MLIoT: An End-to-End Machine Learning System for the Internet-of-Things

Sudershan Boovaraghavan
Carnegie Mellon University
sudershan@cmu.edu

Anurag Maravi
Carnegie Mellon University
amaravi@andrew.cmu.edu

Prahaladha Mallela
Carnegie Mellon University
prahalam@alumni.cmu.edu

Yuvraj Agarwal
Carnegie Mellon University
yuvraj@cs.cmu.edu

ABSTRACT

Modern Internet of Things (IoT) applications, from contextual sensing to voice assistants, rely on ML-based training and serving systems using pre-trained models to render predictions. However, real-world IoT environments are diverse, with rich IoT sensors and need ML models to be personalized for each setting using relatively less training data. Most existing general-purpose ML systems are optimized for specific and dedicated hardware resources and do not adapt to changing resources and different IoT application requirements. To address this gap, we propose MLIoT, an end-to-end Machine Learning System tailored towards supporting the entire lifecycle of IoT applications. MLIoT adapts to different IoT data sources, IoT tasks, and compute resources by automatically training, optimizing, and serving models based on expressive application-specific policies. MLIoT also adapts to changes in IoT environments or compute resources by enabling re-training, and updating models served on the fly while maintaining accuracy and performance. Our evaluation across a set of benchmarks show that MLIoT can handle multiple IoT tasks, each with individual requirements, in a scalable manner while maintaining high accuracy and performance. We compare MLIoT with two state-of-the-art hand-tuned systems and a commercial ML system showing that MLIoT improves accuracy from 50% - 75% while reducing or maintaining latency.

CCS CONCEPTS

• **Computer systems organization** → **Client-server architectures**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Distributed Machine Learning, Internet of Things, Training, Serving

ACM Reference Format:
Sudershan Boovaraghavan, Anurag Maravi, Prahaladha Mallela, and Yuvraj Agarwal. 2021. MLIoT: An End-to-End Machine Learning System for the Internet-of-Things. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '21), May 18–21, 2021, Charlottesvle, VA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3450268.3453522

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

IoTDI '21, May 18–21, 2021, Charlottesvle, VA, USA

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8354-7/21/05.
https://doi.org/10.1145/3450268.3453522

1 INTRODUCTION

There is widespread interest in Internet-of-Things (IoT) enabled environments with the expectation that it can revolutionize healthcare, smart homes/buildings, and manufacturing. This vision is enabled by IoT sensors with rich sensing capabilities [39, 49], cloud platforms like AWS and Azure, and rapid advances in ML software stacks[4–6]. In the context of smart homes and buildings, in particular, the goal is to sense the rich context and activities of the occupants and to detect the events/states of the appliances and equipment in the space [32, 38, 39].However, supporting the diversity of IoT use cases, differences in user and application requirements, as well as the dynamic environments that these devices are deployed in, requires machine learning platforms that can be customized and adapted to the IoT domain. Doing so enables compelling applications such as sensing Activities of Daily Living (ADLs) for the elderly [10, 18, 22] or make smart assistants [3, 27] context driven and more capable.

To detect activities and events in smart environments usually requires machine learning on a variety of IoT sensor data sources. The holy grail is to "train" a generalized ML model once and then "serve" or deploy it to make predictions. In fact, several general-purpose ML serving-only systems exist, [5, 14, 51, 66, 67] including those for IoT relevant audio data [38] or image sources. These systems relying on pre-trained models do not work well for IoT settings for several reasons. First, each IoT deployment/environment is different requiring 'in-situ' (re-)training based on the sensors and events in that setting. For example, multi-modal sensor [39, 49] or audio-based context recognition [21, 38] is affected by the physical space characteristics and changes to the ambient environment. Second, accurate pre-trained models, such as ImageNet or YoLo, need a significant amount of labeled training data, and computation resources, which is atypical in IoT scenarios. Third, these systems are optimized for specific, and often dedicated hardware resources, and do not adapt to changes in resource availability, for example, an edge compute hub (for privacy) as compared to a server on the cloud. Recent work, proposes programming by demonstration (*PbD*) [39, 49], to enable users to train personalized models 'in-situ' in a specific IoT environment. However, these approaches use a single ML model and do not adapt to model drift over time with environmental changes, which requires models that need to be retrained and optimized. Furthermore, these systems are not contextualized to IoT and do not identify which common ML techniques like dimensionality reduction (DR) or hyper-parameter optimizations (HPO) work in

practice. Most importantly, most of the prior works are not "end-to-end" ML systems for IoT scenarios with closely coupled (re-), training and serving components.

To address these limitations, we designed and implemented MLIoT, an end-to-end machine learning system tailored for IoT use cases, supporting the entire lifecyle of initial training, serving, and retraining processes.MLIoT adapts to (heterogeneous) compute resources by performing device selection based on user expressed preferences (policies) and benchmarking device capabilities. MLIoT adapts to different data sources and tasks by automatically training, optimizing, and serving models based on expressive preferences (policies). MLIoT adapts to changes to the IoT environment or compute resources by re-training, and updating the served models on the fly, while maintaining accuracy and performance. MLIoT performs these adaptations dynamically for multiple tasks, each with their own training-serving pipelines and requirements.

In this paper, we make the following contributions:

(1) We present the design and implementation of MLIoT, an extensible end-to-end Machine Learning system contextualized for IoT scenarios. MLIoT provides flexible policy-driven selection of hardware platforms, ML models, and various optimizations for closely coupled training and serving tasks.
(2) We propose numerous optimizations for training and serving and report on their efficacy in an IoT context. These include (a) hyper-parameter optimization and dimensionality reduction of models; (b) lazy training to increase accuracy over time, while reducing latency to start serving; (c) model adaptation to account for drift using corrective feedback from users; and (d) two-stage serving that increases accuracy while improving serving latency.
(3) We evaluate MLIoT on several hardware devices and across a set of expressive IoT benchmark datasets (image, audio, multi-modal sensor data). Our results show that MLIoT is able to service different policy objectives, balancing load across devices while maintaining accuracy and latency. Furthermore, we compare MLIoT with two state-of-the-art hand-tuned systems and a commercial ML system showing that MLIoT improves accuracy from 50% - 75% while reducing or maintaining similar latency.

2 APPLICATIONS AND CHALLENGES

To motivate the challenges unique to IoT settings we present several example IoT applications and their data sources. We then identify the unique challenges for ML systems in IoT settings, which existing general purpose training [17, 63, 66] and serving [14, 51] systems fall short on.

2.1 IoT Application Workloads

Activity Recognition using Multi-Modal sensors: There is rich history of activity and event detection in IoT enabled environments, using non-intrusive ambient sensors [32, 37, 39, 49], or direct instrumentation [8, 64], and even in-direct sensing [12, 30, 57]. These systems use labelled data from sensors such as Accelerometers, Gryoscopes, Microphones, Temperature and Humidity to explicitly train ML models for specific events and test their accuracy on live data. Recent systems[39, 49], incorporate as many as 13 different hardware sensors, extracting over 2,000 features, on a single package. They propose a Programming-by-Demonstration (PbD) approach to train a single ML model to predict the occurrence of various events. Their interface allows end users to add more training data, and retrain the ML model if desired. Given the diversity of IoT relevant hardware sensors present on this platform, we collect and use multiple datasets from the Mites to evaluate MLIoT, as well as compare our accuracy and performance to this state of the art system.

Audio Based Activity Recognition: Given that many activities have an audio signature (e.g. appliances running, faucets being turned on) Ubicoustics proposes using audio only for activity detection, particularly by building a *generalized* a pre-trained deep model [38]. Their system collects audio, processes it into VGG-16 network with 6144 features, and then trains a large DNN. We collect a set of audio benchmarks and compare MLIoT with Ubicoustics.

Object Recognition using Image Data: Several emerging IoT applications apply computer vision algorithms on images, to detect objects, adding labels and bounding boxes[55, 56]. To evaluate MLIoT for image data, we use the popular MNIST dataset which has labelled images for handwritten digits [40].

2.2 Challenges for ML systems in IoT settings

Motivated by the above IoT application and workloads, we identify key challenges for ML Systems geared towards them.

Adapting to Different IoT Application requirements: IoT applications requirements can differ substantially. Deep learning based computer vision applications often need hardware accelerators with significant memory[55]. Activity recognition, using classical models, often need to run on *edge* gateways such as a Rasbperry PI [54] or a SmartThings Hub, [58] with modest compute resources for faster latency and to alleviate privacy concerns. Developers of these applications themselves may have different requirements such as accuracy, latency, cost (cloud vs local inferences), models to use, and even different priorities between their different IoT tasks. MLIoT provides an expressive policy driven mechanisms to balance user requirements with the available resources.

Adapting to Device Capabilities and Resource Availability: The inherent heterogeneity in individual device capabilities (CPU complexity, number of cores, memory, bandwidth, accelerators) affect the performance (training and serving) of ML algorithms. It is essential to thus estimate the comparative performance of the devices available, which MLIoT achieves by benchmarking devices in different conditions and using that information for device selection. Furthermore, the resources available on individual devices also change as different IoT ML tasks run on them, each with their own requirements. MLIoT provides load balancing and various dynamic adaptation mechanisms such as changing the models served, for meeting these requirements.

Adapting to Changes in Environmental Context: Existing ML systems [5, 14, 51, 66, 67] often rely on generalized ML models, which are infeasible in IoT settings since each environment is unique and models needs to be contextualized to that environment. Furthermore, IoT systems are also affected by changes in the ambient environment, both temporary or permanent (e.g. changes to the

physical layout of a home, affecting audio based sensing). MLIoT enables both the initial training, and retraining and re-optimizations of the models based on user driven corrective feedback.

Customization to IoT data types: There are well known techniques to improve the accuracy and the performance of ML systems, such as hyper parameter optimization, creating model ensembles, dimensionality reduction, multi-stage models. However it is unclear which of these techniques (and specific algorithms) work on IoT data types, either individually or in combination. In MLIoT we apply all these techniques on a variety of IoT data types showing their efficacy and the associated tradeoffs.

Need for an End-to-End ML service for IoT: Supporting the lifecyle of an IoT application comprises of multiple key steps: initial training (e.g. using a PbD approach), model evaluation and optimizations, model serving, adapting and retraining models with additional data or changes to resources, and then managing multiple concurrent IoT applications. While individual pieces to support this lifecyle exist, to the best of our knowledge none of them provide an end-to-end solution like we envision with MLIoT.

3 RELATED WORK

We organize the prior work on Machine Learning systems into three categories: Serving-only (§3.1), Training-only (§3.2), and hybrid training-serving (§3.3). We refer to [44] [16] for an extensive survey of this topic and also compare and contrast several prior works with MLIoT in Table 1.

3.1 Large-scale ML Serving Systems

There are several general purpose prediction serving systems from the industry and academia which aim to facilitate model deployment [14, 15, 38, 51, 65, 67]. These systems place the trained models in containers and optimize model inference requests. Clipper [14] aims to deploy pre-trained ML models in containers and optimizing serving performance using request batching and caching to reduce latency. It also employs user feedback to select and combine the output of one or more deployed models to improve accuracy. Inferline [15] provisions and executes prediction pipelines subject to latency constraints, leveraging adaptive batching and autoscaling to reduce latency. TensorFlow Serving [51], a commercial grade serving system, is designed to deploy models as TensorFlow pipelines [24] which are executed in black box containers. Other serving systems have focused on applications content recommendation systems [67], speech recognition [41] and activity recognition [38] all of which have highly customized, application specific models. Several commercial systems targeted towards IoT also exist such as Amazon AWS IoT greengrass [5], Google Cloud Vision AI [26], Amazon Rekognition [4] or Azure IoT [47]. Some of them leverage edge and cloud resources to serve models with low latency, and to save costs. These commercial systems are vertically integrated, focusing on serving predictions from a single model or framework, or specific hardware only. These systems fail to address one or more key requirements for IoT scenarios: focus on serving static pre-trained models, no adaptation to environmental changes, do not support heterogeneous devices, or have limited, if any, support for policies.

3.2 Large-scale ML Training Systems

Most of the training focused systems [11, 17, 63, 66] optimize for deep neural network models with many hyperparameters, which is very resource and time intensive. Project Adam [11] investigates distributed training based on available resources while Helix [66] and KeystoneML [63] use various techniques to optimizes the ML training workflow.

Commercial systems such as Google Vizier [23], similarly optimize DNNs, focusing on a variety of techniques to 'tune' the network parameters to improve accuracy and performance. Google AutoML [25] views learning to build a network itself as a machine learning problem, applying techniques such as reinforcement learning for the task. These systems are geared towards producing high-quality and complex deep networks for domains such as object recognition, and translation. These training only systems assume large corpus of training data for an expensive, infrequent, training tasks and optimize for efficient distributed model training. In contrast, in IoT scenarios the available data is sparse to train complex models, the trained models need to be specific to the sensor sources, environmental context, application requirements and thus need more closely coupled (re-)training and serving systems.

3.3 ML Training/Serving Hybrid Systems

Several recent efforts aim to simplify ML development through a general-purpose machine learning system with both training and serving of models [2, 5, 6, 13, 15, 26, 34, 35, 42, 68, 69]. Some of these systems that share similar goals of MLIoT are the end-to-end "ML Platforms" that run at commercial settings. Systems such as Uber's Michelangelo ML [35] and Facebook's FBLearner Flow [34] serve as ML-as-a-service platform which is optimized for their internal use cases. Uber's Michelangelo ML is optimized for their real-time requirements, allowing production models to use features extracted from streams of live data. FBLearner Flow allows reusable ML workflow that allows ML models to be modified and reused in different products. On the other hand, Google's TFX [6], provides Tensorflow-based [24] toolkits for data preparation, periodic model evaluation to improve performance and reliability and extends TensorFlow Serving [51] to serve the models with TensorFlow-based learners. Such systems generally run on the cloud incurring a higher cost for better workload environments and restrict users to a specific set of algorithms or libraries, so users are on their own when they step outside these boundaries. Similarly, Other commercial IoT tailored systems such as Google's Cloud IoT [26], Microsoft's ML.net [2] and AWS IoT Greengrass [5] are in-house proprietary systems focused on specific hardware and ML algorithms. Academic approaches such as Velox [13] and InferLine [15] propose managing the lifecycle of model training, serving, and updating but they are intended towards modeling efficient execution of ML pipelines to reduce cost or meet the SLO constraints. They are not designed to adapt to unpredictable operational environments. Ultimately, many of these systems do not address the challenges specific to IoT applications as mentioned earlier in Section §2.

Differentiation of Our Approach: Our MLIoT vision and approach differs from prior work along several key aspects, with a summary provided in Table 1. MLIoT is designed as an end-to-end

Table 1: Comparing expressiveness of MLIoT with prior work on Machine Learning systems. While some systems do address some of the requirements of IoT scenarios, to the best of our knowledge MLIoT is the only one that supports all of them.

	Related Work	**Requirements in IoT**						
		Approach	Sources	Adaptive Model Selection	Expressive Polices	Adaptation to Heterogeneous Hardware	Adaptation to Environment	End-to-End
Academic Systems	Clipper [14]	Serving	Any Src	✓	◆	✗	✗	✗
	InferLine [15]	Hybrid	Any Src	✓	◆	✓	✗	✗
	Ubicoustics [38]	Serving	Audio	✗	✗	◆	✗	✗
	Helix [66]	Training	Text	✗	✗	✓	✗	✗
	Project Adam [11]	Training	Image	✗	✗	◆	✗	✗
	KeystoneML [63]	Training	Text	✗	✗	✗	◆	✗
	Mites [49]	Hybrid	Multimodal	✗	✗	✗	✓	✓
	Velox [13]	Hybrid	Any Src	✗	✗	✗	✓	✓
	Laser [67]	Hybrid	Text(Ads)	✗	✗	✗	◆	✓
Commercial	TensorFlow Serving [51]	Serving	Any src	✗	◆	✗	✗	✗
	AWS IoT Greengrass [5]	Serving	Any Src	✗	✗	✓	✗	◆
	Microsoft's ML.Net [2]	Hybrid	Any Src	✗	✗	✗	◆	✓
	Google's Cloud ML [26]	Hybrid	Any Src	✗	✗	✗	✓	✓
	Google's TFX [6]	Hybrid	Any Src	✓	✗	✗	✓	✓
	MLIoT	Hybrid	Any Src	✓	✓	✓	✓	✓

✓: Available ✗: Not Available ◆: Partial features available

IoT focused ML training-serving system. MLIoT is flexible to support wide variety of IoT data sources, train personalized models that can be optimized and re-trained over time. MLIoT supports expressive policy-driven device and model section for both training and serving based on application and user requirements. It is extensible allowing new ML frameworks and algorithms to be added and can leverage heterogeneous hardware platforms.

4 SYSTEM DESIGN AND ARCHITECTURE

The overall architecture of MLIoT is illustrated in Figure 1. It comprises of two logical components, the *Device Selection Layer (DSL)* and *Training-Serving Layer (TSL)*. The DSL has several key roles. It serves as the central authority that manages all the devices or compute resources available to it for scheduling ML tasks. It is also responsible for handling any new request for ML tasks and based on the policy specified by the IoT application, selecting the appropriate resource to serve that task. The DSL also maintains configuration data for each training-serving task, which includes the application training data, the trained ML models, and several other parameters for serving those models. This functionality is important since the same task can be interrupted, or restarted, and can resume serving from where it left off using the state available at the DSL, including running on a different device. We describe the DSL in further detail in §4.1.

The Training-Serving Layer (TSL), is responsible for instantiating the Training-Serving Workers (TSW) for each ML task sent to it. The TSL is also responsible for allocating and managing resources (CPU and memory limits) to individual TSWs. The training-serving workers are responsible for training models, optimizing them, and selecting the set of models for serving based on the specified policies. TSWs are also responsible for keeping track of performance and adapting the models to changes to the resource availability on the device they are running on, or on receiving corrective feedback. Individual TSWs report back metrics like model performance, CPU and memory usage, etc to the DSL which has a holistic view of all

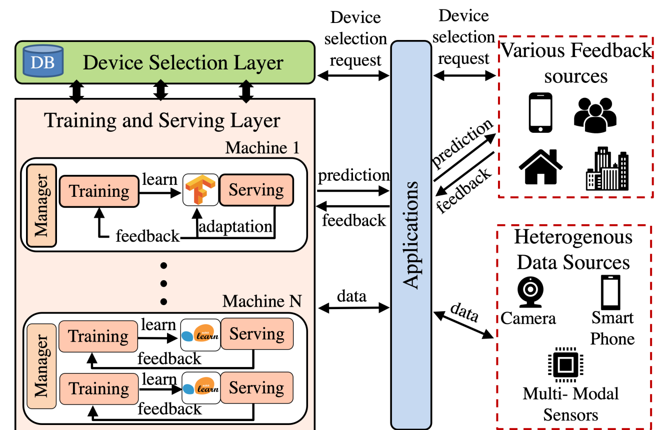

Figure 1: Overall System Architecture of MLIoT

the tasks it is managing and the devices they run on. We describe the TSL and the functionality provided by the TSWs in further detail in §4.2.

4.1 Device Selection Layer (DSL)

IoT application tasks comprise of training and serving ML models. How well these tasks run depends on the device capabilities such as the number of cores available for parallel execution, the amount of memory available, the relative performance of the cores (x86 vs ARM vs an accelerator), etc. Furthermore, IoT applications themselves may have different requirements. For example, an application that senses audio at home may need the ML predictions to be done locally on an in-home hub like a Raspberry-PI or a Samsung Hub. An intrusion detection application [20] may require low latency predictions, while an application that detects falls for the elderly requires high accuracy predictions to reduce false positives. Activity detection scenarios [38, 39] similarly need to be responsive, implying fast training times, to reduce user annoyance of having to

wait. Ultimately, the DSL needs to holistically manage and monitor the compute resources at its disposal and schedule IoT tasks on different devices.

4.1.1 *Benchmarking Device Performance:*

There are numerous devices and platforms that MLIoT can run on, often with different characteristics. These include inexpensive platforms like a Raspberry Pi, or Hardware accelerators for ML such as Google EdgeTPU [28], Nvidia Jetson [50] (or) Intel Neural compute [33]. Alternatively, MLIoT can also run on traditional VMs in the cloud on Google's Cloud Platform [26], or Amazon AWS [4]. To characterize their *relative* performance, MLIoT needs to benchmark each device. Since the IoT application workloads are not known apriori, we collect and use three different *representative* IoT datasets described in Section §2.1 and Table 2 - audio based activity recognition, a multimodal sensor based activity recognition, and an image recognition application. Our insight is that the relative performance of devices on these representative datasets can be used for device selection of new IoT tasks. For these benchmarks, we train and serve a set of classical ML models as well as deep learning models (as applicable), on each device measuring several key metrics:

- **CPU and Memory Utilization:** We collect the average CPU and memory utilization over the entire benchmark execution and normalize the values between 0 and 1.
- **Prediction Latency:** We measure the average time taken by a training serving worker running on a device to receive a prediction request and respond to it for each benchmark.
- **Training Time and Accuracy:** We measure the training time and accuracy for a set of models, for each benchmark.

Notably, we collect these metrics when instantiating a Training-Serving Layer on each device. The overhead of collecting this data for device selection is low and to ensure that it does interfere with any other tasks, we reserve a fraction of the compute resources (10%) for benchmarking only.

4.1.2 *Device Selection Policy:*

The DSL considers several aspects while scheduling tasks on devices. It uses the *benchmark metrics* from devices as mentioned above, as well as *device metadata* such as the number of cores, memory, as well as the presence of accelerators. Since the available resources on a device change based on other co-located tasks on it, the DSL captures *run time metrics* periodically such as the CPU load (C), Memory (M) and Load Average (LA). This information is provided by the TSL running on each device, to the DSL periodically (every 1s). For these metrics, we calculate an Exponential Weighted Moving Average (EWMA) for a window size of 10 samples to reduce transient noise. Based on the benchmark metrics, the runtime metrics, and the device metadata the DSL exposes several policies for IoT developers to specify their application requirement. These policies are either *Static*, based on static values such as CPU core count etc., or *Dynamic* based on metrics that change such as the CPU load.

- Static Policies:
  - GPU-CPU: Use a machine with GPU (or) a CPU.
  - Max-Min: Select devices with the maximum or minimum of number of CPU cores, Memory or Load.
  - Locality: Select devices based on locality, such as an trusted edge device for privacy concerns or optimizing latency.
  - And-Or: Select devices based on a logical combination of different metrics mentioned above.
- Dynamic Policies:
  - Threshold: Select devices based on specifying *Atleast, Atmost or Equal* threshold conditions. e.g. Latency Atmost 40ms.
  - Best Effort: When no application requirements are specified, the best effort policy chooses the device with the most available resources across the runtime and benchmarking metrics.

The DSL compares the metrics it collects from different devices, with the policy requirements of each IoT application, evaluating them in order of arrival. If the application policy can be satisfied, the appropriate device is selected. However, in case the available resources cannot meet the policy requirements of an IoT application, the DSL sends a negative response back to the application.

4.1.3 *Load balancing Training and Serving workers:*

In many cases, the DSL may be able to choose from multiple devices to meet individual IoT application requirements. In these scenarios, especially with an increasing number of concurrent applications, it is important to efficiently load balance tasks across devices. The DSL uses real-time metrics (e.g. CPU and Memory usage, and Load) as inputs to a dynamic load balancing algorithm based on resource weights [62] to address this challenge.

More formally our load balancing algorithm works as follows. Let's assume there are 'n' devices in a MLIoT deployment. The TSL on each of these 'n' devices reports its current system performance metrics periodically (1s). We calculate the load state using the following formula:

$$L_i = w_{cpui} \times cpu_usage_i + w_{memi} \times memory_usage_i + w_{loadAvgi} \times loadAvg_i \quad (1)$$

- L_i is the Load state of ith Training and Serving Layer.
- cpu_usage_i, w_{cpui} is the percentage CPU utilization with its corresponding weight.
- $memory_usage_i, w_{memi}$ is the percentage Memory utilization with its corresponding weight.
- $loadAvg_i, w_{loadAvgi}$ is the system average load with its corresponding weight.

We calculate the weight dynamically at each fixed cycle to calculate load state precisely. To get the weights we find the minimum value of the metric and divide it by the metric value of the current machine as below:

$$w_{cpui} = \frac{min(cpu_usage_1, cpu_usage_2, ...cpu_usage_n)}{cpu_usage_i} \quad (2)$$

The DSL then selects the device with the least load for the particular IoT application.

4.2 Training-Serving Layer (TSL)

The Training-Serving Layer (TSL) runs on each device in a MLIoT deployment. The TSL spawns a separate manager process to create, monitor and manage a pair of Training Workers (TWs) and Serving Workers (SWs) for each IoT application as shown in Figure 2. The TSL uses Linux CGroups (control groups) [9] to allocate and enforce the resource usage for each Training and Serving worker. The

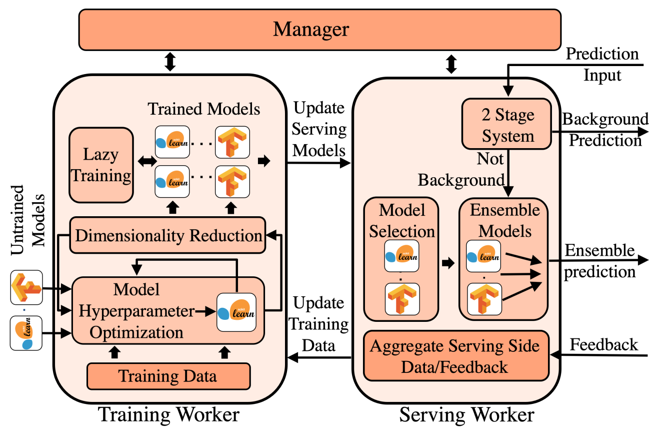

Figure 2: Overall architecture of the Training and Serving Workers. The Training Worker is responsible for training models in parallel, and optimizing hyperparameters, using the input training data. Based on different policies, an ensemble of models is selected and sent to the Serving Worker. The Serving Worker uses the ensemble to make predictions using certainty estimation while aggregating serving side data for feedback from the user.

Training Worker is responsible for the initial model training using labeled data, as well as retraining models when corrective feedback is provided by users to improve accuracy. The Serving Worker obtains the trained models from the TW, creates a model ensemble and performs online predictions. The SW also collects user feedback on the ensemble-based predictions and forwards them to the TW. This close coupling of the TW and the SW for each IoT application is critical to improve accuracy and performance over time, and adapt to dynamic IoT environments. We discuss the Training Worker in §4.2.1 and Serving Worker in §4.2.1.

4.2.1 ***Training Worker (TW)*** *.* A key goal of a TW is to streamline model training to provide high-quality ML models that are tailored to the specific IoT task. The TW uses generic ML model definitions to enable extensibility and allow use of multiple common ML frameworks and their algorithms. The TW selects models and applies various optimizations (discussed below) based on the device resources for faster training, and low latency serving.

Dimensionality Reduction (DR): Dimensionality reduction (DR) is a common technique used in numerous domains to reduce the feature space, reduce model size and complexity, and identifying the most relevant features.

We investigate different DR techniques with the goal of (a) providing better classification accuracy with different types of high dimensional IoT data sources; (b) explore the hyperparameters for various DR algorithms for their efficacy, and finally (c) test whether DR performs efficiently during training. To address these requirements, we evaluate DR methods such as *Principal Component Analysis(PCA)* [36], *Manifold Learning (t-SNE,Isomap,UMAP)* [43, 46, 60] and *Kernel Principal Component Analysis (Kernel PCA)* [48]. PCA [36] captures variations present in the linear subspaces of the original data and computes principal components based on the transformed space with lower dimensions. Though the computation of PCA is inexpensive, it is limited to linear transformations only and thus is ineffective with nonlinear higher dimensional IoT time-series data. Unlike PCA, other DR approaches based on Manifold learning [43, 46, 60] creates a complex nonlinear mapping of the training data based on the overall structure and distribution of data under different distance metrics. t-SNE [43] uses a non-parametric method that does not compute a transformation in the serving data once it is run on the training data. In contrast, Isomap and UMAP are parametric methods with tunable parameters that can be applied to the serving data as well. However, the tunable parameter space is large which leads to a need for an increase in the number of iteration for hyperparameter tuning which takes a longer time than PCA and they converge to different results for every iteration reducing its accuracy.

Kernel PCA [48] overcomes these problems by introducing a kernel function in the PCA algorithm with nonlinear data transformations like UMAP. We compared these DR techniques with different IoT application workloads and found that Kernel PCA with Cosine Kernel was robust at preserving information relevant to the prediction tasks and balanced accuracy and computational overhead (Results in §6.2.2).

Hyper-parameter optimization (HPO): Most ML models expose a rich set of hyperparameters that can be tuned for accuracy and performance. In MLIoT we wanted to explore the efficacy of hyperparameter tuning, especially in combination with DR techniques, for representative IoT data sources. Since the space of hyperparameters for each model can be large and often continuous, it is impractical to enumerate and test all combinations. We explored several techniques like Grid Search, Random Search [7] and Bayesian Search [61] that efficiently reduce the Hyperparameter combinations to search to identify the best hyperparameter to generate a model with high accuracy. We compared the above parameter search techniques and found that Grid Search methods provide better accuracy across different IoT workloads through being computationally expensive.

Figure 2 illustrates the training worker where we perform hyperparameter optimizations for each model, and also do dimensionality reduction for the IoT data itself. We evaluate various combinations of DR and HPO optimizations and report the results in Section §6.2.2. Our results show that DR and HPO used in combination leads to computationally efficient and high accuracy ML models for our representative IoT workloads.

Lazy training: On the serving side, MLIoT uses an ensemble of ML models [19] for better accuracy than using individual models alone. However, using model ensembles requires longer training times to train and optimize all models, leading to higher latency to start serving the first request. To address this, in MLIoT we use a lazy training strategy with a time bound (T) to start serving requests. The parameter T limits the time taken for training models, such that if certain complex models take longer to train they are transferred to a separate training worker to train in the background, in a *lazy* manner, using residual compute resources. To start serving within the specified time bound T, the TW sends the models which have completed to the SW. When the models being trained lazily complete training, the ensemble is re-evaluated. This lazy training strategy increases the accuracy over time while reducing latency to start serving (evaluated in Section §6.2.2)

Model Definitions: MLIoT supports using models from popular ML frameworks such as Scikit Learn [53], TensorFlow [1], PyTorch [52], etc. To enable this extensibility, we designed a flexible Model Definition with a common abstract interface to expose details necessary for both training and serving. This capability allows the use of different frameworks, with uniformity across models while enabling model customization.

TW Model Selection: Similar to Device Selection policies in Section§4.1.2 the training worker selects models based on application policy specification around metrics such as model accuracy and latency, as well as performance metrics such as CPU and memory usage. The goal with TW model selection is to create an ensemble of models to send to the Serving Worker. An example policy can choose three models with the lowest latency, to send to a SW. The TW handles all training related tasks, performing model selection and training, and then continuous model adaptation over time.

4.2.2 *Serving Worker (SW)* .

The Serving Worker performs online predictions using models received from the Training Worker, under the IoT application's policy constraints. The SW also provides a certainty estimate for each prediction based on the models included in the ensemble, for higher accuracy. It also supports a two-stage serving system that is specifically optimized for IoT environments, to improve accuracy due to environmental noise and reduce latency. The SW is also responsible for collecting corrective feedback to send to the TW for model adaptation.

Two-Stage Serving: Most continuous serving IoT scenarios are dominated by periods where no interesting events happens. For example, for activity detection, for 8 hours a day an occupant is likely asleep and in a specific room. In these cases, the IoT ML pipelines are essentially detecting ambient "background". Current IoT systems[39, 49] thus explicitly train for large amounts of background as one of the classes. Furthermore, transient "noise" in the sensor data, is also often misclassified as one of the trained events. Ultimately, this leads to higher latency since all models are evaluated for background and noise events, and lower accuracy when there is noise.

In MLIoT we propose a two-stage serving system. The *First Stage* is a binary classifier with two classes, namely the "Background" and all the other classes of interest for the IoT application. We empirically evaluate several ML models for the first stage and report results for different IoT workloads in Section §6.2.2. We show that Logistic Regression (LR) [45] works best as the first state binary classifier, balancing accuracy and latency. The *Second Stage* uses the model ensemble obtained from the Training Worker and uses it for serving predictions. MLIoT's Two-Stage Serving reduces overall latency when events like background or noise occur.

Ensemble Based Model Prediction: After model selection, the SW uses the final set of models for weighted ensemble prediction, determining model weights based on validation accuracy.

Let the final list of models selected be $m_1, m_2, ..m_n$. We calculate the final ensemble based prediction as:

$$Prediction = argmax_c(\sum_{i=1}^{n} w_i(if\ f(x)_{m_i} = c)) \tag{3}$$

where $f(x)_{m_i}$ is the prediction made by model m_i for input x and w_i is the weight for m_i. The static weights $w_1, w_2, ..w_n$ for the models $m_1, m_2, ..m_n$ are calculated as:

$$w_i = e^{x_i}(\sum_{j=1}^{n} e^{x_j})^{-1} \tag{4}$$

where $x_1, ..x_n$ is the validation accuracy of model $m_1, ..m_n$.

Prediction Certainty Estimation: In several IoT use cases, especially those that actuate devices or send notifications based on ML predictions, knowing the confidence of the predictions can be very useful. To support this, MLIoT provides ensemble-based certainty estimates for each predictionby taking the weighted average of the probability of each class for each model.

$$certainties = (\sum_{i=1}^{n} w_i f_{m_i,c_1}(x), \sum_{i=1}^{n} w_i f_{m_i,c_2}(x), ..., \sum_{i=1}^{n} w_i f_{m_i,c_m}(x)) \tag{5}$$

where $f_{m_i,c_1}(x)$ is the probability that the model m_i predicts for the class c_j on the input x. The estimates are calculated by taking the square magnitude.

Model Adaptation: IoT environments are subject to changes in the ambient environment that affect model accuracy. In addition, users may want to update models with additional training data, including adding a new class or give corrective feedback for an incorrect prediction. MLIoT, performs model adaptation by sending this data to the Training Worker to retrain models. However, naively re-training all models with HPO and DR is expensive and potentially time-consuming. To overcome this challenge, we re-tune each of the models in the ensemble with new data. This significantly reduces the training time and is computationally less expensive. This feedback-based model adaptation allows us to account for changes to the environment and provide better results than using static models.

5 IMPLEMENTATION

We had several goals when implementing MLIoT. We wanted to ensure cross-platform support to run on different devices, extensibility to add different models and frameworks, maintainability of the codebase, and making it scalable and efficient. We implemented MLIoT in Python 3 given its popularity and the availability of several well supported and popular ML frameworks and libraries. To have efficient communication between the different components, in a distributed setting, we tested a number of RPC libraries, selecting Google's gRPC given its maturity, efficiency, scalability, and cross-platform support[29]. Various components in MLIoT also exchange data and models among themselves and we use gRPC Protocol Buffers (Protobuf) extensively for this purpose, again due to its maturity. Additionally, each component is authenticated with each other using gRPCs built-in security primitives. The Device Selection Layer(DSL) also uses a MongoDB data storage layer to store persistent models, training data, and any other state about individual Training Serving Workers. This DB also stores device metrics and other metadata received from the Training Serving Layers. Overall, our implementation of MLIoT is 28,241 lines of code in Python.

6 EVALUATION

We evaluate MLIoT on a set of representative IoT benchmarks, hardware platforms, and varying application requirements. We measure metrics such as accuracy, latency, and resource usage. We describe our experimental setup in Section §6.1. In Section §6.2 we characterize performance across different hardware platforms and compare the efficacy of different training (DR and HPO) and serving (lazy training and two-stage serving) optimizations. In Section §6.3 we present end-to-end evaluations of MLIoT for multiple concurrent IoT benchmark scenarios, using example policies to show how our system load balances and scales while maintaining high accuracy. Finally in §6.4 we compare and evaluate MLIoT with two academic, application-specific, machine learning systems and a commercial general-purpose machine learning system.

6.1 Experimental Setup

Machine Learning Models: We choose several popular ML algorithms and frameworks to show that MLIoT is extensible to use a variety of models. To evaluate MLIoT we select five traditional models (KNN, Ridge Regression, RandomForest, SVM-Linear, SVM-RBF) and two Neural Networks (MLP & XGboost). We integrated the implementations of these algorithms from SKLearn [53], TensorFlow [1], and PyTorch [52] into MLIoT. In our evaluations we use top-1 accuracy, which compares the model's top label choice and assigns a score of 1 for a correct prediction and 0 for an incorrect prediction with no partial scores.

IoT Benchmarks: For our evaluations, we create several representative IoT benchmarks summarized in Table 2. These benchmarks represent IoT applications with labeled data for initial training of models and separate test data (with labels) for calculating accuracy and performance. We chose MNIST, a popular image-based object recognition dataset for the first benchmark[40]. For activity recognition based on rich multi-modal sensors, we collected data using the Mites.io [39, 49] platform with 13 hardware sensors for a set of 16 common residential activities (classes). The Mites.io platform converts the sensor data into 1172 statistical and spectral features. Finally, for activity recognition based on audio we collected and labeled data from a laptop with a microphone, for 14 common residential activities. We use the same features proposed in Ubicoustics[38] wherein audio data from non-overlapping frames (960*ms* each) is converted to a spectral representation to provide log-Mel spectrogram patches of 96×64 bins that form the input to all classifiers [31].

Creating IoT Application workload Traces: Using these IoT benchmarks, we create a set of "application traces" to emulate actual IoT application workloads that each use MLIoT to initiate separate Training-Serving Workers. For these traces, we train all seven ML models using the training data for each benchmark. We then divide the testing data into six parts to create six 'testing application traces' (T1 - T6). For example, every trace for Audio-based activity recognition trains all seven models using 2633 training samples and then uses a subset of the testing samples data (1734/6 = 289) for testing. We use these application traces to evaluate the performance of MLIoT (accuracy, latency) and the efficacy of MLIoT in terms of scheduling to different devices based on application policies and the system workloads (Section §6.3).

Table 2: Summary of IoT Benchmark Data Characteristics

Dataset	Data Type	Features	Training Samples	Testing Samples	Classes
MNIST [40]	Image	28*28	42000	18000	10
Mites [49]	Multimodal	1172	25787	16735	16
Microphone	Audio	96 * 64	2633	1734	14

Table 3: Hardware platforms used in our experiments

Device	Type	Processors	Memory	Average RTT	Local /Cloud
M1	Raspberry-Pi 4	4 x ARM A72	4GB	~3ms	Local
M2	Intel NUC	4 x i5-5250U	8GB	~14ms	Local
M3	Virtual Machine	2 core	4GB	~128ms	Cloud
M4	Virtual Machine	8 core	16GB	~64ms	Cloud
M5	Virtual Machine	16 core	32GB	~84ms	Cloud

Testbed Hardware Platforms: To evaluate MLIoT we set up a five machine testbed (M1-M5 in Table 3) with different hardware configurations, resources, and network latencies (Average RTTs) representing both local (edge) and cloud compute resources. **M1** is an inexpensive Raspberry Pi4 (RPI4) [54] with 4 Cores, 4GB RAM, and 3ms RTT, augmented with the Google's Coral Hardware Accelerator[28] for testing our Device Selection Layer (Section §4.1). **M2** is an Intel NUC desktop with 4 cores, 8GB RAM, and a 14ms RTT . Notably, M1and M2are "local" devices since they are in the same LAN as the client, behind a NAT. **M3, M4, M5** are Virtual Machines (VMs) running on different physical servers. **M3** has 2 Cores, 4GB RAM and 129 ms RTT, **M4** has 8 Cores, 16GB RAM and 64ms RTT and **M5** has 16 Cores, 32GB RAM and 84ms RTT. Our test client machine, which executes all IoT Application traces is a Mac with a dual-core i7 processor and 16 GB RAM.

RTT Calculation: MLIoT uses the Round Trip Time (RTT) from the client to each device, and the prediction latency for the application on each machine for in the device selection layer (Section §6.4). Since our test devices are in the same city, the RTT values were similar. To emulate diversity in RTTs for the test-bed devices we add additional delay during packet network transmission using VirtNet [59].

6.2 Baseline Performance and Accuracy

We first evaluate the different components of MLIoT, to measure their impact in isolation. Specifically, we highlight the differences in model accuracy and performance on different devices used in the Device Selection Layer. We also evaluate our optimizations including DR, HPO, and Lazy-training and compare them to baselines configurations without them.

6.2.1 Device Selection Layer (DSL). The Device Selection Layer (DSL) executes representative IoT benchmarks to collect a set of metrics from each device in a MLIoT deployment. These metrics provide an assessment of each device's relative capabilities and in combination with policies guide device selection. Figure 3 shows the overall CPU usage, memory utilization and training time for all seven models on three different machines (M1, M3, and M4) for three different IoT application workloads. The CPU and memory usage are normalized (Max 1.0) due to different hardware configurations.

We observe that some models (e.g. MLP and Ridge) can better exploit multiple cores to reduce training time. Training time, CPU and memory usage vary widely based on the models, and the relative performance of models is also different per machine. The model

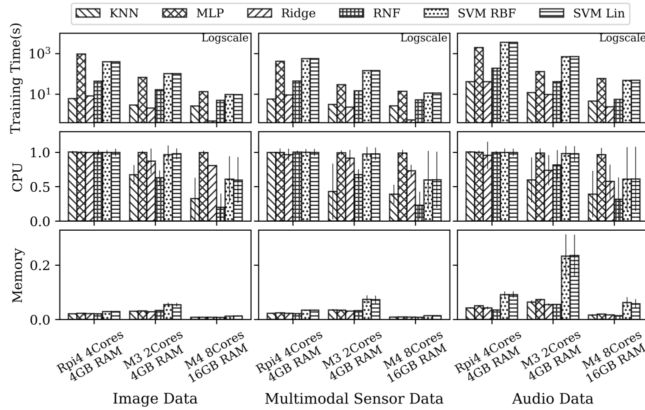

Figure 3: Comparison of machine learning model performance across different data sources. These metrics allows better selection of device and adapting to changing resource environment for the IoT machine learning application.

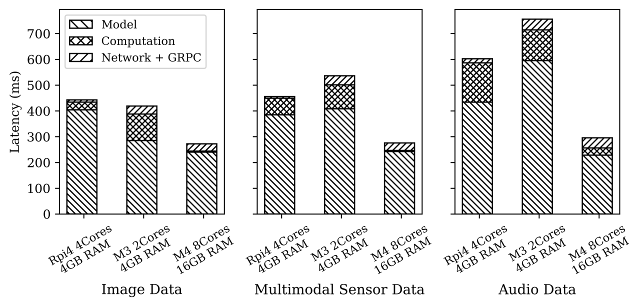

Figure 4: Benchmarking latency across devices.

performance is also dependent on the source data type, for example the Audio data benchmark uses more resources than the others.

Figure 4, breaks down the end-to-end serving latency across machines with more capable devices exhibiting lower values. The latency for a RPI4 with Coral [28] was very high (average 3500ms) despite being optimized for EdgeTPU models, since Coral requires quantized data which consumes significant CPU resources. Thus, we removed Coral from further evaluations. Overall, our results motivate why benchmarking is important for device selection.

6.2.2 Training-Serving Layer (TSL). Next, we evaluate the impact of various MLIoT optimizations on the training worker (DR, HPO, Lazy training) and the serving worker (Two stages, Model Adaptation) by benchmarking them on the testbed hardware machine (M4) with 8 cores and 16GB RAM.

Dimensionality Reduction & Hyper-Parameter Optimization: Figure 5 shows the training time, accuracy and serving latency of using DR and HPO individually, and when used together. The effect of these optimizations are shown for different IoT data types and training set size (200-800 samples) using all the ML models from Section §6.1. We measure training time by training all the models in parallel, while the accuracy and latency values are based on the ensemble based prediction mentioned in Section §4.2.2 measured across different test samples. Individually, DR or HPO provides some benefits. For example, for Audio "HPO-only" provides high

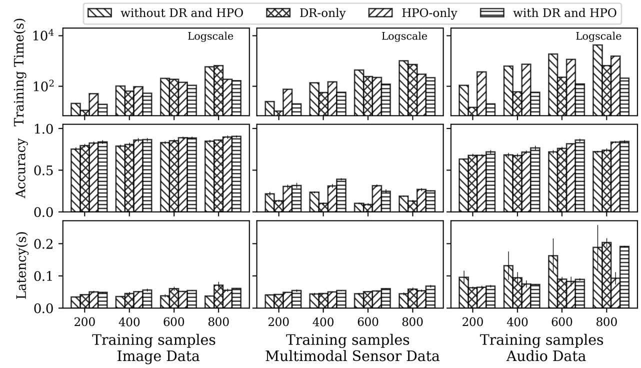

Figure 5: Comparing Training time, Accuracy and Latency using Dimensionality Reduction (DR) (or/and) Hyper Parameter Optimization (HPO) techniques with different training/testing samples for different workloads. Enabling DR and HPO produce models with lower training time and higher accuracy compared to the baseline.

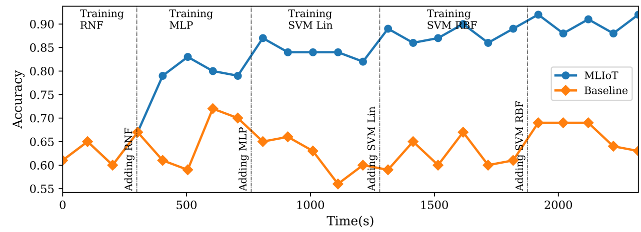

Figure 6: Change in accuracy with Lazy training. The latency to start serving is reduced while accuracy improves as the ensemble is updated with more models trained in the background.

accuracy while reducing serving latency, while "DR-only" reduces training time but increases serving latency due to its computational overhead. Overall, DR and HPO when used together reduce training time, increase accuracy and reduce serving latency across the board as compared to the baseline ("without DR and HPO").

Lazy Training: Figure 6 shows the benefit of lazy training for a sample benchmark (Audio Dataset). MLIoT starts serving quickly, similar to a system serving a single model. Over time MLIoT lazily trains other models and adds them to the ensemble, improving accuracy, with higher serving latency (not shown).

Two-Stage Serving MLIoT uses a two-stage serving system. The 1st stage is a binary classifier to detect the background vs. other classes, while the 2nd stage uses the full ensemble. In Figure 7 we evaluated several binary classifiers for the 1st stage (left) accross different IoT workloads. We also explore the effect of increasing the number of classes in the audio dataset (middle) and the training examples (right). Logistic Regression (LR) [45] works best across the workloads, with low latency and high accuracy. Figure 7 also shows that LR is unaffected by class imbalance (middle) or increase in the number of samples (right). For an example IoT workload, we measure an average serving latency of 65ms with the 2-stage system enabled, as compared to 105ms without it while maintaining the same accuracy.

In Figure 8, we illustrate an audio application IoT workload with two-stage serving enabled and without it as a baseline. Here, the Stage one is a binary classifier trained to just detect "background"

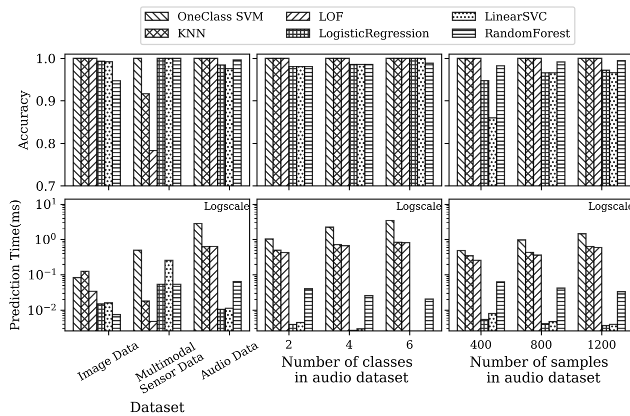

Figure 7: Two-stage serving: Comparing Accuracy and Prediction Latency for different Stage-1 classifiers on different workloads (left). For Audio dataset, we show effect of increase in classes (middle) and Samples (right). Logistic Regression performs best.

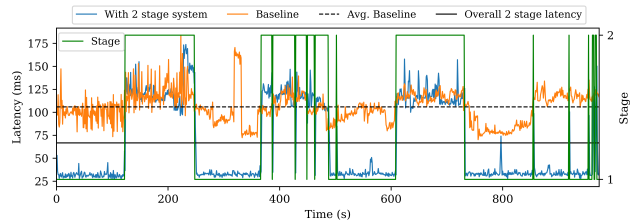

Figure 8: Effect of Two-stage serving on the End-to-End Latency of MLIoT and the corresponding stage where the prediction is made (green line). The Two-stage system's binary model classifies background with lower latency when compared to baseline.

or not, while the Stage two is the normal ensemble-based model prediction. We observe that the average latency with a two-stage system is 65ms, compared to the average latency of the baseline system of 105ms. Furthermore, the prediction accuracy for the two-stage system for this scenario is comparable (close to 100%) to the baseline system without the two stage serving (accuracy graph not shown due to space constraints).

Model Adaptation Next we evaluate how MLIoT adapts to changes in the environment, receiving corrective feedback, and dynamically updating models. We use an audio based activity recognition task, trained on a data set of activities,and then tested on an example activity ("Shaver running"), while introducing different types of background noise into the system. Figure 9, shows the timeline of this application, annotated with times when noise is introduced and the user provides corrective feedback. With corrective feedback (denoted with a blue line) the models in the ensemble are updated and MLIoT is able to predict the activity accurately. In contrast, the accuracy for the baseline case (denoted with an orange line) without model adaptation drops with background noise.

6.3 System Adaptation and Scaling

To assess the overall performance and adaptability of MLIoT, we evaluate MLIoT with different IoT application workloads and policy

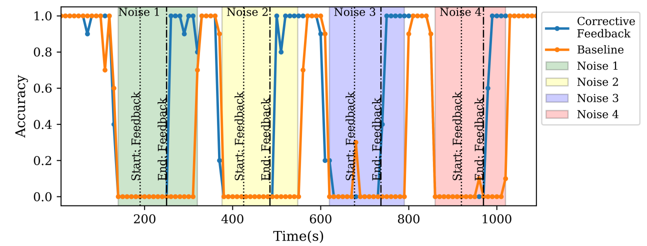

Figure 9: Effect of corrective feedback (dashed lines) on accuracy when additional noise (dotted lines) occur in an IoT environment.

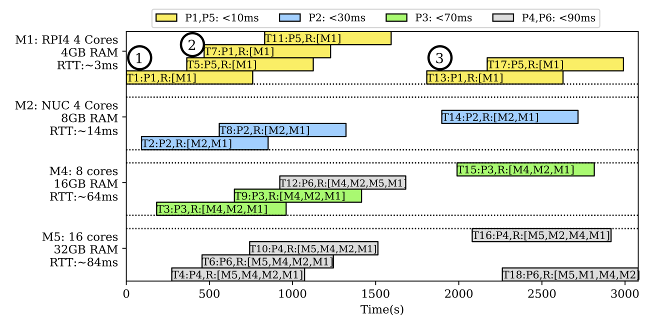

Figure 10: Execution Timeline of MLIoT with six latency-bound application traces (Ts) with four policies having Latency threshold values ranging from 10ms, 30ms, 70ms and 90ms which are executed for three iterations.

requirements Section §6.3.1. We then compare MLIoT with other academic and commercial ML systems.

6.3.1 System Adaptation: An overarching goal of MLIoT is to effectively schedule different IoT applications, that are concurrent, on different devices given their individual policy requirements. We categorize three typical application policy requirements: (a) strictly latency bound, with specified thresholds; (b) those that require "edge" devices in their local network for low latency and for privacy (e.g. speech recognition or activity recognition); and (c) those with more complex requirements as a function of latency, and resource usages and accuracy. For this evaluation, we create a set of machine learning tasks using the six IoT application workload traces described in Section §6.1 , each with different representative policies attached (as discussed in Section 4.1.2). We use four devices (M1, M2, M4, M5) with different RTT values and configurations emulating an example MLIoT deployment (details in Table 3).

Figure 10 shows the execution timeline of a set of latency bound application traces on our test MLIoT deployment. We have three iterations of six application traces (T1-T6, T7-T12, T13-T18), with each trace specifying a policy with a latency threshold of either 10ms, 30ms, 70ms or 90ms. The groups of six traces are spawned at 3 different time intervals as annotated in the graph as (1, 2, and 3). At (1), T1 with a policy of $< 10ms$ is spawned (denoted as T1:P1), and the DSL schedules it on M1 since the RTT from the client to M1 is the lowest (~3ms) among all the devices that met the policy requirement. MLIoT also ensures that there are sufficient residual resources on M1 to run a new ML task using the benchmarking process for each device.As more application traces (T2:P2; T3:P3; T4:P4;) are

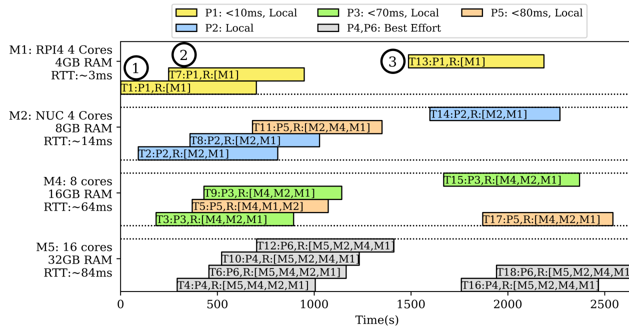

Figure 11: Execution timeline of MLIoT for six latency-bound application traces (Ts) with some traces having additional requirement to run on local devices (M1 and M2).

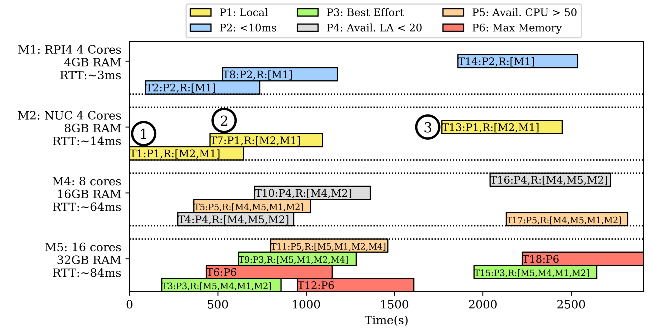

Figure 12: Timeline of the six application traces (Ts) where all the traces have diverse policy requirements. Traces have policy constraints on latency, resource and best effort.

spawned, MLIoT schedules them on different devices depending on their individual policy requirements and resources available on the device. In the second iteration, denoted as (2), we see that most other traces are deployed on devices based on the load balancing metrics adapting to the current resource in the system. For example, T10:P4 has a latency policy such that all the devices qualify but, MLIoT picks M5 (as shown by the Ranks [M5>M4>M2>M1]) due to its hardware configuration. Notably, MLIoT schedules application trace T12:P6 on M4 since the benchmarking values show higher latency values for T12:P6 on M5, M2, and M1 due to the resource consumption of the previous traces. At iteration (3), all other traces have completed and traces T12-18 are thus scheduled on the same devices as those for T1-T6.

Figure 11 shows the execution timeline of application traces with hybrid latency policies.Similar to Figure 10, in Figure 11 we have three iteration of six traces (T1-T18). Some traces come with policies (P1, P3, P5) with latency requirements of $< 10ms$, $< 70ms$, and $< 80ms$ and an additional preference to run locally when possible. Traces with policy P2 restricts picking only local devices (for privacy). The remaining traces with policies P4 and P6 are best effort with no restrictions on latency or locality. At time (1), the first iteration of traces T1 - T6 are spawned and they are scheduled similar to the previous graph; MLIoT picks M1 for T1:P1, M2 for T2:P2, M4 for T3:P3, and M5 for T4:P4, based on latency constraints and available resources. At iteration (2) traces T7-T12 are spawned and we see that the device selection changes due to traces with a local policy. MLIoT tries to pick devices which are local (M1, M2) despite other device satisfying the latency constraint. For example, MLIoT selects M2 over M4 to run the application trace T11:P5 to run the traces locally.

Figure 12 shows application traces with further diverse policy requirements based on on locality, compute resources, and latency metrics. These policy requirements include select devices with available CPU percentage (e.g. > 50%) or a device with maximum available memory. At (1), we see that MLIoT selects M2 over M1 for T1:P1 to satisfy the locality constraint without a latency requirement. T4:P4 is executed on M4 as the load average for M4 was less than 20 and similarly, T5:P5 is run on M4 as the available CPU resources was more than 50. This is indicated by the rank list for T4:P4 as R = [M4,M5 and M2] and T5:P5 as R = [M4>M5>M1>M2]. In the 2nd iteration (2), we see that T11:P5 is run on M5 instead of M4 based on the available resources. Traces (T6:P6, T12:P6, T18:P6) with policy of picking a device with maximum available memory is always executed on M5 given its 32GB RAM configuration.

6.3.2 System Scaling: Finally, we evaluate the scalability and adaptation of MLIoT with an increasing number of co-located IoT tasks and their effect on the end-to-end serving latency. We create these co-located IoT machine learning tasks using the Audio IoT application workload trace described in Section §2.1. For this evaluation, we use three devices (M1, M3, M4) with different device configurations. We also remove any delays that we added in the prior experiments to emulate different RTTs. As a result, the end-to-end latency values for this evaluation are due to the co-located IoT tasks on the same device. We execute 1 to 10 co-located tasks with and without a latency threshold policy (<250ms) for serving on each device. For each IoT task, we initialize both the training and serving instances, load all the models for serving, and then send the data for predictions so as to remove the initial overhead of loading the models in the memory.

Figure 13 (top) shows that as the number of concurrent training serving instances increases from 1 to 10, the average latency also increases almost linearly across all devices. Notably, the standard deviation of the serving latency is low, depicting that the different tasks share the resources on each device fairly. Next, we instantiate the same number of IoT tasks, from 1 to 10, but with a latency threshold policy of (<250ms) for each task (Figure 13 (bottom)). Our results show that MLIoT (DSL) does not schedule any more additional tasks on a machine since servicing them would lead to a higher serving latency than the threshold of 250ms as per the policy. This illustrates how MLIoT adapts to the requirements for the IoT tasks and their policies. For example, the DSL does not schedule more than 4 IoT tasks on M1, or more than 7 tasks on M3, since the average latency would exceed the policy limit of 250ms (for each task) with additional co-located tasks.

6.4 Comparison with other ML Systems

We compare and evaluate MLIoT with two academic, application-specific, ML systems and a commercial general-purpose ML system. For our comparison with application-specific ML systems we use two state-of-the-art activity detection systems, Ubicoustics [38] which uses a *pre-trained* model for audio data and Mites.io [39,

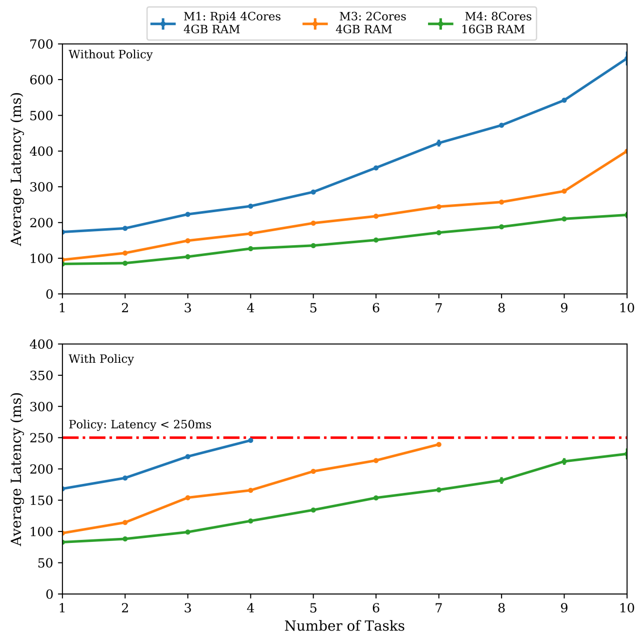

Figure 13: Scalability of MLIoT with an increasing number of co-located IoT tasks on different devices, M1, M3 and M4. The IoT tasks are scheduled without a policy (*top*), and with a latency threshold policy of < 250ms (*bottom*).

49] which uses *supervised* model for multi-modal sensor data. We then compare MLIoT with Google's TensorFlow Extended (TFX) framework [6], a general purpose machine learning system that extends TensorFlow Serving [51] to serve models. We compare the MLIoT with the above systems with two different policies - Lowest Latency and Best-Effort to show the flexibility of MLIoT. The results of the comparison are shown in Table 4.

For fair comparison between these systems we ensure that the testing and training data are the same for each system and they run on the same or similar hardware configuration if running on the cloud. To compare MLIoT with Ubicoustics, we used the pre-trained model that comes with it[38] and for MLIoT, we train all the models with labelled audio data as mentioned in Section §6.1 and Section §2. For both Ubicoustics and MLIoT, we use the same test data for serving. We observe that the "MLIoT-Best Effort" policy improves accuracy to 89% over Ubicoustics (52%), albeit with slightly higher latency: 0.1s (MLIoT) vs 0.08s (ubicoustics). Our "MLIoT-Low Latency" policy picks a 3 model ensemble to reduce latency to 0.06s with comparable accuracy to the best effort policy. To compare the commercial TFX system with MLIoT, we choose similar system configuration as M4on the cloud (8 Cores, 16GB RAM), and trained and tested using the same audio data. For TFX, we chose Tensorflow based classical models, which are the same as the one MLIoT for a fair comparison noting that both systems can be extended to use more complex deep models. We note that the TFX accuracy for this dataset is lower (67%) than either MLIoT best effort or low latency policies, with significantly higher prediction latency of 0.35s. To compare MLIoT with the Mites IoT based activity recognition system, we collected, trained and tested both the systems using the same multimodal sensor data described in Table 2 on the same 8

Table 4: Comparing MLIoT with other ML systems: Ubicoustics [38] Mites.io [49] and a general-purpose system: TensorFlow Extended [6]. Numbers in parenthesis are percentage increase/decrease in accuracy (higher is better) or latency (lower is better).

System	Top 1 Accuracy	Latency (s)
Audio Data		
Ubicoustics [38] - Pretrained Model	0.52	0.08
MLIoT- Best Effort (7 ensemble)	**0.89 (+71%)**	0.1 (+25%)
MLIoT- Low Latency (3 ensemble)	0.86 (+65%)	**0.06 (-25%)**
TFX [6] - General Purpose	0.67 (+29%)	0.35 (+337%)
Multi-modal Sensor Data		
Mites [49] - Supervised Model	0.48	0.05
MLIoT- Best Effort (7 ensemble)	**0.84 (+75%)**	0.09 (+80%)
MLIoT- Low Latency (3 ensemble)	0.72 (+50%)	**0.04 (-20%)**

core machine M4. We observe that the "MLIoT-Best Effort" policy improves prediction accuracy to 84%, compared to the Mites baseline accuracy of 48%, although at a higher latency of (0.09s vs 0.05s). "MLIoT-Low Latency" further reduce latency to 0.04s with accuracy dropping somewhat to 72% as compared to the MLIoT best effort case, but still better than the Mites baseline. Overall, these results show that MLIoT is significantly more accurate than these hand tuned systems[38, 49], while being comparable or better in terms of serving latency.

7 CONCLUSION

In this paper, we design and implement MLIoT, an end-to-end Machine Learning System tailored towards supporting the entire lifecycle of IoT applications from training, efficient serving to re-training based on user feedback. MLIoT integrates multiple distributed components and optimization techniques making our system adaptive, dynamic, and well suited to handle the diversity of IoT use cases. MLIoT provides flexible policy driven selection of hardware platforms, ML models, and various optimizations for closely coupled training and serving tasks. Our evaluations of MLIoT on several hardware devices, and for a set of expressive IoT benchmarks, show that MLIoT is able to service different policy objectives, balancing load across devices while maintaining accuracy and latency.

Acknowledgements: This research has been supported in part by NSF award SaTC-1801472. We would like to thank our shepherd and the IoTDI reviewers for their helpful feedback. We would also like to thank the members of SynergyLabs, especially Dohyun Kim, for their input on the early iterations of this work.

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.
[2] Zeeshan Ahmed et al. 2019. Machine Learning at Microsoft with ML. NET. In *Proc. of the 25th ACM SIGKDD Internat. Conference on Knowledge Discovery & Data Mining*. ACM, New York, NY, USA, 2448–2458.
[3] Amazon. 2020. What Is Alexa? https://developer.amazon.com/en-US/alexa.
[4] Amazon AWS. 2020. Amazon Rekognition – Video and Image - AWS. https://aws.amazon.com/rekognition.
[5] Amazon AWS. 2020. IoT Greengrass. https://aws.amazon.com/greengrass/.
[6] Denis Baylor et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proc. of the 23rd ACM SIGKDD Internat. Conference on KDD* (Halifax, NS, Canada) *(KDD '17)*. ACM, New York, NY, USA, 1387–1395. https://doi.org/10.1145/3097983.3098021
[7] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, Feb (2012), 281–305.
[8] Andreas Bulling, Jamie A Ward, Hans Gellersen, and Gerhard Troster. 2010. Eye movement analysis for activity recognition using electrooculography. *IEEE transactions on pattern analysis and machine intelligence* 33, 4 (2010), 741–753.

[9] Cgroups. 2020. Control Groups — The Linux Kernel. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html.
[10] Carl Chalmers, Paul Fergus, et al. 2020. Detecting activities of daily living and routine behaviours in dementia patients living alone using smart meter load disaggregation. *IEEE Transactions on Emerging Topics in Computing* (2020).
[11] Trishul Chilimbi et al. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 571–582.
[12] Giorgio Conte et al. 2014. BlueSentinel: A First Approach Using IBeacon for an Energy Efficient Occupancy Detection System. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings* (Memphis, Tennessee) *(BuildSys '14)*. Association for Computing Machinery, New York, NY, USA, 11–19. https://doi.org/10.1145/2676061.2674078
[13] Daniel Crankshaw et al. 2014. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. *CoRR* abs/1409.3809 (2014). arXiv:1409.3809 http://arxiv.org/abs/1409.3809
[14] Daniel Crankshaw et al. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proc. of the 14th USENIX Conf. on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 613–627.
[15] Daniel Crankshaw et al. 2018. InferLine: ML Inference Pipeline Composition Framework. *CoRR* abs (2018). arXiv:1812.01776 http://arxiv.org/abs/1812.01776
[16] Everton de Matos et al. 2020. Context information sharing for the Internet of Things: A survey. *Computer Networks* 166 (2020), 106988.
[17] Jeffrey Dean et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[18] Christian Debes et al. 2016. Monitoring activities of daily living in smart homes: Understanding human behavior. *IEEE Signal Process. Mag.* 33, 2 (2016), 81–94.
[19] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *Internat. workshop on multiple classifier systems*. Springer, 1–15.
[20] Mohamed Faisal Elrawy, Ali Ismail Awad, and Hesham F. A. Hamed. 2018. Intrusion Detection Systems for IoT-Based Smart Environments: A Survey. 7, 1, Article 123 (Dec. 2018), 20 pages. https://doi.org/10.1186/s13677-018-0123-6
[21] Antti J Eronen et al. 2005. Audio-based context recognition. *IEEE Transactions on Audio, Speech, and Language Processing* 14, 1 (2005), 321–329.
[22] Anthony Fleury et al. 2010. SVM-Based Multimodal Classification of Activities of Daily Living in Health Smart Homes: Sensors, Algorithms, and First Experimental Results. *Trans. Info. Tech. Biomed.* 14, 2 (March 2010), 274–283. https://doi.org/10.1109/TITB.2009.2037317
[23] Daniel Golovin et al. 2017. Google vizier: A service for black-box optimization. In *Proc. of the 23rd ACM SIGKDD Internat. conference on KDD*. 1487–1495.
[24] Google. 2020. TensorFlow Core | Machine Learning for Beginners and Experts. https://www.tensorflow.org/overview.
[25] Google Cloud. 2020. Cloud AutoML. https://cloud.google.com/automl.
[26] Google Cloud. 2020. Derive Insights via ML. https://cloud.google.com/vision.
[27] Google Nest. 2020. Google Nest Smart Speakers & Displays - Google Store. https://store.google.com/product/google_home.
[28] Google Research. 2020. Coral: toolkit to build AI products. https://coral.ai/.
[29] gRPC. 2020. gRPC-Overview. https://grpc.io/docs/.
[30] Sidhant Gupta et al. 2010. ElectriSense: Single-Point Sensing Using EMI for Electrical Event Detection and Classification in the Home. In *Proc. of the 12th ACM Internat. Conference on Ubiquitous Computing* (Copenhagen, Denmark) *(UbiComp '10)*. ACM, New York, NY, USA, 139–148. https://doi.org/10.1145/1864349.1864375
[31] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, et al. 2017. CNN architectures for large-scale audio classification. In *2017 IEEE Internat. Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 131–135. https://doi.org/10.1109/ICASSP.2017.7952132
[32] Peter Hevesi et al. 2014. Monitoring household activities and user location with a cheap, unobtrusive thermal sensor array. In *Proc. of the 2014 ACM Internat. joint conference on pervasive and ubiquitous computing*. 141–145.
[33] Intel. 2020. Intel® Neural Compute Stick: A Plug and Play Development Kit for AI Inferencing. https://software.intel.com/en-us/neural-compute-stick.
[34] Jeffrey Dunn. 2020. FBLearner Flow: Facebook's AI backbone. https://engineering.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/.
[35] Jeremy Hermann and others. 2020. Meet Michelangelo: Uber's Machine Learning Platform. https://eng.uber.com/michelangelo-machine-learning-platform/.
[36] Ian T Jolliffe. 1986. Principal components in regression analysis. In *Principal component analysis*. Springer, 129–155.
[37] Rushil Khurana et al. 2018. GymCam: Detecting, recognizing and tracking simultaneous exercises in unconstrained scenes. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 1–17.
[38] Gierad Laput, Karan Ahuja, Mayank Goel, and Chris Harrison. 2018. Ubicoustics: Plug-and-Play Acoustic Activity Recognition. In *Proc. of the 31st Annual ACM Symposium on UIST* (Berlin, Germany) *(UIST '18)*. ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/3242587.3242609
[39] Gierad Laput, Yang Zhang, and Chris Harrison. 2017. Synthetic Sensors: Towards General-Purpose Sensing. In *Proc. of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. ACM, New York, NY, USA, 3986–3999. https://doi.org/10.1145/3025453.3025773
[40] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/
[41] Xin Lei, Andrew Senior, Alexander Gruenstein, and Jeffrey Sorensen. 2013. Accurate and compact large vocabulary speech recognition on mobile devices. (2013).
[42] Mu Li et al. 2014. Scaling Distributed Machine Learning with the Parameter Server *(OSDI'14)*. USENIX Association, USA, 583–598.
[43] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.
[44] Mohammad Saeid Mahdavinejad et al. 2018. Machine learning for Internet of Things data analysis: A survey. *Digital Communications and Networks* 4, 3 (2018), 161–175.
[45] P. McCullagh et al. 1989. *Generalized Linear Models*. Chapman & Hall, London.
[46] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Großberger. 2018. UMAP: Uniform Manifold Approximation and Projection. *Journal of Open Source Software* 3, 29 (2018). https://doi.org/10.21105/joss.00861
[47] Microsoft. 2020. Azure IoT. https://azure.microsoft.com/en-us/overview/iot/.
[48] Sebastian Mika et al. 1998. Kernel PCA and De-Noising in Feature Spaces *(NIPS'98)*. MIT Press, Cambridge, MA, USA, 536–542.
[49] Mites.io. 2020. Mites.io: a full-stack ubiquitous sensing platform. https://mites.io/.
[50] Nvidia. 2020. Jetson Nano Developer Kit. https://developer.nvidia.com/embedded/jetson-nano-developer-kit.
[51] Christopher Olston, Noah Fiedel, et al. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).
[52] Adam Paszke et al. 2017. Automatic differentiation in PyTorch. (2017).
[53] F. Pedregosa, G. Varoquaux, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[54] Raspberry Pi Foundation. 2020. Raspberry Pi 4 Model B. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.
[55] Joseph Redmon et al. 2016. You only look once: Unified, real-time object detection. In *Proc. of the IEEE conference on computer vision and pattern recognition*. IEEE, Las Vegas, NV, USA, 779–788.
[56] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. IEEE, Montreal, Quebec, Canada, 91–99.
[57] Manaswi Saha et al. 2014. EnergyLens: Combining Smartphones with Electricity Meter for Accurate Activity Detection and User Annotation. In *Proceedings of the 5th International Conference on Future Energy Systems* (Cambridge, United Kingdom) *(e-Energy '14)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/2602044.2602058
[58] Samsung. 2020. SmartThings. https://www.smartthings.com.
[59] sandilands. 2020. Virtual Networking. https://sandilands.info/sgordon/virtnet.
[60] Vin de Silva and Joshua B. Tenenbaum. 2002. Global versus Local Methods in Nonlinear Dimensionality Reduction. In *Advances in neural information processing systems (NIPS'02)*. MIT Press, Cambridge, MA, USA, 721–728.
[61] Jasper Snoek et al. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. Curran Associates, Inc., Lake Tahoe, NV, 2951–2959.
[62] M. Song, H. Li, and H. Wu. 2015. A Decentralized Load Balancing Architecture for Cache System. In *2015 Internat. Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. IEEE, Xi'an, China, 114–119. https://doi.org/10.1109/CyberC.2015.44
[63] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. 2017. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd Internat. conference on data engineering (ICDE)*. IEEE, IEEE, New York, NY, USA, 535–546.
[64] Edward J. Wang et al. 2015. MagnifiSense: Inferring Device Interaction Using Wrist-Worn Passive Magneto-Inductive Sensors. In *Proc. of the 2015 ACM Internat. Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2750858.2804271
[65] Wei Wang, Jinyang Gao, Meihui Zhang, et al. 2018. Rafiki: Machine Learning as an Analytics Service System. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 128–140. https://doi.org/10.14778/3282495.3282499
[66] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 446–460. https://doi.org/10.14778/3297753.3297763
[67] M. Xu, S. Alamro, T. Lan, and S. Subramaniam. 2017. LASER: A Deep Learning Approach for Speculative Execution and Replication of Deadline-Critical Jobs in Cloud. In *2017 26th Internat. Conference on Comp. Comm. and Networks (ICCCN)*. IEEE, Vancouver, BC, 1–8. https://doi.org/10.1109/ICCCN.2017.8038373
[68] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 2.
[69] Matei Zaharia et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.