Distributed Training for Deep Learning Models On An Edge Computing Network Using Shielded Reinforcement Learning

Tanmoy Sen and Haiying Shen
Department of Computer Science, University of Virginia
Email: {ts5xm, hs6ms}@virginia.edu

Abstract—With the emergence of edge devices along with their local computation advantage over the cloud, distributed deep learning (DL) training on edge nodes becomes promising. In such a method, the cluster head of a cluster of edge nodes schedules all the DL training jobs from the cluster nodes. Using such a centralized scheduling method, the cluster head knows all the loads of the cluster nodes, which can avoid overloading the cluster nodes, but the head itself may become overloaded. To handle this problem, we first propose a multi-agent RL (MARL) system that enables each edge node to schedule its own jobs using RL. However, without the coordination between the nodes, action collision may occur, in which multiple nodes may schedule tasks to the same node and make it overloaded. To avoid these problems, we propose a system called Shielded Reinforcement learning (RL) based DL training on Edges (SROLE). In SROLE, each edge node schedules its own jobs using multi-agent RL. The shield deployed in a node checks action collisions and provides alternative actions to avoid the collisions. As the central shield node for the entire cluster may become a bottleneck, we further propose a decentralized shielding method, in which different shields are responsible for different regions in the cluster and they coordinate to avoid action collisions on the region boundaries. Our container-based emulation experiments show that SROLE reduces training time by up to 59% with 29% lower median resource utilization and reduces the number of action collisions by up to 48% compared to multi-agent RL and the centralized RL. Our real device experiments show that SROLE still reduces the training time by up to 53% with 28% lower median resource utilization than multi-agent RL and the centralized RL.

I. INTRODUCTION

Edge devices are currently used for various applications in many areas including transportation and healthcare [1–4]. These applications often deploy machine learning (ML) frameworks using data collected by the edge devices’ sensors. ML models have transformed into more complex and larger Deep Neural Networks (DNNs). These DNNs are memory and computationally expensive in training due to their complexity and size. On the contrary, an edge device usually does not have sufficient memory or computation resources to conduct the entire DNN model training job. Thus, a DNN is usually trained (or updated) in the cloud, then compressed and deployed on the edge nodes for inference. Such cloud-based training can generate significant delays when the network is intermittent (e.g., disaster, network congestion), and cannot provide data privacy protection [5] for sensitive applications (e.g., medical records) as the data needs to be transferred to the cloud. In this case, distributing the job of training or updating a DNN model among only edge nodes becomes a promising solution.

Recent works [6, 5, 7] for distributed training on edges handle either data parallelism or model parallelism while involving the cloud at a certain stage. Data parallelism methods [5, 7] deploy replicas of an entire neural network on the edges, and these edges have their subsets of training data. Each edge processes its training data subset and synchronizes model parameters in a parameter server running on an initiator edge or the cloud. Model parallelism methods [6] divide a neural network and distribute the shallow (earlier) layers to edges and the deep layers to the cloud and let edges communicate with cloud for model parameters transfer from the previous layer transfer to the next layer. The data parallelism methods sometimes may not be feasible as deploying a large DNN model on a single edge may not be feasible. In contrast, the model parallelism methods suffer from the long delay of communication and intermittent network between edges and cloud.

A concurrent data and model parallelism based deep learning (DL) system can handle these problems. In such a system [8], clusters of edges are created according to geographical locations, and each cluster trains a replica of the DL model using model parallelism based on its locally collected data. Each cluster has a cluster head that has relatively high capacity, and it assigns the partitions of a DNN model (i.e., tasks) to the cluster edge nodes with a goal of minimizing training time. Each edge within a cluster collects its own data and sends the data to the first-layer node, which collects the sensed data from all cluster edges as the training data of the model replica in the cluster. The cluster head assigns tasks to the edges based on the resource demands of the tasks and the available resources of the edges. Thus, the cluster head needs to continuously observe the workload conditions of all the edge nodes in its cluster. With this cluster-wide knowledge, the cluster head can avoid overloading the edges in assigning the tasks. However, such a centralized scheduling method imposes a significant workload on the cluster head, which ultimately impacts the performance of the training. Recently, RL [9, 10] has also been used for such scheduling in the recent times with similar high load. To lessen this load, we first propose a multi-agent RL method (MARL) that enables each edge node to schedule its own jobs among its neighboring edge nodes (i.e., edge nodes in its transmission range) using RL. In MARL, each
edge device works as an independent agent and makes the scheduling decision among its nearby edges, thus relieving the cluster head from the extra burden. However, without the coordination between the edge nodes, action collision may occur, in which multiple nodes may schedule tasks to the same node and make it overloaded.

To avoid this problem, we propose Shielded Reinforce-
ment learning based DL Training on Edges (SROLE) on top of the MARL-based assignment approach. The shielding approach [11] works as a separate monitor that suggests alternative actions to avoid action collision by observing the states and actions that will be taken by the agents. In SROLE, each edge node schedules its own jobs using MARL, and the shield, which is deployed on the cluster head, checks the action collisions among the schedules of the edges in its cluster. Edge nodes report to the shield their action decisions, and it checks action collisions and provides alternative actions to avoid the collisions. However, the computational cost of a centralized shield grows dramatically with the number of edges in a cluster, and it may become a bottleneck for the entire cluster. Thus, we further propose a decentralized shielding method, in which different shields are responsible for different regions in the cluster and they coordinate to avoid action collisions on the region boundaries. Specifically, a large cluster is divided into multiple sub-clusters according to the geographical proximity, and a shield monitors the edges within each sub-cluster and communicates with its neighboring shields for the edges at the boundary of the sub-clusters to avoid action collisions, i.e., unsafe actions. The computational cost of each shield in the decentralized method is lower than that in the centralized shielding as the centralized shield’s workload is distributed to a number of shields.

In summary, the contributions of this work are as follows:

- To avoid overloading a cluster head due to scheduling DL training jobs in its cluster, we initially propose a multi-agent RL-based method (MARL) that enables each edge node to use RL to schedule its own jobs. For a given DL training job, an edge node makes scheduling decisions for DNN partitions among its nearby edges depending on their resource availability to minimize training time.
- To avoid action collisions in MARL, we use the shielding approach in each cluster. The shield in a cluster collects the scheduling decisions of all edge nodes in the cluster, checks the action collisions and provides alternative actions to avoid the action collisions.
- To avoid overloading a shield in a cluster, we distribute the shielding workload to multiple shields in a cluster and each shield is responsible for a sub-cluster. The neighboring shields communicate with each other to avoid action collisions from the edges on the boundaries of sub-clusters.
- We measured the performance of the SROLE system on container based emulation on Amazon EC2 instances. Our evaluation shows that the SROLE system shows up to 59% reduction in training time and up to 48% reduction in the number of action collisions compared to the centralized RL and MARL approach. Our real device experiments also show up to 53% reduction in training time and up to 46% reduction in the number of action collisions in comparison with the centralized RL and MARL approach.

The rest of the paper is organized as follows. Section II presents the related work. Section IV describes our SROLE system. Section V presents the performance evaluation. Finally, Section VI concludes the paper with remarks on our future work.

II. RELATED WORK

Researchers have been studying federated ML training and DNN partition distribution across cloud/fog and edge devices [6, 7, 12–17]. In federated learning [7], edge devices update a trained model on the cloud. Individual edges download a replica of the model and update the models using their own available datasets. Finally, the updated models from individual edges are aggregated through averaging or using a control theorem on the cloud to produce the final model. Many proposed ML inference approaches partition the ML model distributed edge nodes [18–20]. The works in [18, 20] simply consider each or multiple convolution layers as a partition, and the work [19] vertically divides the convolutional layers in a convolutional neural network (CNN). The work in [6] distributes DNN partition across cloud/fog and edge devices to accelerate training or inference. The resource-constraint edge devices run lighter (earlier) layers of the model and the cloud or fog run heavier (later) layers of the model. Resilinet [21] achieves failure-resilient inference in model-parallel ML at the edge. Data parallelism training methods distribute training data among different edges for training and accumulate training updates. Wang et al. [5] analyzed the convergence rate of replicas for ML models such as Support Vector Machine (SVM) and K-means, and accordingly proposed a control method that dynamically adjusts the frequency of global update from each replica to the ML initiator in real time to minimize the learning loss under a fixed computation resource budget for the edges.

To efficiently run ML models on edge devices in terms of inference time, energy and memory, researchers have introduced techniques for compressing the neural networks (NNs) [22–35]. For example, the works in [36–38] find compressed DNN models by formulating optimization problems that meet resource (memory, energy) constraints or minimize inference time while maximizing accuracy or reaching a specified accuracy. Han et al. [39] proposed cumulative pruning of the network connection, weight quantization, and compression through Huffman coding to decrease the size and inference time of an NN model without significant loss of accuracy. Ashok et al. [26] proposed a reinforcement learning based policy that first removes layers from the DNN and later reduces the size of the remaining layers by deleting links. Yao et al. [40] used a pre-trained compressor-critic network to estimate the link weights and drop out the low-weight links.

However, there are few works on scheduling the partitions of a DNN model in model parallelism on the edge in an efficient manner. This paper addresses this problem.
III. BACKGROUND

In the model parallelism, a layer is a DNN unit such as convolutional or fully-connected layer [41]. In each model level, a model partition consists of one or multiple disjoint layers, which can be executed in parallel. These partitions are assigned to the edge nodes based on their available resources. In this paper, we use concurrent data/model parallelism as an example to explain our proposed methods though they also can be applied to model parallelism. The problem we handle is how to schedule the different model partitions (i.e., tasks) to different edge nodes to minimize the training time. In concurrent data/model parallelism, as shown in Figure 1, each cluster is formed by proximity-close edge nodes. Each cluster trains an entire DNN model replica using the cluster’s collected dataset, and partitions the model among the proximity-close cluster edges. The cluster head is responsible for initiating the training, partitioning and distributing the layers of the model to edge nodes, and synchronizing model parameters to generate the final trained model. In order to conduct task scheduling, the cluster head needs to continuously check the resource availability of all the edge nodes in its cluster and also estimate the resource demands of each partition in scheduling the tasks of a job. The resource availability and resource demands are for multiple resources, mainly including GPU or CPU, memory, and bandwidth.

\[
\begin{align*}
    u_k(d_j) &= \frac{D_k(d_j)}{C_k(d_j)}, \\
    \text{where } D_k(d_j) &\text{ denotes the total resource demand of type-k resource of the tasks running on edge } d_j \text{ and } C_k(d_j) \text{ denotes the capacity of type-k resource of edge } d_j. \\
    \text{The system pre-defines } \alpha \text{ (e.g., 0.95) and if } u_k(d_j) > \alpha \text{ for any type-k resource, the edge node is considered as overloaded. We define an edge node’s combined resource utilization as follows:} \\
    u(d_j) = \prod_k u_k(d_j), \quad k = 1, 2, \ldots 
\end{align*}
\]

It measures the overall resource utilization across different types of resources of an edge node.

The cluster head can use RL for the task scheduling and functions as the agent in the RL. RL has three components: state, action and reward. Given the current state, the agent chooses the action that generates the maximum expected reward and receives reward for the action it takes. In this task scheduling scenario, the state is the resource demand of each layer in the DL model and the resource availability of each edge node in the cluster. The action is the schedule that assigns each partition to an edge node. The reward is defined based on the training time of the DNN model; shorter training time leads to higher reward and vice versa. Thus, using the trained RL, the cluster head observes the state and makes the scheduling decision for each DL training job. However, all of these operations create significant overhead on the cluster head. In this paper, we aim to distribute the overhead among the cluster edge nodes while decreasing training time increase due to this decentralized operation.

IV. SYSTEM DESIGN OF SROLE

A. Overview

We propose SROLE that consists of the following components.

- **Multi-agent RL (Section IV-B).** To distribute the job scheduling overhead on the cluster head among the cluster edge nodes, we propose a multi-agent RL-based method (MARL). In MARL, each edge node uses RL to schedule the tasks of its own DL training job without relying on the cluster head.

- **Centralized Shielding for MARL (Section IV-C).** Edge nodes share their neighbors since there are overlaps in the transmission ranges of neighboring edge nodes. Then, edge nodes may schedule their tasks to the same edge node since they do not know the decisions of other edge nodes, which may overload the task assigned node. Thus, we propose a centralized shielding method for MARL, which checks the action collisions and provides alternative actions to avoid the action collisions in a cluster.

- **Decentralized Shielding for MARL (Section IV-D).** As the centralized shield is deployed in one edge node, which may overload it, we propose a decentralized shielding method for MARL. In this method, multiple shields are deployed to the sub-clusters in one cluster and each shield is responsible for its own sub-cluster. Further, the neighboring shields communicate with each other to avoid the action collisions on the boundaries.

B. Multi-Agent RL-based Job Scheduling

The MARL method is similar to the above RL-based method, except each edge node is an agent and the state includes the resource availability of an edge node’s nearby nodes rather than all the edge nodes in the cluster. The RL is initially pre-trained and distributed to each edge node. As a result, each edge node uses the RL to schedule the tasks of its own DL training jobs and keeps training the RL model. In this method, each edge takes the optimal action of assignment of the partitions based on
its observed state. In particular, each edge makes its own local decision on where each layer should be assigned. Based on the decision made by each edge node, the state of the environment changes as the available resources change for the edge node where a layer is assigned. Consequently, based on the local decision taken by one edge or agent, the global decision of all agents influences the overall state. Finally, each edge node has its own long-term reward to optimize, which now becomes a function of the policies of all other agents that are updated based on the global decision. Figure 2 illustrates the working procedure of the multi-agent RL. Each edge node observes the state of the environment and then takes its own action to assign partitions to itself and its neighbors, and then it receives reward. The joint action of the actions of all agents are denoted by $a^* = a_1^* \cup a_2^* \cup \ldots \cup a_n^*$. After the actions are taken, the state $s$ and reward $r$ at the next timestep $t+1$ (i.e., $s_{t+1}$ and $r_{t+1}$) become the state and reward at this timestep $t$ (as indicated with arrows) for the agents to make decisions again.

Now, we explain the corresponding state ($S$), action ($A$) and reward ($R$) for our proposed MARL model by each agent.

**State space.** The state space ($s_t \in S$) consists of the resource demands of all the layers of a DNN model, and the available resources of all of a node’s nearby edge nodes. For each layer, the state includes the CPU resource demand, memory demand, and its data transfer size to each layer in the next level in the DNN model. Besides, the state also includes the utilization of the CPU, memory and bandwidth resource for each device. The state of edges includes the available CPU and memory of each edge, and available bandwidth across each pair of edges at each time $t$. As the continuous values of these resource characteristics result in infinite size of the state space, we discretize the continuous space by dividing their value range into a number (e.g., three) of equal-width ranges: low, medium and high.

We varied the structural parameters of a particular DNN layer structure within reasonable ranges as indicated in [42] and profile the CPU and memory usage in the forward and the backward pass. We use the TensorFlow benchmark tool [43] to profile the usage of all DL components on an edge node. The available resources on an edge device keeps changing accordingly to the layers assigned to the device.

**Action space.** We use $a_t$ to denote $a^*_t$ for simplicity. The action space represented by $a_t = \{a^*_t\} \in A$, ($i = 1, 2, ..., |M|$, $j = 1, 2, ..., |E|$) defines the schedule for all the layers at time $t$, where $M$ denotes the set of the layers in the DNN model, $E$ denotes the set of all nearby edges and $|\cdot|$ means the size of a set. Each element of the action space $A$ defines which edge should be assigned with a certain layer. Action $a^*_t$ is defined as follows for each pair of layer $l_i$ and edge device $d_j$.

$$a_{t}^{i,j} = \begin{cases} 1, & \text{if layer } l_i \text{ is placed in edge } d_j \text{ at } t \\ 0, & \text{otherwise} \end{cases}$$

After $a_t$ is determined, the available resources of edges are updated with the available resources at $t + 1$, and then $a_{t+1}$ is determined, and so on until the last layer is assigned.

**Reward.** Let $a_t$ be the action taken (schedule is made) at time $t$, then the reward function is given by:

$$r_t(s_t, a_t) = \begin{cases} -\gamma, & \text{if memory is violated} \\ \rho, & \text{otherwise} \end{cases}$$

where $\rho$ is a coefficient to control the reward, $\gamma$ is a large constant reward to ensure that a schedule violating the memory limit requirement is not valid. Furthermore, and $O$ denotes the training time of the DNN model. After the job assignment, the states change for the next assignment and the reward is updated for all agents.

### C. Centralized Shielding for MARL

The MARL method cannot ensure that none of the edge nodes get overloaded since different edge nodes may assign tasks to the same node simultaneously based on its original available resources. To handle this problem, we propose a shielding approach on top of the multi-agent RL scheme. Each cluster has a shield deployed in the cluster head that has high resource capacity. It ensures that none of the edges get overloaded by the task assignment from all edge nodes in the cluster. In the centralized shielding method, the shield enforces the safety specification (i.e., avoiding decisions that overload an edge before being sent to the environment) during the RL learning process. After an edge node makes a scheduling decision for its job, it reports its decision to the shield in its cluster. The shield collects the decisions of all edge nodes in its cluster and checks action collisions, i.e., the actions that will make an edge node overloaded by hosting the tasks from multiple edge nodes, and then provides alternative actions to avoid the action collisions.

We hope that the shield restricts MARL agents as less as possible via the minimal interference criteria. These criteria are as follows: (1) the shield only corrects joint action $a^*_t$ if it violates the safety specification, and (2) the shield seeks a safe joint action $\tilde{a}^*_t$ that changes as a few of the agents’ actions as possible in $a^*_t$. Figure 3 shows the working procedure of the centralized shielding built on top of the multi-agent RL scheme. The centralized shield observes the joint action and current state before the action is implemented. If it leads to an unsafe action, i.e., overloading of any edge, the shield suggests a safe action that will be implemented. At the same time, the shield also notifies the edges within the cluster of the safe action and assigns a constant negative reward ($\kappa$).
for their originally decided action that leads to the overload of one device. Accordingly, the reward is redefined as below:

\[
    r_t(s_t, a_t) = \begin{cases} 
        -\gamma, & \text{if memory is violated} \\
        -\kappa, & \text{suggested by the shield} \\
        \frac{\omega}{\sqrt{\sigma}}, & \text{otherwise}
    \end{cases}
\]

In the meantime, the environment changes based on the suggested safe action by the shield and the states and rewards are updated accordingly.

The shield makes a judgement about the potential violations of safety specifications. In details, the shield observes whether the joint action actually changes the resource utilization of any type of resource of an edge to a value higher than the threshold, i.e., \( u_k(d_j) > \alpha \). If this condition is true (criterion (1)), there exists an action collision and the shield will choose alternative safe actions to replace the original actions in the joint action to make the edge node (say \( d_j \)) not overloaded. For each layer \( l_i \) that is assigned to edge node \( d_j \), we define its resource demand weight as follows:

\[
    \omega(l_i) = \prod_k b_k(l_i)/C_k(d_j), \quad k = 1, 2, \ldots
\]

where \( b_k(l_i) \) is the resource demand of type-\( k \) resource of layer \( l_i \) and \( C_k(d_j) \) is the capacity of type-\( k \) resource of edge device \( d_j \). While choosing the safe action, the shield first ranks the layers that are planned to be assigned to the edge node based on their resource demand weights. Then, it picks up the layer with the highest weight and finds a new host edge for it that will not be overloaded after hosting this layer. The purpose of choosing the layer of the highest weight to be rescheduled is to reduce the interference to the original joint action (criterion (2)). The shield repeats this process until the remaining layers will not overload the edge. Specifically, it searches for nearby edge nodes with high resource availability from edge node \( d_j \), and then checks whether any of these edges can host this layer after it accepts other layers that are planned to assign to it in other actions. To quickly find such an edge node, the shield calculates the combined resource utilizations of the nearby edge nodes after they accept other planned layers assigned to them. Next, it orders the nearby edge nodes in the ascending order of their combined resource utilizations and then sequentially picks up the top node to check until it finds such an edge node. The edge node on the top generally has a high available resources and hence is more likely to be able to host the layer. After finding the new host edge, the shield creates an alternative action that assigns the layer to this edge device. As we limit our safe action from the nearby edges of the original edge node in the decided original action and ensure this newly suggested action won’t overload the edge, this new action will not deviate from the previous optimal action greatly.

Algorithm 1 illustrates the pseudocode for the centralized shielding. At each timestep \( t \) (Line 1), the central shield observes the joint action and joint state of all the agents or edges in the cluster (Lines 2-3). For each edge node \( d_j \), it calculates its resource utilization of each resource type (Line 5), ranks the assigned layers on \( d_j \) in descending order of resource demand weight (Line 6), and initializes \( \kappa \) (Line 7). It then checks whether its resource utilization is greater than the pre-defined threshold \( \alpha \) (Line 8). If yes, the shield picks up the layer on the list top, finds and suggests a safe action for scheduling the layer and notifies the layer scheduling edge about the new action and the \( \kappa \) reward for the unsafe action (Lines 9-13). The shield repeats this process until the edge node will not be overloaded after it hosts all layers assigned to it (Lines 8-14).

### D. Decentralized Shielding for MARL

A shield in a cluster is responsible for all edge nodes in a cluster and may become overloaded due to the communication and computation overhead in shielding. Thus, we propose a decentralized shielding method. In the decentralized shielding method, we first divide a cluster to multiple sub-clusters and each sub-cluster consists of geographically proximity-close edge nodes. Then, one shield works for one sub-cluster. Within each sub-cluster, the shield works in the similar way as described in the centralized shielding. There is one additional problem we need to handle. The edge nodes in the boundary of two or more sub-clusters may assign tasks to the same edge node in one sub-cluster, which may overload it, but the shield in this sub-cluster will not receive the actions from its neighboring sub-clusters and hence will not detect the action collision. To solve this problem, the shields of neighboring sub-clusters need to communicate with each other to avoid such a case. Specifically, the neighboring shields first select a delegate to check the action collisions. Then, they send the actions of the edge nodes and the available resources as well as the resource utilizations of the edge nodes in the boundary to the delegate. The delegate uses the same method in the centralized shielding method to check action collision and finds alternative actions. It sends the alternative actions to the neighboring shields, and the shields then forward the alternative actions to the corresponding edge nodes. As
a result, the edge nodes take alternative actions instead of previously determined actions.

V. PERFORMANCE EVALUATION

A. Experiment Setup

Execution. Our proposed system runs on Tensorflow for the execution of the model training through its parameter server strategy. In order to emulate edges with varying resources, we use 25 docker containers. Each cluster has 5 edge nodes. The resource settings of our emulation and real device experiments are indicated in Table I and the resources of the devices were assigned in a round-robin way. The containers are deployed in Amazon EC2 instance of type m5ad.4xlarge. The CPU and memory are configured using commands from docker and the bandwidth between different containers is configured using the tconfig tool [44].

Real experiments. Our real testbed consists of 10 Raspberry Pis; two Pis have 1 GB memory, four other Pis have 2 GB memory and four other Pi has 4 GB memory. They roughly have same CPU but we use the cpulimit [45] command to control the CPU as desired according to Table I. The edges are connected via 2.4 GHz band wireless connection. We use wondershaper [46] tool to control bandwidth among the edges.

ML models and datasets. We run three ML models: GoogleNet Inception, VGG-16, and RNN [47]. We use the MNIST [48] dataset to run the first two models and the Air Quality dataset [49] for the RNN model. The MNIST dataset consists of 70,000 images of handwritten digits and is widely used for training CNN models. The Air Quality dataset contains 9358 instances of hourly averaged responses from an array of 5 metal oxide chemical sensors. One instance refers to the sensor values (as the ML inputs) and the AQI value (as the ML output). Since there are maximally 5 clusters, We divide the dataset to 5 subsets. Each cluster has a subset as the input training data and the data is randomly distributed among the edges in the cluster as their sensed data. We run three DL training jobs of the same type in each cluster initiated by randomly chosen edge nodes.

RL Training. To train the RL models, we need data related to both DNN models and edge nodes. To generate the data related different DNN model structures and we profiled and obtained their resource demands. To generate edge node configuration data, we consider the number of edge nodes in the range of [2,10]. For each edge node, CPU is chosen randomly from range [0.5,2] GHz, memory is randomly chosen from range [64,4096] MB [50] and the bandwidth across pair of edge nodes is randomly chosen from range [128,1000] MBps [51]. Using these data, we train the RL model offline.

Workload and Settings. In all the cases, we trained one DNN model in each cluster and add several other non-ML jobs (PageRank [52]) from the HiBench benchmark to vary available resources on the edges. The workloads were controlled by running multiple PageRank job on these edges in a distributed way. We run x=2,3,...,6 PageRank jobs in each cluster throughout the whole training period to control the workload. Workload of 100% means there are 6 PageRank jobs running simultaneously in the system, and other workloads are defined similarly in the decreasing order, i.e., 5 is 90%, 4 is 80%, and so on. These jobs were run simultaneously with DNN training jobs until the run completes. During these experiments, we either change the number of edge devices or the workload along the x-axis. Unless otherwise indicated, the number of edge nodes is 25 and the workload is 100%. Each run for DNN model experiment was executed for 50 iterations. We repeated each experiment 5 times and plotted the median with the 5th and 95th percentile error bars. During the experiment, we measured the resource utilization of the devices every 10 minutes. In both the cases of emulation and real device, we set the value of the parameters \(\alpha = 0.9, \rho = 1\), \(\gamma = 50\) and \(\kappa = -100\).

B. Compared Methods

MARL. This is a simple multi-agent RL-based method CQ-learning [53] without shielding. In particular, both the evolution of the system state and the reward received by each agent are influenced by the joint actions of all agents. That is, each agent has its own long-term reward to optimize, which now becomes a function of the policies of all other agents.

SROLE-C. This is a multi-agent RL method with an extra centralized shield. In the centralized shielding, there is a single shield to monitor all agents’ joint actions and correct any unsafe action if necessary. That is, the shield observes all the agents and prevents any edge node from being overloaded.

SROLE-D. This is an extension of the centralized shielding. It has multiple shields on multiple sub-clusters in a cluster and each shield is only responsible for the agents in its sub-cluster or a subset of agents in its cluster.

Centralized RL. In the following figures, we use RL for simplicity. This is an RL scheme where the cluster head makes the assignment decision for all the jobs in its cluster. In this method, we assign a negative reward for overloading the memory of a certain device and otherwise, the reward is based on the job completion time.

C. Metrics

Job completion time. This is the training time, which denotes the time period from the time when a job starts to run after scheduling to the time when the training of the whole model completes. This time period contains multiple number of iterations depending on the size of the dataset. In our case, the training for all the models comprises of 50 iterations.

The number of tasks per device. From different runs, we measure the number of partitions for a DNN training

| Environment   | Resource ranges                                                |
|---------------|---------------------------------------------------------------|
| Real edge     | Mem\in\{1024, 2048, 4096\}MB, CPU\in\{0.25, 0.5, 1.0\}Host Ratio |
|               | BW\in\{20, 100\}MBps                                        |
| Container     | Mem\in\{768, 1024, 1536, 2048, 4096\}MB, CPU\in\{0.3, 1.0\}Host Ratio |
|               | BW\in\{50, 100, 200, 500, 1000\}MBps                        |
Figures 4a, 4b, and 4c show the job completion time. D. Experimental Results from Emulation κ)
for unsafe actions.

The number of action collisions. We measure the number of action collisions for the negative reward (κ) for unsafe actions.

D. Experimental Results from Emulation

Job completion time. Figures 4a, 4b, and 4c show the job completion time versus the number of edges for training the VGG-16, GoogleNet, and RNN models, respectively. MARL and RL have similar job completion times, which indicates that the performances of their job schedules are similar. The results imply that MARL still can achieve comparable performance as RL though MARL does not have global information for job scheduling. Both centralized and decentralized shielding methods (SROLE-C and SROLE-D) perform better than RL and MARL without shielding because shielding reduces the number of unsafe actions, i.e., overloading individual devices. As a result, it reduces the job completion time. For VGG-16, GoogleNet, and RNN, SROLE-D shows 36-45%, 35-43%, and 33-44% reduction in job completion time than MARL or RL without shielding. SROLE-D performs 8-13% less than SROLE-C for all three models because the action collisions are checked by multiple shields instead of one, adding extra communication time among the neighboring shields during the DNN training. As a result, SROLE-C saves job completion time by 49-56% for VGG-16, 48-59% for GoogleNet, 47-56% for RNN, respectively in comparison with RL and MARL without shielding. Thus, the shielding can be conducted more efficiently because of observing the resource state of all the edges together. From all the figures, we see that as the number of edges increases, the job completion time increases. This happens because more clusters lead to more time in transferring the model parameters from the clusters to the parameter server for synchronization of the model. Also, the figures show the results do not vary greatly and keep relatively stable.

The number of tasks per edge node. Figures 5a, 5b, and 5c show the number of tasks per node versus different workloads for training the VGG-16, GoogleNet, and RNN models, respectively in the scenario with 25 edges. We plot the median (denoted with different colors) along with the minimum and the maximum number of tasks (denoted with black bars). SROLE-D shows 42-56%, 46-61%, and 41-56% reduction in the median number of assigned tasks per device for VGG-16, GoogleNet, and RNN compared to MARL or RL without shielding. However, the SROLE-C outperforms the SROLE-D by 2-11%. As a result, SROLE-C generates a reduction in the median number of assigned tasks per device by 49-56% for VGG-16, 48-59% for GoogleNet, 47-56% for RNN respectively in comparison with RL and MARL without shielding. From the figure, we also observe that both the SROLE-D and SROLE-C methods show less variance than both MARL and RL without shielding. Both SROLE-C and SROLE-D perform better than other methods because shielding reduces the number of unsafe actions, i.e., overloading on individual devices, and thus distributes the tasks among devices in a more balanced manner. However, SROLE-D performs less than SROLE-C shielding because of taking a higher number of unsafe actions. This phenomenon happens because of not knowing the cluster more completely.

Resource utilization. Figures 6a, 6b, and 6c show the resource utilization of each type of resources for training the VGG-16, GoogleNet, and RNN models, respectively in the scenario of total 25 edges. We plot the median along with the minimum and maximum resource utilizations as error bars. For VGG-16,
GoogleNet, and RNN, SROLE-D shows 12-19%, 11-17%, and 11-15% reduction in the median resource utilization compared to MARL or RL without shielding. However, the SROLE-C outperforms the SROLE-D by 2-14%. As a result, SROLE-C generates a reduction in the median resource utilization by 21-24% % for VGG-16, 48-59% for GoogleNet, 22-29% for RNN, respectively in comparison with RL and MARL without shielding. From the figure, we also observe that both the SROLE-D and SROLE-C show less variance than both the MARL and RL without shielding in terms of resource utilization. Both centralized and decentralized shielding methods perform better than other methods because shielding avoids overloading individual devices. However, SROLE-D performs less than SROLE-C because of taking a higher number of unsafe actions. This happens due to not having the complete knowledge of the cluster, which adds extra computation or communication for the involvement of multiple shields. 

**Average computation time overhead.** Figures 7a, 7b, and 7c show the computation overhead for scheduling (blue bar) and shielding (orange bar) of different methods while training the VGG-16, GoogleNet, and RNN models, respectively. For all the models, the results for computation overhead (scheduling + shielding) are as follow: MARL < SROLE-D < SROLE-C < RL. RL needs the longest decision making time because only one node is responsible for scheduling all jobs in one cluster. MARL greatly reduces the decision making time of RL since...
MARL distributes the scheduling load among the edge nodes in the cluster by letting each edge node schedule its own job among its neighbors. Both RL and MARL do not have any shielding time as they do not have the shielding approach. SROLE-C and SROLE-D are based on MARL, and they have additional shielding components to detect action collisions, thus generating higher decision making time. MARL, SROLE-C and SROLE-D have the same scheduling time since they all use MARL for scheduling. SROLE-D generates 5-8% less shielding time than SROLE-C for all the models. This is because SROLE-C relies on one shield in each cluster, so the shield needs to check all the actions, which needs a long time, and SROLE-D distributes the shielding overload among multiple shields, which expedites the shielding process and reduces both the shielding time and the decision making time.

The number of action collisions. Figure 8 shows how the assigned reward of unsafe action impacts the number of unsafe actions during the training of all the DNN models. SROLE-C performs 31-48% better than the MARL or RL, while SROLE-D performs 27-39% better than MARL or RL for all the three models. This is because the added shield(s) in SROLE-C and SROLE-D coordinate the edges to avoid unsafe actions, but MARL and RL do not have shields. The SROLE-C approach performs 4-7% better than SROLE-D. This is because the global shield in SROLE-C can observe the global environment, compare all actions and suggest alternative actions accordingly to avoid unsafe actions globally. When multiple shields are responsible for sub-clusters in SROLE-D, the information collected by a shield for the boundary nodes may not cover all the unsafe actions, leading to unsafe actions. Though SROLE-C and SROLE-D use the shielding approach, they still produce certain unsafe actions. This is because the resource demands of tasks are time-varying and dynamic and sometimes cannot be accurately predicted, thus leading to the edge node overload.

For all the three models, as the absolute value of the reward of an unsafe action increases, the number of unsafe actions during the whole training period in SROLE-C and SROLE-D decreases and this number in MARL and RL keeps constant. MARL and RL do not use this reward or shielding approach, so their performances are not affected by the reward value. A high penalty for the action collision will help SROLE-C and SROLE-D avoid more unsafe actions.

E. Experimental Results from a Real Device Network

We formed the 10 edge nodes into a network and considered it as a single cluster, and then ran real experiments on the real-device network. From the real experiments, we observe similar performances of the different methods as in the container-based emulation due to the same reasons mentioned above.

Job completion time. Figure 9 shows the job completion time for training all models in the real-device cluster. For these models, SROLE-D performs 32-39% better than MARL or RL without shielding and SROLE-C performs 36-53% better than MARL or RL. SROLE-D performs 4-7% worse than SROLE-C because SROLE-D has additional communication operations between neighboring shields for shielding.

The number of tasks per edge. Figure 10 shows the number of tasks per device for different models from a real-device network. For all the three models, SROLE-D avoid more unsafe actions.

Resource utilization. Figure 11 shows the resource utilization for different models from a real-device network.
SROLE-D shows 18-23% reduction and SROLE-C shows 21-28% reduction in the median in the median of resource utilization. SROLE-D performs 3-5% worse than SROLE-C because the shields in SROLE-D do not have global knowledge of the state of the resources in all edges. Similar to the emulation experiments, the variances of SROLE-D and SROLE-C are lower than those of MARL and RL without shielding for all models.

**Average computation time overhead.** Figure 12 shows the computation overhead for scheduling and shielding of different methods while training all the models. For the shielding time, SROLE-D performs 4-7% better than SROLE-C for real-devices.

**The number of action collisions.** Figure 13 shows how the assigned reward of unsafe action impacts the number of unsafe actions during training the DNN models. SROLE-D shows 27-42% reduction and SROLE-C shows 29-46% reduction in the median in the median of resource utilization. SROLE-D performs 2-6% worse than SROLE-C because the shields in SROLE-D do not have global knowledge of the state of the resources in all edges.

### VI. CONCLUSION

Fully distributed DNN training on edges utilizing concurrent model and data parallelism is a promising way to increase the scalability of DNN model training at resource-constrained edges. However, relying on one edge node to use RL to schedule the model partitions (i.e., distributing the partitions of a large DL model to a set of edges to minimize the training time) among edge nodes is not scalable. In this paper, we propose a multi-agent RL system that enables each edge node to schedule its own jobs. To ensure such distributed job scheduling method will not overload an edge node, we propose using shielding that observes the actions decided by all edge nodes to avoid overloading edge nodes. Relying on one shield is not scalable. Thus, we further propose a decentralized shielding method that relies on multiple shields to conduct shielding in a distributed manner. Our experiments show that our shielding method performs 59% better than multi-agent RL in training time with 29% less median resource utilization of an edge device, and also the multi-agent RL method achieves similar job completion time performance as the centralized RL method. In the future, we will explore using formal method approaches to provide a guarantee of action collision avoidance and explore a method to avoid action collisions caused by decentralized shielding.

### ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants NSF-1827674, CCF-1822965, FHWA grant 693J31950016, Microsoft Research Faculty Fellowship 8300751, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation and workforce development. For more information about CCI, visit cyberinitiative.org. We also thank Ms. Ingy ElSayed-Aly and Dr. Lu Feng for helping us with our initial understanding of RL shielding.

### REFERENCES

[1] G. Boateng, V. G. Motti, V. Mishra, J. A. Batsis, J. Hester, and D. Kotz, “Experience: Design, development and evaluation of a wearable device for mhealth applications,” in Proc. of MobiCom, 2019.

[2] Y. Huang, X. Ma, X. Fan, J. Liu, and W. Gong, “When deep learning meets edge computing,” in Proc. of ICNP, 2017.

[3] J. Wang, Z. Guo, S. Liu, and Y. Xia, “Poster: Maintaining training efficiency and accuracy for edge-assisted online federated learning with abs,” in Proc. of ICNP, 2020.

[4] A. Curtis, A. Pai, J. Cao, N. Moukaddam, and A. Sabharwal, “Healthsense: Software-defined mobile-based clinical trials,” in Proc. of Mobicom, 2019.

[5] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, “When edge meets learning: Adaptive control for resource-constrained distributed machine learning,” in Proc. of INFOCOM, 2018.
"FastGNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in Proc. of NIPS, 2018, pp. 9017–9028.

28. X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in Proc. of CVPR, 2018.

29. S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher, “FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices,” in Proc. of Sensys. ACM, 2018, pp. 278–291.

30. A. Polino, R. Pascaru, and D. Alistarh, “Model compression via distillation and quantization,” in Proc. of ICLR, 2018.

31. A. Malinin, B. Mlodelzeniec, and M. Gales, “Ensemble distribution distillation,” in Proc. of ICLR, 2020.

32. C. Li, J. Peng, L. Yuan, G. Wang, X. Liang, L. Lin, and X. Chang, “Block-wisely supervised neural architecture search with knowledge distillation,” in Proc. of CVPR, 2020, pp. 1989–1998.

33. V. Verma, H. Chen, “Fast Neural architecture search of compact semantic segmentation models via auxiliary cells,” in Proc. of CVPR, 2019, pp. 9126–9135.

34. Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “Model compression and acceleration for deep neural networks: The principles, progress, and challenges,” IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 126–136, 2018.

35. M. R. U. Saputra, P. P. de Gusmão, Y. Almalioglu, A. Markham, and N. Trigoni, “Distilling knowledge from a deep pose regressor network,” in Proc. of ICCV, 2019, pp. 263–272.

36. S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. F. Abdelzaher, “FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices,” CoRR, 2018.

37. H. Yang, Y. Zhu, and J. Liu, “End-to-end learning of energy-constrained deep neural networks,” in Proc. of ICLR, 2019.

38. S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, “On-demand deep model compression for mobile devices: A usage-driven model selection framework,” in Proc. of Mobisys, 2018.

39. S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding,” Proc. of ICLR, 2015.

40. S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, “DeepIot: Compressing deep neural network structures for sensing systems with a compressor-critic framework,” in Proc. of Sensys, 2017.

41. D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, “PipeDream: Generalized pipeline parallelism for dnn training,” in Proc. of SOSP, 2019.

42. S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. F. Abdelzaher, “FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices,” in Proc. of Sensys, 2018.

43. “Tensorflow model benchmark tool description,” https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/benchmark, accessed: 2020-4-27.

44. “tconfig,” https://github.com/thomashj/tconfig.

45. “cpulimit,” https://github.com/osipeng/
cpulimit.

46. “wondershaper,” https://github.com/magnific0/wondershaper.

47. J. Brownlee, “Sequence classification with lstm recurrent neural networks in python with keras,” 2019, https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/.

48. “Keras cnn,” https://keras.io/examples/mnist_cnn/.

49. S. D. Vito, E. Massera, M. Piga, L. Martinoño, and G. D. Francia, “On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario,” Sensors and Actuators B: Chemical, vol. 129, no. 2, 2008.

50. J. Fan, X. Wei, T. Wang, T. Lan, and S. Subramaniam, “Deadline-aware task scheduling in a tiered iot infrastructure,” in Proc. of GLOBECOM, 2017.

51. S. Ten and H. Shen, “Machine learning based timeliness-guaranteed and energy-efficient task assignment in edge computing systems,” in Proc. of ICIFEC, 2019.

52. S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The ibench benchmark suite: Characterization of the mapreduce-based data analysis,” in 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), 2010.

53. Y.-M. De Hauwere, P. Vranckx, and A. Nové, “Learning multi-agent state space representations,” in Proc. of AAMAS, 2010.