RoleML: Role-Oriented Programming Model for Customizable Distributed Machine Learning on Edges

***Abstract*—Edge AI aims to enable distributed machine learning (DML) on edge resources to fulfill the need for data privacy and low latency. Meanwhile, the challenge of device heterogeneity and discrepancy in data distribution requires more sophisticated DML architectures. This calls for a general and standardized programming interface and library that provide support for easy development and testing of various DML architectures. Existing libraries like FedML are designed for specific DML architectures and do not support users to customize new architectures on them.**

In this paper, we present a novel role-oriented programming model, namely RoleML, for general and modular development of DML. RoleML abstracts standardized and reusable roles from the core functionalities of various DML architectures and provides unified interfaces for programming customized DML architectures with ease. Powered by a runtime library, RoleML supports flexible deployment and dynamic configuration of roles and manages their interaction automatically, simplifying the design of complex DML architectures. We demonstrate the wide applicability of RoleML through intensive case studies.

***Index Terms*—programming model, distributed machine learning, edge computing**

I. Introduction

Traditionally, distributed machine learning (DML) tasks are performed on the cloud with large-scale server clusters. Such an approach brings concern for data privacy since all data must be collected to the cloud before training. The emergence of edge computing solves this problem by training the models directly on edge devices and transmitting only model parameters or gradients to the cloud for aggregation. Meanwhile, edge learning also achieves low latency in model inference and training services [1], [2].

However, the heterogeneity and instability nature of edge devices has introduced new challenges to the efficiency of model training. Unbalanced distribution of data across these devices can even have a negative impact on model convergence. This means that traditional DML architectures applied on the cloud, such as Parameter Server and AllReduce, are no longer suitable for edge environments.

To tackle the above challenges, many new architectures for DML have been proposed by both industry and academia, such as Federated Learning [3] and Gossip Learning [4]. In general, these architectures can be classified according to their communication strategies (synchronous, weak synchronous, or asynchronous) and topologies (centralized, decentralized, or fully distributed) [5]. They can increase the training throughput and, therefore, speed up the training process under different network environments. For example, E-Tree Learning [6] leverages a tree structure to support decentralized and localized model aggregation at the edge. The tree is built according to the communication network topology and data distribution across edge devices.

The demand for advanced DML architectures in edge computing calls for a general programming model and framework for customizable DML. Without such a system, DML developers will have to independently implement different architectures, which often result in cumbersome, repetitive coding and diverse runtime environments. It is also difficult to conduct a fair performance comparison between various architectures in this situation. Unfortunately, existing libraries for DML are short of support for such demand: they only implement certain DML architectures based on the requirements of target tasks and provide no interface for customizing new architectures. Popular machine learning libraries like TensorFlow [7] provide synchronous architectures for high-performance parallel training, including Parameter Server [8] and AllReduce [9]. Whereas other edge learning libraries, such as FedML [10], support Federated Learning, which is another typical synchronous and centralized architecture.

To satisfy this demand, we design RoleML, a novel role-oriented programming model for general and modular development of DML. RoleML supports users to easily customize DML architectures as they need with little programming and management effort. It provides a simple yet efficient abstraction, namely *role*, to unify the expression of interactive components that interact with each other within a DML architecture. The core part of a DML architecture can be expressed by the interaction of a few key roles such *Trainer*, *Aggregator* and *Coordinator* with a small number of auxiliary roles. RoleML also allows users to define new roles to extend the functionalities of a DML architecture or change its behavior.

RoleML decouples roles from the worker nodes and uses *relationships* to map the roles to actual worker nodes on runtime. This mechanism is not tied to specific communication or topology configurations, and can thus be applied to develop DML architectures of various kinds. Compared with traditional node-oriented programming where developers program individual nodes, developers use RoleML to program individual functional modules, which can then be assembled into worker nodes on demand. This dramatically improves code reusability and maintainability and simplifies the design of complex DML architectures.

We implement a runtime library for RoleML as a common

execution environment for diverse DML architectures. This library supports dynamic assignment and configuration of roles and unifies inter- and intra-node communication with no performance loss. Users can either specify the role assignments by writing configuration files, or program logic to adjust them on the fly. In addition, RoleML extracts common functionalities from many existing DML architectures and provide standardized role definitions for *Trainer* and *Aggregator*, with out-of-the-box implementations to boost the design of new architectures. We conduct extensive case studies featuring popular DML architectures to show the high customization capability of RoleML.

II. Background and Motivation

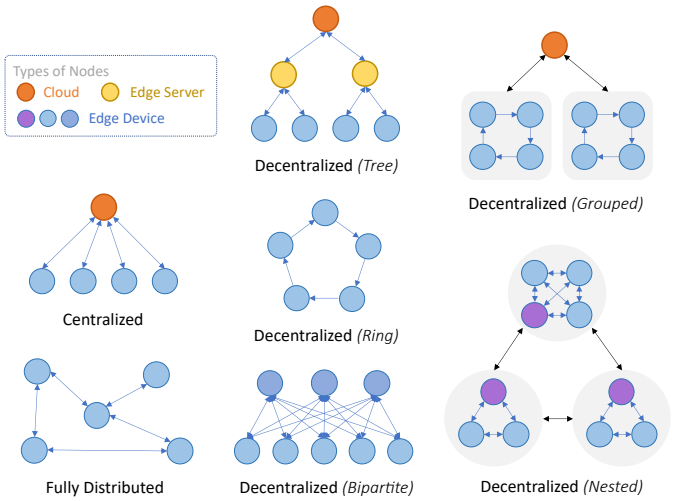
The demand for distributed training is driven by the increase in the size and complexity of ML models. Performing modern training tasks purely on the cloud requires a huge cost to maintain server clusters, although both the software (frameworks) and hardware have been highly optimized. It also challenges data privacy since the performance is guaranteed by having all data available before the training starts. These shortcomings of cloud training can be addressed by leveraging the huge amount of edge servers and devices. Therefore, most DML architectures today are designed for edge training with heterogeneous infrastructures.

There are two types of parallelism in DML: data and model parallelism [5]. In this paper, we mainly consider data parallelism, where the same model is replicated on multiple nodes and trained with different data. The reason is that most DML architectures, if not all, are designed for data parallelism, which is generally applicable to all kinds of models. It is challenging to design an architecture for model parallelism, where the same model is partitioned and trained on different nodes, since the model structures vary greatly.

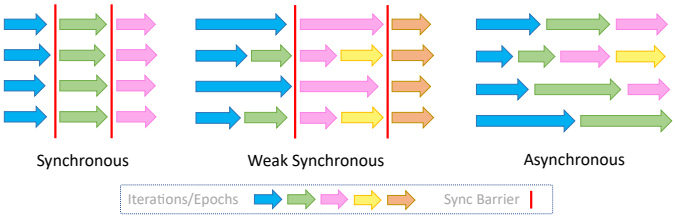
Topology and communication strategy are two main characteristics for classifying DML architectures, as they determine how models trained from different nodes (knowledge) are aggregated. Fig. 1 gives an illustration of the various types of DML architectures. State-of-the-art DML architectures known to us are listed in Table I[1]. Generally speaking, synchronous and centralized architectures are easier to model and implement, but suffer more from low device utilization due to heterogeneity and single point of failure. It is not uncommon to see centralized architectures struggle with performance by waiting for the stragglers to complete training. On the other hand, asynchronous and (fully) distributed architectures provide the nodes with more autonomy and behave more robustly, but may slow down the convergence due to the lack of model consistency.

The pros and cons of various types of DML architecture imply that no single architecture is suitable for every edge training task. However, it is possible to abstract from the commonalities of various architectures (such as actions for training and aggregation) and provide a general programming model and library for the development and testing of DML. Unfortunately, this is currently lacking to the best of our knowledge. We believe the reason is twofold: first, existing libraries typically focus on specific task domains and are therefore exclusively optimized for specific DML architectures, especially the synchronous and centralized ones. For example, FedML [10] provides a complete toolset for all kinds of problems in Federated Learning, but not for its alternative Gossip Learning. Second, the immediate demand for model accuracy hides the potential of asynchronous and distributed architectures in much more complex edge environments. Thus, there is less work on designing supporting systems or libraries for these architectures, restricting the corresponding research.

[1]Note that the topology only represents the flow of data in DML (e.g. model parameters or gradients). Non-centralized architectures like AllReduce may require centralized coordination, but it does not directly participate in training or aggregation.



(a) Classified by topology.



(b) Classified by communication strategy.

Fig. 1. Various types of DML architectures.

TABLE I
State-of-the-Art DML Architectures

Topology	Communication Strategy		
	Synchronous	**Weak Synch.**	**Asynchronous**
Centralized	FL [3] PS [8] HFL [11] PDP [12]	PR-SGD [13] FedAT [14]	SSP [15] APDP [12]
Decentralized	AllReduce [9] WAGMA [16] [a] Moshpit [17] DeDLOC [18]	EL [6]	AD-PSGD [19]
Fully Distributed	D-PSGD [20]	Co-SGD [21] Gossip-PGA [22] RNA [23]	GL [4] EdgeGossip [24]

[a] Centralized when using global averaging.

With the increasing demand for advanced DML architectures, there is now an urgent need for a systematic solution to customizable DML. Motivated by this, we design RoleML, a role-oriented programming model with runtime library. We define our design objectives to be threefold:

- **General**. The programming model and runtime system should be general-purpose and not limited to certain types of topology or communication strategy.
- **Ease of use**. The programming model should bring convenience for developers to implement roles for customized functionalities. Common functionalities in DML should be provided as standardized components (roles). Meanwhile, the runtime system should provide tools for easy deployment and testing of a DML architecture.
- **Modular**. The programming model should be intuitive and provide simple APIs to produce modular and replaceable components (roles) with ease.

III. Role-Oriented Programming Model

A. Role: the Core Abstraction

Distributed systems, including most existing DML architectures, typically follow the procedure-oriented (program for the whole workflow) or node-oriented (program for individual nodes) programming approach. We argue that these approaches are not suitable for general DML development. This is because most advanced DML architectures assign multiple functionalities to a single node, with many of them being common, such as training and aggregation. There are also many customized functionalities designed on demand, such as node grouping. Traditional programming approaches therefore often lead to repetitive and complicated coding, which is contrary to the demand for easy development and testing. RoleML solves this problem with more fine-grained interactive components.

In RoleML, a *role* is defined as a functional component that interacts with each other within a DML architecture. It encapsulates a series of messaging *channels* that provide services, indicate changes in internal status, or respond to the changes in external status. To be more specific, we design three modes of interaction (channels) between roles, including *message*, *task* and *event*. Messages and tasks stand for synchronous and asynchronous function call respectively, which is used in active, directional communication. Events follow the publish-subscribe model with conditional subscription support and are used for passive, unidirectional communication with other non-specific roles. We defer the detailed definition of APIs to Section III-C.

RoleML decouples roles from the working nodes (i.e. the actual devices) and supports dynamic assignment and configuration of roles on demand for a flexible collaboration (described in detail in Section IV). This enables developers to construct nodes in the way of building blocks, which is more consistent with the classic object-oriented paradigm. As such, RoleML dramatically reduces the repetitive work in designing complex DML architectures. We observe that three key roles, namely *Trainer*, *Aggregator* and *Coordinator*, are adequate to represent most of the DML architectures, which will be elaborated in Section III-B. The design principle for roles is that each of them should be dedicated to a single, fine-grained objective or responsibility, which is normally driven by the core information or data that a role maintains. For example, the *Trainer* maintains the model and dataset for training, so it should provide channels for fetching the local model and applying model updates (and training, of course).

TABLE II
Three Key Roles in a DML Architecture

Role	Responsibility	Resources
Trainer	Producer of model updates	Local model and training dataset
Aggregator	Consumer of model updates	Aggregation buffer
Coordinator	Controller of DML workflow	Global model and testing dataset

B. Main Types of Roles

Most of the DML architectures can be expressed by three key roles: *Trainer*, *Aggregator* and *Coordinator*. The *Trainer* and *Aggregator* represent the two basic operations in a DML architecture, while the *Coordinator* defines the overall workflow of the DML architecture[2]. Table II summarizes these roles.

Since training is a general action in DML, we also define abstract classes for models and datasets to make the Trainer workload-agnostic. In addition, it might be feasible to divide the Coordinator into several roles in some advanced DML architectures, such as an *Organizer* role for grouping the nodes in the AllReduce operation.

Besides regular architecture designs, RoleML also allows users to design roles for external components that interact with a DML architecture. For example, a *Monitor* role can be defined to collect the training progress and statistics of Trainers and visualize them, while providing user interfaces to control the overall task. We can also define an *Attacker* role to fetch the gradients produced by Trainers and study different attack algorithms. These functionalities can easily be implemented via event subscription.

C. Programming Interfaces

Core API. In RoleML, each role is represented by a class with a series of messaging *channels* described in Section III-A. All messaging between roles follows multi-value sending and single-value return, which allows the handlers to be defined just like normal functions, improving clarity. For event channels, the "multi-value" part represents the *properties* of an event, which can be leveraged by the subscriber for a conditional subscription. The user can alternatively *store* the events received in certain channels instead of writing a handler, and then use the API to *wait* for the events. Table III

[2]For fully distributed architectures such as Gossip Learning, the Coordinator is assigned to every node and expresses the node behavior, which is generally independent of each other.

summarizes the core APIs. Note that users are responsible for meaningful naming of the channels and preventing conflicts when multiple roles are to be deployed on the same node.

The interaction APIs accept worker nodes as the target since they are the actual entities in communication. When defining the interaction, we use *relationships* to represent the node where the target role will be deployed.

Collective communication. We provide a *Communicator* class as an abstraction of centralized collective communication sessions. It represents a set of target nodes from the view of the center node and provides interfaces to interact with them all at once. Meanwhile, it offers a natural way to manage the worker nodes in a grouped manner, as seen in multiple DML architectures (such as [16], [17]).

Extending a role. To maintain the extensibility of roles, we further introduce the extension mechanism for adding new functionalities to the original roles. Extensions are defined just like roles but must rely on actual roles for execution. This allows a new role to be defined as a combination of the base role and several extensions (rather than multiple inheritance). Using extensions can also separate the loading and usage of data resources, achieving flexible adaptation, especially for the model and the dataset (as exemplified in Section III-D).

D. An Example

Here we take the Federated Averaging (FedAvg) algorithm, a basic variant of Federated Learning, as an example to demonstrate the use of RoleML. We first define the key roles in this section, and show how to connect and deploy them in case studies (Section VI).

Define the Trainer. As shown in Fig. 2, we first define a task channel `train` for the Trainer to perform training (initiated by the Coordinator). It accepts the hyperparameters for the training loop and returns the statistics asynchronously. An event channel `local-update-ready` is defined to signal the completion of training and that the update is ready to be fetched by the Aggregator. We also define two message channels for fetching the local update and overriding the local model respectively. All the handlers must be declared in the `declare()` function, which is part of the role's constructor.

As mentioned in Section III-B, the Trainer calls the abstract interfaces for the model and dataset in the training procedure. Note that the Trainer itself does not contain any logic for loading the model and dataset. This is instead done in a role extension, which in this case puts the model and dataset into the node's shared storage after initialization. The Trainer then retrieves them from the shared storage (via the `conf` property). Such design allows different combinations of model and dataset to be loaded with customizable arguments to ensure compatibility.

Define the Aggregator. The Aggregator in FedAvg is responsible for collecting newly trained models from all the selected trainers and averaging them. The core logic of aggregation is implemented in only five lines of code (see Fig. 3). Here we leverage an API of the Communicator to link the events of training completion to a *pipeline*, consisting of

```python
class Trainer(Role):

  def declare(self):
    self._update = LockedSingleValueBuffer()
    self.add_task_channel('train', self.on_task_train, awaitable=True)
    self.add_event_channel('local-update-ready')
    self.add_message_channel('get-local-update', self.on_get_local_update)
    self.add_message_channel('apply-update', self.on_apply_update)

  def on_get_local_update(self, _):
    return self._update.get()

  @expand_arguments
  def on_apply_update(self, _, update):
    self.conf.get('model').set_model(update)

  @expand_arguments
  def on_task_train(self, _, epochs: int = 1, **options):
    model, dataset = self.conf.get('model'), self.conf.get('dataset')
    dataset.ready.wait()
    result = {}
    for i in range(epochs):
      result = model.train(dataset.all(), **options)
    self._update.set(model.get_model())
    self.emit_event('local-update-ready')
    return result     # accuracy, loss, etc.
```

Fig. 2. FedAvg Trainer in RoleML. For this and any other role, the first argument of each handler after `self` is reserved for the sender node (identified by a name string), which can be used for access control.

```python
@expand_arguments
def on_task_aggregate(self, _, source_names):
    sources = self.create_communicator(*source_names)
    sources.subscribe('local-update-ready', store=True)
    updates = CollectiveBuffer(sources.actors)
    # connect events to a pipeline
    sources.collect_events('local-update-ready',
        ask_for(self, 'get-local-update') | extract_kv() | updates)
    # sync barrier
    updates.ready.wait()
    return self.do_aggregate(updates.get())    # averaging logic

def do_aggregate(self, data): ...
```

Fig. 3. FedAvg Aggregator in RoleML.

a series of actions to fetch the updates from corresponding clients and send them to a collective buffer. The pipeline is automatically executed every time an event arrives. Then, the synchronization barrier is simply expressed by a `wait` operation to the buffer.

Define the Coordinator. The Coordinator in Fig. 4 controls the entire workflow of FedAvg. It selects the training nodes in each communication round, updates their local models, then sends signals to the Aggregator and Trainers respectively to initiate aggregation and training operations.

IV. Runtime Library for RoleML

A. Architecture

Fig. 5 depicts the architecture of RoleML's runtime system. From bottom to top are *Runtime Layer*, *API Layer* and *Application Layer* respectively.

RoleML adopts the Master-Worker architecture. The Master controls the execution of DML and manages all Worker nodes in the network. We should distinguish the Master from the *Coordinator* role, which defines the workflow of a DML architecture. However, the Coordinator can be deployed together with the Master, as long as the environment is compatible.

TABLE III
CORE API FOR ROLE PROGRAMMING IN ROLEML

Type	API	Description
Action Declaration	**add_message_channel**(*name*, *handler*)	Register a message channel for the given handler.
	add_task_channel(*name*, *handler*, *awaitable*)	Register a task channel for the given handler. The flag *awaitable* indicates whether the result of task execution can be fetched.
	add_event_channel(*name*)	Register an event channel, which is required before triggering events.
Action Definition	*reply* = **send_message**(*target*, *channel*, *message*)	Send a message to the target node and wait for a reply. The target node is usually identified via relationship.
	future = **send_task**(*target*, *channel*, *message*)	Send a task to the target node. If the task is awaitable, a *future* will be returned with result available via *future*.result().
	subscribe(*target*, *channel*, *options*)	Subscribe to an event channel of the target node. The user can either connect the events to a handler or store the events (and then **wait_for_event()** when needed). Conditions can be specified to filter out unwanted events based on the properties.
	unsubscribe(*target*, *channel*)	Unsubscribe from an event channel of the target node.
	emit_event(*channel*, *info*)	Trigger an event in *channel* and send the *info* to eligible subscribers (filtered by condition matching).
	event = **wait_for_event**(*target*, *channel*)	Block until the corresponding event is triggered. Require subscription.

```python
@expand_arguments
def run(self, _, num_rounds: int = 75, select_ratio: float = 0.1, **options):
    model, test_set = self.conf.get('model'), self.conf.get('dataset')
    for i in range(num_rounds):
        lucky_dog_names = sample(trainer_names, select_ratio)
        selected_clients = self.create_communicator(*lucky_dog_names)
        selected_clients.broadcast_message(
            'apply-update', attachments={'update': model.get_model()})
        aggregation_task = self.send_task(
            '/aggregator', 'aggregate', {'source_names': lucky_dog_names})
        selected_clients.broadcast_task('train', options)
        update = aggregation_task.result()
        model.set_model(update)
        test_result = model.test(test_set.all())
        self.emit_event('round-completed', properties=test_result)
```

Fig. 4. FedAvg Coordinator in RoleML.

The *API Layer* is based on the *Runtime Layer* described above. This layer exposes interfaces for implementing roles and extensions, and we also provide many standardized built-in roles for Trainer and Aggregator (see Section V). These standardized roles serve as references for new role designs as well as a quick starter of new architectures. Other roles, including the Coordinator, are left for the users to customize.

In the *Application Layer*, we provide unified interfaces for adapting the model and dataset. Deployment tools are also provided to build a DML architecture out of roles and deploy them in the Runtime Layer.

The current version of RoleML runtime is implemented in Python 3.9 with around 3200 lines of code. The use of Python is mainly due to its excellent portability, which greatly fits the need for edge computing. It is also the first choice for implementing the model and dataset.

B. Design of Worker Nodes

In RoleML, each worker node operates independently as a runtime environment for the roles deployed. When a node starts, a *control proxy* is first initiated to manage the status of the node (including all assigned roles and active relationships) and accept updates to the status (e.g. new role assignments), which can also be done via interaction (e.g. with the Coordinator). Therefore, the control proxy implements the same interfaces as a role, similar to the concept of init container in

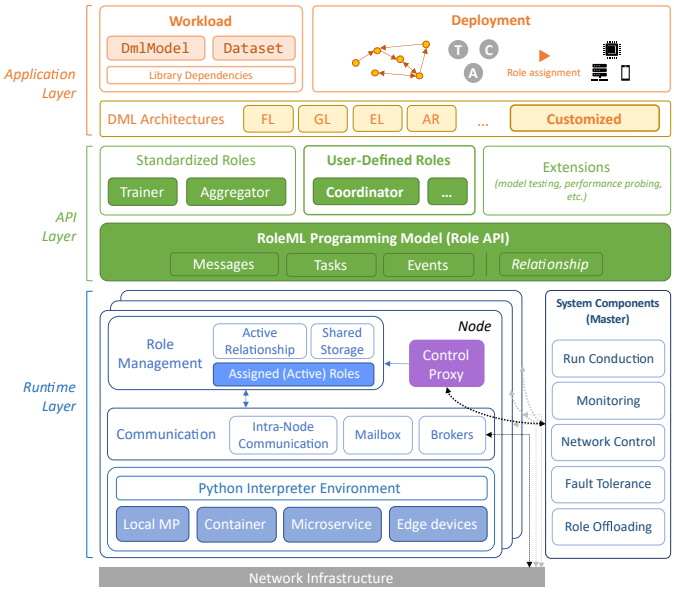


Fig. 5. Architecture of RoleML runtime.

Kubernetes. Any change to the role's status during a DML run will trigger the corresponding event, which can be subscribed by the roles on top. This way, a role's behavior can further be simplified via *relationship-driven* actions.

There are two storage components inside a worker node. The *Shared Storage* is a public dictionary for configuration variables; it also brings convenience to the data exchange between a role and its extensions. The *Mailbox* caches incoming role events and provides corresponding thread events when the user is waiting for certain role events.

RoleML's roles are designed to be execution-environment-agnostic. Currently, a worker node is implemented as a single process, which is very useful in local development. In the future, we will implement more types of worker nodes, including containers and cloud platforms for microservices.

Communication middleware. In order to achieve dynamic role assignments, we unify the communication within and

between nodes. The node maintains a dictionary of channel handlers of all roles, mapping all intra-node communication to the corresponding function calls. Exceptions, in this case, will be directly propagated to the invocating sender. This requires no transmission over the network.

For inter-node communication, the *message broker* and *event broker* are two separate components for different kinds of interaction. They manage the (de)serialization of all data types and identify the communication entities (i.e. the nodes). Currently, we have implemented brokers for the HTTP protocol. Support for more edge and IoT protocols such as MQTT will be provided in the future.

C. System Functionalities

Fault tolerance. RoleML aims to improve the efficiency of any DML architecture running on top by reducing the impact of failures at the edge, which can be caused by either the devices or the network links. Therefore, we need to encapsulate *general* functionalities for both error detection and handling strategies.

For error detection, it is desired to accurately determine the type of an error before taking recovery actions. To determine if a task is executing correctly, we simply check if an exception is raised. To prevent a node from not responding (in which case the task may still be executed properly), we can either add specific heartbeat channels to the role or monitor the event emission of the role. For example, if a role is not emitting certain events for a long period of time but the heartbeat does not fail, the task can be considered frozen. It is also possible to let the edge nodes probe the status of each other and report to the master if needed.

Then, proper handling strategies can be applied based on the type of error. If a task fails, the user can simply choose to *rerun* or *resign* the task (e.g. skip a round of AllReduce). Another option is to *reassign* the role to another node (i.e. task offloading, discussed below). However, if a task is executed normally and only the network link is broken, *retransmission* is probably preferred. Alternatively, the task result can be *retained* for use in the next call if it just produces too slow (as the straggler). Generally, these strategies can work in concert with roles for node monitoring or coordination. In future work, we will further elaborate on the fault model and provide a systematic solution for fault tolerance.

Role offloading. Many edge computing tasks obtain performance speedup by offloading sub-tasks between devices. We believe that this is an emerging trend in future DML architectures. Therefore, RoleML supports offloading roles between different worker nodes. During the offloading, all the channels, including extensions, as well as the status of a role are transmitted via the control proxies. RoleML manages the task dependencies and suspend necessary workflows so that relationships can be updated correctly.

It is challenging to determine the target node for offloading since multiple factors must be taken into account, including node performance (reflected by the computing devices such as GPU), network bandwidth and delay, as well as data privacy and training time constraints. We will continue to explore a proper formulation of the offloading problem considering specific challenges in edge training, such as data privacy.

Network control. RoleML separates a node's active relationship and its communication links. In future versions of RoleML, we will take this advantage and add more functionalities for network control, such as link shaping and relay transmission. Specific problems in edge training will also be addressed, such as hole punching under NAT conditions [18].

D. Deploying a DML Application

Before the start of a DML run, the user should specify configurations of all initial nodes in a YAML file, including role assignments, relationships and network connections. Python modules containing the required roles and all library dependencies (including the ones for model and dataset) must be prepared first on each target node.

Nodes must be started first before the above configurations are deployed via the control proxies. We provide scripts for starting independent nodes. A user-defined entry-point function, which typically consists of message and task calls to certain roles, is used to initiate the DML run. We provide a command-line interface (CLI) for users to deploy and start the DML application with flexible arguments. For nodes started during the DML run, initial roles should be provided to perform handshakes with existing nodes.

V. Standardized Roles

Many DML architectures share the same set of actions for training and aggregation, which makes it possible to extract these actions and design standardized roles that can be directly applied to new architecture designs. In RoleML, we provide 2 types of Trainer and 6 types of Aggregator out of the box as standardized components.

A. Standardized Trainers

Trainers are considered bottom producers in a DML network topology and generate model updates. Depending on the way of production, the two following Trainers are designed.

Planned Trainer: This Trainer trains the model according to a customizable training plan, which is essentially an iterator of hyperparameters. Results of each round (e.g. accuracy and loss) will be fed back to the training plan. This allows the user to alter the training procedure on the fly, for example, to achieve dynamic learning rates or early exit[3]. This Trainer is suitable for a majority of DML architectures.

Continuous Trainer: This Trainer iterates over the local dataset with a given batch size to compute and accumulate gradients until a stop signal is received (or the iteration finishes). Progress events are regularly emitted for tracking purposes, e.g. when a global large batch size is desired [18].

[3]The `StopIteration` exception indicating the end of the training plan can be raised at any time.

B. Standardized Aggregators

We design Aggregators for different aggregation schemes, including one-to-one, one-to-many and many-to-many.

Collective Aggregator (many-to-one). This Aggregator collects updates from multiple sources and averages them. To avoid unnecessary communication, the aggregated update will not be actively returned to the sources. Despite the aggregation being centralized, this Aggregator can also be used in non-centralized DML architectures, such as [22].

Layered Collective Aggregator (many-to-one). This is an enhanced version of Collective Aggregator suitable for hierarchical topologies. It adds extra channels to act like a Trainer, with the aggregated model available as an update for upper-layer Aggregators. We provide a special *Layered Planned Trainer* to work in concert with this Aggregator, in case the user wishes to assign both roles to the same node.

Cumulative Aggregator (many-to-one). This Aggregator continuously listens to updates via a message channel and aggregates any updates received immediately.

AllReduce Aggregator (many-to-many). This type of Aggregator usually has multiple instances work together to perform a complete AllReduce algorithm, with a *Step Coordinator* to control the progress of participating nodes. A user-defined function is required to get a slice of the model given the index and the total number of slices. We provide implementations for butterfly and ring AllReduce algorithms.

Push Aggregator (one-to-many). This Aggregator actively pushes the updates produced by Trainer to other Aggregators in neighbor nodes and stores updates from other Aggregators before fetched by other roles.

Exchange Aggregator (one-to-one). This Aggregator is for exchanging updates with neighbor nodes (peer-to-peer aggregation). It can either work in active mode (finding an idle neighbor and initiating an exchange process) or passive mode (accepting the exchange request).

VI. Case Studies and Experiments

A. Centralized DML: Federated Learning

Federated Learning [3] is a popular DML architecture for distributed training with localized data. A detailed description of the vanilla Federated Learning is presented in Section III-D, where we implement the three key roles for the Federated Averaging algorithm. Now we deploy these roles to actual nodes and demonstrate customized training with RoleML. Since this is a centralized architecture, we assign Aggregator and Coordinator to the server and Trainer to the clients, as shown in Fig. 6.

Many FL architectures prefer to choose "faster" clients more often since it increases the training throughput. To obtain the performance of the clients, we add a *Training Time Prober* extension to the Trainer. It defines an extra task channel to measure the time consumption of one training epoch. Model transmission time, on the other hand, can be measured by calling the corresponding getter and setter channels of the Trainer. This can be implemented as an extension for the

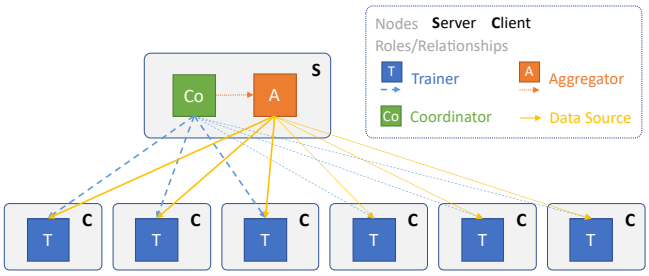


Fig. 6. Topology structure and role assignments in Federated Learning.

Aggregator. Both extensions can then be managed by a new role named *Client Selector* that interacts with the Coordinator and provide channels for probing and client selection. To apply other client selection strategies (such as random selection), just replace the Client Selector with another implementation exposing the same messaging channels.

B. Decentralized DML: E-Tree Learning

E-Tree Learning [6] is a novel decentralized DML architecture aiming to improve training convergency and model accuracy with Non-IID data. It leverages a tree structure to organize the aggregation of model updates. Therefore, each node in the tree can be mapped to a role instance of either the Trainer (leaf node) or the Aggregator (non-leaf node). In practice, we use *Layered Planned Trainer* and *Layered Collective Aggregator* respectively in hierarchical aggregation. We also need an extension for each Trainer to initiate new training rounds after receiving model updates, which can be sourced from either the first-level or the root Aggregator. It enables the use of diverse training plans for clients with different data distributions.

E-Tree Learning also adopts localized aggregation to save communication costs. Each aggregation operation is conducted in one of the data source nodes. This means that the same working node in the network may appear in multiple layers of the aggregation tree, which can easily be implemented by assigning multiple role instances to the same worker node, including Trainer and/or Aggregator, distinguished by a layer ID. Fig. 7 gives an example, where we deploy the Trainer and two instances of the Aggregator (and a Coordinator) to the worker node representing the root of the aggregation.

E-Tree Learning further proposes dynamic client clustering to adapt to the changing edge environments. In RoleML, this is achievable with dynamic role assignments. To adjust the aggregation tree, just assign new Aggregators representing the corresponding layers and terminate old ones if necessary. Then, configure the relationships before the Aggregators subscribe to new data sources. The whole process can be conducted by a dedicated role that interacts with the Coordinator.

C. Fully Distributed DML: Gossip Learning

Gossip Learning [4] is widely considered an alternative to Federated Learning in more heterogeneous environments. In

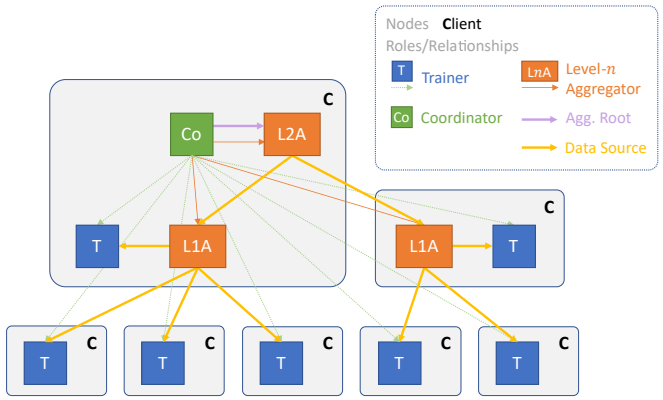


Fig. 7. Topology structure and role assignments in E-Tree Learning.

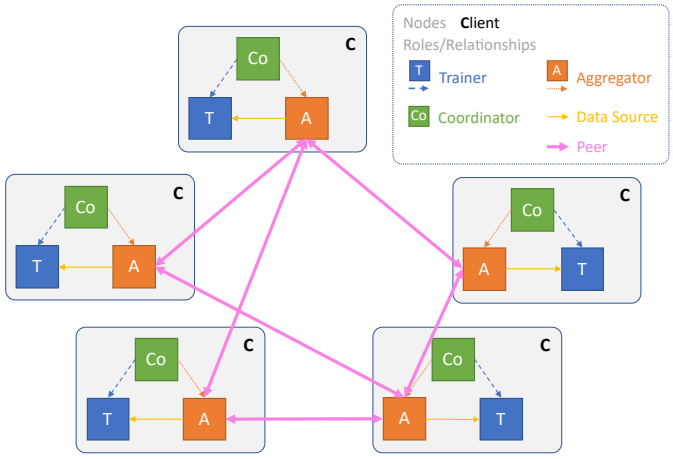


Fig. 8. Topology structure and role assignments in Gossip Learning.

this architecture, each node trains its local model independently and propagates model updates to neighbors in an active way. Since it is fully distributed, we assign *Planned Trainer*, *Push Aggregator* and Coordinator simultaneously to each client, as shown in Fig. 8. The Coordinator is responsible for passing updates received by the Aggregator (after averaging with the local model) to the Trainer and initiating the next round of training. Recall that the Aggregator automatically fetches the updates produced by the Trainer. Therefore, the Coordinator links the Aggregator back to the Trainer with another data path to form a complete loop of data flow.

D. Experiments

We now conduct some experiments to demonstrate how to use RoleML for easy testing of DML architectures.

To perform evaluation, we can easily write a *Tester* role extension for the Coordinator which holds the global model (or local model for Gossip Learning). There are two ways to initiate the evaluation: subscribing to the Trainer's event for training completion or providing a task channel for the Coordinator to call manually after aggregation completes. We recommend the latter since it avoids the conflicts between training and testing. We also need another *Loader* extension

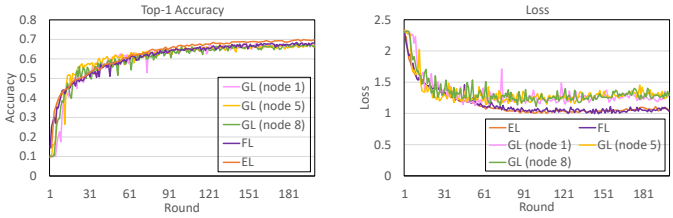


Fig. 9. Evaluation result of three DML architectures.

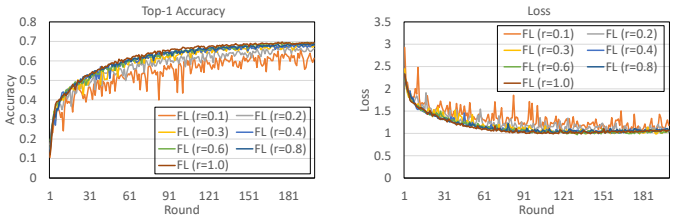


Fig. 10. Impact of different client selection ratios for Federated Learning.

to load the test dataset. Make sure these extensions are loaded before the Coordinator.

In our first experiment, we make a comparison between Federated Learning, E-Tree Learning and Gossip Learning. We apply these architectures respectively to train the same CNN model with about 550K parameters in the same network with 10 client nodes (and a server added for Federated Learning). More concisely, we adopt a client selection ratio of 0.4 for Federated Learning, a two-level aggregation for E-Tree Learning (frequency ratio 2:1), and a random topology for Gossip Learning with each client having three to four neighbors. The dataset chosen is CIFAR-10 [25], randomly divided into 10 equal parts and distributed to the clients.

We display our result in Fig. 9. After 200 training rounds, E-Tree Learning achieves the highest accuracy thanks to the utilization of all clients in every communication round. Federated Learning performs slightly worse, which we will study later. Gossip Learning produces higher fluctuations in the beginning stage and achieves lower accuracy than the others, which is within the expectation for a fully distributed architecture. This experiment shows that developers can use RoleML to conduct a fair comparison between DML architectures.

In our second experiment, we continue to explore the impact of the client selection ratio in Federated Learning. We rerun the test with different ratios and show the result in Fig. 10. In a nutshell, a higher selection ratio leads to higher stability and accuracy. This can be explained by more data being used in each round which helps balance the data distribution. A selection ratio of 1 (full utilization) yields a similar result as E-Tree Learning. This experiment shows that RoleML can be used to conduct further testing on a DML architecture.

E. Supports of More DML Architectures

Finally, we select several other architectures of different types and briefly discuss how to implement them in RoleML with some important details.

AD-PSGD [19]. This is a fully-distributed DML architecture that trains asynchronously, but aggregates in a weak synchronous manner: model updates are exchanged and averaged between peers. Like Gossip Learning, we assign all key roles to every peer, but with *Exchange Aggregator* selected. The Coordinator updates the local model after aggregation and applies gradients once they are produced by the Trainer. Access conflicts to the Trainer's model can be resolved by a mutex in the Coordinator.

WAGMA-SGD [16]. This architecture uses grouped AllReduce and global averaging in turn for aggregation. Therefore, we assign *AllReduce Aggregator* and *Planned Trainer* to the clients and *Collective Aggregator* to the central server. The *Step Coordinator* is also needed on every client since the AllReduce operation is initiated by any Trainer that finishes first. The *Coordinator* on each client needs to interact with each other to determine the initiator of AllReduce; while the *Coordinator* on the server just monitors the training steps and initiates the global averaging regularly.

HFL [11]. This is a Federated-Learning-like architecture using a hierarchical topology to organize nodes. Unlike E-Tree Learning, the height of its aggregation tree is fixed to three, with FedSGD and FedAvg algorithms applied respectively for the two levels of aggregation. To implement this architecture, we simply use the *Collective Aggregator* for aggregation since the layered structure can be managed by *Coordinators* deployed together with each Aggregator.

VII. Related Work

A. Mainstream Machine Learning Frameworks

Modern mainstream ML frameworks, such as TensorFlow [7] and PyTorch [26], typically come with built-in distributed support. However, it is mainly intended for high-performance homogeneous clusters (such as multi-GPU) and therefore only some classic architectures such as Parameter Server and AllReduce are implemented. To apply these architectures in distributed training, the logic for communication strategies and network topology must be directly embedded into the original training loop, making it difficult to switch between different DML architectures given the same training loop. It can as well lead to security problems since the server code is directly exposed to the client, and is therefore unsuitable for edge learning. On the other hand, RoleML decouples the DML architecture and the training loop and assigns separate roles to the server and client respectively, improving both security and flexibility.

B. DML-Specific Systems

Recent years have witnessed the emergence of diverse systems that provide support for certain DML architectures. They typically rely on mainstream ML frameworks for underlying tasks such as model construction, and design new programming interfaces for distributed training.

DIANNE [27] is a distributed framework mainly used in IoT and reinforcement learning. It provides ten components for common distributed training tasks (e.g. DNN modules, dataset and agent), implemented as microservices under the Java OSGi specifications. DIANNE supports model parallelism and provides a GUI for codeless neural network construction and workflow definition. However, codeless development also limits its capability to define more sophisticated workflows for DML.

FedML [10] is initially a research and benchmark framework for Federated Learning. It adopts a client-oriented programming model to decouple the communication strategy from the training loop logic, but provides only one type of messaging (peer-to-peer sending) which leads to extra effort in development. Whereas in RoleML, role-oriented programming and different kinds of role interaction make development easier. In addition, RoleML is designed for all kinds of DML architectures, not just limited to Federated Learning. It is worth noting that FedML is currently under a major evolution that focuses on Federated Learning in more scenarios (such as cross-silo) and now provides an open platform for MLOps.

EdgeTB [28] is a hybrid testbed toward edge distributed learning. It is the first to propose the concept of role-oriented development, classifying DML nodes into *Trainer*, *Aggregator* or an integration of both roles, but it does not provide any standard for defining roles. Moreover, EdgeTB encapsulates execution environments into each role, which limits code reusability. In contrast, RoleML provides an effective encapsulation for the well-defined roles. It also decouples the runtime from the role definition and provides an abstraction layer for the model and dataset in which the user manages library dependencies, improving code reusability and clarity of DML architectures.

VIII. Conclusion

The demand for advanced DML architectures in edge computing calls for a general programming model and framework for customizing different architectures. However, existing systems only support certain DML architectures and cannot be extended to others. This paper presents RoleML, a role-oriented programming model for easy development and testing of diverse DML architectures. RoleML provides *role* as the common abstraction of various interactive components in DML, with *Trainer*, *Aggregator* and *Coordinator* being the three key roles that create the core part a DML architecture. With an intuitive programming interface, users can easily define customized roles to extend the functionalities of a DML architecture. We also implement a runtime system for RoleML that supports dynamic role assignment and configuration, simplifying the programming of complex DML architectures. Finally, we conduct extensive case studies and experiments to demonstrate the use of RoleML in various DML architectures.

REFERENCES

[1] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A survey on edge computing systems and tools," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1537–1562, 2019.

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

[3] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Singh and J. Zhu, Eds., vol. 54. PMLR, 20–22 Apr 2017, pp. 1273–1282.

[4] I. Hegedűs, G. Danner, and M. Jelasity, "Gossip learning as a decentralized alternative to federated learning," in *Distributed Applications and Interoperable Systems*, J. Pereira and L. Ricci, Eds. Cham: Springer International Publishing, 2019, pp. 74–90.

[5] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Computing Surveys*, vol. 53, no. 2, Mar 2020.

[6] L. Yang, Y. Lu, J. Cao, J. Huang, and M. Zhang, "E-tree learning: A novel decentralized model learning framework for edge AI," *IEEE Internet of Things Journal*, vol. 8, no. 14, pp. 11 290–11 304, 2021.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[8] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27, 2014.

[9] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018. [Online]. Available: http://arxiv.org/abs/1802.05799

[10] C. He, S. Li, J. So, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr, "FedML: A research library and benchmark for federated machine learning," *Advances in Neural Information Processing Systems, Best Paper Award at Federate Learning Workshop*, 2020.

[11] M. S. H. Abad, E. Ozfatura, D. Gunduz, and O. Ercetin, "Hierarchical federated learning across heterogeneous cellular networks," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 8866–8870.

[12] G. Zhao, T. Zhou, and L. Gao, "A proactive data-parallel framework for machine learning," in *2021 IEEE/ACM 8th International Conference on Big Data Computing, Applications and Technologies (BDCAT '21)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 69–79.

[13] H. Yu, S. Yang, and S. Zhu, "Parallel restarted SGD with faster convergence and less communication: Demystifying why model averaging works for deep learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 5693–5700.

[14] Z. Chai, Y. Chen, A. Anwar, L. Zhao, Y. Cheng, and H. Rangwala, "FedAT: A high-performance and communication-efficient federated learning system with asynchronous tiers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. Association for Computing Machinery, 2021, pp. 1–16.

[15] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," *Advances in Neural Information Processing Systems*, vol. 26, 2013.

[16] S. Li, T. Ben-Nun, G. Nadiradze, S. Di Girolamo, N. Dryden, D. Alistarh, and T. Hoefler, "Breaking (global) barriers in parallel stochastic optimization with wait-avoiding group averaging," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1725–1739, 2020.

[17] M. Ryabinin, E. Gorbunov, V. Plokhotnyuk, and G. Pekhimenko, "Moshpit SGD: Communication-efficient decentralized training on heterogeneous unreliable devices," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 18 195–18 211.

[18] M. Diskin, A. Bukhtiyarov, M. Ryabinin, L. Saulnier, q. lhoest, A. Sinitsin, D. Popov, D. V. Pyrkin, M. Kashirin, A. Borzunov, A. Villanova del Moral, D. Mazur, I. Kobelev, Y. Jernite, T. Wolf, and G. Pekhimenko, "Distributed deep learning in open collaborations," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 7879–7897.

[19] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 3043–3052.

[20] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[21] J. Wang and G. Joshi, "Cooperative SGD: A unified framework for the design and analysis of communication-efficient SGD algorithms," *CoRR*, vol. abs/1808.07576, 2018. [Online]. Available: http://arxiv.org/abs/1808.07576

[22] Y. Chen, K. Yuan, Y. Zhang, P. Pan, Y. Xu, and W. Yin, "Accelerating gossip SGD with periodic global averaging," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 1791–1802.

[23] D. Yang, W. Rang, and D. Cheng, "Mitigating stragglers in the decentralized training on heterogeneous clusters," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 386–399.

[24] R. Han, S. Li, X. Wang, C. H. Liu, G. Xin, and L. Y. Chen, "Accelerating gossip-based deep learning in heterogeneous edge computing platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1591–1602, 2020.

[25] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[27] E. De Coninck, S. Bohez, S. Leroux, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, "DIANNE: a modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure," *Journal of Systems and Software*, vol. 141, pp. 52–65, 2018.

[28] L. Yang, F. Wen, J. Cao, and Z. Wang, "EdgeTB: A hybrid testbed for distributed machine learning at the edge with high fidelity," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2540–2553, 2022.