Pushing Down Machine Learning Inference to the Edge in Heterogeneous Internet of Things Applications

Anthony Thomas[1] Yunhui Guo[1] Yeseong Kim[1] Baris Aksanli[2]
Arun Kumar[1] Tajana S. Rosing[1]
[1]University of California, San Diego [2]San Diego State University
{ahthomas, yug185, yek048, arunkk, tajana}@eng.ucsd.edu, baksanli@sdsu.edu

ABSTRACT

Networked applications with heterogeneous sensors on the edge, popularly called the Internet of Things (IoT), are a growing source of data. Such applications now use machine learning (ML) to make streaming predictions. The current dominant approach to deploying ML in most IoT applications is monolithic–features from all sensors are collected in a centralized cloud-based tier to stitch the whole feature vector for ML inference. This approach has high communication costs, which wastes energy on the edge devices and often bottlenecks the network. In this work, we study an alternative approach that mitigates such issues by "pushing down" ML inference queries through a hierarchy of devices to the edge as much as possible. Our approach presents a new technical challenge of rewriting ML inference without significantly reducing prediction accuracy. We provide the first comprehensive characterization of several popular ML models in terms of their amenability to such push down rewrites based on their communication cost–computation cost-accuracy trade-offs. We introduce novel exact rewrite algorithms for some popular models that preserve accuracy. We also create novel approximate variants of other models that still offer high accuracy. Our rewrites reduce communication cost by up to 90%, while a real prototype on a common edge device in IoT networks shows that our techniques reduce energy use and latency by up to 67%.

PVLDB Reference Format:
Anthony Thoma, Yunhui Guo, Yeseong Kim, Baris Aksanli, Arun Kumar, and Tajana S. Rosing. Pushing Down Machine Learning Inference to the Edge in Heterogeneous Internet of Things Applications. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

The explosion in the number and variety of networked sensors, collectively called the Internet of Things (IoT) is causing a proliferation of data in applications ranging from "smart" homes and cities to personalized healthcare and wildfire monitoring [1, 2, 3]. To analyze data at this scale and speed, machine learning (ML) techniques are typically used by such applications [4, 5, 6]. ML models help predict a quantity of interest for the IoT application, e.g., human activity, power consumption, wildfire events, and traffic congestion events. In a *heterogeneous* IoT application, this prediction is performed by combining information from heterogeneous sensors that sense different *features*. Such ML models are periodically trained offline and then deployed for predictions on live streaming data. In the state of the art, for a single prediction request, all types of sensors *communicate* their respective features over a network to a centralized cloud-based application tier. The cloud tier then *stitches* together the whole *feature vector* (essentially, a streaming key-key join) and then performs *ML inference* using a given learned ML model. Such "monolithic" inference is illustrated in Figure 1(B.ii).

The monolithic approach has at least three key issues. First, communication is expensive in terms of *energy cost* on the edge devices. The more data they have to communicate, the more rapidly battery life is drained, which makes it more cumbersome and costly to operate the IoT application. Second, in some applications, the volume of data transmitted can saturate the network, either crippling the system or leading to increased costs from provisioning extra resources. Third, some applications have privacy constraints that might be difficult to satisfy, e.g., a "smart" home user may not want their raw data to leave the home.

To mitigate the above issues, we recently introduced a modular "hierarchical" approach for heterogeneous IoT applications that moves computation out of the cloud and closer to the edge [7]. Our approach introduced new software running on edge devices that we call "context engines" that compute a reduced representation of input data. Input to these context engines may come from sensors connected by a cheap short-range link (e.g., direct contact or Bluetooth) or the output of other context engines. Thus, context engines can be organized in a multi-layer hierarchy that abstracts the raw data successively, as illustrated by Figure 1(A). Edge devices running context engines have CPUs and bi-directional wireless communication - a representative example of such a device common in IoT applications is Raspberry Pi 3 [8].

While hierarchical architectures have been studied before for homogeneous sensor networks [5, 9] and for simple SQL aggregates in heterogeneous IoT networks [10, 11], our work is the first to study the problem of pushing *ML inference queries* down through a hierarchical heterogeneous IoT net-

Proceedings of the VLDB Endowment, Vol. 12, No. xxx

DOI: https://doi.org/TBD

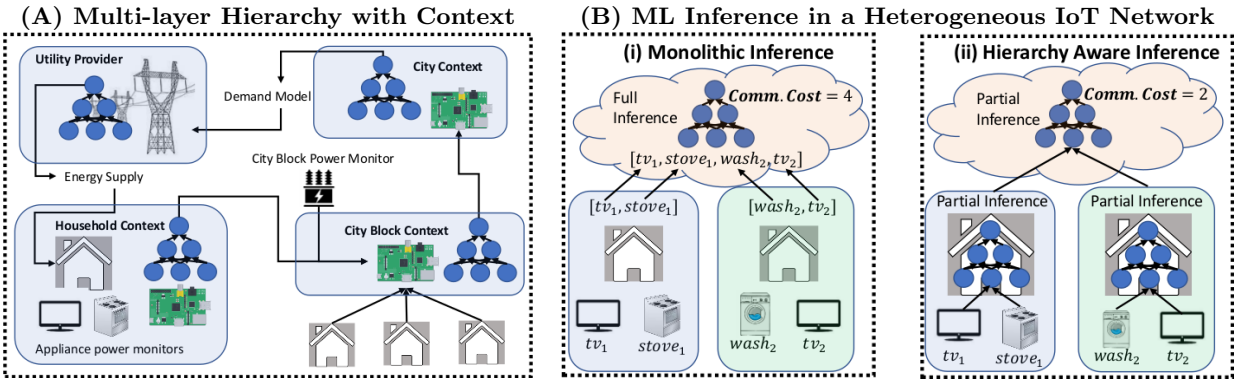

Figure 1: (A) An example heterogeneous IoT application for predicting the (future) power consumption of a neighborhood. It has various sensors from different appliances within homes and a context engine at the bottom tier of a home (the edge devices). There is a hierarchy of aggregation tiers for clusters of homes (city block and city) and finally, a predictive model running in the cloud tier. In the monolithic approach, each edge device communicates all of its sensed feature values up the hierarchy to the cloud tier, where ML inference is performed for further action. (B.i) Simplified example of monolithic inference in a simple 2-tier hierarchy with appliance-specific power use features. All features are communicated to the cloud, which stitches together the full feature vector for ML inference. (B.ii) In our proposed hierarchy-aware approach, ML inference is rewritten and "pushed down" through the hierarchy closer to the edge devices that may contain multiple sensors. This could substantially reduce communication cost, save energy on the edge devices, and ease network congestion. In this example, the *abstract communication cost* goes down from $d = 4$ to $m = 2$ numbers.

work in depth. In its simplest form, the hierarchy has only the sensors, a single layer of context engines, and the cloud tier. Our goal is the following: *Rewrite ML inference to perform partial inference on each context engine using only the features gathered by its local sensors, followed by post-computation at the cloud tier to obtain the prediction result.* Figure 1(B.ii) provides a simplified example of such "pushed down" inference. Our approach could reduce communication cost, since edge devices now communicate a smaller amount of data, which lower their energy use, improves battery life, and reduces network congestion.

Realizing the full generality of our above approach requires tackling the following technical question: *How to push down ML inference without significantly affecting prediction accuracy?* This is analogous to push-down query rewrites in RDBMSs for queries with joins [12]. But there are two key differences. First, instead of reducing computation cost, we seek to *reduce communication cost*. Second, unlike relational queries, ML predictions are inherently statistical and imprecise, which allows for *new accuracy trade-offs*. We showed recently that it is easy to push down linear regression inference exactly [7]. Its push down rewrite is similar to the recent idea of "factorized learning" for pushing ML training down through joins [13, 14, **?**, **?**]. While our setting can be seen as a streaming join (an edge device is a virtual table), factorized learning techniques are not applicable here for two reasons. First, ML inference often has different data access patterns than learning, necessitating new rewrites. Second, factorized learning reduces computation cost by creating large intermediate states to avoid computational redundancy. In our case, there is likely no computational redundancy (the join is key-key), while large intermediate states could raise communication cost.

In this paper, we take a first principles approach to the above technical question and characterize a large number of popular ML models in terms of their amenability to pushing down inference. We face two technical challenges. First, there is immense *diversity in the types of ML models* popular in practice [15]. Thus, rewrites designed for one model may not apply to others. This requires us to group the ML models in a more mathematically abstract way based on the data access patterns of their inference computations. Second, if a model is not amenable to the push down rewrite we seek, we need to *devise approximations that still offer high accuracy*. Handling all possible ML models is beyond the scope of one paper. We pick a set of supervised classification and regression models, listed in table 1 that are popular for real-world ML tasks based on the recent Kaggle survey of data scientists [16] and which are representative of major types of ML models vis-a-vis their data access patterns for inference.

To tackle the first challenge, we create a *taxonomy* of ML models based on their amenability to pushing down inference in terms of communication cost, accuracy, and potential extra computation cost. We identify three groups, as listed in Table 1: (1) Models that can be pushed down fully with maximum communication savings, (2) Models that are not amenable to push down rewrites in general, and (3) Models that can be pushed down but the rewritten inference might have prohibitively high communication cost in practice. The first group includes linear models with no feature interactions, Naive Bayes, and decision tree-based models. We introduce a *novel rewrite algorithm* for tree models that ensures prediction accuracy is unaffected, while communication cost is minimized. The second group includes models with dependencies between features such as linear models with feature interactions and Bayesian Networks. The third group includes such popular models as kernel support vector machines (SVMs) and most forms of neural networks. Naive push-down rewrites for these models might have too high a communication cost. Thus, in response to the second challenge of introducing approximations, we present several novel variants of "hierarchy aware" neural networks that enable users to gracefully trade off between communication cost and accuracy.

Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first work to characterize the amenability of many popular ML

models to having their inference queries pushed down to the edge in a hierarchical heterogeneous IoT setting. We create a taxonomy of models based on communication cost–accuracy–computation trade-offs involved.

- We present a novel rewrite algorithm for full push down of decision tree inference. We explain why many other popular ML models are not amenable in general to efficient push downs. We present novel hierarchy-aware variants of popular neural network models that support accuracy-communication cost trade-offs.
- We conduct an extensive empirical study with three datasets from real IoT applications and analyze the accuracy–communication cost trade-offs of all pushed down ML models. Overall, we find that our techniques can reduce communication costs by up to 90% without affecting accuracy significantly.
- We prototype our ideas on a Raspberry Pi 3, a popular edge device in IoT networks. Our techniques yield reductions in latency and energy use of up to 67% relative to the monolithic setting.

Outline. Section 2 explains our problem setting more formally. Section 3 dives deep into each ML model and explains how we push down its inference. For the sake of readability, we provide a brief background on each model just prior to discussing our push-down rewrites for it in Section 3. Section 4 describes the real-world datasets and presents our empirical evaluation. We discuss other related work in Section 5 and conclude in Section 6.

2. SETUP AND PRELIMINARIES

We present some terminology and notation that will be needed for the rest of this paper. We then introduce our abstract *communication cost model* along with the lower and upper bound that are applicable to any ML inference query in our setting.

2.1 Setting, Terminology, and Notation

As mentioned earlier, in a heterogenous IoT deployment, there are multiple types of sensors that sense different physical features. Typically, more than one sensor is attached to a given physical device on the edge. These edge devices run an application that we call a "context engine" (CE) which computes an aggregated representation of its inputs. In the most general setting, these devices might be connected to higher-level tiers in a whole *hierarchy* of devices running context engines that are eventually connected to a program running in the cloud [7]. For simplicity, and without loss of generality, we assume the simplest possible hierarchy: an edge device (to which sensors are attached or connected cheaply) running a single context engine, and the cloud. Thus, we refer to a context engine and the device on which it runs synonymously.

Formally, we are given an IoT deployment with a set of m edge devices that together sense a total of d features. Without loss of generality, we can assume all feature values communicated are numbers, either because the features are numeric or because a numeric encoding is used for categorical features. Typically, $d \gg m$, since an edge device can sense even dozens of features. In general, the entire *feature vector* will be available together only at the cloud tier, since each edge device has access to only some of the features. The sensors can produce their feature values either periodically (say, at 10Hz) or when a real-world event occurs–this frequency of sensing is orthogonal to the focus of this paper.

Groups of ML models	Cost
(1) Full Push Down in General	
Vanilla GLMs and Linear SVMs	lm
Naive Bayes	lm
All trees (CART, RandomForest, XGBoost)	m
(2) No Push Down in General	
GLM with interactions	d
General Bayesian Networks	d
(3) Expensive Push Down in General	
Non-linear Kernel SVMs	lp
Projection Pursuit Regression (PPR)	mh
Fully connected Multi-Layer Perceptron	mh
Fully connected LSTM Network	$4mh$

Table 1: Abstract communication costs of *pushed down inference* for the various ML models studied in this paper. This is for the most general case of l-class classification; for regression, we set $l = 1$. m is the number of edge devices. d is the total number of features (typically, $d \gg m$). p is the number of "support vectors" in a kernel SVM. h is the number of "hidden units" in PPR and the first layer of the neural models. A cost of c means c numbers (say, floats) need to be communicated to execute one inference query.

Symbol	Meaning
X	Feature set/vector
Y	Prediction target
m	Number of edge devices
d	Total number of features
E_i	i^{th} edge device ($i = 1$ to m)
X_i	Subset of X sensed by E_i
$\mathcal{D}_z$	Domain of feature $z \in X$
$\mathcal{D}_Y$	Domain of target Y
l	Number of classes (= 1 for regression)
$\boldsymbol{x}$	A feature vector instance
$\boldsymbol{x}_{\|Z}$	Projection/sub-feature vector ($Z \subset X$)
$f(\boldsymbol{x})$	Given ML inference query

Table 2: Notation used in this paper.

The set of features is denoted X (also treated as a sequence/vector). The *target* is denoted Y. Its domain, denoted $\mathcal{D}_Y$, is $\mathbb{R}$ for regression and a general discrete set S of classes for classification, with $l = |S|$ denoting the number of classes. The domain of a feature $z \in X$ is denoted $\mathcal{D}_z$. We denote the m edge devices by $E_1, \ldots, E_m$. An edge device E_i only has access to the features $X_i \subsetneq X$. The set of feature sets X_i is a *partition* of the feature set X, i.e., $X = \cup_{i=1}^{m} X_i$ and $X_i \cap X_j = \phi, \forall i \neq j$. A specific vector of feature values for X is denoted $\boldsymbol{x} \in \mathbb{R}^d$. One can view an edge device as a virtual table, and the stitching together of the feature vector $\boldsymbol{x}$ as an on-the-fly "streaming" join of m tables. We denote the projection of $\boldsymbol{x}$ onto $Z \subset X$ by $\boldsymbol{x}_{|Z}$. For our purposes, ML inference can be viewed as just applying a (potentially complex) function $f(\boldsymbol{x})$, where $f : \mathbb{R}^d \rightarrow \mathcal{D}_Y$. We call the computation of $f(\boldsymbol{x})$ an *ML inference query*. Table 2 summarizes our notation.

Problem Statement. Given a heterogeneous IoT deployment with m edge devices that together sense a partition $\{X_i\}$ of a feature set X, we need to execute an ML inference query $f(\boldsymbol{x})$. The currently dominant monolithic approach communicates all of $\boldsymbol{x}$ up the hierarchy to the cloud tier. Since such queries can arrive as a high-velocity streams involving many devices, the monolithic approach could throttle the network and/or the cloud tier, while always communicating large feature vectors could reduce the battery life of edge devices. Thus, our goal is to *reduce the overall communication cost of executing $f(\boldsymbol{x})$, i.e., not have to communicate all of $\boldsymbol{x}$*. Two key additional desiderata in achieving this goal are *not sacrificing ML prediction accuracy significantly* and *not imposing high additional computational cost.*

2.2 Our Approach and Cost Model

High-level Approach. Inspired by the query optimization literature on push-down rewrites, our approach is to "rewrite" the ML inference query $f(\boldsymbol{x})$ in a way that we can compute it partially on each edge device E_i and remove the need to communicate of the entire $\boldsymbol{x}_{|X_i}$ from E_i. Since such rewrites will be heavily dependent on the computational structure of f, in this paper, we characterize for the first time a large number of popular ML models based on the structure of their inference function f. In particular, our analysis of their *data access pattern* with respect to X yields a novel *trichotomy* in their amenability to such push-down rewrites. This trichotomy is summarized by table 1.

Abstract Communication Cost Model. To crisply explain the communication benefits, or lack thereof, of push-down rewrites for $f(\boldsymbol{x})$, we need an abstract communication cost model that is generic and independent of the specifics of the hardware, the IoT application, and the ML model. To this end, we just *count the number of numbers* communicated by all edge devices. For simplicity sake, we assume all numbers communicated have the same fixed bit length (e.g., 64 bits for double precision floating points or long integers). In this model, the *upper bound* for communication cost is $|X| = d$, which is the cost of the monolithic approach (communicating the raw $\boldsymbol{x}$). The *lower bound* for communication cost is clearly m, the number of edge devices, since each E_i needs to communicate at least 1 number for each inference query in general. Thus, the maximum gain possible is d/m. This ratio could be substantial, e.g., in an activity prediction IoT application we describe in Section 4, this ratio is about $6x$. This gap also gives us flexibility in communicating more numbers than just m (but still less than d) to the end of improving ML prediction accuracy–as we show later, this flexibility is crucial for some ML models.

3. PUSH-DOWN REWRITES FOR ML INFERENCE QUERIES

We now dive into the rewrite procedures and techniques for the ML models organized by their respective groups.

3.1 Easily Pushable Models

The first group of ML models are those whose inference query $f(\boldsymbol{x})$ can be rewritten *exactly* without much extra communication or computation and without any loss in prediction accuracy. We start by characterizing how the rewrite works for such models in an abstract manner. We then explain how many popular simple ML models fit this characterization.

DEFINITION 3.1. Linearly Decomposable. *Let $l = 1$ (regression). Fix X. An ML inference query $f(\boldsymbol{x})$ is* linearly decomposable *iff there exist $d+1$ scalar functions g_0 and g_z (for each $z \in X$) s.t. $f(\boldsymbol{x}) = g_0(\sum_{z \in X} g_z(\boldsymbol{x}_{|\{z\}}))$.*

PROPOSITION 3.1. *If f is linearly decomposable, it can be fully pushed down with a communication cost of m.*

The proof is in the push-down rewrite itself, which is as follows. On edge device E_i that senses features X_i, we precompute $u_i = \sum_{z \in X_i} g_z(\boldsymbol{x}_{|\{z\}})$ and communicate just u_i. Thus, E_i only needs to store the code (and parameters) for g_z on its local storage, not the entire ML model. The cloud tier then finishes the inference query by computing $g_0(\sum_{i=1}^{m} u_i)$, which is equal to $f(\boldsymbol{x})$. Thus, the communication cost is m. In general, however, if there are $l \geq 2$ classes, we might need to communicate l numbers per edge device for the ML inference query. This brings the overall communication cost to lm. For some models, as we explain below, the cost can still be just m (instead of $2m$) for binary classification.

Note that the above rewrite does *not* affect ML prediction accuracy at all, since we obtain the same value of $f(\boldsymbol{x})$ in the cloud-tier for further processing. Moreover, it is clear these such inference queries trivially generalize to multi-layer hierarchies that have aggregator devices in between the edge and cloud. We now explain how many popular ML models such as GLMs, linear SVMs, and Naive Bayes are linearly decomposable.

3.1.1 GLMs and Linear SVMs

Background. Generalized Linear Models (GLMs), which includes linear and logistic regression, are widely used due to their simplicity, interpretability, and speed [16, 17]. We already showed in prior work that linear regression is trivially pushable. We now show that the same idea works directly for logistic regression and linear SVMs too. Essentially, GLMs and linear SVMs model the data using a hyperplane $\boldsymbol{w} \in \mathbb{R}^d$ that separates the classes (for classification) or predicts the numeric target (for regression). The co-efficients of the hyperplane are learned from training data. During inference, $\boldsymbol{w}$ becomes a fixed vector. In particular, the ML inference query f for GLMs and linear SVMs just needs to compute the inner product $\boldsymbol{w}'\boldsymbol{x}$. Subsequently, linear SVM just checks its sign, while logistic regression applies the sigmoid function to compute the binary class probabilities.

Push-Down Rewrite and Cost. Clearly, f is linearly decomposable with $g_z = \boldsymbol{w}_{|\{z\}}\boldsymbol{x}_{|\{z\}}$ and g_0 being the identity function. The communication cost is m.

Multi-Class Classification Cost. If $l > 2$, linear SVMs and logistic regression are typically trained with the so-called one-versus-all approach [15]. This approach reduces multi-class classification into l binary classifications (one for each class label). Since each of those binary classifications have linearly decomposable inference queries, the whole multi-class case is also linearly decomposable. Essentially, g_z becomes a vector-valued function (l numbers instead of 1) instead of a scalar function and the communication cost becomes lm.

3.1.2 Naive Bayes

Background. Naive Bayes is a probabilistic classifier that models the joint distribution of the target and the features by assuming conditional independence of the features given the class label [17]. In particular, its inference query $f(\boldsymbol{x})$ computes the following: $argmax_{y \in \mathcal{D}_Y} P[Y = y] \Pi_{z \in X} P[z = \boldsymbol{x}_{\{z\}} | Y = y]$. Here, the probability table $P[Y]$ and the d conditional probability tables $P[z|Y]$ (for each $z \in X$) are learned from training data. During inference, these probability tables are fixed. Thus, the inference query just looks up the appropriate probability values based on feature values in $\boldsymbol{x}$. Typically, the inference query is computed as additions of log probabilities to avoid floating point underflows. That is, f becomes $argmax_{y \in \mathcal{D}_Y} \log(P([y]) + \sum_{z \in X} \log(P[z = \boldsymbol{x}_{\{z\}} | y])$. If $z \in X$ is a numeric feature, $P[z|Y]$ does not have discrete entries but the parameters of some assumed class-conditional distributions, e.g., Gaussian distributions. The distributions' parameters (e.g., means and variances) are estimated during training.

Push-Down Rewrite and Cost. Once again, f is linearly decomposable, albeit with a vector-valued g_z that computes $\log(P[z = \boldsymbol{x}_{\{z\}} | y])$ for each $y \in \mathcal{D}_Y$. As for g_0, it adds the vector-valued $\log(P([y])$ and computes the *argmax* over $y \in \mathcal{D}_Y$. The communication cost is lm. Note that the cost is still the same even if all features are numeric or if there is a mixture of numeric and categorical features.

3.2 Complex Push Down: Tree Models

Background. A decision tree recursively partitions the d-dimensional feature space into hyperrectangles and approximates the data generating process using a constant for each hyperrectangle [17]. This partitioning is captured using a tree data structure. An internal node has a boolean expression of the form $z < \alpha$, where $\alpha \in \mathcal{D}_z$ is learned from training data. If z is categorical, most decision tree learning algorithms, e.g., CART [18], create multiple children for that internal node (one child per value of z). A leaf node carries the predicted class label (for classification) or the predicted number (for regression). Thus, a decision tree can be viewed as a losslessly compressed representation of a large number of conjunctive predicates, one per path from the root to a leaf. The ML inference query for a decision tree just traces a path from root to leaf based on the sequential application of predicates on the feature values in $\boldsymbol{x}$. Figure 2 presents an example of a decision tree for binary classification.

Complex variants of decision trees called *tree ensembles* learn multiple trees on the same data and help improve prediction accuracy compared to a single tree [17, 15]. Popular tree ensemble models include RandomForest and "boosted" trees such as AdaBoost and XGBoost [19, 20], with XGBoost being especially popular for structured data [16]. RandomForest creates a large collection of trees (say, 100) trained on random subsets of features in X. Boosted trees learn a sequence of trees by iteratively re-weighting the training examples to give more emphasis to examples incorrectly classified by previous trees [21]. These differences in the training process are not relevant for our purposes; we only need the following information–an inference query with tree ensembles simply performs inference with every tree in the collection and compute the majority vote or weighted average (or a similar aggregating procedure) to get the final prediction.

3.2.1 Novel Push Down Rewrite and Cost

Algorithm 1: Offline prepocessing phase for trees

```
Input: T: learned decision tree (or trees)
foreach E_i do
    v_i = ()                                   //List of arrays
    foreach z ∈ X_i in convention order do
        v_i.Append(Sort(T.getSplitPoints(z)))
    Store v_i on E_i
```

Algorithm 2: Online pushed down query phase for trees

```
//List of sorted interval arrays v_i stored on each E_i
foreach E_i independently in parallel on the edge do
    //x_{|X_i} sensed locally on E_i
    foreach z ∈ X_i do
        β_z ← Interval number for x_{|{z}} based on v_i(z)
    β_i ← Pack {β_z} in convention order
    Communicate β_i to cloud tier
//The following happens in the cloud tier
Initialize x' as empty feature vector
foreach β_i obtained from the edge do
    u ← Unpack interval indices into array
    foreach z ∈ X_i in convention order do
        x'_{|{z}} ← Random sample from interval u(z)
Compute ML inference query f(x')
```

At first glance, it might seem impossible to push down decision tree inference in our setting: an arbitrary inference path from root to leaf might involve multiple features in X, not all of which might be available on a single edge device. For instance, in Figure 2, panel (A), red nodes correspond to the features locally available to device 1 while black nodes correspond to those of device 2. The dotted black line indicates the inference path for the feature vector in the bottom of panel (A) and requires evaluating predicates for the features of device 1 and device 2.

We now present a novel algorithm for pushing down the inference query for a decision tree. Our key insight is the following interesting property of inference with trees: we do not really need the exact feature values to compute $f(\boldsymbol{x})$; we only need to know which *partition* of the feature space an example falls under! Using this insight, we devise a push down rewrite algorithm that almost always hits the communication cost lower bound of m. Crucially, our rewrite is exact, i.e., prediction accuracy is unchanged. We start with an intuitive description of our algorithm, with Figure 2 providing a simplified illustration. It has two phases: an *offline preprocessing* phase that is performed once after training the tree. It creates a list of *sorted arrays* for all features based on the learned model. The *online pushed down query* phase uses these arrays and simple arithmetic functions to effect the push down rewrite. Algorithms 1 and 2 present these phases of our algorithm more precisely.

Offline preprocessing phase. The offline phase proceeds as follows. We are given a learned decision tree T. For each feature $z \in X$, extract the set of all *split points* on z that arise in T. For instance, if an internal node has the split condition $z < \alpha$, then α is a split point for z. Sort all split points, say p of them for z in T, and store them in an array data structure $\boldsymbol{v}(z) = [\alpha_1, \alpha_2, ..., \alpha_p]$. This array defines

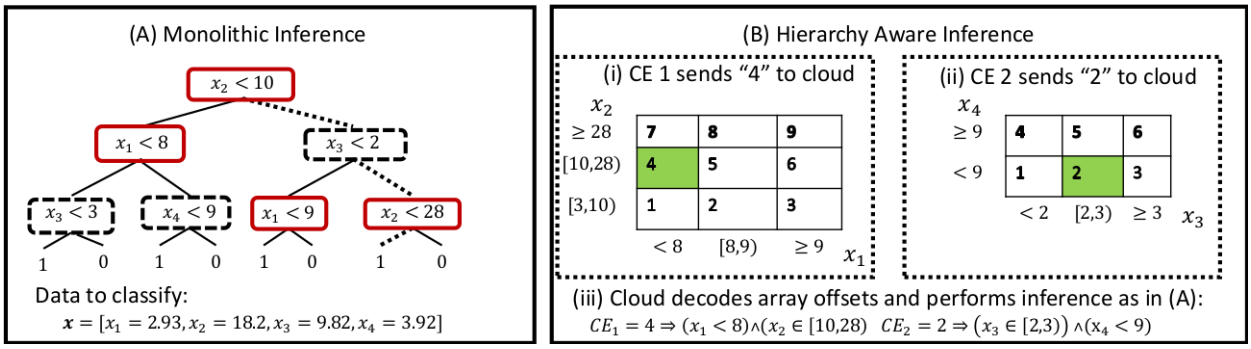

Figure 2: (A) Illustration of standard monolithic inference for a simplified binary classification decision tree. Red solid nodes are features on edge device 1; black dashed nodes, on edge device 2. The dashed edges show the inference path for the given $\boldsymbol{x}$. (B) Illustration of our pushed down/hierarchy-aware inference. (B.i, B.ii) Edge devices 1 and 2 compute offsets in to their local arrays based for the partition of the local feature spaces this example falls into. (B.iii) The cloud tier unpacks the offsets and performs inference.

intervals that *partition* $\mathcal{D}_z$. We assign *sequential integers* (0 to $p-1$) to these intervals. If z is categorical, we can simply map its domain values in lexicographic order to sequential integers (0 to $|\mathcal{D}_z| - 1$). Each edge device E_i that senses X_i locally stores the list of sorted arrays $\boldsymbol{v_i}$ (for all $z \in X_i$) and a linear ordering convention for X_i (say, lexicographic order of feature names). The same information is kept in the cloud tier as well, along with T.

Online pushed down query phase. Recall that the full feature vector is $\boldsymbol{x}$ but on E_i, we only have the values of $\boldsymbol{x}_{X_i}$. For each $z \in X_i$, we determine which interval partition number/index $\boldsymbol{x}_{|\{z\}}$ falls in to. This calculation is easily done with a binary search for each z on its sorted interval array. Alternatively, we can also use a local hash table instead of sorted arrays to determine the interval partition indices. We then pack all these indices into a *single number* using simple integer multiply-add arithmetic using the lengths of the sorted arrays and the linear ordering of X_i. This is possible because the sorted arrays are fixed beforehand by the offline preprocessing phase. E_i then communicates only this single packed number to the cloud tier. Thus, the overall communication cost is just m, regardless of l.

The cloud-based application unpacks the number received from each E_i using simple modulo arithmetic to obtain the exact interval partitions for each $z \in X_i$. Using this interval partition information, the application then samples a feature value for each z at random from its respective interval partition. Thus, it reconstructs a modified feature vector $\boldsymbol{x'}$. It then applies T on $\boldsymbol{x'}$, wherein we are guaranteed that $f(\boldsymbol{x}) = f(\boldsymbol{x'})$ because the internal nodes of T only perform checks based on the same interval partitions. If the model is a tree ensemble, the application uses $\boldsymbol{x'}$ for each tree in the ensemble. We provide a formal proof of correctness for our algorithm in the technical report due to space constraints [**?**]Anthony: I will add this once I have set up the project webpage (tonight).

Tree ensembles. To handle ensembles, we use the following approach. Create the sorted interval arrays for each trees in the ensemble for each $z \in X$ as before. But then, during the offline preprocessing phase, *merge* the respective sorted arrays of z across all trees in the ensemble. Thus, we will have just one sorted array for z overall for the ensemble. The rest of the algorithm, as well as the online pushed down query phase, all remain the same, while the communication cost remains m.

Bit length. A subtle but minor consideration is the *bit length* of the number sent by each E_i. This bit length is entirely a function of the number of split points in T for each z and the size of the largest $|X_i|$. For example, suppose the largest $|X_i|$ is 15. If the number of split points for each feature is 9, we need to represent up to $9^{15} \approx 2^{48}$ index combinations, which can be done with 64 bits (e.g. a single number). But if the number of split points is 30, we would need $30^{15} \approx 2^{74}$ index combinations, which requires 2 numbers (128 bits in total), bringing the cost of this push down to $2m$. We show in section 4.3 that one number typically suffices in practice even for boosted trees.

3.3 No Push Down in General

The second group in our trichotomy are models whose inference queries are not amenable to an exact push down in general. That is, one might have to approximate these models in an IoT network hierarchy-aware manner to get a lower communication cost than d. We now explain why this is the case for each model and mention some special cases where some push down is still possible.

3.3.1 *Linear Models with Feature Interactions*

Background. Since GLMs and linear SVMs are linear models, data scientists often include *interaction features* to improve their predictive power [15, 17]. The most common feature interactions are *pairwise products* of each pair of features in X, including squares of features. This leads to a quadratic blowup in the total number of features, i.e., from d to $d + d +^d C_2 = d(d+3)/2$. In general, one could also add ternary products or higher-order product terms.

Rationale for No Push Down. All pairwise feature products that straddle different edge devices are not computable in general locally. But even if we have just 2 edge devices, every feature in each device participates in at least one feature interactions with a feature from the other device. Thus, all devices have to communicate all their features. If L1 regularization is applied during training, all interaction features which straddle context engines *may* become zero. In this special case, the remaining feature interactions are device local and push-down is possible.

3.3.2 *Bayesian Networks*

Background. A Bayesian Network is a probabilistic classifier that encodes conditional independences among X and Y using a directed acyclic graph (DAG) [15]. The DAG is either specified manually using domain knowledge or learned from training data. The inference query works as follows. $f(\boldsymbol{x}) = argmax_{y \in \mathcal{D}_Y} P[Y = y]P[X = \boldsymbol{x}|Y = y]$, where the joint probability $P[X|Y]$ is factorized into a product of conditional probabilities with subsets of X defined by the DAG (Naive Bayes is a special case of a DAG that allows for full factorization to individual features). $P[Y]$ and various conditional probability tables over subsets of X are learned from training data. During inference, these tables are fixed and f looks up the appropriate probability values based on $\boldsymbol{x}$. The rest of the inference mechanics (using log probabilities, summation, etc.) are similar to Naive Bayes.

Rationale for No Push Down. Unlike Naive Bayes, a general Bayesian Network's DAG might have dependencies between features in X that are on different E_i. In this sense, it is similar to the linear model with feature interactions and thus, f cannot be pushed down fully in general. But if all feature dependencies are local to various E_i, we can push down f in a manner analogous to Naive Bayes; the cost will be lm. Depending on the DAG, partial push down (i.e., communicate some but not all features in X) is possible; the cost will be between lm and d but it is tedious to express in closed form. In general however, the DAG could have too many dependencies across features in different X_i, which means it might be cheaper to just communicate all of $\boldsymbol{x}$.

3.4 Expensive Push Down in General

The final group in our trichotomy are models whose inference queries are amenable to an exact push down in general, but the communication cost of the rewritten query will almost always be prohibitively high. Thus, one might again have to approximate these models in an IoT network hierarchy-aware manner to reduce the communication cost from d. In the next subsection, we present novel approximations for the neural networks in particular, but we continue to focus on the feasibility and efficiency of exact push down rewrites in this subsection.

3.4.1 Non-linear SVMs

Background. Support vector machines (SVMs) with non-linear kernels implicitly transform examples to a high-dimensional space using special functions of inner products between two examples called "kernels" and learn a separating hyperplance in that space [15, 17]. Training a non-linear kernel SVM (or simply, kernel SVM) results a weight being assigned to each training example. All examples with non-zero weights are called "support vectors." A learned kernel SVM retains all support vectors and their weights and uses them for inference. More precisely, given p support vectors $(\boldsymbol{x_j}, y_j)$ and their respective weights α_j ($j = 1$ to p), the inference query for a kernel SVM for binary classification is as follows: $f(\boldsymbol{x}) = sign(\sum_{j=1}^{p} \alpha_j y_j K(\boldsymbol{x}, \boldsymbol{x}_j)) - b$, where K is the kernel and b is a learned parameter called the margin. Commonly used K are the polynomial kernel $K(\boldsymbol{a}, \boldsymbol{b}) = (a'b + 1)^\gamma$ and Gaussian kernel $K(\boldsymbol{a}, \boldsymbol{b}) = \exp(-\gamma \|a - b\|_2^2)$, where γ is a hyper-parameter.

Push-Down Rewrite and Cost. The inference query computes inner products of all p support vectors with $\boldsymbol{x}$ and applies a non-linear function K on each number. While the inner products themselves are amenable to a full push down a la linear models, it is impossible in general to push down the aggregation over the p support vectors. Thus, the communication cost for the pushed down rewrite is p. In general, p can be very large, say, some non-trivial percentage of the training set size (e.g., thousands). The multi-class case for SVMs is even worse, since it is typically handled with the one-versus-all approach [15]; this raises the cost to lp. Overall, it will almost always be cheaper to just communicate all d features for kernel SVMs.

3.4.2 Projection Pursuit Regression (PPR)

Background. PPR is a semi-parametric statistical model for regression that is a pre-cursor to general multi-layer perceptrons but typically much faster to train [17, 22]. It computes weighted inner products over X and learns both these weights and h non-parametric non-linear functions g_j ($j = 1$ to h), typically splines, from training data. The hyper-parameter h is typically called the *number of hidden units*. The inference query $f(\boldsymbol{x})$ is as follows: $\sum_{j=1}^{h} g_h(w_h^T \boldsymbol{x})$. The "hidden-units" ($g_h$) in PPR are learned *sequentially* with each subsequent hidden unit fitting residual variance not explained by the previous units. As with multi-layer perceptrons, which we will discuss shortly, PPR has a "universal approximation" property in that it can approximate any prediction function well, which typically translates to high prediction accuracy [23].

Push-Down Rewrite and Cost. The inference query for PPR can be pushed down fully, since it only computers inner products over $\boldsymbol{x}$. PPR is typically applied to regression problems but can be applied to classification as well. For a regression problem, the communication cost of the pushed down execution is hm. In practice, h is typically between two and a dozen [24]. Thus, PPR may or may not be expensive depending on how hm compares with d, but it will almost surely be less expensive to push down compared to kernel SVMs. Later in Section 4, we show how even with a single-digit value of h, PPR is able to provide near-best prediction accuracy. The primary virtue of PPR compared to the MLP is this parsimony in terms of the number of hidden units.

3.4.3 Neural Networks

Background on MLPs. The multi-layer perceptron (MLP) is a neural network model usable for both regression and (multi-class) classification. It is a powerful non-convex generalization of logistic regression [17, 15] that transforms $\boldsymbol{x}$ by a dataflow of functional transformations organized as *layers* with vectors flowing between layers. Each layer either multiples the incoming vector by a weight matrix (and adds an offset vector) or applies a non-linear element-wise scalar function called an "activation function" such as sigmoid or ReLU [25]. MLPs can be visualized as a graph; the internal nodes are called *hidden units*. Figure 3 (A) provides an illustration. Each hidden unit computes an inner product of the incoming vector with one row of the weight matrix. MLPs are trained using an algorithm called back-propagation with stochastic gradient descent. For our purposes, the training details of MLPs are not relevant. The query $f(\boldsymbol{x})$ also operates in a layer-wise manner, and we only need to consider the access pattern of the computations over X at the first layer. Formally, the computation at the first layer is as follows: $\sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = g(\boldsymbol{x})$ (say), where $\boldsymbol{W}$ is an $h \times d$ weight

matrix, $\boldsymbol{b}$ is an $h \times 1$ offset vector, and g is a vector-valued function. Note that $\boldsymbol{W}$ and $\boldsymbol{b}$ are fixed after training. The rest of the computations in f use the above result of g for further processing.

Background on LSTMs. A recurrent neural network (RNN) is a twist on the standard MLP that models sequential dependencies in the data, including temporal dependencies. RNNs introduce "loops" which allow prior examples to influence the current prediction. The most common form of RNN is the so-called long short-term memory (LSTM) network. The interested reader is referred to [26] for a thorough overview. Since most IoT applications produce time series data with temporal dependencies, we include LSTMs in our analysis. For our purposes, it suffices to know that the inference query for an LSTM computes 4 matrix-vector products in the first layer, each of which has the same access pattern as the single g above for an MLP.

Push-Down Rewrite and Cost. As with PPR, full push down of the inference query is indeed possible for fully-connected MLPs and LSTMs too, but the rewritten execution might be prohibitively costly. To see why, we need to consider only the first layer's computation over X, viz., $\boldsymbol{W}\boldsymbol{x}$. Similar to a GLM, this product is easy to push to the edge devices by sub-selecting appropriate columns of $\boldsymbol{W}$ corresponding to each E_i. Thus, each E_i only needs to send h (resp. $4h$) numbers for, bringing the total cost to mh (resp. $4mh$) for the pushed down MLP (resp. LSTM). This rewrite also trivially generalizes to a multi-tier hierarchy, since each tier only performs partial summations needed to compute the full matrix-vector product. Of course, depending on h, this cost of mh (or $4mh$) might still be less than d, but in practice, h is usually at least a few dozen and is commonly in the hundreds. This is because having too few hidden units might hurt the prediction accuracy of neural networks [27]. We observe that the MLP/LSTM need not really be fully connected at the first hidden layer; we could impose "partial connectivity" to reduce the cost. This observation motivates our next subsection on approximating these neural network models in an IoT network hierarchy-aware manner.

3.5 Approximate Hierarchy-Aware Neural Networks

We now dive into novel approximate variants of MLPs and LSTMs that enable us to support a *graceful trade off* between the communication cost affordable and prediction accuracy. Our approach is to "wire in" a partial connectivity structure based on the given IoT application's network hierarchy-imposed partitioning of X. This is related to prior work in the ML literature on "partitioned" neural networks [28], but our work differs in that we do not aim to improve training efficiency but rather inference communication cost in the IoT setting. We also analyze our approximate MLP architecture formally using standard learning theoretic concepts to explain its accuracy behavior.

3.5.1 Novel Hierarchy-Aware MLP

Our modified MLP architecture is illustrated by Figure 3(C). On each E_i, we have a *local MLP* that only looks at X_i. These local MLPs act as *lossy compression* functions for X_i. The length of their compressed representation, denoted as κ, is a tunable knob to control the accuracy-communication cost trade-off. While it is simple to have different knobs for each E_i, we assume the same κ for all E_i for the sake of simplicity. At the cloud tier, we have a fully connected *global MLP* that uses the *concatenation* of all the compressed representations as the input. We train this whole modified MLP end to end using standard backpropagation. Thus, by respecting the IoT network's hierarchy in the MLP architecture itself, we can potentially improve prediction accuracy, while still enabling users to control the communication cost. If $\kappa = 1$, the total cost hits the lower bound of m. If $\kappa = d/m$, the total cost hits the upper bound of d. In general, κ can be set to a value in between these extremes if it is tuned as a hyper-parameter during training with standard cross-validation [17].

Extension to LSTMs. Our approach to making the MLP hierarchy-aware is applicable to LSTMs too, but we get at least two different architectures. In the first, illustrated by Figure 3(D), each edge device uses a local MLP to compress X_i as before, while the cloud tier has a *global LSTM* with the recurrent connections. In the second, illustrated by Figure 3(E), we use a *local LSTM* on the edge devices to exploit potential autocorrelation of the features in X_i, i.e., the compressed representation sent by E_i also depends on previous time steps implicitly. The second architecture is more complex than the first and thus, it is more susceptible to overfitting. However, with enough training data, it is possible for the second architecture to have better prediction accuracy owing to the "bias-variance trade-off" inherent in ML [17, 29].

3.5.2 Formal Analysis of Hierarchy-Aware MLPs

We now formally analyze the accuracy behavior of our hierarchy-aware MLPs relative to its monolithic counterpart using standard concepts from ML theory [29]. We start with brief intuitive explanations of these concepts. The *hypothesis space* of an ML model is the set of all prediction functions it can represent. The test error of an ML model has three components: bias, variance, and noise. *Bias* captures the error caused by the hypothesis space not being complex enough; this could cause *underfitting*. *Variance* captures the error introduced by the hypothesis space being overly complex; this could cause *overfitting* to the training set. A central concept in ML theory is the *bias–variance trade-off*: one typically has to trade off a decrease in bias for an increase in variance or vice versa when training ML models. Models with larger hypothesis spaces tend to have lower bias and higher variance. *Regularization* and/or *feature selection* help achieve better dataset-specific sweet spots for test error. The *generalization error* of an ML model is the gap between the test and train error. For binary classifiers, the *VC dimension* of a hypothesis space is a way to quantify its "capacity" for accurately classifying a set of labeled examples. Models with higher VC dimensions tend to have higher variance and are prone to overfitting.

Using the above concepts, we explain how the hiearchy-aware MLP has a smaller hypothesis space but possibly, better generalization behavior. We state our observations in the following two propositions. Due to space constraints, we present more technical details of our analysis, including the proofs, in the technical report [**?**]. We empirically validate these observations later in Section 4.3.

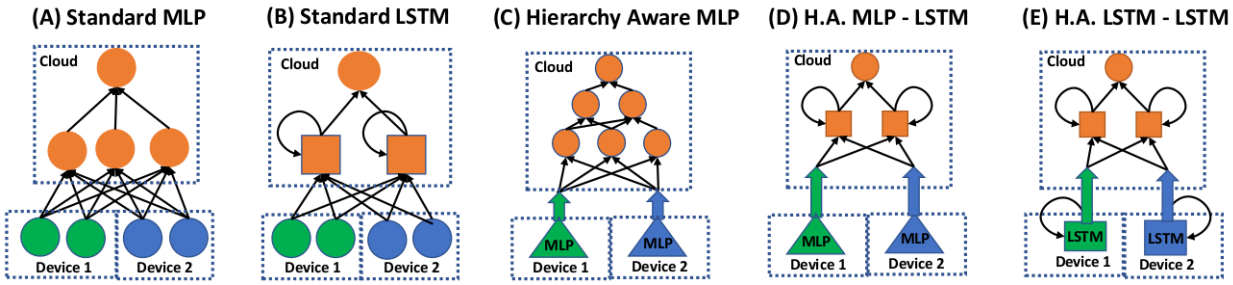

Figure 3: Simplified illustrations of various neural network architectures considered in this paper. "H.A." stands for "Hierarchy Aware." Orange nodes reside in the cloud; green nodes, on edge device 1; blue nodes, on edge device 2. Each edge device has 2 features. (A) Standard fully connected MLP. In the monolithic approach, each device sends 1 number per orange node in the first hidden layer. (B) Standard fully connected LSTM; the loops represent recurrent connections. In the monolithic approach, each device sends 4 numbers to each orange node in the first hidden layer. (C) Our new H.A. MLP. (D) Our new H.A. LSTM with local MLPs. (E) Our new H.A. LSTM with local LSTMs. In all three of the H.A. neural models, each edge device sends only 1 number to the cloud.

PROPOSITION 3.2. *Fix X, Y, and a partitioning of X with $m < d$. Let $\mathcal{P}$ be a hierarchy-aware MLP as described earlier and let $\mathcal{M}$ be a fully-connected MLP with the same number of hidden layers, hidden units in each layer, and activation functions. Let $\mathcal{H}_{\mathcal{P}}$ and $\mathcal{H}_{\mathcal{M}}$ be their respective hypothesis spaces. Then, we have: $\mathcal{H}_{\mathcal{P}} \subset \mathcal{H}_{\mathcal{P}}$.*

Intuitively, the above result holds because $\mathcal{P}$ can be viewed as a fully-connected MLP in which weights on the connections straddling edge devices are forced to be zero. Next, we have a consequence of the above proposition for binary classification. This result hinges on the relationship between the VC dimensions of $\mathcal{H}_{\mathcal{P}}$ and $\mathcal{H}_{\mathcal{P}}$.

PROPOSITION 3.3. *Let $\Delta_{\mathcal{P}}$ and $\Delta_{\mathcal{M}}$ be the upper bound on the gap between the test and train errors for $\mathcal{P}$ and $\mathcal{M}$ respectively presented in Theorem 3.2 in [13]. Then, we have: $\Delta_{\mathcal{P}} < \Delta_{\mathcal{M}}$.*

In practice, the above results suggest that it is possible for $\mathcal{P}$ to have higher training errors than $\mathcal{M}$ but also lower test errors. Basically, our modification to the MLP acts as a form of regularization that rebalances the bias-variance trade-off in a way that respects the IoT hierarchy. Unfortunately, ML theory does not yet offer the machinery to analyze regularization in neural networks in a fine-grained manner [**?**]; thus, we leave a deeper analysis to future work.

4. EXPERIMENTAL EVALUATION

We now evaluate our pushed down ML inference techniques on a suite of three real-world datasets from heterogeneous IoT use cases. Our goal is three-fold. (1) Validate that our pushed down approach reduces communication costs compared to the monolithic approach, while either not affecting accuracy or supporting a graceful trade-off with accuracy. (2) Explain the accuracy behavior of our hierarchy-aware neural networks. (3) Validate with a real prototype implementation on a Raspberry PI device that our techniques yield energy and latency savings. Note that our goal is *not* to compare which ML model is the best for each task, but rather compare each model's monolithic version with our pushed down (hierarchy-aware) version for that model.

4.1 Tasks and Datasets

Task/Dataset	Type	n	d	m
Urban Energy Demand	Reg.	125,549	387	52
Human Activity	Class. (5)	804,228	52	4
Server Performance	Reg.	24729	60	5

Table 3: Task/dataset statistics. "Reg." is regression and "Class." is classification - the number in parenthesis denotes the number of classes. n is the number of unique data points, d is the number of features ("time-lagged" features derived on edge devices are not counted in d), and m is the number of edge devices (context engines).

We start with brief descriptions of each IoT use case's motivation, dataset description, and the methodology for ML-based predictive modeling. Table 3 provides the key dataset statistics. Due to space constraints, we present more deatils on the use cases and the specific hyper-parameter "grids" used for ML training in our technical report [**?**].

4.1.1 *Urban Energy Demand Prediction*

Motivation. To forecast consumer electricity demand, prior works in the IoT community have proposed instrumenting individual appliances in houses with power-use sensors to gather real-time data on energy use [30, 31, 1, 4]. In our context engine-based approach, each house can be outfitted with a context engine that aggregates the raw sensor data for that house alone. In a dense urban area in which a neighborhood may have hundreds (if not thousands) of housing units, one might need additional layers in the hierarchy of aggregation to avoid throttling the network. Such a multi-layer hierarchy was shown in Figure 1(A).

Dataset. This dataset is from a commercial data broker service [32]. After some pre-processing to remove houses with too many missing values, we have 52 houses observed over the period 2014-01-1 to 2016-12-31. Each house has a set of appliances instrumented with sensors that record the average energy used by that appliance in increments of 15min. The number of unique appliance types across all houses is 58, with the total number of features (one per appliance) being 387. We additionally have average energy use across all appliances from all houses. Unfortunately, not all houses are spatially co-located, which means transmission and distribution losses of energy might be non-trivial.

Thus, we simulate neighborhood energy use as a sum over all houses plus a quadratic term capturing these energy losses as per [33]; this is our regression target.

Methodology. We predict Y for evey one hour given current and prior energy use data from each appliance. For all ML models except LSTM, we augment the feature vector X with two standard time series techniques for feature extraction [34]. First, we use a finite "lag" model that concatenates features from up to three previous time stamps. Second, we add indicator variables for hour-of-day (1 to 24) and week-of-year (1 to 52) to capture longer term cyclical trends. These derived features can be easily constructed either on the edge device or in the cloud tier; thus, for a fair comparison, we omit them from the communication cost of the monolithic approach.

4.1.2 Human Activity Prediction

Motivation. The public health community is increasingly interested in using IoT for real-time monitoring of individuals at risk for early-onset dementia [35, 36]. The early warning signs of dementia may manifest in subtle behavioral changes that are difficult for caregivers to detect from infrequent in-person visits. In-home sensor networks can help build a profile of typical behavioral patterns that can be analyzed automatically, with unexpected changes flagged for review by caregivers. A major component of such an IoT application is "activity detection," which identifies a human subject's current activity (e.g., cooking, sitting, etc.) based on "inertial measurement units" (IMU) from accelerometers.

Dataset. We use the publicly available PAMAP2 dataset, which has activity measurements from 9 subjects [37]. We drop two subjects with too many missing values. Each subject was outfitted with three IMU sensors mounted on the wrist, chest, and ankle and one heart-rate monitor. Each IMU produces 17 numeric features, yielding a total 52 features. This is a multi-class classification task with 12 activity classes (running, sitting, walking, etc.). For our purposes, we group these activities into 5 broad classes.

Methodology. Such human subject-oriented IoT applications use the so-called "leave-one-subject-out" nested cross-validation (LOSO-NCV) method to evaluate ML accuracy. We pick 1 subject as the test set; the rest 8 are used for training and validation (for hyper-parameter tuning). This process is repeated for each subject and the average accuracy is reported. This dataset has high temporal correlations, and as suggested in [37], we capture this variation by augmenting X with aggregated temporal features: slide a rolling window of length 5.12s with a stride of 1s to compute the mean, median, and variance of each individual feature. Once again, we omit these derived features from the communication cost of the monolithic approach for a fair comparison.

4.1.3 Server Performance Prediction

Motivation. This scenario demonstrates that our techniques are also applicable to use cases on wired networks, not just wireless networks. This task predicts the performance of a program on a server in a cluster. Such performance models are useful for schedulers to dynamically allocate resources and/or migrate tasks to improve cluster utilization. These models rely on data from "performance measurement units" (PMU) on each server that send many features such as number of executed instructions, number of cache misses, etc. [38]. For large clusters, communicating such features continually to a central server can throttle the network, hurting application performance. In our hierarchical approach, each server can be viewed as an "edge device," with racks, aisles, or even whole data centers being higher levels in the hierarchy. Pushing down ML inference and reducing communication cost in this hierarchy reduces network congestion.

Dataset. We instrumented 6 servers to record 12 PMU events as reported in [38]. The PMU events were collected for a set of 26 Apache Spark benchmark applications running tasks such as PCA, clustering, and MLPs [39, 40]. PMU data was sampled at a rate of 5Hz. AK: 200ms (5Hz) or 200Hz?

Methodology. The performance metric to predict is the instruction-per-second (IPS) of the benchmark application on a given server using the other 5 servers. This can help schedulers decide whether to migrate a part of the application to the given server, when it is already running on the others. Since such decisions need to be done in real-time, we follow the same convention as prior work [38, 41] in using the raw PMU event features.

4.2 ML Training Details

We used the standard ML practice of "grid search" for tuning the hyper-parameters of each ML model along with nested cross-validation [42]. Due to space constraints, we only mention what hyper-parameters were tuned; the exact grid values are provided in the technical report [**?**]. For all GLMs, we use elastic net regularization and tune both L1 and L2 regularizers. For XGBoost, we use 100 trees and tune its maximum depth hyper-parameter. We implemented PPR from scratch in Python NumPy; we use a cubic spline with 2 knots as its "activation function." All neural models (MLPs and LSTMs) are trained using TensorFlow [43] with ReLU activations, Adam optimizer, and a fixed reasonably sized architecture (number of layers and neurons/memory cells). We tuned the step-size, L2 regularizer, and dropout probability. Based on Section 3.3, we omit kernel SVMs and Bayesian Networks, since they have high cost anyway. We also skip linear SVMs, Naive Bayes, and other tree models for brevity sake, since they offer no new additional insights over the set of models shown.

4.3 Results and Discussion

Overall Accuracy and Communication Costs. Figure 4 presents the overall accuracy (reported as prediction errors; lower is better) and communication costs (again, lower is better) for all datasets. Recall that the cost is the abstract number of numbers communicated by all edge devices for a single ML inference query (see Table 1). Overall, we see that for each dataset, at least one of our pushed down/hierarchy-aware ML models attains comparable or better accuracy relative to the best monolithic ML model, while still yielding much lower communication costs. Thus, these results validate our core claim in this paper: *our push down techniques for ML inference queries can reduce communication costs substantially, while still yielding high accuracy.*

Our hierarchy-aware MLP achieves the best or near-best combination of accuracy and cost. But unsurprisingly, no single ML model dominates on all tasks, but this point is

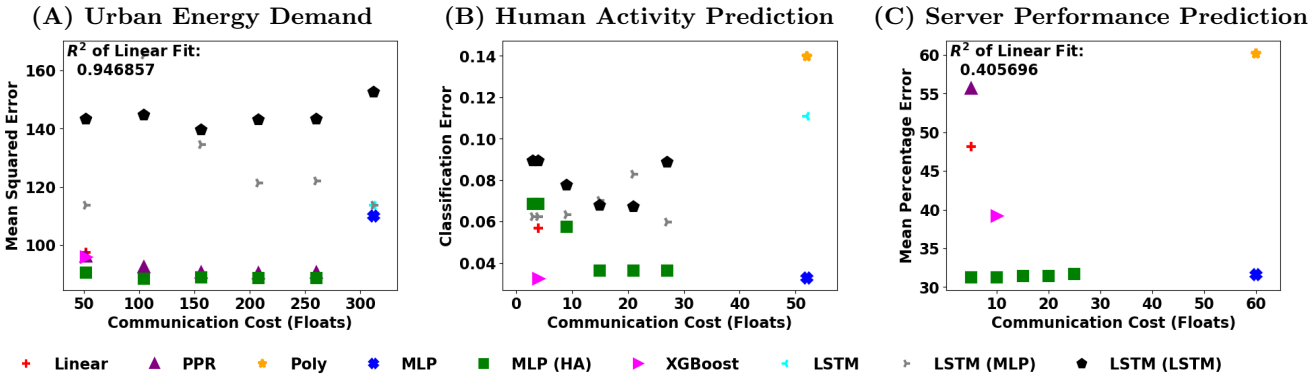

Figure 4: Overall trade-offs of prediction errors and communication cost for various ML models on the datasets listed in Table 3. Lower is better for both axes. Recall that the upper bound for cost is d (number of features), which is hit by the monolithic approach; lower bound is m (number of edge devices). These bounds are also listed in Table 3. Linear is a GLM without feature interactions; Poly is a GLM with order 2 interactions. MLP and LSTM are the fully connected (monolithic) MLP and LSTM respectively. MLP (HA) is our hierarchy-aware MLP. LSTM (MLP) and LSTM (LSTM) are our hierarchy-aware LSTMs with MLPs and LSTMs respectively on the edge devices. XGBoost has a cost of m on all 3 datasets thanks to our push down rewrite (Section 3.2). For MLP (HA), LSTM (MLP), LSTM (LSTM), and PPR we also vary the compression factor κ (Section 3.5.1) to show its accuracy-cost trade-off.

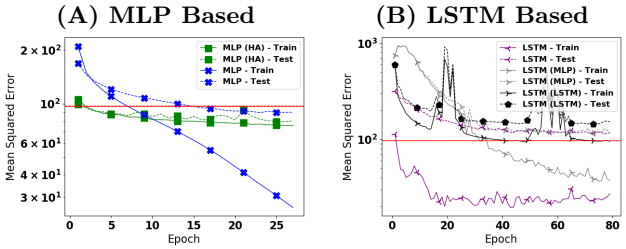

Figure 5: Train and test errors over epochs for the (A) MLPs and (B) LSTMs on the Urban Energy Demand dataset. The red horizontal line is the test error of the linear model.

well-known [**?**] and also orthogonal to this paper's focus. XGBoost has lower errors on the Human Activity task, but is worse than the hierarchy-aware MLP on the other tasks even though it hits the cost lower bound thanks to our push down rewrite. The linear models, which are linearly decomposable, have significantly higher errors even though they too hit the cost lower bound. The polynomial model has higher errors than the linear models, which suggests more overfitting. PPR performs well on the Urban Energy Demand task, but overfits severely on the Server Performance task. All the LSTM models, be it monolithic or hierarchy-aware, report high errors across the board, also due to overfitting. We verify this issue with a drill-down experiment shortly. As such, LSTMs are typically used for complex sequential prediction tasks such as natural language processing and seem to be an overkill for the time series tasks of Figures 4(A) and (B).

Drill-Down Results On both the Urban Energy Demand and the Server Performance tasks, our hierarchy-aware MLP yields both the lowest error and lowest cost. We now drill into this model's accuracy behavior on the Urban Energy Demand dataset to understand its behavior. Figure **??**(A) plots the train and validation (proxy for test) errors at each epoch during training for the chosen hyperparameter combination. Consistent with our formal analysis (Section 3.5.2), the hierarchy-aware MLP overfits substantially less than the monolithic one (test-train error gap is smaller). Thus, in spite of converging to a worse training error, the hierarchy-aware MLP yields a better test error due to its better generalization behavior. We leave further theoretical analysis of this interesting behavior to future work.

On the Human Activity task, we observed larger variations in the test errors across subjects than across models. Due to space constraints, the exact errors are tabulated in the technical report [**?**]. Thus, although the hierarchy-aware MLP and XGBoost fare well on average, we cannot draw statistically significant conclusions about the differences between the models. Finally, we also computed the bit lengths needed for each edge device for our pushed down rewrite of the XGBoost model on all datasets. The average bit lengths needed were 30 bits, 55 bits, and 73 bits for the three datasets in that order. Thus, our novel push down rewrites (Section 3.2) ensure low communication cost for a complex model such as XGBoost without affecting its accuracy.

4.4 Implementation and Measurements on RPI

We implementated a prototype of our pushed down inference for the 2 best performing models–XGBoost and hierarchy-aware MLP (with a compression factor of $\kappa = 1$)–on a Raspberry Pi 3 (RPi3) device, which runs our context engine software [44]. The RPi3 communicates with a server using MQTT protocol, which is widely used for IoT networks [45]. We measure the latency and energy consumption of the RPi3 in both the monolithic case and our pushed down case. We plot the percentage reductions yielded by our approach. We vary the number of ML inference requests per second on the x axis. We report the results for 3 different bandwidths: 5000, 700, and 200 kbps, corresponding to WiFi, Bluetooth, and constrained Bluetooth respectively. These bandwidths reflect typical levels of network congestion. Under these network conditions, the RPi3 is able to send up to 400 samples per second for the monolithic case. Figure **??** presents the results for the Human Activity task.

Overall, we see that our pushed down ML inference often reduces both system energy and latency, especially when the

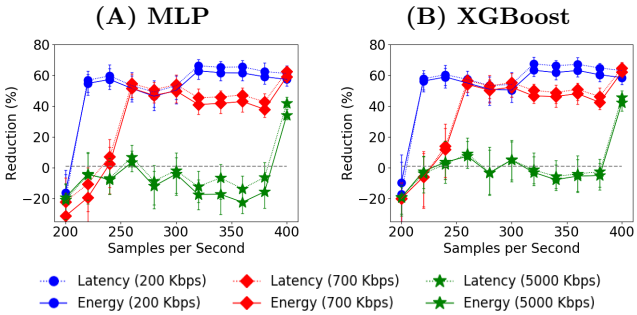

Figure 6: Percentage reduction of latency and energy consumption from monolithic to our pushed down ML inference on the Human Activity dataset. "Samples per second" is the number of ML inference requests received per second.

network bandwidth is low. The reductions in latency and energy consumption are up to 67% and 63%, respectively. Considering that a swarm of IoT devices often need to send data fast in typically harsh network conditions in the real world [46], these results suggest our approach could indeed be beneficial to IoT applications. Still, the actual savings are lower than the theoretical cost reduction of d/m =13x, as per Table 1. This suggests that hardware and networking overheads not captured in our abstract communication cost model matter for real-world performance. We leave an exploration of such hardware-level issues to future work. Finally, we note that the hierarchy-aware MLP and XGBoost exhibit similar trends for both energy and latency reduction. Thus, the extra computation cost incurred by the device-local MLP in our hierarchy-aware MLP is not significant.

5. RELATED WORK

Cloud-Edge Computing. The tradeoff between cloud and edge computing is well studied in general [9, 47, 48, 49]. However, this work has not generally addressed ML algorithms with complex, nonlinear, data access patterns. [50], and more recently [51], proposed techniques to determine the optimal layer in a deep neural network at which to partition computation between the edge and cloud such that energy use and latency were minimized but focused on fully connected networks. The cloud-edge computing tradeoff has gained attention in industry as well with Google's recent introduction of "federated learning" which learns a centralized model over a massive collection of distributed devices (e.g. smartphones) [52, 53]. Federated learning differs from our work in that it assumes all features are available locally on a device and focuses on *training* a centralized model.

Sensor Data Fusion. The field of "Sensor Data Fusion" (SDF) studies methods for integrating data streams from heterogeneous sensors. [54, 54] and [55] discuss ML based techniques. Most closely related to our work is [5] who proposed a hierarchy aware convolutional neural network architecture for image classification. However, their approach differs from ours in that they assume *homogneous* sensors in the form of video cameras which implies each device can perform a *complete* inference computation. Data is passed up the hierarchy only to obtain a more accurate prediction. By contrast, we assume that device local inference is not possible in general and address a larger spectrum of model types.

Partitioned Neural Networks. Partially connected networks have been widely explored in the literature and *a priori* constrain some number of weights in the network to be zero [56]. However, existing literature [57, 28] has focused primarily on settings in which connectivity could be chosen ad-hoc, or to maximize some objective which differs from our setting in which the constraints are physically imposed.

Compressive Sensing. Compressive sensing (CS) is a technique which utilizes a statistically motivated sampling procedure to represent a d dimensional vector using $p \ll d$ coordinates without losing too much information [58, 29]. CS and related sampling techniques have been widely applied to data reduction in sensor networks: [59, 60, 6]. From our perspective, the output of CS is just another feature vector and so fits naturally in the framework we describe.

Query Rewriting for ML. The RDBMS community has long studied techniques to optimize ML training and inference. [61, 62, 63] presented systems which perform logical rewrites to optimize linear-algebra based ML for execution in distributed environments. [64, 65, 66, 67] studied efficient rewrites for nearest neighbor classification, recommender systems and matrix factorization. [13, 68] and [14] presented rewrites for pushing learning of GLMs and similar linear-algebra based procedures through joins. However, the aforementioned work has not considered the hierarchical setting on which we focus. [10] studied query rewrites for ML in heterogeneous hierarchical sensor networks, but only considered simple linearly decomposable queries.

6. CONCLUSIONS AND FUTURE WORK

As the use of ML in heterogeneous Internet of Things applications grows, efficient management of IoT data for ML inference is a pressing challenge for the data management community. In this work, we study a new approach to IoT stacks that moves computations closer to the edge via a hierarchy of devices and reduces the cost of data movement. To enable such an approach, we tackle the problem of rewriting ML inference computations and pushing them down to edge devices that collect different subsets of features. We established a trichotomy in the amenability of ML models to such push-down rewrites in terms of the communication cost, accuracy, and computation cost. By introducing novel push-down rewrites, both exact and approximate, for a variety of popular ML models, we demonstrate substantial savings in communication cost, energy use, and latency, while still achieving high accuracy.

We see two key avenues for future work. From the theoretical standpoint, our analysis of the hierarchy-aware MLPs showed a connection to regularization. It will be interesting to characterize that connection more rigorously to help data scientists reduce model building effort, say, exploiting more domain knowledge about their data. From the systems standpoint, our encouraging results with the RPI prototype suggest it will be interesting to deploy our ideas on a real IoT hierarchy and study new network-oriented optimizations.

7. REFERENCES

[1] Aftab Khan, James Nicholson, Sebastian Mellor, Daniel Jackson, Karim Ladha, Cassim Ladha, Jon Hand, Joseph Clarke, Patrick Olivier, and Thomas Plötz. Occupancy monitoring using environmental &

context sensors and a hierarchical analysis framework. In *BuildSys@ SenSys*, pages 90–99, 2014.

[2] Jaganathan Venkatesh, Baris Aksanli, Christine S Chan, Alper Sinan Akyurek, and Tajana Simunic Rosing. Modular and personalized smart health application design in a smart city environment. *IEEE Internet of Things Journal*, 2017.

[3] P Rajalakshmi and S Devi Mahalakshmi. Iot based crop-field monitoring and irrigation automation. In *Intelligent Systems and Control (ISCO), 2016 10th International Conference on*, pages 1–6. IEEE, 2016.

[4] Baris Aksanli. Accurate and data-limited prediction for smart home energy management. *Proceedings of the ASME 2018 Power and Energy Conference*, 2018.

[5] Surat Teerapittayanon, Bradley McDanel, and HT Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 328–339. IEEE, 2017.

[6] Bojan Milosevic, Jinseok Yang, Nakul Verma, Sameer S Tilak, Piero Zappi, Elisabetta Farella, Luca Benini, and Tajana Simunic Rosing. Efficient energy management and data recovery in sensor networks using latent variables based tensor factorization. In *Proceedings of the 16th ACM international conference on Modeling, analysis & simulation of wireless and mobile systems*, pages 247–254. ACM, 2013.

[7] Jagannathan Venkatesh, Christine Chan, Alper Sinan Akyurek, and Tajana Simunic Rosing. A modular approach to context-aware iot applications. In *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*, pages 235–240. IEEE, 2016.

[8] Raspberry Pi Foundation. Raspberry pi 3 specifications. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`.

[9] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 73–78. IEEE, 2015.

[10] Hannes Grunert and Andreas Heuer. Rewriting complex queries from cloud to fog under capability constraints to protect the users' privacy. *Open Journal of Internet Of Things (OJIOT)*, 3(1):31–45, 2017.

[11] Keiichi Yasumoto, Hirozumi Yamaguchi, and Hiroshi Shigeno. Survey of real-time processing technologies of iot data streams. *Journal of Information Processing*, 24(2):195–202, 2016.

[12] Clement T Yu and Weiyi Meng. *Principles of database query processing for advanced applications*. Morgan Kaufmann Publishers Inc., 1998.

[13] Arun Kumar, Jeffrey Naughton, Jignesh M Patel, and Xiaojin Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data*, pages 19–34. ACM, 2016.

[14] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment*, 10(11):1214–1225, 2017.

[15] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[16] Kaggle survey: The state of data science and ml. `https://www.kaggle.com/surveys/2017`. Accessed January 31, 2018.

[17] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[18] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[19] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[20] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[21] Yoav Freund and Robert E Schapire. Game theory, on-line prediction and boosting. In *Proceedings of the ninth annual conference on Computational learning theory*, pages 325–332. ACM, 1996.

[22] Jerome H Friedman and Werner Stuetzle. Projection pursuit regression. *Journal of the American statistical Association*, 76(376):817–823, 1981.

[23] Lee K Jones. A simple lemma on greedy approximation in hilbert space and convergence rates for projection pursuit regression and neural network training. *The annals of Statistics*, pages 608–613, 1992.

[24] Jeng-Neng Hwang, Shyh-Rong Lay, Martin Maechler, R Douglas Martin, and Jim Schimert. Regression modeling in back-propagation and projection pursuit learning. *IEEE Transactions on neural networks*, 5(3):342–353, 1994.

[25] Chris Bishop, Christopher M Bishop, et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[26] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[27] Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

[28] Douglas P Sutton, Martin C Carlisle, Traci A Sarmiento, and Leemon C Baird. Partitioned neural networks. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 3032–3037. IEEE, 2009.

[29] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[30] Sean Barker, Aditya Mishra, David Irwin, Emmanuel Cecchet, Prashant Shenoy, and Jeannie Albrecht. Smart*: An open data set and tools for enabling research in sustainable homes. *SustKDD, August*, 111(112):108, 2012.

[31] J Zico Kolter and Matthew J Johnson. Redd: A public data set for energy disaggregation research. In *Workshop on Data Mining Applications in Sustainability (SIGKDD), San Diego, CA*, volume 25,

pages 59–62, 2011.

[32] Pecan street dataport. https://dataport.cloud/, 2018.

[33] Ali Nourai, VI Kogan, and Chris M Schafer. Load leveling reduces t&d line losses. *IEEE Transactions on Power Delivery*, 23(4):2168–2173, 2008.

[34] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.

[35] Hande Alemdar, Halil Ertan, Ozlem Durmaz Incel, and Cem Ersoy. Aras human activity datasets in multiple homes with multiple residents. In *Proceedings of the 7th International Conference on Pervasive Computing Technologies for Healthcare*, pages 232–235. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[36] Yonatan Vaizman, Katherine Ellis, and Gert Lanckriet. Recognizing detailed human context in the wild from smartphones and smartwatches. *IEEE Pervasive Computing*, 16(4):62–74, 2017.

[37] Attila Reiss and Didier Stricker. Introducing a new benchmarked dataset for activity monitoring. In *Wearable Computers (ISWC), 2012 16th International Symposium on*, pages 108–109. IEEE, 2012.

[38] Yeseong Kim, Pietro Mercati, Ankit More, Emily Shriver, and Tajana Rosing. P4: Phase-based power/performance prediction of heterogeneous systems via neural networks. In *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*, pages 683–690. IEEE, 2017.

[39] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[40] Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, et al. Bigdl: A distributed deep learning framework for big data. *arXiv preprint arXiv:1804.05839*, 2018.

[41] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Lacross: Learning-based analytical cross-platform performance and power prediction. *International Journal of Parallel Programming*, 45(6):1488–1514, 2017.

[42] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[43] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[44] Wanlin Cui, Yeseong Kim, and Tajana S Rosing. Cross-platform machine learning characterization for task allocation in iot ecosystems. In *Computing and Communication Workshop and Conference (CCWC), 2017 IEEE 7th Annual*, pages 1–7. IEEE, 2017.

[45] MQTT. Mqtt protocol.

[46] Chun-Wei Tsai, Chin-Feng Lai, Ming-Chao Chiang, Laurence T Yang, et al. Data mining for internet of things: A survey. *IEEE Communications Surveys and Tutorials*, 16(1):77–97, 2014.

[47] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[48] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[49] Zhefeng Jiang and Shiwen Mao. Energy delay tradeoff in cloud offloading for multi-core mobile devices. *IEEE Access*, 3:2306–2316, 2015.

[50] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 615–629. ACM, 2017.

[51] Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. https://arxiv.org/pdf/1802.03835.pdf, 2018. ArXiv Preprint.

[52] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[53] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, 2017.

[54] David L Hall and James Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1):6–23, 1997.

[55] Bahador Khaleghi, Alaa Khamis, Fakhreddine O Karray, and Saiedeh N Razavi. Multisensor data fusion: A review of the state-of-the-art. *Information Fusion*, 14(1):28–44, 2013.

[56] D Elizondo and Emile Fiesler. A survey of partially connected neural networks. *International journal of neural systems*, 8(05n06):535–558, 1997.

[57] Sanggil Kang and Can Isik. Partially connected feedforward neural networks structured by input types. *IEEE transactions on neural networks*, 16(1):175–184, 2005.

[58] Saad Qaisar, Rana Muhammad Bilal, Wafa Iqbal, Muqaddas Naureen, and Sungyoung Lee. Compressive sensing: From theory to applications, a survey. *Journal of Communications and networks*, 15(5):443–456, 2013.

[59] Qing Ling and Zhi Tian. Decentralized sparse signal recovery for compressive sleeping wireless sensor networks. *IEEE Transactions on Signal Processing*, 58(7):3816–3827, 2010.

[60] Si-Yao Fu, Xin-Kai Kuai, Rui Zheng, Guo-Sheng Yang, and Zeng-Guang Hou. Compressive sensing approach based mapping and localization for mobile robot in an indoor wireless sensor network. In *Networking, Sensing and Control (ICNSC), 2010 International Conference on*, pages 122–127. IEEE, 2010.

[61] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[62] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[63] Sebastian Schelter, Andrew Palumbo, Shannon Quinn, Suneel Marthi, and Andrew Musselman. Samsara: Declarative machine learning on distributed dataflow systems. In *Machine Learning Systems workshop at NIPS*, 2016.

[64] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. Efficient index-based knn join processing for high-dimensional data. *Information and Software Technology*, 49(4):332–344, 2007.

[65] Defu Lian, Cong Zhao, Xing Xie, Guangzhong Sun, Enhong Chen, and Yong Rui. Geomf: joint geographical modeling and matrix factorization for point-of-interest recommendation. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 831–840. ACM, 2014.

[66] Jie Bao, Yu Zheng, and Mohamed F Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *Proceedings of the 20th international conference on advances in geographic information systems*, pages 199–208. ACM, 2012.

[67] Rana Forsati, Mehrdad Mahdavi, Mehrnoush Shamsfard, and Mohamed Sarwat. Matrix factorization with explicit trust and distrust side information for improved social recommendation. *ACM Transactions on Information Systems (TOIS)*, 32(4):17, 2014.

[68] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1969–1984. ACM, 2015.