# *EDDL*: A Distributed Deep Learning System for Resource-limited Edge Computing Environment

Pengzhan Hao
Computer Science Department
Binghamton University
Binghamton, New York, USA

Yifan Zhang
Computer Science Department
Binghamton University
Binghamton, New York, USA

## ABSTRACT

This paper investigates the problem of performing distributed deep learning (DDL) to train machine learning (ML) models at the edge with resource-constrained embedded devices. Existing solutions mostly focus on data center environments, where powerful server-class machines are interconnected with ultra-high-speed Ethernet, and are not suitable for edge environments where much less powerful computing devices and networks are used. Due to the resource constraint on computing devices and the network connecting them, there are three main challenges for performing edge-based DDL: (1) susceptibility to struggling workers, (2) difficulty of scaling up to a large training cluster, and (3) frequent changes in training device availability and capability. To address these challenges, we design and implement *EDDL*, an edge-based DDL system, with ARM-based ODROID-XU4 and Raspberry Pi 3 Model B boards. We evaluate the prototype *EDDL* system by performing edge-based mobile malware detection and classification on a large Android APK dataset. The evaluation results show that *EDDL* can efficiently train deep learning models with consumer-grade embedded devices and wireless networks while incurring small overhead.

**ACM Reference Format:**
Pengzhan Hao and Yifan Zhang. 2021. *EDDL*: A Distributed Deep Learning System for Resource-limited Edge Computing Environment. In *The Sixth ACM/IEEE Symposium on Edge Computing (SEC '21), December 14–17, 2021, San Jose, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3453142.3491286

## 1 INTRODUCTION

**Edge-based DDL training is desirable**. We advocate performing DDL at the edge to train deep neural network (DNN) models due to the following observations.

*First*, we observe that co-located mobile device users, such as users within a university, an office campus, or a hospital, usually share similar interests, exhibit similar behavior patterns, or even are targeted by similar threats. Take mobile app preferences as an example, social networking, entertainment, and gaming apps are popular among university students; professionals from financial institutions are likely to have apps related to work productivity, finance, and their companies installed on their devices. Co-located users may also be targeted by the same or similar security threats as mobile apps are becoming increasingly susceptible to targeted attacks [22, 55, 56], where malware are targeting specific groups of users by masquerading as legitimate apps that are likely used by the victims [5, 52].

*Second*, we observe that ML-based solutions that take training data in a global manner are less effective to deal with localized testing data than solutions that are generated based on training data collected locally. This is because ML classification models work best if the distributions of the testing data match those of the training data. We present a motivation study which demonstrates this observation in §3.

Based on the first two observations above, we argue that it is advantageous to generate ML models by training with data collected locally from the same group of co-located users.

*Third*, most of the existing ML-based solutions require processing and storing the training data collected from the users centrally on the cloud. However, there are growing concerns on the trustworthiness of cloud when it comes to processing/storing user data [3, 4, 17, 25, 54, 66]. Indeed, a series of recent user data leakage incidents [13, 18, 27, 46] warrant prudent designs for user privacy protection.

Because of the third observation above, we argue that it is desirable to move storing/processing of user data and training ML models from cloud to the infrastructures owned and trusted by the users. The edge computing infrastructure, which takes advantage of the computing resources at the edge of networks, is a natural and ideal fit for this purpose.

**We focus on embedded edge devices**. The edge devices we aim to utilize are the existing edge infrastructure devices, such as wireless access points and base stations, and end user devices, such as smartphones and tablets, both of which are mostly embedded devices. We do not consider dedicated servers deployed at the edge for a couple of reasons. First, embedded edge devices are far more universal than dedicated edge servers and are mostly underutilized. Second, most of the existing DDL solutions focus on data center environment where powerful server-class machines are interconnected via ultra-high-speed Ethernet [1, 9, 14, 35, 48, 61]. These solutions are likely to well suit the environment with dedicated high-performance edge servers. However, there have been few works investigating the issues of training DNN models using embedded devices which are connected with consumer-grade wireless networks. Therefore, our goal to investigate the practical issues of supporting efficient DDL at the edge with resource-limited embedded devices.

*SEC '21, December 14–17, 2021, San Jose, CA, USA*

ACM ISBN 978-1-4503-8390-5/21/12…$15.00
https://doi.org/10.1145/3453142.3491286

**Challenges**. To achieve the above goal, we design *EDDL*, a system for DDL training by utilizing embedded edge devices. Like many existing solutions, *EDDL* adopts the *parameter server* approach as the way of coordinating the training nodes [8, 11, 14, 24, 35, 61]. With *EDDL*, edge devices participating a training job form a *training cluster*. There are two roles for the training devices: *worker* and *leader*. The training data is divided into shards such that *worker* devices can concurrently work on their own shards of training data using mini-batch Stochastic Gradient Descent (mbSGD) [36, 60], and synchronize updates of the model's parameters to the leader device. The *leader* device essentially works as a parameter server [10, 23, 35, 37]: it performs aggregation of worker parameter updates and sends back the new global parameters to the workers. An *epoch* of training is said to be done when the whole set of the training data has been trained on once. The training is repeated for multiple epochs until a desirable model accuracy is achieved.

While *EDDL* share similarities with the existing DDL works, it faces three unique challenges which stem from the resource constraints of the edge devices and the underlying network.

• *Challenge 1: EDDL is more susceptible to struggling workers than existing solutions.* Struggling workers are those which take longer time to consume their shards of training data than the other workers. They prolong the time needed for an epoch of training and thus the whole training process. Edge-based DDL is impacted more by struggling workers because of edge devices' native workloads (such as routing/switching workloads for wireless APs), which can exhibit a large variance on different edge devices, and thus leaving a highly variable amount of computing resource for the DDL training.

• *Challenge 2: it is difficult for EDDL scale up a training cluster because of the network bandwidth constraint at the leader device.* With the parameter server architecture, all worker devices transmit their updated model parameters to the leader device for aggregation. As a result, the network bandwidth capacity at the leader device becomes a bottleneck of scaling up the number of worker devices. Compared to cloud-based DDL systems, the bandwidth bottleneck is much more significant for edge-based DDL systems like *EDDL*. This is because cloud-based systems run in data center environment where high performance computing nodes are connected via ultra-high-speed Ethernet (e.g., commonly at hundred Gigabit level [32, 53]), whereas mobile/edge devices have a much lower interconnecting speed (e.g., 100 to 200 Mpbs of real-world speed with recent WiFi standards [2, 47]).

• *Challenge 3: EDDL needs to deal with training device changes more often than traditional DDL systems.* Here training device changes refer to the cases like device joining or leaving a cluster and sudden training capability decline/boost in training device. Due to the resource constraints and mobility of the edge devices, dealing with training devices changes is critical to *EDDL*'s design.

**How *EDDL* addresses the challenges**. To address the first challenge above (i.e., susceptibility to struggling workers), we design a dynamic training data distribution mechanism with which shards of training data are distributed to the workers by the leader during run time. Compared with the majority of the existing solutions, which statically partition training data among the workers [8, 11, 14, 35, 61, 65], our mechanism can minimize the impact of struggling workers on training time efficiency (§4.2).

To address the second challenge (i.e., network bandwidth bottleneck at the leader), we have three key designs (§4.3):

*(1) We adopt synchronous parameter update (PU) by default.* The conventional wisdom suggests that fully asynchronous PU [9, 14, 35, 38, 45, 50] and asynchronous PU with bounded staleness [10, 11, 23, 61] are more time-efficient that synchronous PU because individual training workers do not need to wait for other workers' parameters to be synced to the leader. However, we find that sync PU works better than the async counterpart for resource-limited edge computing environment. The reason is that when compared to sync PU, async PU shortens time needed for an epoch of training, but it also increases the number of epochs needed for convergence. As a result, it requires a large training cluster for async PU to outperform the sync counterpart. Unfortunately, in an edge computing environment, before the training cluster can reach the size larger enough to favor async PU, the network bandwidth at the leader is saturated, meaning further increasing cluster size does not reduce, or even worsen, the time needed for an epoch of training. We call the training cluster is saturated when the leader bandwidth is saturated. In this case, further increasing the number of training devices does not help improve the overall training time.

*(2) We design a run-time method to detect if a training cluster is saturated.* For the reason described above, it is important to detect if a training cluster becomes saturated as new workers are joining it. We design a mechanism to accurately achieve such detection during run time. To the best of our knowledge, we are the first to study the issue of detecting cluster saturation in DDL training.

*(3) We propose the mechanism of leader role splitting (LRS) to help scaling up training cluster size after the leader's bandwidth is saturated.* The idea is to adaptively split the leader role among multiple workers such that each worker also works as the leader of a subset of training devices. Our evaluation result shows that LRS can significantly reduce overall training time when training cluster becomes saturated.

To address the third challenge (i.e., need of dealing with training device changes), we take the dynamics of devices, such as joining/leaving the cluster and change of computation capabilities, into consideration in the designs of training cluster formation, training nodes management, and leader role transfer (§4.4).

**Contributions**. In summary, we make the following contributions in this paper.

• We design and implement *EDDL*, an edge-based distributed deep learning system for training DNN models. We identify and address three main challenges of applying DDL in an edge computing environment with resource-limited computing devices. Departing from the conventional wisdom, we demonstrate that synchronous parameter update actually works better than the asynchronous counter part in an environment with limited network bandwidth. The mechanism of detecting training cluster saturation during run time is first of its kind to the best of our knowledge.

• We implement a *EDDL* prototype system using ARM-based ODROID-XU4 boards [21] and Raspberry Pi 3 Model B boards [49], and conduct a comprehensive set of experiments on a 16-worker-node testbed to evaluate and study the practical issues and implications of running *EDDL*.

## 2 RELATED WORK

**Distributed deep learning**. There have been a plethora of recent works which focus on improving distributed deep learning (DDL) from theoretical perspective or on implementing and supporting real-world DDL systems. Among these works, some adopt a centralized approach, which uses parameter servers to coordinate training workers and synchronize trained gradients/parameters [1, 8, 9, 11, 14, 23, 35, 37, 61, 67], while others opt for a decentralized approach [39–41]. Some works exploit data parallelism by employing multiple computing nodes to consume training data in parallel [8, 11, 14, 35, 61], while others take advantage of model parallelism which divides a large model into small parts so that individual computing nodes can be used to process different parts of the model [14, 29, 30, 61].

**Federated learning and traditional DDL**. Due to the high computing resource demand for training DL models, most research work of deep learning on embedded devices have been focusing on DL inferences [20, 31, 42, 43, 62, 64]. Federated learning (FL), which is a type of DDL proposed by Google [6, 44], is a promising way of generating DL models on resource-constrained embedded devices. There are three main differences between FL and traditional DDL. *First*, a main motivation behind FL is to protect user data privacy. To this end, FL generates ML models by training on private user data on individual devices which can be heterogeneous, unbalanced, and non-independent and identically distributed (non-i.i.d.), whereas traditional DDL assumes that the training datasets are identically distributed and centrally stored. *Second*, training devices in FL are usually connected via wide area networks (WANs), and are loosely-coupled in the training. On the contrary, training nodes in traditional DDL are connected via a high-speed local area network (LAN), and are tightly-coupled in the training process, which is the reason that traditional DDL incurs shorter training time than FL. *Third*, training devices in FL are usually consumer-grade devices and have lower computation capacity than those in traditional DDL which are usually high performance servers.

*EDDL* is similar to traditional DDL in that it targets DDL scenarios where training devices are connected via the same edge network infrastructure (e.g., a LAN) and can be tightly-coupled to accomplish a training task. But it is also different because the training devices in our scenarios are resource-limited consumer-grade mobile devices. An example of such an scenario is that college mobile users living in the same campus can face similar security threats as mobile apps are becoming increasingly susceptible to targeted attacks [22, 55, 56]. Those users may utilize the *EDDL* system which coordinates the participating user devices to generate mobile malware detection models based on the data collected locally within the campus. Since the users use the same campus LAN for communication, they can be tightly-coupled to complete the training tasks efficiently. One benefit of *EDDL* is that, as shown in our motivation study (§3), DL models trained based on local data perform better in dealing with local testing data (e.g., detecting the targeted attacks to college app users) than those trained based on the data collected globally.

*EDDL* is similar to FL in that both aim at using consumer-grade mobile devices for DL model training. The differences are largely the same as the ones between traditional DDL and FL as discussed previously. Generally speaking, on-site edge-based DDL, such as the one DDL implements, complements FL in that it can better serve users who use the same edge network infrastructure and share similarities like similar interests and behaviors, while FL is better suited for users who are loosely-coupled over wide area networks. In this paper, we focus on training nodes communication topology, parameter update impact and efficiency in performing DDL within resource-constrained edge environments. Our findings should also apply to existing federated learning solutions if training nodes are tightly coupled.

**Synchrony for parameter update**. For the DDL solutions which adopts the centralized approach and utilizes data parallelism, they use one of the three ways to perform parameter synchronization: synchronous, fully asynchronous, and asynchronous with bounded staleness. The synchronous approach synchronizes all the training workers in a way that the training progress is moved forward in a batch-by-batch manner [1, 8, 10]. With the fully asynchronous approach, the parameter server aggregates a set of locally trained parameters/gradients as soon as they are received [9, 10, 14, 19, 35, 45]. The asynchronous with bounded staleness approach limits the progress difference between the fastest and the slowest training workers so as to enable faster convergence [10, 11, 23, 61].

The existing works suggest that the fully asynchronous parameter update approach and the asynchronous with bounded staleness approach work better than the synchronous approach because they are impacted less by struggling workers and scale better [9–11, 14, 23, 35, 38, 45, 50, 61]. However, our experiments suggest that for DDL systems where bandwidth capacity of training nodes is limited, sync PU is preferred over async PU in terms of training time efficiency. We present the further analysis in §4.3 and §6.3 later.

**Dealing struggling training workers**. As indicted above, one major challenge of DDL system that adopt the synchronous parameter update approach is dealing with the existence of struggling workers. Chen et. al. [8] uses backup workers to replace struggling workers. However, this approach may not work well in practice because backup workers may not be available. Project Adam [9] and Li et. al. [35] deals with struggling workers by simply terminating them because losing a small of data does not affect the training. This approach works well for large DNN models with large amount of training data, but may not work well for small models like our scenario.

## 3 THE MOTIVATION STUDY

A key observation motivating us to investigate edge-based DDL training is that ML models trained based on data collected locally (short as *local models*) perform better in local data inference than those trained based on global training data (short as *global models*). We have conducted a study to demonstrate this observation. In this study, we examine how edge-based DDL training would help improve the effectiveness of mobile malware defense solutions. The existing ML-based solutions take app samples from users globally. The universalness of the training data renders these solutions less effective to deal with localized testing data, such as apps that are used by users in the same community (e.g., users in the same university, who are likely to share similar app interests and be targeted by

**Table 1: Selection/distribution of APKs for CAU scenario 1 (for training local models L1 to L6).**

| Category name (index) | APK number | APKs used when training | | | | | |
|---|---|---|---|---|---|---|---|
| | | L1 | L2 | L3 | L4 | L5 | L6 |
| *Tools* (1) | 1,899 | 1,000 | 200 | 400 | 600 | 800 | 600 |
| *Entmt.* (2) | 1,651 | 800 | 1,000 | 200 | 400 | 600 | 600 |
| *Brain and Puzzle* (3) | 1,066 | 600 | 800 | 1,000 | 200 | 400 | 600 |
| *Lifestyle* (4) | 1,034 | 400 | 600 | 800 | 1,000 | 200 | 600 |
| *Education* (5) | 835 | 200 | 400 | 600 | 800 | 1,000 | 600 |
| Total | 6,485 | 3,000 | 3,000 | 3,000 | 3,000 | 3,000 | 3,000 |

**(a) Benign APKs**

| Family name (index) | APK number | APKs used when training L1 to L6 |
|---|---|---|
| *Mecor* (3) | 1,820 | 875 |
| *Youmi* (4) | 1,301 | 625 |
| *Fusob* (5) | 1,277 | 613 |
| *Kuguo* (6) | 1,199 | 576 |
| *BankBot* (7) | 648 | 311 |
| Total | 6,245 | 3,000 |

**(b) Malware APKs**

**Table 2: Selection/distribution of APKs for CAU scenario 2 (for training local models SL1 and SL2).**

| Category name (index) | APK number | APKs used when training SL1 | APKs used when training SL2 |
|---|---|---|---|
| *Entmt.* (2) | 1,651 | 360 | 800 |
| *Education* (5) | 835 | 360 | 250 |
| *Communication* (14) | 583 | 360 | 250 |
| *Music & Audio* (16) | 534 | 360 | 250 |
| *Social* (17) | 465 | 360 | 250 |
| Total | 4,068 | 1,800 | 1,800 |

**(a) Benign APKs**

| Family name (index) | APK number | APKs used when training SL1 and SL2 |
|---|---|---|
| *Fusob* (5) | 1,277 | 543 |
| *Kuguo* (6) | 1,199 | 510 |
| *BankBot* (7) | 648 | 276 |
| *Jisut* (8) | 560 | 238 |
| *DroidKungFu* (9) | 546 | 233 |
| Total | 4,230 | 1,800 |

**(b) Malware APKs**

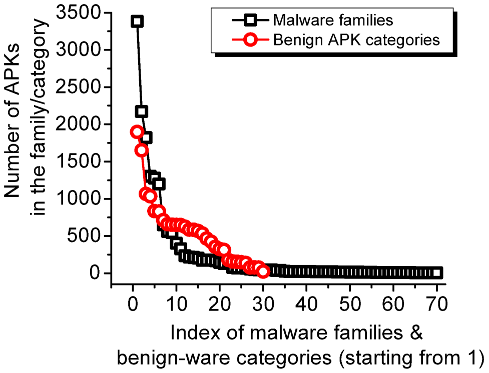

**Figure 1: Distributions of malware family sizes and benign APK category sizes.**

the same security threats). We simulate edge-based mobile malware defense scenarios where malware detection/classification (d/c for short) models are constructed by training on app data collected from the users of the same community.

**The Android APK dataset**. Our study is based on a large Android app dataset, which consists of 16,710 malware APKs and 16,425 benign APKs. The malware APKs are obtained from the Android Malware Dataset [58], and are classified into 70 malware families. The benign APKs were crawled from Google Play during the period from April to July 2014, and can be put into 30 categories based on app functionalities (e.g., News, Shopping, Finance, etc.). We index all the malware families in an ascending order of family size (i.e., number of APKs in the family), and did the same thing for all the benign APK categories. Figure 1 shows the distributions of malware family sizes and benign APK category sizes.

***PerNet*: a deep neural network for mobile malware detection and classification**. We design a multilayer perceptron (MLP) [59] based DNN, which is named "*PerNet*" because it determines whether an app is a malware based on the permissions used in the app. The input to *PerNet* is the subject APK's Boolean vector of Android permissions. *PerNet* examines all the permissions defined by the Android system, which are 427 in total. Therefore, there are 427 neurons in the input layer. The input layer connects to five fully-connected hidden layers, whose number of neurons are 512, 512, 256, 256, and 128 respectively. The output layer has 71 neurons, which contain the Boolean classification results for the 71 classes (i.e., the 70 malware families and the benign class). The activation function used in *PerNet* is rectified linear unit (ReLU) [7].

**Generating the local models**. We simulate the scenarios of app usage in different communities based on which local malware d/c models are generated. Since users from the same community are likely to use similar types of apps, we select several benign APK categories (out of the 30 categories) to simulate the commonly used app types in a community. We also choose several malware families (out of the 70 families) to simulate the malware attacks targeting the users in the community. Specifically, we simulate two *community app usage* (short as "CAU" below) scenarios as follows.

- For CAU scenario 1 (summarized in Table 1), we pick the five largest benign categories, which together have 6,485 APKs, and select 3,000 of them for training local models. To match the total number of APKs in the five benign categories, we choose five malware families, which contain 6,245 APKs in total, and also select 3,000 of them in training local models. To evaluate the effects of different training data distributions on malware d/c accuracy, we generate six local models (L1 to L6), of which the 3,000 training benign APKs follow different distributions among the five benign categories. For example, the 3,000 benign APKs are linearly distributed among the five categories when training local models L1 to L5. For the local model L6, the 3,000 benign APKs are evenly divided among the 5 categories. All the 6 local models share the same set of

**Table 3: Inference testing results of CAU scenario 1 and 2.**

| | CAU scenario 1 | | | | | | | CAU scenario 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | L1 | L2 | L3 | L4 | L5 | L6 | Global | SL1 | SL2 | Global |
| Classification accuracy | 99.0% | 98.97% | 98.94% | 98.95% | 98.95% | 98.94% | 96.93% | 99.67% | 99.72% | 95.37% |
| Classification recall | 97.97% | 97.7% | 97.83% | 97.83% | 97.85% | 97.83% | 93.75% | 99.35% | 99.45% | 90.89% |
| Classification precision | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100 % | 100% |
| Detection accuracy | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 99.94% | 99.94% | 99.94% |

3,000 training malware APKs, which are randomly chosen from the five malware families. We select training APKs in the above way because for the same set of malware APKs, the detection accuracy may vary with the environment they are targeting. Different environments can have different mixtures of benign APKs, depending on the interests of the community.

- For CAU scenario 2 (summarized in Table 2), we simulate app usage in schools. We select five benign app categories which are commonly used by students, and use 1,800 APKs from these categories for training the local models. Five malware families whose total number of APKs match that of the five benign categories are selected to train the local models. Two local models (SL1 and SL2) are generated for the scenario 2: SL1 is trained on benign APKs which are uniformly distributed among the five categories, and SL2's training benign APKs are dominated by one benign category. Similar to CAU scenario 1, the training malware APKs are randomly chosen from the five malware families.

**Local testing data**. The local testing data are drawn from the benign categories and malware families which are associated with the CAU scenarios. We randomly choose 20% from each of these categories/families that were not used in the training process. As a result, for CAU scenario 1, we select 2,544 APKs (1,297 benign and 1,247 malware) as the testing APKs; for CAU scenario 2, 1,654 APKs (814 benign, 840 malware) are selected as the testing APKs.

**Generating the global model**. The global mode is trained based on all the APKs in the dataset except the testing APKs.

**Study results**. Table 3 presents the simulation study results. An app is considered to be correctly *classified* if the *PerNet* model correctly puts the app into one of the 71 classes (i.e., benign and the 70 malware families). An malware app is considered to be correctly *detected* if the model labels it to one of the malware families (regardless the correctness of the labeling). An benign app is considered to be correctly *detected* if the model rules it as benign.

We can see from Table 3 that for CAU scenario 1, the six local models (L1 to L6) enjoy a classification accuracy around 99%, whereas the global model's classification accuracy is 96.9%. For CAU scenario 2, both local models (SL1 and SL2) achieve more than 99.6% of classification accuracy, which is over 4% higher than the global model. All the local models have significant higher classification recall than global model, while both the local and global models achieve 100% classification precision. This indicates that local models perform better than global models on correctly classifying true malware targeting the communities, but both are unlikely to misclassify a benign app as malware.

It is worth noting that there have been several recent studies which achieve personalized DL models via different approaches, such as meta-learning [28], multi-task learning [51, 63] and transfer learning [57]. These solutions usually first train a global model which is then adapted to individual users based on their local data. *EDDL* complements these solutions in that it does not require training of global models which may not be feasible due to lack of global data. With *EDDL*, local-community-oriented DL models can be generated based on the data collected locally.

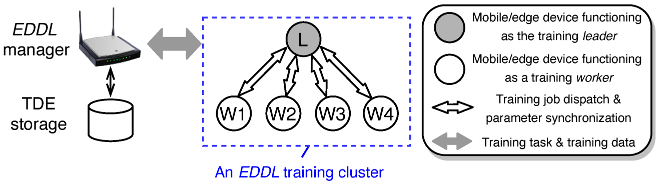
EDDL manager
TDE storage
L
W1
W2
W3
W4
An EDDL training cluster
Mobile/edge device functioning as the training leader
Mobile/edge device functioning as a training worker
Training job dispatch & parameter synchronization
Training task & training data

**Figure 2: The basic setup of *EDDL*'s distributed training.**

# 4 *EDDL* SYSTEM DESIGN

## 4.1 The setup of *EDDL*'s distributed training

The basic setup of *EDDL*'s distributed training is shown in Figure 2. There are three main entities: the *EDDL training cluster*, the *EDDL manager*, the *training data entry storage*.

The ***EDDL* training cluster** consists of edge or mobile devices that are participating in a training task. In the following, we refer to these devices as the training nodes. There are two roles in for the training nodes: leader and worker. As introduced in §1, individual worker nodes train on their own portions of training data entries (TDEs) using mini-batch Stochastic Gradient Descent (mbSGD) [36, 60], and synchronize the updated model parameters to the leader node. The leader node acts as a parameter server [10, 23, 35, 37] which aggregates and sends back the parameters received from the workers. We refer to the number of worker nodes in a training cluster as the *size of the cluster*. In the example shown in Figure 2, the training cluster has a size of four.

The ***EDDL* manager** performs three tasks: working as the portal device to collect training data entries (TDEs) from users within the same community, initializing training tasks, and relaying TDEs to the training cluster during a training. Edge infrastructure devices, such as wireless APs, are ideal to work as *EDDL* managers because the *EDDL* manager needs to communicate to the leader device of a training cluster, which can be any user device, and requires minimal amount of computation.

A **Training data entry (TDE)** is a piece of labeled data which consists of a feature vector and a classification label. Take the the edge-based malware defense (§3) as an example, the feature vector of a TDE is the permission vector of an app, and the label tells

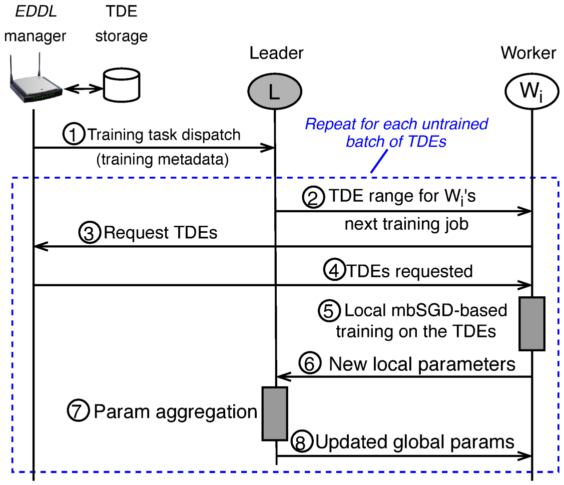
EDDL manager
TDE storage
Leader
Worker
L
W_i
Repeat for each untrained batch of TDEs
① Training task dispatch (training metadata)
② TDE range for W_i's next training job
③ Request TDEs
④ TDEs requested
⑤ Local mbSGD-based training on the TDEs
⑥ New local parameters
⑦ Param aggregation
⑧ Updated global params

**Figure 3: Dynamic training data distribution in *EDDL*.**

whether the app is benign or the type of malware. Another example is that for an edge-based news recommendation, the feature vector of the TDE could be user preferences and time when a piece of news is viewed, and the label is the headline of the news. The **TDE storage** stores TDEs contributed by users of the same community. Compared to cloud storage, the TDE storage is owned and trusted by the community users.

## 4.2 Dynamic training data distribution

Existing distributed DNN training solutions usually statically partition training data among the training workers [8, 11, 14, 35, 61, 65]. The static partitioning approach is unlikely to work well for *EDDL* because of two reasons. *First*, as discussed previously, training workers in our scenario have high variation in available computing resource. As a result, static training data partitioning can lead to large differences in time needed for individual workers to finish one epoch of training (i.e., to consume their partitions of data once), which would slow down the overall training significantly. *Second*, static training data partitioning cannot well deal with training node changes, such as nodes leaving and joining, which is not uncommon in our scenario.

To address the above problems, *EDDL* adopts a dynamic training data distribution approach, which allows the leader to distribute training data based on the training progresses on different workers. The overall training workflow with the dynamic training data distribution mechanism is shown in Figure 3. The *EDDL* manager initiates a training task, whose goal is to generate a new or updated malware d/c model, by dispatching the training metadata, which contains the indexes of all the TDEs, to the leader of a training cluster (step ①). The leader then dispatches training jobs, each of which requests a worker to perform the forward pass and backpropagation on one batch of training data as in mini-batch stochastic gradient descent (mbSGD) [36, 60]. For each worker which is ready for a new training job, the leader sends it the indexes of a TDE batch (step ②). The worker then fetches the batch of TDEs from the *EDDL* manager (steps ③ and ④), and performs local training on them (step ⑤). After the local training is completed, the worker sends the new local parameters to the leader (step ⑥), which aggregates them with other worker's latest local parameters or the latest global parameters (depending on (a)synchrony of the aggregation) to generate the updated global parameters (step ⑦), and sends them back to the worker. The steps ② to ⑧ above repeat until a epoch of training is completed (i.e., the entire set of TDEs are trained on once). Multiple epochs of training are needed to obtain a model with desired accuracy.

The above design dynamically dispatches training data range for each training job to workers based on their training progresses instead of statically partitioning the data among them. As a result, fast-working workers do not need to wait for struggling workers to finish their shards of data in an epoch of training, which consequently lead to improved training time efficiency.

To reduce the overhead of distributing training data in real time, we adopted an optimization in our system implementation that after the leader device receives the indexes of all the TDEs from the *EDDL* manager (i.e., step ①), it broadcasts this info to all the workers. Each worker then prefetches the whole set of TDEs before the training process starts. This way, each worker device can start the local training right after receiving info of the TDE batch it needs to work on from the leader (i.e., step ②), and therefore to forego steps ③ and ④ to speed up the whole training process. Here prefetching the entire training dataset is a feasible optimization for edge-based DDL scenarios due to two reasons. First, it is a one-time operation which occurs before the training starts and thus has little impact on overall training time. Second, compared to conventional DDL which trains ML models based on a massive amount of data collected globally, the training data size of edge-based DDL is relatively small because the data are collected from only one local community.

## 4.3 Scaling up *EDDL* training cluster size

As discussed in §1, a challenge for *EDDL* to apply DDL training in an edge environment is the difficulty of scaling up the number of training nodes in a cluster. Our approach of addressing this challenge is three-fold: *first*, we adopt synchronous parameter update as the default approach for aggregating parameter updates from workers; *second*, we design a practical method to detect if a training cluster is saturated when scaling up its size; *third*, we propose the approach of adaptive leader role splitting to further scale up training cluster size after cluster saturation is reached. In the following, we first introduce the concept of synchronous and asynchronous updates, and then the three designs of scaling up *EDDL* training cluster size.

**(A)synchrony of parameter update**. The operations performed in steps ⑥, ⑦ and ⑧ in Figure 3 are referred to as *parameter update* (PU). *EDDL* supports two types of PU: synchronous and asynchronous. With *synchronous* PU, after the leader dispatches training jobs to the workers, it waits for all the workers' new local parameters before performing aggregation on them and sending back the new global parameters. With *asynchronous* PU, whenever the leader receives local parameters from a worker, it aggregates them with the latest global parameters to generate new ones and sends them back to the worker. If the leader receives a worker's local parameters while an aggregation process is under way, it restarts the aggregation to include the newly-arrived parameters.

***EDDL* adopts synchronous PU by default**. The reason of this design is that, according to our experiments (§6.3 later), we find that in distributed deep learning systems where bandwidth capacity of training nodes is limited, sync PU is preferred over async PU in terms of training time efficiency. Our finding here contradicts many recent works which suggest sync PU is inferior to fully async PU [9, 14, 35, 38, 45, 50] or async PU with bounded staleness [10, 11, 23, 61]. We analyze the cause of the disagreement below.

With the parameter server architecture, all the worker nodes in a training cluster send their updated model parameter to the leader node for aggregation. As the number of worker nodes increases, the network capacity at the leader node will become saturated, after which time further increasing worker nodes would only deteriorate the overall training time because of the fast increasing network time for parameter synchronization. We call the number of workers in a training cluster when the leader node's network bandwidth is saturated the "*saturation size*" of the training cluster.

In the meantime, although async PU takes less time in one epoch of training ("epoch time" for short) than sync PU, both our experiment results and the recent works suggest that async PU needs to take more epochs to converge (i.e., to reach the desired training accuracy). For a given training cluster, async PU incurs less overall training time (which is the product of epoch time and epoch number) than sync PU when the cluster size is large enough, such that the reduction in epoch time outweighs the disadvantage in epoch number. We name this sufficiently large cluster size with which async PU incurs less overall training time than sync PU the "*inflection size*"[1] of the training cluster.

For edge-based DDL, the inflection size of a training cluster is likely to be notably higher than the cluster's saturation size because of the limited network bandwidth at the leader node. In other words, when scaling up the training cluster size, the leader node's bandwidth is saturated long before the inflection size is reached. As a result, async PU usually has worse performance than sync PU in terms of overall training time. The experiment results which support the above analysis will be presented later in §6.3.

**Detecting training cluster saturation size**. Since increasing the number of worker nodes in a training cluster after the cluster's saturation size is reached would deteriorate the overall training time, it is important to be able to detect when saturation size is reached when scaling up the size of the cluster. However, detecting training cluster saturation size is not trivial because the saturation size varies as the available network bandwidth at the leader node changes. Therefore, an online approach is needed rather than an offline one.

We observe that before a training cluster is saturated, the following inequation should hold:

$$B_l > \frac{S_{param}}{t_{sync}} \times n, \tag{1}$$

where $B_l$ is the leader's available network bandwidth, $n$ is the number of workers, $S_{param}$ is the size of parameters which are needed to be synchronized for a batch of training, and $t_{sync}$ is the average time needed for parameter synchronization in a batch of training. The above inequation means before cluster is saturated, $B_l$ has surplus after accommodating bandwidth consumption from the workers.

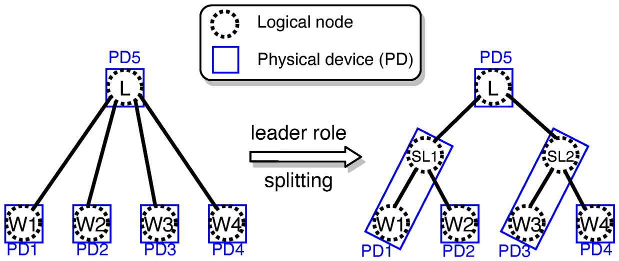

**Figure 4: Leader role splitting example in a 5-node cluster.**

We define *bandwidth usage coefficient* (*BUC*) as:

$$BUC = \frac{n}{t_{sync}} \tag{2}$$

According to the inequation (1), the following should hold before saturation is reached:

$$BUC < \frac{B_l}{S_{param}}, \tag{3}$$

which means if a new worker were added, the new *BUC* would be larger than the current one. Therefore, our way of detecting cluster saturation as workers are added is allowing leader node to monitor how the *BUC* value changes after adding a new worker: if the new *BUC* is larger than the old value, it means the cluster has not yet been saturated; otherwise the cluster is saturated. It is worth noting that the value of $t_{sync}$ can be easily obtain by the leader node during runtime. The experiment later in §6.4 shows that the runtime metric of *BUC* is able to accurately detect when the saturation size is reach while increasing the size of a training cluster.

**Adaptive leader role splitting**. One of our ongoing efforts is investigating how to adjust the topology of a training cluster after the saturation size is reached, such that further scaling up the cluster can be beneficial. Our current design adopts a two-level tree structure where the leader locates at the root level and new workers are always directly connected to the leader. The drawback of this design is that cluster saturation size is determined by the available bandwidth of a single node (i.e., the leader). A promising direction is to adaptively (e.g., when leader's network bandwidth is saturated) split the current leader role among several worker nodes which can then work as "sub-leaders". Figure 4 shows an example of leader role splitting (LRS) in a training cluster with 5 devices. In the example, the leader device (PD5) splits its leader role among PD3 and PD5 to allow them to work as sub-leaders which aggregate parameters for a portion of the workers, and relays the partially-aggregated parameters to the top leader for final aggregation. In order to enable LRS without requiring more physical devices, PD3 and PD5 also work as workers (each of which is implemented as a separate process from the sub-leader process). The preliminary

[1] It is named this way because when the number of worker devices in the training cluster is smaller than the cluster's inflection size, sync PU achieves better overall training time than async PU. When there are more worker devices in the training cluster than the cluster's inflection size, and async PU performs better.

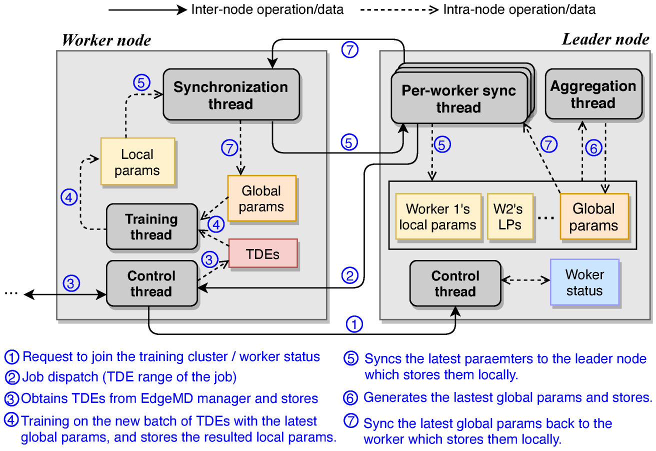
Inter-node operation/data
Intra-node operation/data
Worker node
Leader node
Synchronization thread
Local params
Global params
Training thread
TDEs
Control thread
Per-worker sync thread
Aggregation thread
Worker 1's local params
W2's LPs
Global params
Control thread
Woker status
① Request to join the training cluster / worker status
② Job dispatch (TDE range of the job)
③ Obtains TDEs from EdgeMD manager and stores
④ Training on the new batch of TDEs with the latest global params, and stores the resulted local params.
⑤ Syncs the latest paraemters to the leader node which stores them locally.
⑥ Generates the lastest global params and stores.
⑦ Sync the latest global params back to the worker which stores them locally.

Figure 5: Implementation of leader and worker nodes.

evaluation result (§6.5) shows that LRS is a promising approach to scaling up training cluster size for edge-based DDL where training cluster saturation size is usually small.

## 4.4 Training cluster formation and training nodes management

Owners of edge devices have strong incentives to participate in the distributed training for the following three reasons. *First*, the edge devices may greatly be benefited by using the trained models. *Second*, organizations such as companies and schools, which can be benefited most by edge-based DDL solutions (such as the one presented in §3), may mandate their employees to participate. *Third*, the training can be done while the edge devices are idle, which alleviates the concern of training interfering normal workloads on the devices.

Formation of a training cluster is initiated by the *EDDL* manager when it has collected sufficient TDEs. Edge devices that are selected to participate in the training process send their hardware specification and runtime resource statistics, such as CPU and network interface utilization, to the *EDDL* manager, which uses such information to pick a device as the leader of the training cluster. The leader node of the training cluster is responsible for training nodes management which has two main goals. *First*, the leader node needs to be aware of available computation resource changes on different training nodes, so that the leader role can be transferred to the node with the most abundant available computation resource. *Second*, training node dynamics, such as joining and leaving the training cluster, need to be properly managed. This goal is easy to achieve because training data is dynamically distributed to workers in a batch-based manner and workers synchronize model parameters with the leader after each batch of training. As a result, the leader does not need to maintain states for workers. Each worker can be treated the same regardless when it joined the cluster.

## 5 *EDDL* SYSTEM IMPLEMENTATION

**Prototype system hardware**. We implemented a prototype system using two single-board computer (SBC) embedded platforms which have similar computation capability as today's edge devices, such as smartphones and access points. One such platform is ODROID-XU4 [21], which is equipped with a 2.1/1.4 GHz 32-bit ARM big.LITTLE octa-core processor and 2GB memory. The other platform is Raspberry Pi 3 (RP3) Model B board [49], which comes with an ARM 1.2 GHz 64-bit quad-core processor and 1 GB memory.

**Software environment**. The operating system running on the above SBC platforms is Ubuntu 18.04 with Linux kernel 4.14. We use Dlib [16], a C++ library which provides implementations for a wide range of machine learning algorithms and tools, to implement the core deep learning functionalities, such as SGD. We choose the Dlib library because it is written in C/C++, and thus can be easily and natively used by embedded devices for good performance.

**Leader/worker node implementation**. Figure 5 demonstrates the implementations of leader and worker nodes. To support dynamic leader role transfer, the implementation logic of both leader and worker are carried in each ODROID and RP3 device, such that all training nodes can work as leader or worker depending on runtime needs.

The implementation logic of both leader and worker nodes consists of multiple threads, each of which is implemented as a POSIX thread to complete a specific functionality. For a new worker node to join a training cluster, the control thread in the worker node sends a join request to the control thread in the leader node (step ①). For each new worker node, the leader launches a synchronization thread, through which it starts a new job for the worker by sending the indexes of a batch of TDEs to the worker's control thread (step ②). The worker then fetches the said TDEs from *EDDL* manager and stores them locally (step ③). Our implementation adopts the optimization mentioned before, which allows worker node to prefetch the entire set of TDEs, and thus avoids the need of fetching TDEs from *EDDL* manager for every new job. The worker control thread also communicates its status (e.g., CPU and network bandwidth utilization) to the leader thread, which can use the info to manage training nodes (e.g., to initialize leader role transfer). When a new batch of TDEs become available, the training thread within the worker starts the forward and backward training passes on the new TDEs, and stores the resulted (local) parameters (step ④), which are fetched and synchronized to the leader node by the worker's synchronization thread (step ⑤). The leader node is responsible for aggregating the locally trained parameters from different workers to generate the latest global parameters (step ⑥). For synchronous PU, the leader waits for the local parameters from all the workers until aggregating them to generate the global parameters. For asynchronous PU, the leader aggregates the newly received local parameters with the current global parameters to generate the latest ones without waiting for other workers. Once new global parameter become available, the leader sends them back to individual workers through the corresponding per-thread sync thread (step ⑦). In our implementation, the TDE indexes of a new training job are piggybacked when leader sends updated global parameters back to workers (i.e., the info sent in steps ② and ⑧ is delivered with one network message). After worker node receives the latest global parameters and the indexes of the next batch of TDEs, it starts a new training cycle by repeating the steps ② to ⑦ above.

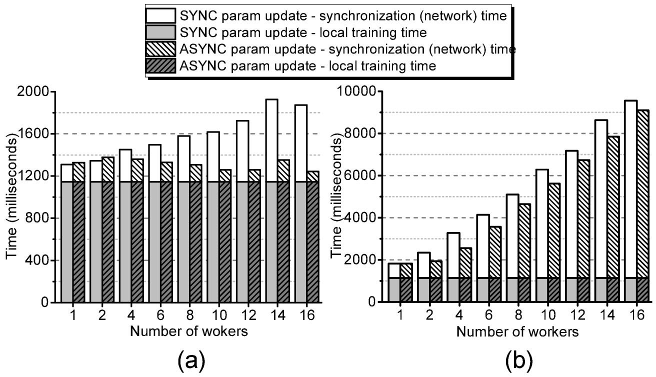

**Figure 6: Time for training a batch of 128 TDEs (y-axis) with training clusters of difference sizes (x-axis). (a) Cluster node's network bandwidth is 1000 Mbps. (b) Cluster node network's network bandwidth is 100 Mbps.**

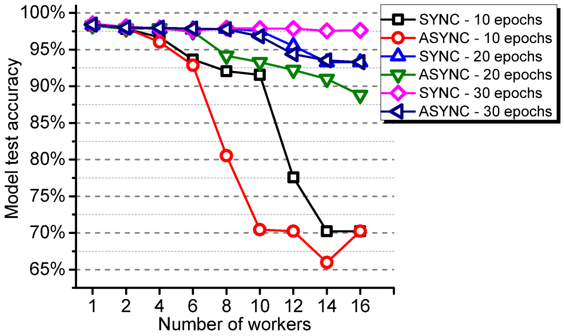

**Figure 7: Model test accuracy after 10/20/30 epochs of training (y-axis) with training clusters of different sizes (x-axis).**

# 6 EVALUATION AND INSIGHTS

We conducted experiments to evaluate the real-world performances of our *EDDL* prototype system.

**Training devices**. The training nodes used in the evaluation experiments are ODROID XU4 devices.

**Network setting**. To enable controllable network bandwidth between leader and worker devices, the training nodes are connected via an Ethernet switch, which emulates wireless communication speeds by varying the bandwidth.

**DNNs and training datasets**. We conducted experiments on two DNN models and their associated datasets. The first DNN is the *PerNet* introduced in §3. We trained *PerNet* using 6,656 TDEs (i.e., TDEs derived from 6,656 APKs which were selected following the same way of training the local model L1 of the CAU scenario 1 (see Table 1 in §3 previously). The size of the 6,656 TDEs is 6.3 MB. As a comparison, the size of all the 33,135 TDEs used in training/testing the global model (§3) is about 30.7MB. The testing APKs were also selected using the same way as described in §3. The second DNN is the LeNet [33] and was trained using the MNIST dataset [34] which has about 60,000 grey-scaled images of handwritten digits with about 47 MB in total. Both of the datasets chosen have around several thousands to tens of thousands data entries, which match the scale edge-based DDL scenarios where models are trained based on the data collected from local communities.

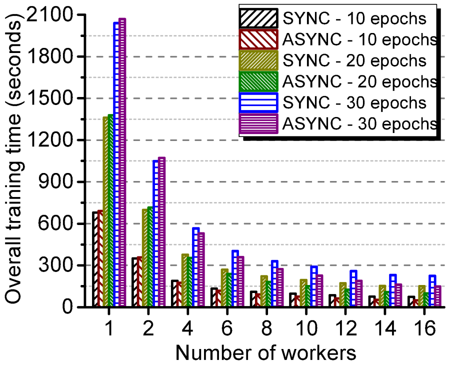

**Figure 8: Training time for 10/20/30 epochs (y-axis) with training clusters of different sizes (x-axis) (1000 Mbps network bandwidth).**

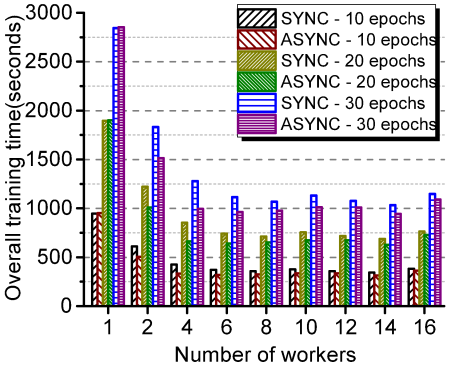

**Figure 9: Training time for 10/20/30 epochs (y-axis) with training clusters of different sizes (x-axis) (100 Mbps network bandwidth).**

The findings presented in the following are supported by the experiment data on both *PerNet* and LeNet. Due to the space limit, we use the data obtained from the *PerNet* experiments to discuss the findings.

## 6.1 Batch training time

Figure 6 shows the average time needed to train a batch of 128 TDEs, which consists of local training time and (parameter) sync time, with different number of workers. The measurement was performed under two network bandwidth settings for cluster nodes: 1000 Mbps and 100 Mbps. From the figure we can see that worker's local training time stays unchanged regardless parameter update (PU) synchrony or cluster node bandwidth setting differences. The figure shows the following results about the relationship between batch training time PU synchrony.

• *Async PU incurs smaller batch training time than sync PU.* This is because sync PU requires each worker to wait for the leader to

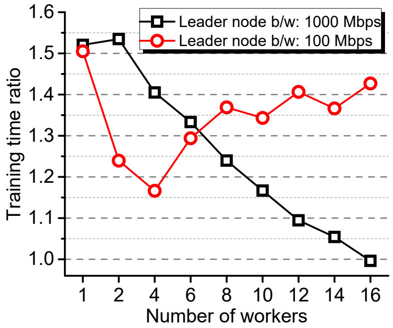

**Figure 10: Training time ratio of the ASYNC-30 model over the SYNC-20 model (i.e., $\frac{ASYNC-30}{SYNC-20}$ training time ratio) (y-axis) with training cluster with different sizes (x-axis).**

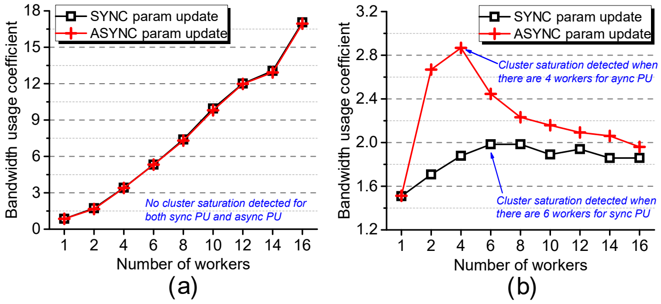

**Figure 11: Bandwidth utilization ratio (*BUC*) (y-axis) vs. cluster size (x-axis). (a) When cluster node's bandwidth is set to 1000 Mbps, no cluster saturation with both sync and async PU. (b) When cluster node bandwidth is set to 100 Mbps, cluster saturation is detected when cluster size increases to 6 and 4 for sync PU and async PU respectively.**

aggregate parameter updates from the entire cluster before starting the next batch of training, while async PU does not have this requirement.

• *The leader node's network bandwidth capacity is shared among the workers. Thus, after the leader's bandwidth is saturated, adding more workers would proportionally increase sync time as well as batch training time.* Figure 6 also shows that batch training time with cluster node bandwidth of 100 Mbps is significantly higher than that with cluster node bandwidth of 1000 Mpbs, especially for clusters with more workers. The difference is caused by the increase of (parameter) sync time. For example, when 16 workers are used, the ratios between sync time and batch training time for sync/async PU are 0.88/0.87 with bandwidth of 100 Mbps, compared to 0.38/0.07 with bandwidth of 1000 Mbps.

## 6.2 Model accuracy

Figure 7 shows the results of model test accuracy after 10, 20 and 30 epochs of training with different number of workers. We can see the following results from the figure.

• *It requires more epochs of training for async PU to converge (i.e., to reach a desired model test accuracy) than sync PU.* For example, the models trained with async PU after 30 epochs (i.e., the "ASYNC-30 epochs" line) have almost the same accuracy as the models trained with sync PU after 20 epochs (i.e., the "SYNC-20 epochs" line). This finding is consistent with recent literature [1, 8, 12].

• *Sync PU achieves better model test accuracy than async PU under the same condition (i.e., same number of training epochs and same training cluster size).* For example, models with SYNC-30 achieve notably higher accuracy than those with ASYNC-30 when the cluster size is larger than 8.

• *With the same number of epochs of training, model test accuracy declines as size of the training cluster increases, but the variation range is much smaller when the model is trained for more number of epochs.* In other words, it requires more epochs of training for large training clusters to reach a desired model test accuracy. For example, after 20 epochs of training using async PU (i.e., ASYNC-20), an accuracy of 95% can be achieved with a training cluster of 4 workers, while only an accuracy of 88% is achieved with a 16-worker cluster.

## 6.3 Overall training time

Figure 8 and Figure 9 demonstrate how overall training time is affected by the different factors. We have the following finding about overall training time and cluster size.

• *Increasing number of workers is helpful for reducing overall training time until network bandwidth of the leader node becomes a bottleneck.* For example, as shown in Figure 9, with cluster node network bandwidth set at 100 Mbps, increasing training cluster size beyond 6 does not only does not reduce, it may also increase, the overall training time (e.g., the overall training times with the 16-worker cluster are larger than those with smaller clusters)

• *In DDL systems where network bandwidth of the training nodes is limited, sync PU is preferred over async PU in terms of training time efficiency.* For example, recall that in Figure 7, it was shown that the model trained with async PU for 30 epochs (short as "ASYNC-30 model" below) achieves almost the same test accuracy as the model trained with sync PU for 20 epochs (short as "SYNC-20 model" below) under training clusters of different sizes. Here, according to Figure 8 and 9, ASYNC-30 model needs more time than SYNC-20 model in most of the cases.

We find that the time difference for training SYNC-20 and ASYNC-30 models actually varies depending on network bandwidth of the leader as well as size of the training cluster. Figure 10 plots the training time ratio of ASYNC-30 model over the SYNC-20 model for training cluster of different sizes. We can see that when the leader node's network bandwidth is limited (i.e., 100 Mbps), the $\frac{ASYNC-30}{SYNC-20}$ training time ratio decreases toward 1 as number of workers increases from 1 to 4. However, when there are more than 4 workers in the training cluster, the $\frac{ASYNC-30}{SYNC-20}$ training time ratio increases as cluster size grows. The reason is that network bandwidth of the leader is saturated when there are 4 workers (i.e., the saturation size of the cluster is 4). Thus, adding more workers would cause a significant increase in network sync time when training a batch of TDEs. The increase of batch training time in this case overshadows the benefit brought by adding more workers (which is reducing the number of batches to be trained by each worker).

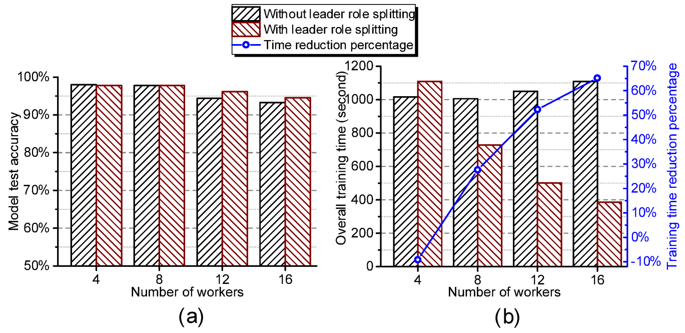

**Figure 12: Leader role splitting's effect on (a) Model accuracy, and (b) Training time.**

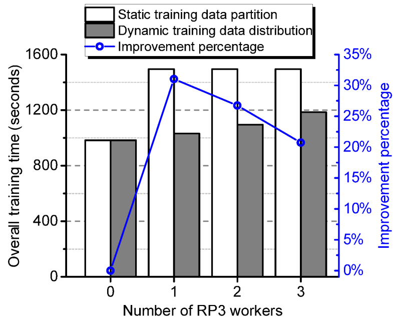

**Figure 13: TDEs dynamic distribution vs. static partition.**

## 6.4 Training cluster saturation size detection

Figure 11 shows how the *BUC* value changes in the experiments discussed previously. When leader node's bandwidth is sufficient (i.e., 1000 Mbps), *BUC* increases as number of workers increases, which matches our observation that there is no cluster saturation with sync and async PU when leader's bandwidth is 1000 Mbps. When leader node's bandwidth is limited (i.e, 100 Mbps), *BUC* increases as the worker number grows until reaching 6 and 4 workers for sync and async PU, which again matches our observation on cluster saturation size with 100 Mbps bandwidth (which can be seen in Figure 9 that the overall training time for sync and async PU stops decreasing when there are more than 6 and 4 workers in the cluster respectively).

## 6.5 Leader role splitting (LRS) for scaling up training cluster size after saturation

Figure 12 shows the effect of LRS for training clusters with 4/8/12/16 workers. In this experiment, training devices' network bandwidth is set to 100 Mbps, and async PU is used for the case of without LRS, of which the saturation size is 4 workers as we analyzed previously. For the case of with LRS, sync PU is used between workers and sub-leaders, and async PU is used between sub-leaders and the top leader. If the cluster topology with LRS in Figure 4 is notated as "1-2-4" (meaning 1 top leader, 2 sub-leaders, and 4 workers which are evenly under the sub-leaders), the cluster topologies with LRS in this experiments are "1-2-4", "1-2-8", "1-3-12" and "1-4-16", respectively. The results reported here are for 30 epochs of training. Figure 12 (a) suggests that LRS achieves similar to slightly better model accuracy as when LRS is not applied. Figure 12 (b) demonstrates that topologies with LRS incurs much less training than those without, especially when the severity of cluster saturation is high. From the above results it can be seen that LRS is a promising approach to deal with saturated clusters. We are working on addressing other fundamental challenges, such as how to practically and systematically achieve the optimal topology.

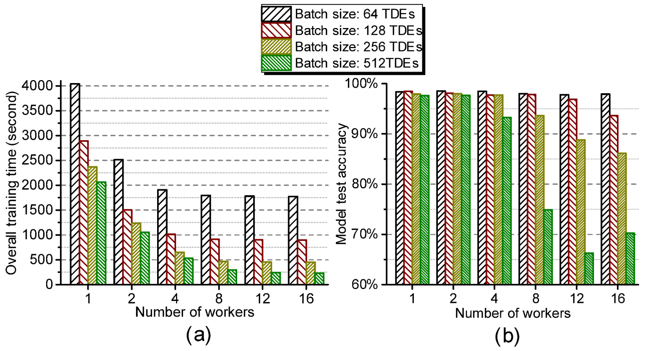

**Figure 14: Impact of batch size on (a) training time, and (b) model accuracy.**

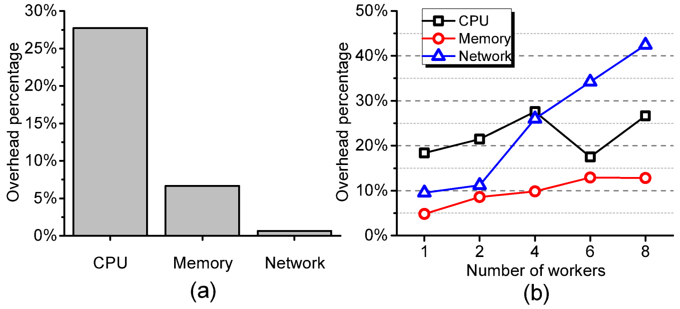

**Figure 15: Overhead on normal workload caused by training workload on (a) worker node, and (b) leader node.**

## 6.6 Evaluating the effects dynamic training data distribution

We compared our dynamic TDEs distribution approach with the traditional static training data partition approach. In this experiment, an 8-worker cluster is used to trained a *PerNet* model with aysnc PU for 30 epochs. We allow some of the workers to be "struggling workers" by using RP3 devices, which have significantly lower processing power than ODROID devices. With static training data partition, the training TDE set is evenly divided into eight parts, each of which is fed to an individual worker. Figure 13 shows the result of training time for the two different approaches. We can see that our TDEs dynamic distribution approach is notably more time-efficient than the static partition approach when there are struggling workers in the training cluster.

## 6.7 Impact of training batch size

*EDDL* adopts mbSGD for training the *PerNet* model. Here we evaluate the impact of batch size on training time (Figure 14 (a)) and model accuracy (Figure 14 (b)). The experiment performs training for 30 epochs with different batch sizes and cluster sizes. Cluster node's network bandwidth is set to 100 Mbps, and async PU is used. Figure 14 (a) shows that training time decreases as batch size gets bigger. This is because larger batch size means less number of parameter aggregation and synchronization. However, as demonstrated in Figure 14 (b), training with large batch size can lead to significant model accuracy drop for training clusters with four or more workers. The default batch size of our prototype system is 128 TDEs, which strikes a good balance between training time and model accuracy.

## 6.8 System overhead

The training nodes in our system are edge devices which have their own normal workloads. Here we evaluate how *EDDL* training workload affects the normal workloads on these edge devices. We construct three normal workloads which are heavy on usage of CPU, memory, and network respectively. The CPU and memory workloads are constructed using `sysbench` which is a scriptable database and system performance benchmark [15]. The network workload is constructed using the `iperf` utility [26]. We run each of the above workload while issuing a training workload which trains a *PerNet* model using ODROID devices. The network bandwidth of cluster nodes is set to 100 Mbps. Figure 15 (a) shows the result for worker node. The major overhead caused by the training is seen on the CPU workload, which is a slowdown about 27%. This is because the main computation resource used by a worker node is CPU. Figure 15 (b) shows the overhead results for leader node. We can see that when the number of workers increase from 1 to 8, the CPU workload and the memory workload suffer from slowdowns of 5%-13% and 19%-26%, respectively. The major overhead for leader node is on the network workload, which is about 10% to 43%. Given the above result, it is preferable to run training workload on edge devices when they are idle, especially for devices serving as training leaders.

## 7 CONCLUSION

In this paper, we advocate edge-based DDL with which machine learning models are trained based on the data collected locally from users serviced by the same edge infrastructure. We designed *EDDL*, an edge-based DDL system which addresses multiple challenges of performing DDL on edge environments where computing devices are resource-constrained embedded devices connected via consumer-grade wireless networks. The proposed *EDDL* system has been implemented with ARM-based ODROID-XU4 and Raspberry Pi 3 Model B boards. We further conducted a case study of enabling edge-based mobile malware defense on our 16-device *EDDL* prototype system, which demonstrated the effectiveness and efficiency of the *EDDL* system.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for their tremendously valuable feedback and Dr. Diego Perino for shepherding the paper revision. This work was supported in part by NSF Award #1943269.

## REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI* (2016).

[2] Ahmed, M. What is The Actual Speed of My WI-FI Network? https://blog.vtsl.net/vtsl-blog/what-is-the-actual-speed-of-my-wifi-network.

[3] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. SCONE: Secure linux containers with intel SGX. In *USENIX OSDI* (2016).

[4] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with haven. In *USENIX OSDI* (2014).

[5] Beachx, A. Targeted Mobile Malware Aimed at Android. https://www.trushieldinc.com/targeted-mobile-malware-aimed-at-android/.

[6] Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konecný, J., Mazzocchi, S., McMahan, B., Overveldt, T. V., Petrou, D., Ramage, D., and Roselander, J. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems (MLSys)* (2019).

[7] Brownlee, J. A Gentle Introduction to the Rectified Linear Unit (ReLU). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[8] Chen, J., Pan, X., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981* (2016).

[9] Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project adam: Building an efficient and scalable deep learning training system. In *USENIX OSDI* (2014).

[10] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *ACM EuroSys* (2016).

[11] Dai, W., Kumar, A., Wei, J., Ho, Q., Gibson, G., and Xing, E. P. High-performance distributed ml at scale through parameter server consistency models. In *AAAI* (2015).

[12] Dai, W., Zhou, Y., Dong, N., Zhang, H., and Xing, E. P. Toward understanding the impact of staleness in distributed machine learning. In *ICLR* (2019).

[13] Davis, J. Carbon Black may be leaking terabytes of customer data (UPDATED). https://www.healthcareitnews.com/news/carbon-black-may-be-leaking-terabytes-customer-data-updated.

[14] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. In *NIPS* (2012).

[15] Debian Manpages. Sysbench - multi-threaded benchmark tool for database systems. https://manpages.debian.org/unstable/sysbench/sysbench.1.en.html.

[16] dlib.net. Dlib C++ Library. http://dlib.net/.

[17] Dorschel, A. Rethinking Data Privacy: The Impact of Machine Learning. https://medium.com/luminovo/data-privacy-in-machine-learning-a-technical-deep-dive-f7f0365b1d60.

[18] Gressin, S. The Marriott data breach. https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach.

[19] Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., and Ré, C. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487* (2016).

[20] Han, S., Shen, H., Philipose, M., Agarwal, S., Wolman, A., and Krishnamurthy, A. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *ACM MobiSys* (2016).

[21] Hardkernel. ODROID XU4. http://www.hardkernel.com/main/products/prdt_info.php.

[22] Hardy, S., Sonne, B., Crete-Nishihata, M., Dalek, J., Deibert, R., and Senft, A. Permission to Spy: An Analysis of Android Malware Targeting Tibetans. https://citizenlab.ca/2013/04/permission-to-spy-an-analysis-of-android-malware-targeting-tibetans/.

[23] Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS* (2013).

[24] Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G. R., Gibbons, P. B., and Mutlu, O. Gaia: Geo-distributed machine learning approaching LAN speeds. In *USENIX NSDI* (2017).

[25] Hunt, T., Zhu, Z., Xu, Y., Peter, S., and Witchel, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *USENIX OSDI* (2016).

[26] iPerf The ultimate speed test tool for TCP, U., and SCTP. Opendocument technical specification. https://iperf.fr/.

[27] Isaak, J., and Hanna, M. J. User data privacy: Facebook, cambridge analytica, and privacy protection. *IEEE Computer 51*, 8 (2018), 56–59.

[28] Jiang, Y., Konečný, J., Rush, K., and Kannan, S. Improving federated learning personalization via model agnostic meta learning. *arXiv preprint arXiv:1909.12488* (2019).

[29] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang,

L. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM ASPLOS* (2017).

[30] Kim, J. K., Ho, Q., Lee, S., Zheng, X., Dai, W., Gibson, G. A., and Xing, E. P. Strads: a distributed framework for scheduled model parallel machine learning. In *ACM EuroSys* (2016).

[31] Lane, N. D., Bhattacharya, S., Georgiev, P., Forlivesi, C., Jiao, L., Qendro, L., and Kawsar, F. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *IEEE IPSN* (2016).

[32] Lanner. Terabit Ethernet: The New Hot Trend in Data Centers. https://www.lanner-america.com/blog/terabit-ethernet-new-hot-trend-data-centers/.

[33] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[34] LeCun, Y., Cortes, C., and Burges, C. J. THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[35] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *USENIX OSDI* (2014).

[36] Li, M., Zhang, T., Chen, Y., and Smola, A. J. Efficient mini-batch training for stochastic optimization. In *ACM KDD* (2014).

[37] Li, M., Zhou, L., Yang, Z., Li, A., Xia, F., Andersen, D. G., and Smola, A. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop* (2013).

[38] Lian, X., Huang, Y., Li, Y., and Liu, J. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NIPS* (2015).

[39] Lian, X., Zhang, C., Zhang, H., Hsieh, C.-J., Zhang, W., and Liu, J. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *NIPS* (2017).

[40] Lian, X., Zhang, W., Zhang, C., and Liu, J. Asynchronous decentralized parallel stochastic gradient descent. In *ICML* (2018).

[41] Luo, Q., Lin, J., Zhuo, Y., and Qian, X. Hop: Heterogeneity-aware decentralized training. In *ACM ASPLOS* (2019).

[42] Mao, J., Yang, Z., Wen, W., Wu, C., Song, L., Nixon, K. W., Chen, X., Li, H., and Chen, Y. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2017).

[43] Mathur, A., Lane, N. D., Bhattacharya, S., Boran, A., Forlivesi, C., and Kawsar, F. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *ACM MobiSys* (2017).

[44] McMahan, B., and Ramage, D. Federated Learning: Collaborative Machine Learning without Centralized Training Data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[45] Mitliagkas, I., Zhang, C., Hadjis, S., and Ré, C. Asynchrony begets momentum, with an application to deep learning. In *Annual Allerton Conference on Communication, Control, and Computing (Allerton)* (2016), IEEE, pp. 997–1004.

[46] Morris, D. P. Google's Privacy Whiplash Shows Big Tech's Inherent Contradictions. https://www.wired.com/story/googles-privacy-whiplash-shows-big-techs-inherent-contradictions/.

[47] Mugerwa, S. Why advertised WiFi router speeds are different from actual real-world speeds. https://www.dignited.com/37840/wifi-router-speeds-theoretical-vs-actual-real-world-speeds/.

[48] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for DNN training. In *ACM* (2019).

[49] RaspberryPi. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[50] Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS* (2011).

[51] Smith, V., Chiang, C., Sanjabi, M., and Talwalkar, A. S. Federated multi-task learning. In *NIPS* (2017).

[52] Targett, E. Mobile Malware on the Rise, Warns McAfee. https://www.cbronline.com/news/smartphone-malware-mcafee.

[53] TitanPower. Data Center Network Speed Is Getting Faster! https://www.titanpower.com/blog/data-center-network-speed-is-getting-faster/.

[54] Trach, B., Krohmer, A., Gregor, F., Arnautov, S., Bhatotia, P., and Fetzer, C. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research* (2018).

[55] Villeneuve, N. Linsanity Leads to Targeted Malware Attacks. https://blog.trendmicro.com/trendlabs-security-intelligence/linsanity-leads-to-targeted-malware-attacks/.

[56] Villeneuve, N. Trends in targeted attacks. Tech. rep., Micro Trend, 2011.

[57] Wang, K., Mathews, R., Kiddon, C., Eichner, H., Beaufays, F., and Ramage, D. Federated evaluation of on-device personalization. *arXiv preprint arXiv:1910.10252* (2019).

[58] Wei, F., Li, Y., Roy, S., Ou, X., and Zhou, W. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)* (Bonn, Germany, 2017), Springer, pp. 252–276.

[59] Wikipedia. Multilayer perceptron. https://en.wikipedia.org/wiki/Multilayer_perceptron.

[60] Wikipedia. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[61] Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., and Yu, Y. Petuum: A new platform for distributed machine learning on big data. In *ACM KDD* (2015).

[62] Yao, S., Hu, S., Zhao, Y., Zhang, A., and Abdelzaher, T. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *WWW* (2017).

[63] Zantedeschi, V., Bellet, A., and Tommasi, M. Fully decentralized joint learning of personalized models and collaboration graphs. In *International Conference on Artificial Intelligence and Statistics* (2020).

[64] Zeng, X., Cao, K., and Zhang, M. Mobiledeeppill: A small-footprint mobile deep learning system for recognizing unconstrained pill images. In *ACM MobiSys* (2017).

[65] Zhang, H. Intro to distributed deep learning systems. https://medium.com/@Petuum/intro-to-distributed-deep-learning-systems-a2e45c6b8e7, 2018.

[66] Zheng, W., Dave, A., Beekman, J. G., Popa, R. A., Gonzalez, J. E., and Stoica, I. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX NSDI* (2017).

[67] Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. Parallelized stochastic gradient descent. In *NIPS* (2010).