EdgeMove: Pipelining Device-Edge Model Training for Mobile Intelligence

Zeqian Dong\(^2\), Qiang He\(^1,2\), Feifei Chen\(^3\)*, Jin Hai\(^1\), Tao Gu\(^4\), Yun Yang\(^2\)

\(^1\)School of Computer Science and Technology, Huazhong University of Science and Technology, China
\(^2\)Department of Computing Technologies, Swinburne University of Technology, Australia
\(^3\)School of Information Technology, Deakin University, Australia
\(^4\)School of Computing, Macquarie University, Australia
zdong.qhe,yyang@swin.edu.au, feifei.chen@deakin.edu.au, tao.gu@mq.edu.au, hjin@hust.edu.cn

ABSTRACT

Training machine learning (ML) models on mobile and Web-of-Things (WoT) has been widely acknowledged and employed as a promising solution to privacy-preserving ML. However, these end-devices often suffer from constrained resources and fail to accommodate increasingly large ML models that crave great computation power. Offloading ML models partially to the cloud for training strikes a trade-off between privacy preservation and resource requirements. However, device-cloud training creates communication overheads that delay model training tremendously. This paper presents EdgeMove, the first device-edge training scheme that enables fast pipelined model training across edge devices and edge servers. It employs probing-based mechanisms to tackle the new challenges raised by device-edge training. Before training begins, it probes nearby edge servers’ training performance and bootstraps model training by constructing a training pipeline with an approximate model partitioning. During the training process, EdgeMove accommodates user mobility and system dynamics by probing nearby edge servers’ training performance adaptively and adapting the training pipeline proactively. Extensive experiments are conducted with two popular DNN models trained on four datasets for three ML tasks. The results demonstrate that EdgeMove achieves a 1.3×-2.1× speedup over the state-of-the-art scheme.

CCS CONCEPTS

• Computing methodologies → Machine learning; Parallel computing methodologies; • Human-centered computing → Ubiquitous and mobile computing.

KEYWORDS

Web of Things, edge computing, machine learning, edge intelligence

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WWW '23, May 1–5, 2023, Austin, TX, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9416-1/23/04... $15.00
https://doi.org/10.1145/3543507.3583540

1 INTRODUCTION

Machine learning (ML) is powering an increasing variety of mobile and Web-of-Things (WoT) applications [20, 33], e.g., personalized recommendations [93], visual assistance [44], video analytics[32], etc. It is shifting ML model training and inference from the central cloud (i.e., cloud-only in Fig. 1) to the network edge [2, 45, 46, 83, 92], enabling real-time data analytics and decision making with privacy preservation [4, 61, 74, 90].

A series of ML techniques have been proposed to support model training on mobile devices (i.e., On-Device in Fig. 1). For example, transfer learning can transfer the knowledge learned by an edge device’s ML model to other edge devices’ ML models to save on training expenses [6, 85]. Federated learning allows edge devices to train a model collaboratively without revealing their private training data [4]. However, the sizes of popular ML models have been increasing rapidly to pursue a higher model accuracy [55]. Many resource-constrained edge devices like smartphones and smart cameras cannot afford to train large models because they can be easily overwhelmed by the high computational overheads incurred [3, 9, 60, 88]. Many techniques, e.g., model compression [10], model pruning [47, 54], and sparse training [88], as well as tools, e.g., TensorFlow Lite [17] and PyTorch Mobile [1], have been developed to support model training on resource-constrained edge devices. However, it is difficult and often impossible to obtain a high model accuracy with these techniques [65].

An alternative solution is the device-cloud training scheme (see Fig. 1), which splits a model into two parts, one (and usually a small one) to be trained locally on the edge device and the other to be
transmitted in between instead of the user’s private data. EdgeMove facilitates pipelined device-edge schemes, this paper presents EdgeMove, a novel scheme that facilitates pipelined device-edge training (i.e., Device-Edge in Fig. 1). It splits a model into two partitions, one for training on the edge device and the other on an edge server. Intermediate features are transmitted in between instead of the user’s private data. EdgeMove offers the following advantages over existing training schemes.

- Compared with the cloud-only training scheme and the edge-only training scheme [2], EdgeMove protects users’ data privacy because it does not upload users’ private data to the cloud or edge servers, which is subject to potential abuse, legal subpoenas, and extra judicial surveillance [71]. EdgeMove also protects users’ model privacy because it offloads only part of a model for training, while the cloud-only training scheme and edge-only training scheme open the door to many cyber attacks [7, 58, 59, 73].
- Compared with the device-cloud training scheme [15, 61, 74, 87], EdgeMove significantly reduces the delays in the training process introduced by the communication latency between the local model part and the external (offloaded) model part. The key is to take advantage of the low-latency access to edge servers’ computation resources. In Fig. 2, we can observe that training a model under EdgeMove (Device-Edge) takes much less time than under the device-cloud training scheme. Please note this experiment is conducted under relatively stable network conditions without user mobility. In a real-world scenario where users move and network conditions vary, EdgeMove can achieve an even greater advantage in model training time (§5).

**Contributions.** Our work makes the following main contributions.

- EdgeMove is the first scheme devised to facilitate pipelined device-edge ML model training across edge devices and edge servers.
- In an edge computing system, an edge server’s training performance replies on its GPUs available, its workloads, and the network condition, which often vary unpredictably over time. Model training strategies formulated offline can quickly be invalidated by these system dynamics. To tackle this challenge, EdgeMove probes the training performance of an edge device’s nearby edge servers, and bootstraps model training with an approximate model partitioning (§3).
- During the training process, system resources often vary over time, sometimes mildly, sometimes considerably. For example, a more powerful GPU may become available. Edge servers’ workloads may fluctuate. Network conditions may also change. These variations are usually more drastic when the user moves. EdgeMove probes nearby edge servers’ training performance adaptively and adapts the training pipeline proactively according to system dynamics at runtime (§4).
- To evaluate the performance of EdgeMove, extensive experiments are conducted in an edge computing system with 2 popular ML models trained on 4 datasets for 3 types of ML tasks. The results demonstrate EdgeMove’s superior performance over existing training schemes. Compared with the state-of-the-art training scheme, EdgeMove achieves a speedup of 1.5×-2.1× in training ML models for image classification, 1.2×-2.0× for text classification, and 1.2×-1.8× for audio classification.

2 **BACKGROUND AND CHALLENGES**

2.1 **Parallelisms for ML Model Training**

A series of parallelism schemes have been proposed to accelerate ML model training. Data parallelism partitions training data across multiple workers [5, 40, 41, 80]. Specifically, training data is partitioned into multiple subsets, one for each worker, so that workers can train their ML model replicas in parallel. To accommodate the increasingly large and complex ML models, model parallelism partitions an ML model across multiple workers [13, 27]. It allows each worker to their model partitions individually and the entire model collectively. Data parallelism and model parallelism can also...
and GPUs, available on its nearby edge servers are limited due to their small physical sizes [2, 38, 89]. In addition, it is unknown to the edge device what GPUs and how many of them are available on nearby edge servers for training its ML model. Furthermore, an edge server’s training performance relies on not only the specs of the GPUs available but also its workloads and the network condition. For example, a fully loaded edge server may not be able to guarantee its training performance as expected. While many pipeline schemes see GPU clusters as white boxes, the edge devices in an edge computing system have to consider edge servers as black boxes, particularly when serverless computing is implemented to relieve edge devices of resource allocation and server management [14, 42, 63, 67, 75]. EdgeMove tackles this challenge by probing the end-to-end training performance of an edge device’s nearby edge servers at the beginning of the training process for pipeline construction (§3).

**Runtime Dynamics.** Most existing pipeline schemes do not adapt model partitionings during the entire training process, assuming that GPUs’ training performance always remains stable. This may be true in a dedicated GPU cluster but is unrealistic in edge computing systems. An edge server’s resources are shared by multiple tenants like YouTube and Uber [24, 37, 68]. The availability of its GPUs may vary during the training process. Its workloads and network conditions may also vary during the training process, depending on the user demands within its coverage. These system dynamics can undermine or improve edge servers’ runtime training performance, especially when the edge device moves. To tackle this challenge, EdgeMove monitors the end-to-end training performance of the worker, i.e., the edge server training the ML model at the moment, probes the end-to-end performance of nearby edge servers adaptively, and adapts the training pipeline proactively (§4).

### 2.3 Device-Cloud ML Model Training

Yao et al. proposed the first scheme to enable device-cloud training across an edge device and a cloud server [87]. Their scheme consists of an offline phase and an online phase. In the offline phase, the edge device evaluates its own training performance, the cloud server’s training performance, and the network conditions. Then, Device-Cloud splits the ML model into two partitions, one to be trained locally and the other to be trained by the cloud server. The experimental results presented in [87] reveal that the network condition is critical to the end-to-end training performance. Thus, the key idea of Device-Cloud is to find the partition point that minimizes the time taken to transmit intermediate features between the edge device and the cloud server. Similar to EdgeMove, Device-Cloud recognizes the potential changes in the network conditions during the training process. In the online phase, it monitors the uplink and downlink data rate, and evaluates the training pipeline. If it is suboptimal, Device-cloud enumerates all the partition points based on the current network conditions and adapts the training pipeline accordingly. Device-cloud is not suitable for device-edge training for two main reasons.

- Device-Cloud assumes a single powerful cloud server as the external worker with fixed specs. The experimental results presented in [87] suggest offloading as many model layers as possible to the cloud for training. However, in an edge computing system, edge servers are heterogeneous and their available resources vary over...
time [2, 21, 25]. It is not always the optimal solution to offload the most model layers for training. In addition, as shown in Fig. 3, device-edge communication latency factors into the performance of the device-edge pipeline. To optimize the pipeline performance, EdgeMove sees edge servers as black boxes, and evaluates their end-to-end training performance, taking into account the computation dimension and the communication dimension in both training pipeline construction (§3) and adaptation (§4).

- Similar to its offline training pipeline construction, Device-Cloud’s online training pipeline adaptation only considers the communication dimension, i.e., the changes in uplink and downlink data rates. However, the dynamics in edge servers’ computation resources must also be considered at runtime, especially when the clients move (§2.1). In addition, Device-Cloud re-evaluates network conditions and pipeline performance at every training iteration. This incurs excessive (and often unaffordable) resource consumption on edge devices, and more so in an edge computing system where multiple edge servers may be available as the external worker. Taking both computation and communication into account, EdgeMove monitors the current edge server’s end-to-end training performance at runtime by training cycles (1 cycle = 500 training iterations) (§3), probes nearby edge servers’ end-to-end training performance when it is necessary, and adapts the training pipeline when it is worthwhile (§4).

To enable device-edge training, EdgeMove consists of two phases, i.e., training pipeline construction (§3) and adaptation (§4). It may complement existing model training schemes. For example, PipeDream and GPipe can be implemented on edge servers to accelerate the training of model partitions offloaded from clients under EdgeMove.

### 3 Offline Pipeline Construction

Similar to pipelined cloud-only training [27, 55, 57, 64], pipelined device-cloud training and device-edge training both require splitting the target ML model into two partitions, one for local training and the other for external training. The key is to find the optimal partitioning that maximizes the pipeline throughput (and minimizes the training time). Existing pipeline schemes profile the target model before model partitioning, assuming that external workers’ training performance is known and unchanged (§2.2). For example, PipeDream [55] obtains the estimates of each model’s compute time and output size based on the performance of the GPUs. Then, it employs dynamic programming to find the optimal partitioning progressively from the lowest model layer to the highest. Device-Cloud [87] also profiles the target model layer by layer but on the edge device and the cloud server. Then, it runs tests, layer by layer, through the device-cloud pipeline to find the optimal partitioning. These sophisticated profiling and partitioning techniques are unsuitable for device-edge training because the system dynamics can quickly invalidate the previously-optimal partitioning and undermine the pipeline performance (§2.1). An illustrative example can be found in Appendix B. In addition, Device-Cloud uploads the entire target model to the cloud server for profiling, which reveals the model architecture and opens the door to cyber attacks (§1).

#### 3.1 Model Profiling

EdgeMove also profiles the target model \( M \) on the client and its nearby edge servers. It sends a test model \( M_t \) and a test dataset \( D_t \) to each nearby edge server for training. When they complete the training cycle (§4) and return the results, the client obtains the estimates of their end-to-end training performance, measured by the time taken to complete a training epoch plus the RTT (Round Trip Time). The edge server with the best performance is selected as the worker. In the meantime, other nearby edge servers register the client’s training task. When they can improve their training performance with a more powerful GPU or more pipelined GPUs, they notify the client, which may activate a training pipeline adaptation (§4). This way, the client does not constantly have to probe nearby edge servers’ available GPUs at runtime.

Unlike PipeDream or Device-Cloud, EdgeMove does not transmit the entire model to external workers to probe their training performance. Instead, it produces the testing models by obfuscating the model parameters. This protects the edge device from many cyber attacks [7, 58, 59, 73] by preventing the reveal of the true model architecture. The test dataset \( D_t \) comprises randomly generated samples to protect the user’s data privacy. For example, a dummy image generator[^3] is used to generate dummy images for profiling image classification models in our study.

### 3.2 Model Partitioning

As discussed in §3, system dynamics may quickly invalidate the optimal model partitioning found offline. Thus, knowing the impracticality of the sophisticated model partitioning techniques employed by PipeDream and Device-Cloud in edge computing systems, EdgeMove employs a lightweight partitioning technique to bootstrap pipelined device-edge model training. Its main idea is to commence model training rapidly with an approximate partitioning, followed by online adaptation (§4.2). Let \( t_d \) denote the time taken by the edge device to complete a training cycle in model profiling. \( S \) denote the set of nearby edge servers, and \( t_{e,n} \) denote the time taken by the \( n \)-th nearby edge server in \( S \). We calculate their per-layer training time as follows:

\[
\begin{align*}
\bar{t}_d &= \frac{t_d}{L}, \\
\bar{t}_{e,n} &= \frac{t_{e,n}}{L}, \forall n \in S
\end{align*}
\]

where \( R \) is the number of training iterations in a training cycle (§4), and \( L \) is the number of layers in the target model \( M \).

The objective of model partitioning under EdgeMove is to maximize the pipeline throughput, similar to existing pipeline schemes. The key is throughput balance, i.e., for the edge device and the edge server to sustain roughly the same throughput. Noting that the edge device’s training performance is lower than the edge server, EdgeMove partitioning objective is to ensure that the time taken to process a minibatch (including forward and backward passes) roughly equals to the time taken to complete the backward pass of the previous minibatch plus the forward pass of the next minibatch.

Take Fig. 4 for example, which compares an example EdgeMove pipeline with a PipeDream pipeline for device-edge training. There should be \( t_{ul}(b_2) + t_{fp}(b_2) + t_{bp}(b_2) + t_{dl}(b_2) = t_{fp}(b_1) + t_{fp}(b_3) \), where \( ul \) stands for uplink, \( dl \) stands for downlink, \( fp \) stands for forward pass, and \( bp \) stands for backward pass.

[^3]: Many off-the-shelf techniques can be employed. In our experiments, we employ Neurofuscator [59] available at https://github.com/zlijingtao/Neurofuscator.

[^1]: https://github.com/FabianBeiner/PHP-Dummy-Image-Generator


4 ONLINE PIPELINE ADAPTATION

At runtime, EdgeMove monitors the pipeline performance and adapts the training pipeline accordingly. It may adapt the training pipeline in two ways: 1) adapting model partitioning across the client and the worker; 2) adapting the entire training pipeline with a new worker and potentially a new model partitioning.

4.1 Pipeline Monitoring

At runtime, system dynamics like changes in the network conditions and worker’s workload conditions may undermine the performance of the pipeline §2.2. To optimize the pipeline, the client monitors the worker’s training performance by inspecting its end-to-end training time, and adapts the training pipeline accordingly. For example, if the worker is taking more time than before to train its model partition, it may disrupt the throughput balance between the client and the worker ($\S 3$) and undermine the pipeline performance. To restore the throughput balance, one or more model layers need to be transferred from the worker to the client.

The training epoch can conduct the inspection (i.e., after every training epoch), similar to how many techniques adapt model training [26, 34, 49, 82]. However, pipeline adaptation by the epoch (i.e., the processing of all the minibatches) is unsuitable for device-cloud training. It is an overly low granularity. Take VGG-16 [72] for example. It takes an RTX3080Ti GPU 24 seconds to complete a training epoch on the CIFAR-10 dataset [35] with a batch size of 32. In device-edge training, during such a long period, the system dynamics may have already undermined the pipeline performance immensely.

Device-Cloud [87] goes to another extreme by inspecting network conditions by the training iteration (i.e., after processing each minibatch). This granularity is too high for device-edge training. Fig. 5 shows the per-iteration computation time (1 Iteration per Circle) for training a VGG-16 model and a ResNet-50 model. We can see significant and constant fluctuations across different training iterations. There are various possible causes, e.g., the worker’s workload dynamics, the model’s inner nature, and sample characteristics. These fluctuations easily activate excessive pipeline adaptations, which require pipeline flushes, model re-partitioning, and model performance degradation.

DeviceMove employs Algorithm 1 (in Appendix) to find an approximate model partitioning for model $M$. Based on the result, $M$ can be split into two model partitions, $M_d$ for the client and $M_c$ for the worker. They are deployed accordingly to establish a device-edge training pipeline. This concludes offline pipeline construction and training commences.

4.2 Partitioning Adaptation

As discussed in §3.2, EdgeMove employs a lightweight technique to find an approximate model partitioning for bootstrapping model training. After training commences, EdgeMove starts monitoring pipeline performance for pipeline adaptation ($\S 4.1$). One of EdgeMove’s key online tasks is adapting the approximate model partitioning to optimize pipeline performance. It employs a novel partition adaptation technique to achieve this objective. The main idea is to move the partition point by one layer (to a lower layer or a higher layer) after each training circle based on pipeline performance changes until the optimal partition point is found. It continues cycle-by-cycle over the entire session with the worker until worker adaptation ($\S 4.3$). The pseudocode is presented in Appendix 2.

EdgeMove runs the algorithm when training commences. Let $c_1$ denote the first training circle. At the end of $c_1$, the algorithm measures the end-to-end training time (i.e., the overall time for the client and the worker to complete $c_1$), and moves partition point randomly by one layer, across either the previous layer or the next layer (Line 2-3). Please note that if the partition point is
after the first layer, it moves across the next layer. At the end of $c_2$, the algorithm again measures the end-to-end training time and compares it with the performance in $c_1$ (Line 10-12). If it takes less time in $c_2$ than in $c_1$, the movement of the partition point at the end of $c_1$ was correct. The algorithm will move the partition point across one layer in the same direction to approach to optimal partition point (Line 11). This continues until the end-to-end training time stabilizes, which indicates that EdgeMove found the optimal partition point for balancing the client and the worker’s throughput (§3.2). Please note that the partition point may move back and forth around the optimal partition point. To avoid such movements, the algorithm will move the partition point only when the expected performance difference is more significant than those caused by previous movements in the same direction.

4.3 Worker Adaptation

By balancing the client and the worker’s throughput, partitioning adaptation can cope with mild variations in pipeline performance. However, when the pipeline performance decreases drastically, partitioning adaptation may struggle to ensure pipeline performance. The client and the worker can both contribute to a drastic pipeline performance decrease. For example, when the client switches to an energy-saving mode, e.g., Android’s "Battery Saver" mode [18], its computation power declines. The decline could be so significant that even offloading most model layers to the worker through partitioning adaptation (§4.2) cannot restore pipeline performance to a satisfactory level. Similarly, when the worker’s end-to-end performance degrades drastically due to unexpected runtime events like network condition deterioration, workload bursts, cyber attacks and client movement, training the fewest model layers on the worker may not be able to ensure the pipeline performance either. It also violates the objective of device-edge training, i.e., to leverage edge servers’ computation power. EdgeMove tackles these challenges with an adaptive worker adaptation technique that finds a new worker to replace the previous one. The pseudocode is presented in Appendix 3.

Worker adaptation requires probing nearby edge servers’ end-to-end training performance online with model profiling (§3.1). The client can monitor pipeline performance constantly (§4.1) because it consumes little computation resources. However, it is impractical for the client to probe nearby edge servers constantly. Under EdgeMove, the client records the worker’s worst end-to-end training performance at the end of every training circle (§4.2), denoted by $t_{\text{new}}$. At the end of each training cycle, the client inspects the worker’s performance against $t_{\text{new}}$ to determine whether it is necessary to probe nearby edge servers’ training performance. When the worker’s training performance is worse than $t_{\text{new}}$, the client probes nearby edge servers’ training performance (including the worker’s) in the same way as offline model profiling (§3.2). In the meantime, model training proceeds to the next circle, asynchronously, with the online probing process.

When the probing results come back from the nearby edge servers, the client compares their training performance with the worker’s performance obtained at the end of the current training circle. If the worker is outperformed, the client selects the edge server with the best training performance as the new worker. Then, it partitions the ML model and builds a new training pipeline with the technique presented in §3.2. If the worker is not outperformed by any other nearby edge server, the client may retain the current training pipeline, wait for the next opportunity, or simply terminate the training process.

When profiling a model, the client registers its training task with probed edge servers (§3.1). If an edge server’s training performance improves at runtime, it can notify the client. This may also activate a worker adaptation, which compares the edge server’s performance against the worker’s.

Remark. Worker adaptation incurs migration overheads. The client needs to terminate the training pipeline and build a new one. These take time. However, in the long term, these once-off overheads are worthwhile compared with the performance gains from training with a more powerful new worker. In addition, a pub/sub system is needed to allow the client to register its training task with probed edge servers. Such a system can be built easily based on pub/sub services offered by Google [19], Microsoft [33], Amazon [52], etc. It barely consumes edge servers’ resources and can benefit other applications running in the edge computing system. Without this pub/sub system, the client has to wait until it probes a nearby edge server to discover its improved training performance. This slightly reduces EdgeMove’s responsiveness to system dynamics but does not fundamentally undermine its usefulness.

5 EVALUATION

5.1 Experiment Setup

System Setup. An edge computing system is built with 125 virtual machines deployed in a private data center as edge servers with a coverage radius of [300, 500] meters, each with a 4-core vCPU, 8-16GB RAM, and a V100 GPU or a 3080Ti GPU. These virtual machines are each assigned a location extracted from the widely-used EUA dataset [36] to simulate the Melbourne CBD area powered by edge computing. An Amazon p3.2xlarge EC2 instance with 8 vCPUs, 61GB RAM and a V100 GPU is hired as the cloud server to enable Device-Cloud [87]. A Google Pixel 6a connects to the system via a Wi-Fi 6 router as the client. We ran tests on the system and measured the network conditions. The network latency between the client and the cloud server is [100, 170] milliseconds. The network latency between the client and the edge servers is [5-40] milliseconds, similar to the 5G network latency in the U.S. reported by Ericsson in August 2022 [48]. At runtime, an edge server may be able to improve its training performance at runtime with a more powerful GPU or more pipelined GPUs (§3.1). From the client’s perspective, it is no different from discovering a new edge server nearby. Thus, in the experiments, edge servers’ GPU resource dynamics are not implemented.

Client Movement. In the experiments, the client moves along one of the ten randomly-created trajectories across the Melbourne CBD. These trajectories include different numbers of turns at different locations. Two example trajectories are illustrated in Appendix C. When the client moves, its nearby edge servers are identified based on the distance between them - those that cover the client are considered its nearby edge servers.

[[https://github.com/swinedge/eua-dataset]]

Baseline. EdgeMove is evaluated against three representative training schemes, including a baseline and two state-of-the-art schemes.

- **On-Device** [87]. Under this baseline scheme, ML models are trained only on the client’s edge device.
- **Device-Cloud** [87]. This is the state-of-the-art scheme for device-cloud training. In the experiments, it builds training pipelines across clients and the cloud server. First, it profiles the target model layer by layer, similar to PipeDream, and finds the optimal model partitioning for device-cloud training, considering both computation and communication conditions. At runtime, it monitors the network conditions and adapts model partitioning based on uplink and downlink data rates without worker adaptation. A detailed discussion about this scheme can be found in §2.3.
- **PipeDream-E** [55]. PipeDream is the state-of-the-art scheme for pipelining ML model training across GPUs in a cluster. Several schemes have adapted PipeDream to various ML training scenarios [56, 57, 86]. These schemes are not implemented in the experiments because they share the same core idea with PipeDream. PipeDream-E adapts PipeDream to device-edge training by building training pipelines across clients and edge servers. The experiments randomly select one of the client’s nearby edge servers as the worker offline (and online when the client leaves the worker’s coverage area). Then, it profiles the model layer by layer and finds the optimal model partitioning across the client and the worker without considering device-edge network conditions. After that, it builds a training pipeline across the client and the worker without partitioning adaptation or worker adaptation. More details about PipeDream can be found in §2.1.

Performance Metrics. Acceleration is the main objective of pipeline training [27, 55, 87]. In the experiments, we evaluate EdgeMove’s training time-to-accuracy, i.e., the time taken to train a model to the target accuracy, same as [27, 55, 87].

5.2 Experimental Results

Overall Performance. Table 1 summarizes the average time taken for the clients to train a model to the target accuracy under different schemes, as well as their speedups over On-Device. The following key findings can be derived from the table.

- **On-Device** takes the most time to train a model in all the cases. This is not surprising because it does not leverage powerful workers or pipeline parallelism to accelerate model training.
- **Powered by pipeline parallelism and external workers, PipeDream-E and Device-Cloud both reduce the training time considerably, evidenced by their significant speedups over On-Device. This confirms the effects of pipelining model training across the client and a powerful external worker.**
- **PipeDream-E and EdgeMove both outperform Device-Cloud evidently. Their advantages come from the much lower device-edge communication latency compared with device-cloud communication latency (§1). This indicates the fundamental advantage of device-edge training over device-cloud training.**
- **EdgeMove outperforms PipeDream-E remarkably. The main reason is that it factors into three elements that are neglected by PipeDream-E: 1) edge servers’ heterogeneous GPU resources; 2) network conditions between the client and different edge servers; and 3) runtime system dynamics in both computation and communication. This tells us that EdgeMove can properly tackle the challenges discussed in §2.**

Model Convergence. Fig. 6 illustrates clients’ model convergence under different training schemes. Despite the models and the datasets, we can see that EdgeMove is always the first to complete the training process, with much time to spare compared with its competitors. Looking closely at the early stage of the training process, we can see that EdgeMove’s model accuracy increase is relatively slow, even slower than PipeDream-E. About 50-75 seconds in, the increase rate starts to pick up. This is EdgeMove’s “warm-up” phase, where partitioning adaptation (§3.2) kicks in to find the optimal partitioning. After the warm-up phase, EdgeMove’s model accuracy increase remains well above every other training scheme, making it always the first to converge the model.
To evaluate EdgeMove's ability to adapt to client speed, we conduct an experiment where clients move along the same trajectories at twice the speed in the previous experiments. Fig. 7 demonstrates the clients' average per-circle training time in the first 200 training circles. As expected, we can observe that EdgeMove's per-circle training time is much lower than its competitors. In the beginning, it takes more time to complete a training circle than PipeDream-E. This confirms that the approximate model partitioning (§4.2) EdgeMove obtains offline to bootstrap model training is indeed not optimal. Luckily, through partitioning adaptation (§4.2), EdgeMove manages to find the optimal model partitioning across the client and the worker rapidly. This is evidenced by the quick decrease in its per-circle training time after model training commences. In Fig. 7, we can observe peaks in EdgeMove's per-circle training time, as well as other training schemes, some larger than others. We investigated and found that most of the small ones were caused by partitioning adaptation (§4.2), while most of the large ones were caused by worker adaptation (§4.3). We can also observe peaks in PipeDream-E's per-circle training time, which appear when the client is no longer within the worker's coverage and has to find a new worker. EdgeMove's peaks are much smaller than PipeDream-E's on average. This indicates the ability of EdgeMove to adapt partitioning or worker timely in response to system dynamics.

Adaptation to Client Speed. The clients in an edge computing system may operate at different speeds. For example, they may travel in different ways, on foot, by bicycle or car. System dynamics are usually more significant when a client is travelling at a higher speed. To evaluate EdgeMove's ability to adapt to client speed, we conduct an experiment where clients move along the same trajectories at twice the speed in the previous experiments. Fig. 8 compares the training time-to-accuracy in these two cases. We can see that when the clients move faster, On-Device and Device-Cloud take almost twice the speed in the previous experiments. PipeDream-E's per-circle training time, which appear when the client is no longer within the worker's coverage and has to find a new worker. EdgeMove's peaks are much smaller than PipeDream-E's on average. This indicates the ability of EdgeMove to adapt partitioning or worker timely in response to system dynamics.

Adaptation Overheads. Partitioning adaptation (§4.2) and worker adaptation (§4.3) incur overheads, measured by the time taken to upload new model layers to the worker and the time taken to upload model partitions to new workers. Fig. 9a compares the number of partitioning adaptations EdgeMove needs to train a model and compares the number of worker adaptations it needs against PipeDream-E. PipeDream-E replaces the worker with a new worker reactively only when it leaves the worker's coverage area (§5.1). To respond to system dynamics timely, EdgeMove replaces the worker proactively based on its end-to-end training performance against other nearby edge servers’ (§4). Thus, it evidently needs more worker adaptations than PipeDream-E. Fig. 9c compares EdgeMove's adaptation time (i.e., time for adaptation) and training time (overall training time minus adaptation time) against PipeDream-E. We can see that EdgeMove spends more time adapting than PipeDream. However, its overall training time is much less than PipeDream-E. This indicates that its proactive adaptations pay off with significant speedup gains.

6 CONCLUSION AND FUTURE WORK
This paper presented EdgeMove, a novel scheme that pipelines device-edge machine learning (ML) model training. To tackle the challenges raised by the edge computing system dynamics, it bootstraps model training with an approximate training pipeline constructed based on edge servers’ end-to-end training performance. At runtime, it monitors pipeline performance and adapts the training pipeline accordingly. Compared with state-of-the-art schemes, EdgeMove completes model training up to 2.4x faster across a range of ML tasks.

ACKNOWLEDGMENTS
This research is partially funded by Australian Research Council Discovery Projects (DP180100212 and DP200102491). Feifei Chen is the corresponding author of this paper.


A  PSEUDOCODE

Algorithm 1: Model Partitioning

**Input:** $t_d$: client’s probed training time; $t_e$: worker’s probed training time; $R$: size of training circle; $L$: number of layers in the test model; $L_s$: number of layers in the target model; $t_{dd}$: uplink transmission time; $t_{dl}$: downlink transmission time

**Output:** $l_d$: number of layers for client

1. $l_d \leftarrow \frac{t_d}{R_sL_s}$; /* evaluate client’s training performance */
2. $\overline{t_e} \leftarrow \frac{t_e}{R_sL_s}$; /* evaluate worker’s training performance */
3. $l_d \leftarrow (L_s + \overline{t_e} + RTT)/(\overline{t_d} + \overline{t_e})$; /* find approximate partition point */
4. $l_d \leftarrow \text{round}(l_d)$
5. return $l_d$

Algorithm 2: Partitioning Adaptation

**Input:** $F$: training pipeline; $l_d$: number of layers for client

**Output:** $l_d$: partitioning adaptation result

1. **Function** Partition Adaptation($F, l_d$)
   1. /* initialize */
   2. $d \in \{+1, -1\}$; /* set random direction */
   3. $i \leftarrow 1$, $l_i \leftarrow l_d$
   4. $F(l_i)$; /* build pipeline */
   5. obtain $t_i$ from $F(l_i)$; /* obtain training time in $c_i$ */
   6. /* adapting */
   7. while $l_i$ not stable do
   8.      $l_i \leftarrow l_i - d$; /* move partition point */
   9.      $F(l_i)$ with $l_i$; /* adapt pipeline */
   10.     obtain $t_i$ from $F(l_i)$; /* obtain training time in $c_i$ */
   11.     if $t_i > t_{i-1}$ then
   12.        $d \leftarrow -d$; /* reverse adaptation direction */
   13.     end
   14. end
   15. return $l_d$

Algorithm 3: Worker Adaptation

**Input:** $w$: worker adaptation result

**Output:** $w$: worker adaptation result

1. **Function** Worker Adaptation($F, M_l, W$)
   1. /* initialize */
   2. probe($W, M_l$); /* probe nearby edge servers */
   3. $w \leftarrow \text{argmin}(\text{recv}(W))$; /* find best edge server */
   4. obtain $l_d$ with Algorithm 2 for $w$
   5. $t_{wst} \leftarrow F(l_d)$; /* monitor training performance by cycle */
   6. For each training cycle
   7.      $t_{cur} \leftarrow F(l_d)$; /* obtain training time */
   8.      if $t_{cur} > t_{wst}$ then
   9.         probe($M_l$); /* probe nearby edge servers */
   10.        $w' \leftarrow \text{argmin}(\text{recv}(W))$; /* find best worker */
   11.        if $w = w'$ then
   12.            $t_{wst} \leftarrow t_{cur}$; /* update worst training time */
   13.        else
   14.            $w \leftarrow w'$; /* change worker */
   15.            obtain $l_d$ with Algorithm 2 for $w$
   16.            $t_{wst} \leftarrow F(w, l_d)$
   17. end
   18. end
   19. return $w$;

B  OPTIMAL PARTITIONING POINT

Fig. 10 shows the optimal partition points for training a VGG-16 across a device-edge pipeline under four different network conditions. We can see that the optimal partition point varies drastically with the network condition.

Figure 10: Overall training time for VGG-16 across a device-edge pipeline with different partition points under different network conditions. In this figure, a bar represents the overall training time with the corresponding partition point. The star indicates the optimal partition point that results in the least overall training time.
C  EXAMPLE CLIENT TRAJECTORIES

Figure 11: Example client trajectories across Melbourne CBD with 125 edge servers.

D  MODELS AND DATASETS

Table 2 summarizes the characteristics of the models and datasets used in the experiments.

Models. VGG-16 [72] and ResNet-50 [23] are conducted on four datasets in the experiments to perform three different ML tasks.

- **Image Classification.** VGG-16 is trained on the CIFAR-10 dataset and ResNet-50 is trained on the CINIC-10 dataset to classify images into 10 classes. The size of the training circles is 500 iterations.

- **Text Classification.** VGG-16 is trained on the AG News dataset to classify articles into 4 classes. The size of the training circles is 1,000.

- **Audio Recognition.** ResNet-50 is trained on the Speech Commands dataset to classify audio records into 12 classes. The size of the training cycles is 200 iterations.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Task</th>
<th>Model</th>
<th>Cycle Size</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>Image Classification</td>
<td>VGG-16</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>CINIC-10</td>
<td>Image Classification</td>
<td>ResNet-50</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>AG News</td>
<td>Text Classification</td>
<td>VGG-16</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>Speech Command</td>
<td>Audio Recognition</td>
<td>ResNet-50</td>
<td>200</td>
<td>12</td>
</tr>
</tbody>
</table>

Datasets. Four datasets are used in the experiments.

- **CIFAR-10 [16]** is the baseline dataset for tiny image classification. It contains 50,000 training image samples and 10,000 test image samples, all sized 32×32. They are equally divided into ten mutually exclusive classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

- **CINIC-10 [12]** is an augmented dataset for image classification. It is constructed from two popular sources: ImageNet and CIFAR-10. The images are resized to 32×32 and the dataset contains 90,000 images in each of its three subsets, including the training subset, the validation subset, and the test subset. It is 4.5 times larger than CIFAR-10.

- **AG News [91]** is a dataset built from AG’s corpus of news articles. It contains 30,000 training articles and 1,900 test articles from the 4 largest classes of AG’s Corpus, including “World”, “Sports”, “Business”, and “Sci/Tech”.

- **Speech Command [81]** is an audio dataset of spoken words to help train and evaluate audio recognition systems. It includes 64,727 audio files classified into 12 different classes.