(t)}$ alone: π_G, $\pi_{\sigma_{\mathrm{v}}}$, and $\pi_{\sigma_{\mathrm{h}}}$ produce stage count $G^{(t)}$, layer partition $\sigma_{\mathrm{v}}^{(t)}$, and chunk count $\sigma_{\mathrm{h}}^{(t)}$ (Lines 12 to 14), with $\sigma_{\mathrm{v}}^{(t)}$ rounded by PropRound to enforce the hard constraint $\sum_g L_g^{(t)} = L$. Decisions that must identify specific nodes are additionally conditioned on $\mathbf{H}^{(t)}$: Score_h selects the hub and $\pi_{\mathcal{W}}$ assigns workers stage by stage via attention over node embeddings, which allows the same trained policy to handle any network size N at execution time (Lines 15 to 19). Layer sets $\{\mathcal{L}_g^{(t)}\}$ follow deterministically from $\sigma_{\mathrm{v}}^{(t)}$ (Line 20). The design of the decoding procedure guarantees feasibility of the resulting split configuration by construction. Therefore, the controller need not check whether the split configuration is feasible after generation, given current measurements of system resources and topology.

V. Experimental Setup

A. Network and Channel Model

Each episode samples a connected RGG instance with static link capacity B_{ij} drawn log uniformly from $[5\,\mathrm{Mbit/s},\ 200\,\mathrm{Mbit/s}]$ per edge and node compute capacity drawn uniformly from $\{1, \ldots, 10\}$. Link capacity evolves according to the GE model and node compute availability according to the MMPP model. All parameters are listed in Table I.

B. Training and Evaluation Protocol

Training proceeds in the two consecutive phases using PPO [18] with Generalized Advantage Estimation (GAE), i.e., pre-training (Phase 1), where 50 RGG instances are generated per node count $N \in \{4, \ldots, 10\}$, and a fine tuning (Phase 2), where training continues for 10^7 additional steps on $N \in \{4, \ldots, 20\}$, with graphs of $N \in \{11, \ldots, 20\}$ assigned twice the sampling weight. Training uses a single fixed seed for both phases. The seed sets the initial weights of the actor-critic network and also drives the random draws used during training, which pick the topology instance sampled at each rollout step and generate the G-E link-bandwidth and MMPP compute-load trajectories for that topology. For evaluation we generate 10 RGG test topologies per node count $N \in \{4, \ldots, 20\}$ that are held out from the 50 training instances used in Phase 1. Within a trajectory the sampled RGG topology itself is held fixed, only the link capacity and node compute availability evolve according to the G-E

Algorithm 3: GReMS Training Phase

Input: Steps per rollout S, update epochs K, number of rollouts $\mathcal{R}$, discount factor γ, topology distribution $p(\mathcal{G})$, learning rate η, initial weights θ, ϕ
Output: Trained actor π_θ (weights θ), critic V_ϕ (weights ϕ)

for $n \in \{1, \dots, \mathcal{R}\}$ **do**
  // init buffer $\mathcal{B}$, action memory a_0, reward memory r_0
  $\mathcal{B} \leftarrow \emptyset, \quad a_0 \leftarrow \mathbf{0}, \quad r_0 \leftarrow 0$
  for $t = 1$ **to** S **do**
    $s_t \leftarrow$ `ObserveSystem()`,
    $(\mathbf{c}^{(t)}, \mathbf{H}^{(t)}) \leftarrow$ `EncodeState`(s_t, a_{t-1})
    // sample action stochastically for exploration
    $a_t \leftarrow$ `DecodeConfig`$(\mathbf{c}^{(t)}, \mathbf{H}^{(t)})$,
    $V_t \leftarrow \mathrm{MLP}_{\mathrm{critic}}(\mathbf{c}^{(t)}, r_{t-1})$
    // measure end-to-end latency
    $(\Delta, s_{t+1}) \leftarrow$ `Dispatch`(a_t),
    $r_t \leftarrow -\kappa \log \Delta$
    // add to training buffer
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(s_t, a_{t-1}, a_t, r_t, V_t)\}$
  $(\{\hat{A}_t\}, \{R_t\}) \leftarrow$ `ComputeAdvantages`$(\mathcal{B}, \gamma)$
  // K epochs of minibatch actor/critic updates
  for $k \in \{1, \dots, K\}$ **do**
    // re-encode: rollout activations stale after gradient step
    $(\mathbf{c}^{(t)}, \mathbf{H}^{(t)}) \leftarrow$ `EncodeState`(s_t, a_{t-1}),
    $V_t \leftarrow \mathrm{MLP}_{\mathrm{critic}}(\mathbf{c}^{(t)}, r_{t-1})\ \forall \mathcal{M} \subseteq \mathcal{B}$
    // actor loss $\mathcal{L}_\pi$, critic loss $\mathcal{L}_V$ on $\mathcal{M}$
    $\mathcal{L}_\pi \leftarrow \mathbb{E}_{\mathcal{M}}[-\log \pi_\theta(a_t \mid s_t) \cdot \hat{A}_t]$,
    $\mathcal{L}_V \leftarrow \mathbb{E}_{\mathcal{M}}[(V_t - R_t)^2]$
    // update actor
    $\theta \leftarrow \theta - \eta \nabla_\theta (\mathcal{L}_\pi + \mathcal{L}_V)$
    // update critic
    $\phi \leftarrow \phi - \eta \nabla_\phi \mathcal{L}_V$
return θ, ϕ

and MMPP models. Each evaluation instance is run over $S = 10$ independent trajectory sequences of $T_{\mathrm{eval}} = 50$ steps. GReMS is evaluated on each test topology under 5 independent draws of the stochastic MMPP and G-E trajectory. Key hyperparameters are listed in Table I.

C. Baselines and Metrics

DiSNet [8] solves a continuous relaxation of the joint partitioning and allocation problem at each inference request, recomputing it at every step. It ranks the hub's neighbors by their compute capability and takes a fixed-size subset of the top performers then discretizes the resulting partition heights and widths to a feasible split. Even though DiSNet is polynomial in the number of devices and layers for each inference request, it is executed from scratch at every step using only the currently measured link throughput and compute capability, and it does not leverage the history of how these evolve over time.

MoDNN [7] distributes DNN layers evenly across all N devices as a fixed partition and does not adapt to runtime channel or load conditions.

EnSplit [9] is adapted from its original 6G setting to the IoT environment using the same latency and capacity model as DiSNet and MoDNN. At each inference step, EnSplit extracts a four-node chain from the shortest path between source and destination in the IoT graph, then exhaustively evaluates all valid contiguous layer-to-node assignments and selects the minimum-latency split, without exploiting nodes off the shortest path or applying horizontal splitting. Like DiSNet, this search is repeated from scratch at every step using only the current channel state, with no learned generalization across steps or topologies. DiSNet and MoDNN apply fixed decision rules with no learned parameters and therefore require no tuning.

TABLE I
EXPERIMENT PARAMETERS

Parameter	Value
Splittable DNN	VGG-16 [19]
GNN architecture	GraphSAGE [20]
Discount γ, GAE λ	0.99, 0.95
PPO clip, Entropy coef. ε	0.20, 0.02
Mini-batch size $\lvert\mathcal{M}\rvert$	512
Episode length	10
Learning rate η (Phase 1, 2)	$3 \cdot 10^{-4}$, $3 \cdot 10^{-5}$
Training steps (Phase 1, 2)	$3 \cdot 10^{7}$, 10^{7}
Num. of devices N (Phase 1, 2)	$\{4, \dots, 10\}$, $\{4, \dots, 20\}$
p_{LH}, p_{HL}, q_{HL}, q_{LH}	0.05, 0.20, 0.15, 0.40
$\beta^H_{\min}$, $\beta^H_{\max}$, $\beta^L_{\min}$, $\beta^L_{\max}$	0.80, 1.00, 0.15, 0.25
$\ell^L_{\min}$, $\ell^L_{\max}$, $\ell^H_{\min}$, $\ell^H_{\max}$	0.10, 0.35, 0.55, 0.80
B_{ij}[Mbit/s], ρ_w	$\log$-$\mathcal{U}(5, 200)$, $\mathcal{U}\{1, \dots, 10\}$

We report the empirical *mean inference latency* over all evaluation sequences and time steps $\bar{\Delta} = \frac{1}{S \cdot T_{\mathrm{eval}}} \sum_{s=1}^{S} \sum_{t=1}^{T_{\mathrm{eval}}} \Delta^{(s,t)}$, where $\Delta^{(s,t)}$ is the latency at step t of sequence s, and the *relative latency reduction* $S_b = \frac{\bar{\Delta}_{\mathrm{GReMS}} - \bar{\Delta}_b}{\bar{\Delta}_b}$ to quantify GReMS's advantage over each baseline b.

VI. RESULTS

A. Inference Latency

Figure 6 shows mean inference latency $\bar{\Delta}$ versus network size N on RGG topologies. The dominant trend is scale-invariance: GReMS stays flat near 2.5 s across $N \in \{4, \dots, 20\}$, varying by less than 10%. MoDNN falls steeply from roughly 32 s at $N = 4$ to 14 s by $N = 8$, then levels off, suggesting its static assignment wastes less capacity as the network grows but cannot catch GReMS. DiSNet similarly declines from 10 s to 7 s before plateauing, maintaining a gap of roughly 2.8× relative to GReMS despite recomputing the optimal configuration at every step. EnSplit is the closest baseline but trends mildly upward from 4.2 s at $N = 4$ to 5.9 s at $N = 20$, as its fixed chain topology cannot exploit the additional routing diversity that GReMS's GNN policy leverages to keep latency constant.

B. Relative Latency Reduction

Figure 7 shows the relative latency reduction $\mathcal{S}_b$ of GReMS against each baseline across $N \in \{4, \dots, 20\}$. Against MoDNN, the reduction remains the largest throughout.

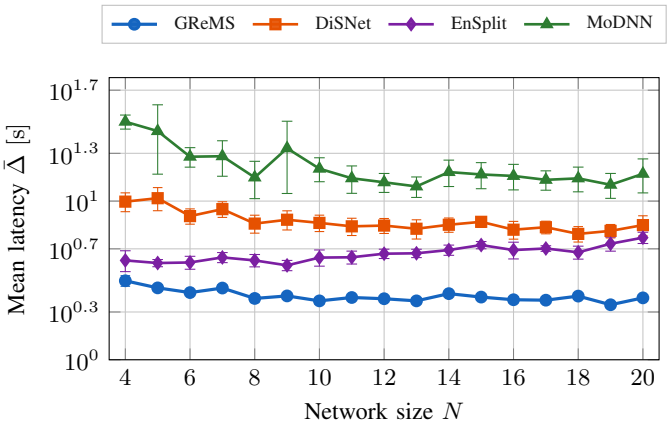


Fig. 6. Mean inference latency $\bar{\Delta}$ against network size N on RGG topologies (log-scale y-axis). Error bars denote 95% confidence intervals: 5 seeds $\times$ 10 networks for GReMS, 10 networks for baselines.

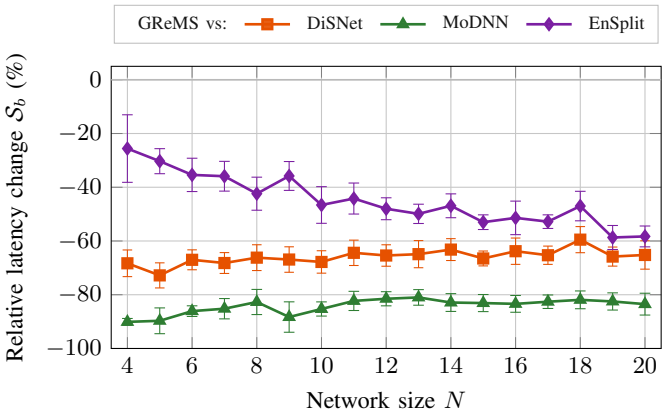


Fig. 7. Relative latency reduction $\mathcal{S}_b$ of GReMS against each baseline on RGG topologies. Error bars show propagated 95% confidence intervals over 5 seeds $\times$ 10 networks (GReMS) and 10 networks (baselines).

MoDNN always splits every one of the spatial layers across the same fixed set of neighbors of a hub node that is given in advance rather than chosen, and runs the three FC layers on that hub alone. Its split ratio is set purely by each neighbor's compute weight, and every worker is timed against a single transfer rate obtained by averaging all neighbor link bandwidths together, so a weak link is never singled out or avoided, it only drags the shared average down for the whole group. GReMS, instead, learns to route around exactly such links and to choose where the hub and the FC layers sit, which explains why the gap stays large and only narrows slightly as denser graphs give MoDNN a larger neighbor set to average and split work over. Against DiSNet, the reduction stays large and comparatively stable, declining only mildly. DiSNet reoptimizes only the hub and the horizontal split ratio at each step, while its vertical partition is fixed to four layer groups set in advance, regardless of topology or channel state. Since this rigidity does not depend on N, the resulting shortfall relative to GReMS's jointly learned action stays flat. Against EnSplit, the gap grows steadily with N. EnSplit restricts every decision to a chain of four nodes independently of the graph size. As N grows, an increasing share of the network's nodes, links, and routing diversity falls outside what EnSplit can ever use, while GReMS's policy, conditioned on its GNN, scales its plan to the full topology.

C. Temporal Adaptation Under Dynamic Conditions

Figure 8 shows per-step mean latency over 50-step sequences under jointly varying G-E and MMPP conditions. Every curve remains level from step zero with no warm-up or accumulating drift, confirming that channel dynamics does not destabilize any policy within a 50-step episode.

During the Pre-training phase ($N \in \{4, \ldots, 10\}$), GReMS achieves the lowest latency, followed by EnSplit, DiSNet, and MoDNN. MoDNN's fixed partition exposes latency directly to every channel and load fluctuation without any policy correction. In the Fine-tuning phase ($N \in \{11, \ldots, 20\}$), GReMS shifts only marginally, demonstrating stable generalization to

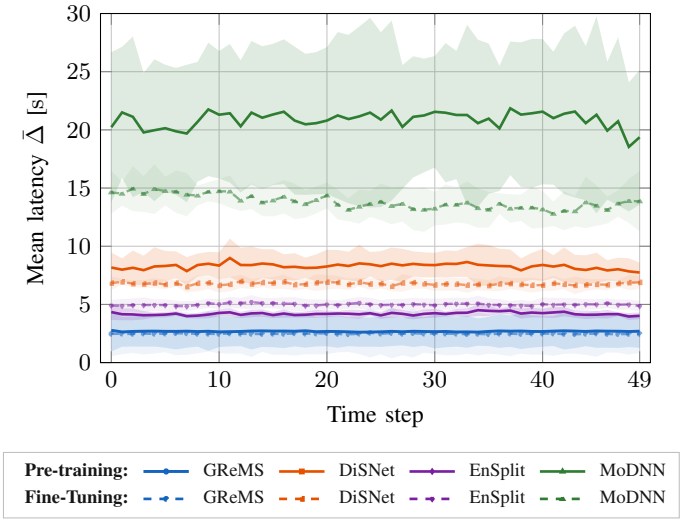


Fig. 8. Per-step mean inference latency $\bar{\Delta}$ over 50-step sequences on RGG networks under dynamic G-E and MMPP conditions. Shaded bands show ± 1 standard deviation across topology scales. Solid curves and darker bands: pre-training phase ($N \in \{4, \ldots, 10\}$, 3500/700 repetitions for GReMS/baselines). Dashed curves and lighter bands: fine-tuning phase ($N \in \{11, \ldots, 20\}$, 5000/1000 repetitions).

larger graphs. DiSNet drops to approximately 6.8 s as its per-step optimization benefits from the additional routing options, while EnSplit rises slightly to approximately 5.0 s as the fixed chain topology becomes a tighter bottleneck at larger scale. MoDNN incurs the largest decrement because distributing a fixed partition across more nodes reduces per-node load even without adaptive routing. Despite this improvement, GReMS maintains margins of 2.7$\times$ over DiSNet, 2.0$\times$ over EnSplit, and 5.6$\times$ over MoDNN, and the gap to EnSplit widens relative to the pre-training phase.

D. Sensitivity to Link Bandwidth

Figure 9 shows mean inference latency as a function of uniform link bandwidth B on $N = 10$ RGG topologies. Methods with static or sequential partitions are far more bandwidth-sensitive than GReMS, and GReMS leads at every

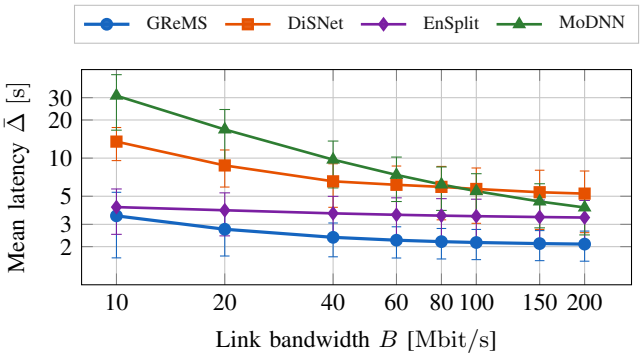


Fig. 9. Mean inference latency $\bar{\Delta}$ as a function of uniform link bandwidth B on $N = 10$ RGG topologies (both axes log-scale). Error bars show ± 1 standard deviation across 5000 step level observations.

tested level. MoDNN exhibits the steepest drop, falling from $31.1\,\mathrm{s}$ at $10\,\mathrm{Mbit/s}$ to $4.1\,\mathrm{s}$ at $200\,\mathrm{Mbit/s}$, a $7.6\times$ reduction driven by large intermediate tensor transfers across each hop. GReMS also falls steeply, from $3.5\,\mathrm{s}$ to $2.1\,\mathrm{s}$, but saturates above roughly $100\,\mathrm{Mbit/s}$, at which point compute time rather than transmission becomes the binding constraint. DiSNet follows a similar downward trajectory from $13.5\,\mathrm{s}$ to $5.2\,\mathrm{s}$, and its gap to GReMS narrows as bandwidth grows, from roughly $3.9\times$ at $10\,\mathrm{Mbit/s}$ to roughly $2.5\times$ at $200\,\mathrm{Mbit/s}$. EnSplit is the least sensitive because its fixed chain transmits fewer and smaller partitions per hop. Nevertheless, GReMS outperforms EnSplit at every bandwidth level, with a margin that grows from roughly 15% at $10\,\mathrm{Mbit/s}$ to roughly 38% at $200\,\mathrm{Mbit/s}$ as its flexible routing increasingly exploits the available capacity.

VII. Conclusion

We presented GReMS, a stateless GNN+PPO agent for adaptive split inference on dynamic IoT mesh networks. By casting split-point selection and layer partitioning as a joint per-step graph-level decision, GReMS decouples inference latency from network size and topology. Its message-passing architecture aggregates per-node capacity and link quality into embeddings that transfer across graph sizes and topology families without architectural changes. On RGG networks under G-E and MMPP dynamics, GReMS achieves average speedups of $3.0\times$, $6.6\times$, and $1.9\times$ over DiSNet, MoDNN, and EnSplit across $N \in \{4, \ldots, 20\}$. These gains hold steady across all 50 steps of each episode, confirming that GReMS continuously tracks the network state rather than committing to a fixed plan.

Acknowledgment

This work has been funded by the SNSF project "Efficient Distributed Intelligent Applications in Mobile-Network Dynamics (eDIAMOND)" with grant number 223443.

References

[1] I. Behnke and H. Austad, "Real-time performance of industrial iot communication technologies: A review," *IEEE Internet of Things Journal*, vol. 11, no. 5, p. 7399–7410, Mar. 2024. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2023.3332507

[2] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," 2018. [Online]. Available: https://arxiv.org/abs/1812.00564

[3] Y. Gao, M. Kim, S. Abuadbba, Y. Kim, C. Thapa, K. Kim, S. A. Camtepe, H. Kim, and S. Nepal, "End-to-end evaluation of federated learning and split learning for internet of things," 2020. [Online]. Available: https://arxiv.org/abs/2003.13376

[4] M. K. Aswad, F. A.-B. Beetro, and A. A. Keshlaf, "Workload balancing model for heterogeneous iot nodes architectures," in *2024 2nd International Conference on Electrical Engineering and Automatic Control (ICEEAC)*, 2024, pp. 1–5.

[5] A. Cilfone, L. Davoli, L. Belli, and G. Ferrari, "Wireless mesh networking: An iot-oriented perspective survey on relevant technologies," *Future Internet*, vol. 11, no. 4, 2019. [Online]. Available: https://www.mdpi.com/1999-5903/11/4/99

[6] A. Sehgal, R. Agrawal, R. Bhardwaj, and K. K. Singh, "Reliability analysis of wireless link for iot applications under shadow-fading conditions," *Procedia Computer Science*, vol. 167, pp. 1515–1523, 2020, international Conference on Computational Intelligence and Data Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920308280

[7] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 1396–1401.

[8] E. Samikwa, A. D. Maio, and T. Braun, "Disnet: Distributed micro-split deep learning in heterogeneous dynamic iot," *IEEE Internet of Things Journal*, vol. 11, no. 4, pp. 6199–6216, 2024.

[9] S. Majumdar, E. Samikwa, K. Samdanis, E. Pateromichelakis, E. H. Nezhad, and T. Braun, "Ensplit: Dynamic drl energy-aware split inference for ai-based ue apps in 6g networks," in *2026 IEEE 12th International Conference on Network Softwarization (NetSoft)*, 2026, pp. 82–89.

[10] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, "Bottlefit: Learning compressed representations in deep neural networks for effective and efficient split computing," in *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, June 2022, p. 337–346. [Online]. Available: http://dx.doi.org/10.1109/WoWMoM54355.2022.00032

[11] Y. Matsubara, M. Mendula, and M. Levorato, "A multi-task supervised compression model for split computing," 2025. [Online]. Available: https://arxiv.org/abs/2501.01420

[12] Y. Matsubara *et al.*, "Sc2 benchmark: Supervised compression for split computing," *arXiv preprint arXiv:2203.08875*, 2023.

[13] V. Ninkovic, D. Vukobratovic, D. Miskovic, and M. Zennaro, "Comsplit: A communication-aware split learning design for heterogeneous iot platforms," *IEEE Internet of Things Journal*, vol. 12, no. 5, pp. 5305–5319, 2025.

[14] X. Feng, R. Jia, C. Luo, V. C. M. Leung, and W. Xu, "Slwf: A split learning without forgetting framework for internet of things," *IEEE Internet of Things Journal*, vol. 12, no. 9, pp. 12 008–12 020, 2025.

[15] N. Akar, M. Bastopcu, S. Ulukus, and T. Başar, "Modeling interfering sources in shared queues for timely computations in edge computing systems," 2024. [Online]. Available: https://arxiv.org/abs/2408.01327

[16] A. H. El Fawal, A. Mansour, and A. Nasser, "Markov-modulated poisson process modeling for machine-to-machine heterogeneous traffic," *Applied Sciences*, vol. 14, no. 18, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/18/8561

[17] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis, "The quadratic assignment problem," in *Handbook of Combinatorial Optimization*, D.-Z. Du and P. M. Pardalos, Eds. Kluwer Academic Publishers, 1998, vol. 2, pp. 241–337.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[20] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018. [Online]. Available: https://arxiv.org/abs/1706.02216