2019 IEEE International Conference on Edge Computing (EDGE)

DEFT: Dynamic Edge Fabric environmenT

Seamless and automatic switching among resources at the edge of IoT network and cloud

Fatemeh Jalali [1], Timothy Lynar [2], Olivia J. Smith [3], Ramachandra Rao Kolluri [1] Claire V. Hardgrove [4], Nick Waywood [1], Frank Suits [1]
[1] IBM Research Australia, [2] The University of New South Wales, [3] Telstra Corporation Ltd, [4] The University of Sydney

***Abstract*—The number of IoT devices at the edge of the network is increasing rapidly. Data from IoT devices can be analysed locally at the edge, or they may be sent to the cloud. Currently, the decision to deploy a task for execution at the edge or in the cloud is not decided as the data are received. Instead the decision is usually based on pre-defined system design and corresponding assumptions about locality and connectivity. However, the mobile environment has rapid, sometimes unpredictable changes and requires a system that can dynamically adapt to these changes. An intelligent platform is required that can discover available resources (both nearby and in the cloud) and autonomously orchestrate a seamless and transparent task allocation at runtime to help the IoT devices achieve their best performance given the available resources.**

We propose a new platform, DEFT (Dynamic Edge-Fabric environmenT), that can automatically learn where best to execute each task based on real-time system status and task requirements, along with learned behavior from past performance of the available resources. The task allocation decision in this platform is powered by machine learning techniques such as regression models (linear, ridge, Lasso) and ensemble models (random forest, extra trees). We have implemented this platform on heterogeneous devices and run various IoT tasks on the devices. The results reveal that choosing proper machine learning approaches based on the tasks properties and priorities can significantly improve the overall performance of selecting resources (either from the edge or cloud) dynamically at runtime.

I. Introduction

The Internet of Things (IoT) includes numerous heterogeneous sensors and devices that can communicate with each other and with the Internet. The main computation platforms for IoT networks are cloud computing and edge computing. Cloud computing refers to large storage and computation resources located in geographically distant locations from the IoT local network, whereas edge computing refers to local storage and computation resources found within the IoT local network, close to the IoT devices and sensors. The compute resources at the edge of an IoT network can be as small as a single board computer (e.g. Raspberry Pi Zero W, wearable devices) or as large as a home server if the IoT network is in a fixed location such as a building. Some IoT applications work on cloud computing platforms, some on the edge computing, and some on hybrid edge-cloud based on the rules written by application developers at design time rather than at runtime [1]. The majority of IoT applications cannot change their platforms at runtime or choose the right resources automatically and dynamically [1], [2], [3], [4], [5].

All authors were working with IBM Research Australia (22/60 City Road, Southbank VIC 3006 Australia) during working on this research project. The corresponding author's email is: fa.jalali@au1.ibm.com

The fixed and predefined platforms may work well for stationary IoT applications with non-mobile resources and a stable environment, such as IoT applications inside buildings that servers are not mobile and connected to constant power. However, mobile IoT applications deal with rapid changes in bandwidth and connectivity and limitations on power and computation, and even with an isolated system within a single building, greater autonomy and adaptability of the devices would simplify the addition of new devices and restructuring of the system as large scale changes are needed.

Mobile IoT devices in such a dynamic environment should be able to choose their resources at runtime dynamically, selecting between the nearby resources at the edge of the IoT network and the cloud in order to attain the best overall performance. To achieve this goal, we propose a Dynamic Edge-Fabric environmenT (DEFT) with the following characteristics:

a. The network is decentralised and IoT devices may act as either task requesters or task performers.
b. IoT devices can automatically join a trusted IoT network and discover available resources.
c. IoT devices connected to this network get updates on the status of nearby resources within the local IoT network as well as those in the cloud.
d. The IoT nodes in this platform are powered by machine learning techniques to predict the behaviour of other nodes for tasks' assignment based on the real-time status of the environment (connectivity, bandwidth, etc.) and available nodes (e.g. CPU, memory, remaining battery).
e. The tasks may have a variety of requirements, such as low latency or minimum bandwidth usage. The choice of resource to use for deployment of a task should not simply be greedy and ask for the most powerful resource available - and instead should match the resource optimally to the goals and requirements of the task.

The proposed platform comprises a collection of heterogeneous nodes representing devices with some combination of sensing and computational capability in an IoT environment. Each task from a particular IoT device (known as a *task requester*) can be assigned to a *task receiver* at runtime. The *task receiver* can be any of the following possibilities based on the task objectives and the environment status:

a. The requester node itself
b. Another nearby node in the local IoT network
c. A resource in the cloud

The selection of the task receiver is made based on a machine learning model running on the requester node that learns over time where to assign its tasks at runtime. In this work, the requester node is powered by regression models (linear, ridge, Lasso) and ensemble models (random forest, extra trees) to make the task allocation decisions. The task requester must be able to predict how each of the available task receivers would perform a given task and use that prediction to determine the best match of receiver to the given task.

A. *Example application*

A simple example of the proposed system is a wearable device such as a smart watch shown in Figure 1 with a rechargeable battery that is itself capable of some level of local computation and communication. The dashed lines in the figure show unstable wireless connection that can be connected or disconnected based on the location of the smart watch and its proximity to other resources.

The ideal scenario is that the device can handle all the tasks and computations itself. However, if the available computation power is insufficient to perform a task, or if the battery is currently too low for such a computation, the task should be assigned to a nearby resource or to the cloud automatically. In the case of nearby nodes, it would depend on the location of the mobile IoT device; if the person wearing the wearable device is at home or in a car, the system should be able to automatically connect the smart watch to the home or car network and find the available resources at each location (e.g. a laptop, set-top unit, home server, or car computation module) and offload the task to a proper resource. If the person is far from home or car (e.g. walking), the system should automatically identify the person's portable devices, such as a smart phone, and allocate the task to the portable device. If nearby resources are busy or unavailable the task should be allocated to the cloud. In this platform, the smart watch has a machine learning model for which the inputs are the type

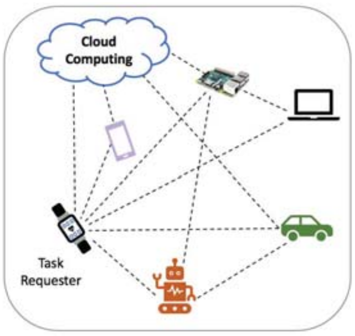

Fig. 1: An example of the DEFT system. The smart watch is a task requester and other available resources, either nearby devices or the cloud, can be a task receiver.

of task, available resources, and environment status - and the output is the best available resource(s) for the task. In this case the task allocation between the resource at the edge and the cloud is automatic at runtime and occurs in a seamless, autonomous, and transparent manner.

B. *Role of data and predictions*

Unlike systems that must calculate the performance of a given task on a given node, we propose the use of machine learning to *predict* the performance on a computational node given the current state of that node. To enable autonomous application of machine learning as a task running repeatedly, we investigate the use of ongoing cross-validation to choose the best machine learning approach. We also explore different approaches to select which node should complete a given task, in terms of both the performance of that task and the system's confidence in future predictions. The results show that the techniques we have suggested can improve overall performance of the system by significantly reducing the time taken to perform tasks due to sensible selection of nodes and states where the goal is time reduction. The results also indicate that a single machine learning model cannot predict the task completion time (or energy consumption) of various types of tasks. Therefore, it is necessary to apply the best machine learning model for each type of task.

C. *Paper organization*

This paper is organized into sections as follows: Section II surveys previous researches that has been done on task allocations for mobile IoT networks. In section III, we describe the system model, goals and the architecture of the proposed environment. Section IV presents the details of the learning algorithms and Section V reports the results. Finally, we summarize the paper in Section VI and highlight a few open research directions in the relevant areas that are of significance and require further attention.

II. Related Work

The idea of task allocation to other resources (offloading computation) has received considerable attention in the research community since mobile computing has become pervasive. The main reason is that small mobile devices are limited by local computation and power resources (i.e. portable batteries). Data and computation offloading are used to give an opportunity to mobile devices to complete heavy computation tasks without consuming significant amounts of battery, local memory and CPU.

Computation can be sent to remote resources in geographical distance data centres in the cloud or to nearby resources at the edge of the mobile network. In the following subsections we review both categories of research related to our work and then specify how our work is different.

A. *Computation offloading to remote resources (Cloud)*

The majority of previous research on computation offloading focuses on sending data and computation only to the cloud [1], [2], [3], [4], [5], [6] or to the fog, [7] which is a small version of the cloud located somewhere closer to the IoT devices but still outside of the IoT local network.

The decision as to whether an application or a part of an application is offloadable can be made statically during the design of the applications. In this case, the part(s) of application that can be offloaded are annotated by application developers manually during the development phase. For example, if the programmers know that the application requires heavy computation, they can mark it as a offloadable. Phone2Cloud [1] is one example that uses a static method to choose to run an application locally or send it to the cloud.

Other work such as ThinkAir [2], MAUI [3], CloneCloud [4], and Jade [5] make a dynamic decision at runtime to offload an application to the cloud. In this case, applications' performance, real-time conditions of the network and local hardware specifications are monitored. The final decision is made in real-time according to the applications' types and available resources. Some work such as Cloudlet [8] has not clearly mentioned that the offloading is static or dynamic. The frameworks which make dynamic decisions are discussed below.

ThinkAir [2] introduces a framework to offload mobile computation to the cloud. This framework profiles the applications characteristics when running on the mobile phone and also in the cloud. Additionally, it monitors the network and mobile phone status at runtime to accommodate changing computational requirements dynamically. This framework focuses on the elasticity and scalability of the cloud and parallel execution of offloaded tasks using multiple virtual machine (VM) images.

MAUI [3] proposes a framework that optimises energy consumption of a mobile device by offloading heavy computation tasks dynamically to a remote resource. The framework considers the trade-off between the energy consumed by local computation versus data transmission energy consumption for remote execution. This framework only focuses on the energy saving and it does not consider other metrics such as application performance or delay when offloading the data and computation.

CloneCloud [4] presents a framework for improving the battery life and performance on mobile devices through offloading intensive components to the cloud using a cloned virtual machine (VM) image. This framework uses an offline static analysis of various scenarios with different conditions on both the mobile device and the cloud to determine offloadable parts of the application. The result of this analysis goes to a database to identify which parts should be sent to the cloud. This solution is limited by the number of environmental conditions in the offloading pre-processing. In addition, the database should be updated manually as a new application is added to the system.

Jade [5] is a framework for maximising the use of energy-aware offloading for mobile applications while minimising the burden on the application development for programmers. This framework monitors the application and mobile device status and decides where to run the application automatically to reduce the energy consumption of the mobile phone. The main focus of Jade is saving energy.

The work on Cognitive IoT gateways [6] proposes dynamic switching between edge/fog and the cloud at runtime. The authors used machine learning (SVM: Support Vector Machine) to identify the type of IoT applications. The mobile phone status (e.g. CPU, memory, remaining battery) and the network condition (e.g. bandwidth) were continuously monitored and the offloading decision was made based on multi-objective optimization on the type of running application and the system real-time conditions.

All the above studies assume that mobile and IoT devices are connected to the Internet a good fraction of the time, but that is not always the case in mobile computing environment. Therefore, it is required to consider neighbourings nodes on which to offload tasks and computation.

B. Computation offloading to nearby resources (Edge)

Some other work considers decentralised and short range communications to offload the computation to nearby resources [9], [10], [11], [12], [13], [14].

In [9], [10], the authors used device-to-device (D2D) communication to offload tasks to nearby mobile devices instead of cloud data centres. This paper studied the performance of distributing tasks to several nearby mobile nodes and retrieving tasks after execution. The neighbour nodes in these two studies could be reached with more than one hop and other intermediate nodes (relays) might be used to transfer the tasks to the new mobile device. It may cause some delay compared to one hop communication.

The authors of [11] proposed one-hop D2D communication to offload tasks to neighbouring mobile devices with better cellular service quality and then to the cloud. The aim in this work is not to find nearby peers that can serve the computation, but instead it uses nearby peers with better cellular service quality to transfer the computation and tasks to the cloud.

In [12], the computation can be offloaded to the mobile peers or to the servers located outside the local IoT network. The offloading decision is made using successive convex approximation (SCA) method and geometric programming (GP) subject to minimizing the weighted energy consumption while maintaining the application latency requirements.

Aura [13] is a highly localised and ad hoc computing platform which is built upon the IoT and other computing devices in the nearby physical environment. Aura offloads tasks to untapped computing resources in the surrounding environment using an incentive and contact based model that uses a reward for rating and penalizing IoT worker nodes based on task performance, task offloading, and partial state.

The recent work in [14] proposes a distributed task allocation scheme among nodes at the edge of IoT network as well as the cloud. This work uses a combination of machine learning and heuristic methods in an auction-based system in order to provide near-optimal task allocation. In this study, only one machine learning model (linear regression model) is used.

C. Contributions

This proposal platform takes insights from prior work on computation offloading explained in the previous sub-sections. The platform monitors and profiles tasks, network and hardware properties continuously to make dynamic decisions at runtime. Unlike the majority of prior works that rely on evaluating the performance of a given task on a compute node, we propose the use of machine learning to *predict* the performance on a computational node given the current state of that machine and its inherent capabilities.

With this prediction our platform can decide where to offload the computation based on the task objectives and real-time conditions of the whole system. Various machine learning models are applied to learn the system, the application type and environment characteristics over time in order to make proper decisions dynamically to save energy, reduce delay and improve performance of the mobile IoT applications.

In addition, our proposed platform can automatically discover peer resources that are available nearby in the Edge or remote in the cloud. The proposed idea in this work is implemented on real devices and evaluated via practical experiments.

III. System Model

In this section we discuss the goals of designing DEFT (Dynamic Edge-Fabric environmenT) and describe the architecture of the whole system.

A. Design goals and architecture

The design of DEFT framework is based on three key objectives. The first objective is to improve computational performance as well as the power efficiency of IoT devices. The second is to embrace the rapid changes in mobile IoT environment and react dynamically to the changes at runtime. The third is to discover the available resources, either nearby resources or the cloud, automatically and use them to allocate tasks when required.

In this IoT ecosystem, there are various heterogeneous IoT nodes such as smart phones, Raspberry Pi's, drones, robots, wearable devices as shown in Figure 1.

Each IoT node can communicate with other nearby IoT nodes via a short range wireless communication such as Wi-Fi, Bluetooth, ZigBee, etc. The IoT nodes can be directly connected to the Internet (e.g. the mobile phone shown in Figure 1) or they can be connected to the Internet via other nearby nodes (e.g. the robot shown in Figure 1).

Each node in this system can be either a *task requester* or a *task receiver*. All nodes broadcast their available computation resources (CPU, Memory, etc.) and power (battery) to the nearby nodes in their range to inform them on the available nearby resources. When a *task requester* wants to perform a task, the following steps would be done:

- The *task requester* node monitors the status of the task to identify the task type and task objectives which is shown in Figure 2 as component number 1 (inside a yellow circle).
- At the same time, the *task requester* checks its own available resources (CPU, memory and battery) which is marked as component 2 in the figure.
- It also checks the status of the network highlighted as component 3 in the figure.
- The *task requester* discovers other nodes' availability (nearby nodes or cloud) and gets the status of the nearby nodes. This component is marked with number 4 in Figure 2.
- The data that gathered from components 1-4 are used as inputs for the machine learning part marked as component number 5 in Figure 2. It runs machine learning algorithms and uses historical data to see how to run that specific

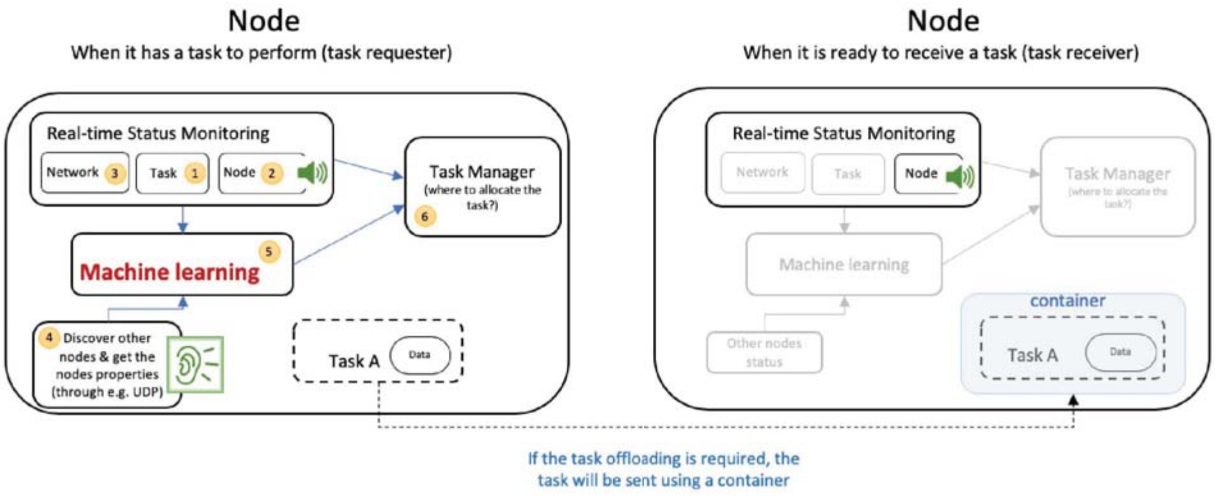
Node
When it has a task to perform (task requester)
Real-time Status Monitoring
Network 3
Task 1
Node 2
Task Manager
(where to allocate the task?)
6
Machine learning
5
4 Discover other nodes & get the nodes properties (through e.g. UDP)
Task A
Data
Node
When it is ready to receive a task (task receiver)
Real-time Status Monitoring
Network
Task
Node
Task Manager
(where to allocate the task?)
Machine learning
Other nodes status
container
Task A
Data
If the task offloading is required, the task will be sent using a container

Fig. 2: Key components of the DEFT framework. Some components (e.g. machine learning, task manager, etc.) are inactive on the task receiver node.

task efficiently (will be discussed in Section IV). It is worth noting that machine learning only happens on *task requester* nodes and it is inactive on *task receiver* nodes.

- Then, the task manager marked as component number 6 selects the *task receiver* to perform the task. The *task receiver* might be the requester node itself, one of nearby nodes or the cloud.
- If task offloading is required, the task and the data will be sent to the *task receiver* using a container as shown in Figure 2. The *task receiver* downloads the container to perform the task if it was not downloaded previously.

B. Properties that affect resource allocation

It is obvious that task distribution is worthwhile only when the local node consumes more time and/or power than the allocation overhead to the nearby nodes or to the cloud. The allocation decision and process can strongly be impacted by parameters such as the task properties, network properties, the computation nodes (local node, nearby nodes and cloud server) properties and user's preferences.

All the components have properties that should be considered in detail since they highly impact the decisions of dynamic task allocation. We discuss them as follows:

1) Node properties: Each node in this platform is capable of some level of computation. Each node periodically broadcasts its available computation and power resources. For the computation, the CPU (Central Processing Unit) type, the max CPU frequency (e.g. 4 MHz), CPU utilization from 1-100%, and the memory currently in use are the required parameters to be used. For the power, the parameters such as source of power (from constant connectivity to the grid or using battery), the max battery capacity and the remaining battery (if it is powered by a battery) are required to be continuously monitored and logged.

2) Network properties: The underlying network of an IoT environment impacts on the speed and reliability of the connections. In order to know the speed and the required time to transfer or receive data from nearby devices or the cloud, the type of wireless connection among the IoT devices (e.g. Bluetooth, Wi-Fi, ZigBee), communication protocols (e.g. UDP, TCP, MQTT) and available bandwidth are required to be considered.

In this work, User Datagram Protocol (UDP) is used for discovery of new nodes, and for updating the status of nodes to the rest of the community. The discovery happens as follow:

- Devices broadcast their available status on CPU, memory and etc. using UDP.
- Each device repeats the broadcast when the status of the device changes beyond a given threshold value.
- The task requesters receive the broadcast messages and then send the broadcast messages to the machine learning model to decide where to perform the tasks.

3) Task properties: The type of tasks plays an important role on task allocation and offloading decisions. The tasks can be categorized based on different properties such as the computation (heavy computation, medium computation, light computation) or the time completion (delay-sensitive or delay tolerant) or the volume of traffic (light traffic such as sensor data, medium traffic such as images, high traffic such as video) or storage demand.

For example the task allocation decision for a delay sensitive and heavy computation task could be different compared to a delay tolerant task with the similar required computation.

4) User preferences: In addition to all discussed properties, the users can also introduce their preferences and force some

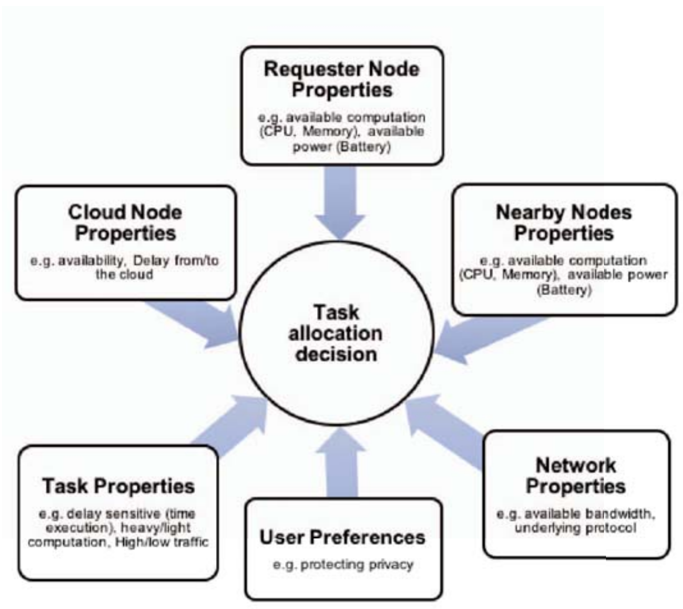
Requester Node Properties
e.g. available computation (CPU, Memory), available power (Battery)
Cloud Node Properties
e.g. availability, Delay from/to the cloud
Nearby Nodes Properties
e.g. available computation (CPU, Memory), available power (Battery)
Task allocation decision
Task Properties
e.g. delay sensitive (time execution), heavy/light computation, High/low traffic
User Preferences
e.g. protecting privacy
Network Properties
e.g. available bandwidth, underlying protocol

Fig. 3: Impactful parameters on Task Allocation in IoT

applications to be done always locally or to be sent to the cloud. As an example, protecting users' privacy can be a reason to run application at the edge.

IV. Machine learning algorithms

A. Data types and acquisition

The machine learning task in our framework is for a single node to be able to predict the performance (specifically the time and energy usage in the given context) of a single task on a potentially unknown compute node based only on the latter's system parameters. The system parameters incorporate static, dynamic and calculated data. Most of the data we collect and use for the machine learning framework is readily available on most compute nodes. The only bespoke data collected is the benchmark data, which refers to two benchmarks from *sysbench* [15], memory and CPU, each run on a single core or multiple cores. We require each node on the system to run these short benchmarks and record their run times as static data for the system.

The remaining static data we collect are the maximum CPU speed, total memory, and number of CPUs. The dynamic data we collect are current CPU speed (less than maximum on a Raspberry Pi, which throttles CPU when hot), % availability of CPU and memory, ping-time from the requesting node, and a flag specifying whether or not the requesting node is the considered node (`is_me`). The calculated data consists of maximum CPU speed $\times$ % available, current CPU speed $\times$ % available, memory % available, total memory $\times$ % available, and number of CPU cores $\times$ current CPU speed $\times$ % available.

B. Regression methods

Predicting the outcomes of running a task on a given node is a regression task. In this paper, we compare both linear and tree-based ensemble regression approaches. The linear regression approaches we consider are standard least-squares linear regression as well as regularised regression, with either ridge or LASSO regression. For tree-based ensemble methods we use both random forest regression and extra (extremely randomised) trees. We use background theory and programming from *scikit-learn* libraries [16] for machine learning throughout this work.

When using a linear regression approach, the presence of strongly collinear input variables can be very detrimental. Due to many of our parameters being static parameters, that only take on one value for each node in the network, there are many strongly correlated parameters in our data. We therefore consider a greatly reduced subset of input variable to be used for all linear models. This subset is chosen such that the absolute value of correlation between any two different variables is no higher than 0.7. For our particular data set, the smaller list of variables selected is CPU and memory % availability, benchmark times for CPU multi core and memory single core, average ping time over 3 attempts, current CPU $\times$ % availability of CPU, and the previous variable multiplied by the number of cores. It should be noted that these variables were chosen manually for this paper. However, an automatic process could manage this feature selection.

C. Cross-validation

Within the DEFT framework, a machine learning model for a particular task is repeatedly fitted to data and is used to make predictions for a number of compute nodes. One of these nodes will then be chosen to have the task assigned to it, meaning that the true performance of the node in that state will be discovered and can be added to the training data. For a given task, it is not clear *a priori* what type of model will be most accurate or what hyper-parameter values should be set for it. We therefore allow for a mode of operation that incorporates cross-validation on the fly. Specifically, a set of model types and associated hyper-parameter values is given, along with a frequency f of cross-validation and an initial model. After every f new data points are found, k-fold (with $k = 3$) cross-validation is used to compare the performance of each model type with each relevant set of hyper-parameters. The model with the best measured performance is used until another f data points are added to the training set when the cross-validation is repeated.

D. Explore-exploit trade-off

Another important aspect of this regression model is that the DEFT framework controls what data is added to the training set. This means that when choosing the node to be given the next task, there is a trade-off between exploitation (choosing the apparently best node) and exploration (choosing a node that will potentially improve the accuracy of the model). The node we choose should be based on a combination of predicted performance and areas of low certainty.

We manage the explore exploit trade-off using lower confidence bounds by defining performance on the basis of prediction intervals, drawing inspiration from various works [17], [18], [19]. The α prediction interval of a model for a given input (e.g. $\alpha = 95\%$) represents a range in which there is an α (95%) probability of the next data point occurring. This range incorporates both the implicit variability of data and uncertainty in the parameters guiding the model. By specifying the predicted performance of a node as the best value within an α prediction interval, the lower confidence bound, we incorporate both prediction and uncertainty, leaving us more likely to find a new high quality point than if we always selected the node with the best predicted performance.

Prediction intervals for single output linear regression at a given input point $\overline{x}$ depends on the training data and the mean squared error (MSE) of the fit to training data as well as $t_{\alpha/2,df}$, the t statistic with value $\alpha/2$ at with df degrees of freedom, calculated as the number of data points less the number of features (including the bias term). If the training inputs are $(\mathbf{X}, y)$ where $\mathbf{X}$ includes a bias feature (where all values are 1), and the prediction of the model at $\overline{x}$ is $\overline{y}$, then the α prediction interval is $\overline{y} \pm t_{\alpha/2,n-p} \times \sqrt{MSE\left(1 + x(\mathbf{X}^T\mathbf{X})^{-1}x^T\right)}$.

For ensemble methods, we approximate prediction intervals using the ideas of quantile regression trees. Consider the probability distribution of future data points by individually calculating the predictions from each member of the ensemble (each tree). The $\alpha/2$ quantile of these predictions is an approximation to the lower confidence bound.

Node i	Node type	Processor type	Cpu frequency (max)	Memory (max)	# cores	Benchmark runtime
1	Raspberry Pi Model 3B	ARM Cortex-A53	1.2GHz	1GB LPDDR2	4	1.8419 s
2	Server	Intel Xeon E5	2.6 GHz	1GB	1	2.3993 s
3	Raspberry Pi Model 3B	ARM Cortex-A53	1.2GHz	1GB LPDDR2	4	1.9303 s
4	Raspberry Pi Model 3B	ARM Cortex-A53	1.2GHz	1GB LPDDR2	4	3.1521 s
5	Server	Intel Xeon E5	2.6 GHz	1GB	1	2.2743 s
6	Macbook pro	Intel Core i7	3 GHz	8GB DDR4	4	0.3121 s
7	Cloud	Intel Xeon E5	2.6 GHz	64GB	16	2.3353 s

TABLE I: Configuration, static properties and benchmark run results of nodes participating in the Dynamic Edge Fabric environmenT (DEFT) experiment.

V. Evaluations and Results

To analyse the various machine learning approaches, we investigate:

- which simple machine learning methods are sufficient to make accurate predictions,
- how many observations are required to make an accurate prediction of task completion time,
- the impact of repeated cross-validation, and
- the impact of various methods of choosing the next node - random, the best node, or using prediction intervals (α).

We implemented a multi-node IoT network with fluctuating load by continuously and randomly deploying background jobs which tax either memory, CPU or neither resource on each node. The experimental DEFT analysed in this section consists of nodes listed along with their static properties in Table I.

We then repeatedly submitted five tasks from a single node in the IoT network to every node in the network (including itself). These tasks are: a dockerised face detection task, larger versions of the two sysbench tasks used for benchmarking, finding the 35^{th} Fibonacci number without caching, and repeatedly inverting a random 1000×1000 matrix.

We collected data about the system parameters on each node prior to submitting each task, as well as data on the task completion time of each task on each node. For Raspberry Pis, we also have data on energy usage. Task completion time is measured as the time between submitting a task and receiving confirmation of task completion. The task schedule was simplified by waiting for each task to complete on all nodes before submitting the next task. This provided a complete dataset about which node would have been the best node to complete each task under a range of system loads, which is used to evaluate the decisions made by each machine-learning method.

Figures 4 and 5 show the distribution of the total duration and energy usage, respectively, for each node i for two of the tasks. It is clear that different task types tax nodes differently.

A. Prediction performance

In order to define the best learning algorithm, standard scoring metrics such as mean squared error (MSE) can be used. The regression problem solved here is for the purpose of selecting which of a number of nodes should be assigned a task. That is, the regression model has succeeded if the node with the best prediction is also the node with the best actual performance. We define a performance metric $\mathcal{M}$ by repeatedly selecting ten data instances, predicting their performances and choosing the best. We record the percentage of times the predicted best data point matches the data point with the best actual performance. A random algorithm would have a 10% performance on this metric while a perfect predictor would achieve 100%.

B. Training set & prediction accuracy

In order to establish an understanding of the right amount of data required by each machine learning model to make a reasonably accurate prediction, we inspect learning curves for the various regression models. Here we show results for the CPU test task. The accuracy metric used here is the MSE as described in IV and V-A. Training curves plot accuracy on an evaluation data set compared to the number of training examples. This is representative of a dynamic learning system as the number of training examples steadily increases. Figure 6 shows example training curves for the regression models. From the learning curves, it can be seen that ridge regression in Figure 6 has the best MSE parameter but needs a little more than a hundred data points to reach reasonable MSE. Random forest regression, as shown in Figure 6, needs much fewer data points to reach a reasonable MSE but it never converges to the performance of a ridge regression, for the given data set.

From Figure 6 it is seen that Lasso regression is very unstable for the given data set, while the other linear methods both reach a point where there is no significant difference

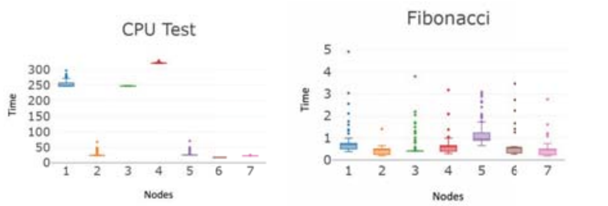

Fig. 4: Distribution of task completion times [seconds] for seven different types of nodes for two of the considered tasks.

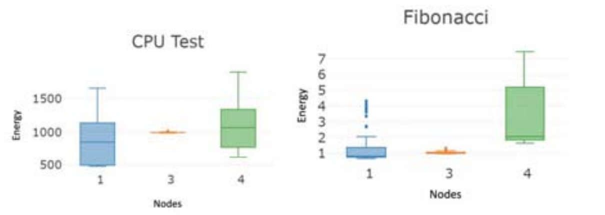

Fig. 5: Distribution of energy usage [mJ] for three different Raspberry pi's under two sample tasks (CPU test, Fibonacci).

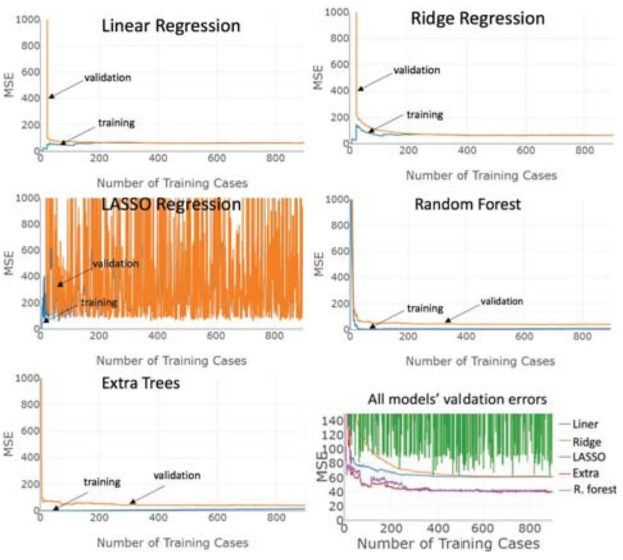

Fig. 6: MSE [s^2] of duration [s] under all machine learning models for training (blue) and validation (orange) sets. The validation errors of all models are compared.

between the error on training data (the best this model can achieve) and its performance on unseen data. The ensemble based methods as shown in Figure 6 have a noticeable gap between the training error and validation error meaning there may be room for further improvement. The combined results in Figure 6 show that the two ensemble methods have lower error than the linear based methods for this specific task. For other tasks, the performances are different, with linear regression performing best for some. It is also clear that for this task of testing CPU, the best performing approach changes between extra trees and random forest regression.

Remark 1. *It is important that the predictions can be made quickly, with minimal data regarding the other nodes, and with as few training samples as possible. In present context, the amount of time and energy that should be expended on finding the best location for each task may also depend on the amount of time and energy used by the task itself.*

Figure 7 shows the evolution of our percentage correct metric as in Figure 6 with increasing number of samples. As seen earlier, the LASSO regression is very unstable, while the other methods all perform well, eventually reaching around 80% correct performance, although in the case of ridge regression, it takes almost all 900 available data points before reaching that level. The ensemble methods again perform better on this data set.

Remark 2. *The results of various machine learning methods on the data set for the CPU intensive task show that ensemble methods (random forest, and extra trees) outperform the regression models since they have have lower error compared to the regression and reach a reasonable MSE although the training and validation sets never converge. For other types of tasks, the performances are different, with linear regression performing best for some.*

C. Impact of repeated cross-validation

Our next set of experiments investigate the use of ongoing cross-validation to make the approach adaptable to different data sets. Figure 8 shows the MSE for the training and validation data sets with and without performing cross-validation every 100 iterations. For Fibonacci and matrix inversion tasks, the initial linear model is the best model and is unchanged with or without cross-validation. For the CPU test and face detection tasks however, the cross-validation mainly chooses random forest regression, with occasional extra trees. For the memory test task, cross-validation also generally selects random forest, although it uses ridge regression for one where the validation error is low for that. The time taken for cross-validation in these experiments is around 15 seconds, meaning it should used sparingly, and could itself be a good task for offloading depending on system availability.

Remark 3. *Cross-validation in this work chooses the method and parameters that perform most accurately on the training set. In highly time constrained environments, it could also choose on the basis of training speed, looking for the fastest model with accuracy close to the best accuracy. It may, for example, result in choosing the faster extra trees over random forest regression.*

D. Impact of explore-exploit methods

Figure 9 shows the impact of managing the explore exploit tradeoff with varying prediction interval values (α) on the CPU test task with linear regression. It shows the results of 250

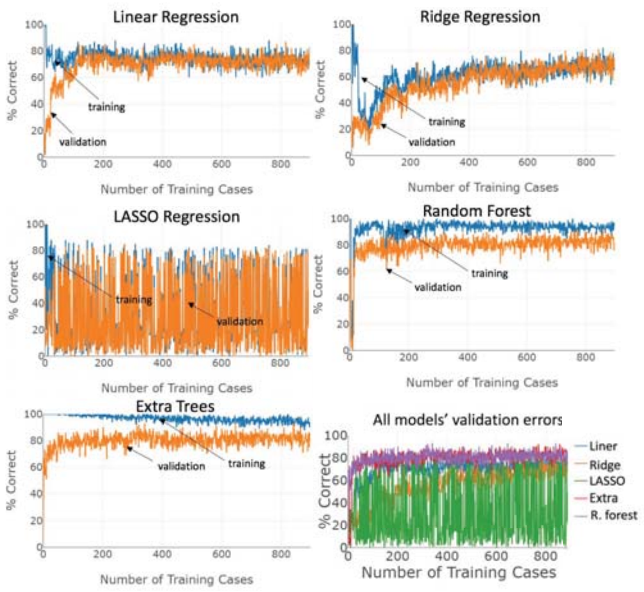

Fig. 7: The percentage of times that machine learning models predict accurately the “best node” again using the CPU test task. The validation errors of all the models are compared.

Task	Method	Data collection time	Train (fit) time	Predict time	Cross-val time	Average MSE
Face Detection	random	4,417	0.251	0.000	6.646	20.6
	best	1,551	0.214	0.007	6.529	32.3
	$\alpha = 0.5$	1,713	0.192	0.053	6.573	71.6
	$\alpha = 0.95$	1,690	0.208	0.059	6.482	45.2
Fibonacci	random	172	0.052	0.000	6.643	0.136
	best	122	0.128	0.004	6.520	0.150
	$\alpha = 0.5$	129	0.038	0.017	6.981	0.158
	$\alpha = 0.95$	128	0.105	0.031	6.580	0.138
Matrix Inversion	random	165	0.093	0.000	6.940	0.280
	best	123	0.095	0.003	6.905	0.291
	$\alpha = 0.5$	119	0.165	0.042	6.799	0.279
	$\alpha = 0.95$	126	0.100	0.029	7.273	0.287
CPU Test	random	34,981	0.245	0.000	6.965	47.3
	best	7,842	0.145	0.005	6.456	209
	$\alpha = 0.5$	7,912	0.141	0.042	6.772	207
	$\alpha = 0.95$	8,540	0.198	0.058	6.643	165
Memory Test	random	1,323	0.122	0.000	7.008	0.717
	best	852	0.125	0.004	6.733	0.999
	$\alpha = 0.5$	846	0.211	0.061	6.624	0.881
	$\alpha = 0.95$	849	0.180	0.051	6.396	1.24

TABLE II: Results (time [s]) from the combined system for each task averaged across five runs of selecting 250 data points.

iterations, in each of which 10 data points (available resources for computing the task) are randomly selected and one is chosen to be added to the training set using the given selection method. Each of these is shown as the average of 10 runs. Figure 9 (top) shows the total time spent on completing all of the tasks in the training set while Figures 9 (bottom) shows the MSE on the training set and validation sets as the training set size increases. It is clear that selecting randomly, as expected, leads to much more time being required for the tasks in the training set, while leading to better accuracy on the validation set as the training data is more varied. The other methods, which have a much less diver training set are between choosing the predicted best and choosing with lower confidence bounds, the most effective option varies by training task and learning method, although in this case, lower confidence bounds with $\alpha \in \{0.25, 0.5, 0.75\}$ appears to be the best performing choice

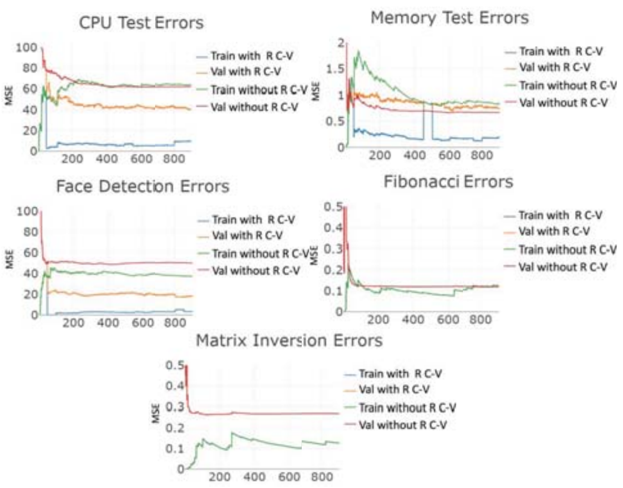

Fig. 8: The mean squared error [s^2] with and without repeated cross-validation (R. C-V) every 100 steps for each task. The initial model is linear regression.

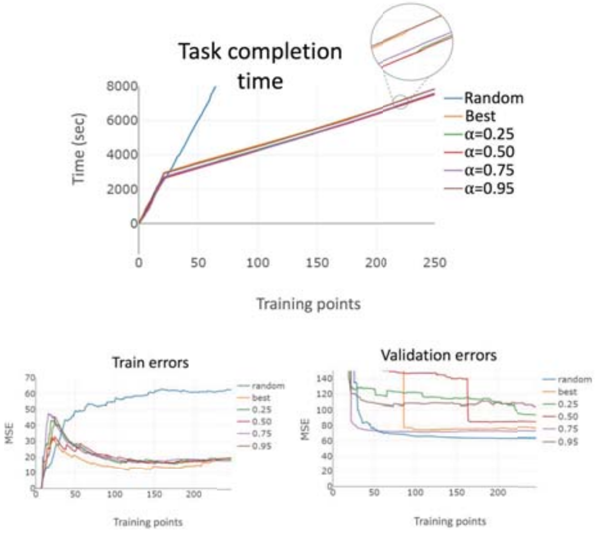

Fig. 9: Various selections (random, best, prediction interval values $\alpha = 0.25, 0.50, 0.75, 0.95$) of the target node and the impact of the training set size (0-250) on tasks completion time (top), MSE of the training set (bottom left) and validation sets (bottom right).

of methods.

All methods in this paper have been combined in our final set of tests. These tests are initialised with 20 randomly chosen data points (available resources) and then cross-validation is performed. 250 new data points are chosen using $\alpha = 0.5$, the best (predicted) performance, or random selection with cross-validation being repeated every 100 iterations. This is repeated five times and averaged for each of our considered tasks. Table II shows the incurred time to collect all data as well as the average time for fitting (training) models, predicting and finding the prediction interval and cross-validating as well as the average MSE across the last 100 iterations.

The most striking result here is that choosing nodes based on predicted times is significantly faster than choosing them randomly. In the case of the CPU test, it decreases the time from nearly 35,000 seconds to under 8,000 seconds. The prediction times are all short which is important as prediction occurs most often in our system, once for every considered node for each task instance to be allocated. Unsurprisingly, the lower confidence bound methods have greater prediction times due to calculating the prediction interval rather than the average prediction. Whether or not this difference matters will depend on the system requirements. For some methods, it provides better results but this is not universal. The average fit (training) times are also short, all well under a second, although for ensemble based approaches this can be relatively high, around 0.25 seconds. In our experiments, the fit times for ensemble approaches grow roughly linearly with the number of training points, meaning that with many training points online learning (updating rather than re-fitting the model) would be valuable, see [14] for example. For cross-validation, the time taken is fairly consistent. It depends on how long fitting to the data takes as well as how many different models are being considered. The average MSE for these models follows our intuition, with the the more varied randomly collected data showing lower error than the focused collection methods.

Remark 4. *To limit the cost of making predictions, a system with many nodes would only make predictions about those nodes that are, in some sense, the best candidates. Selecting top nodes can be based on simple proxies, such as currently available CPU or memory.*

VI. Conclusion and Future Work

We have proposed a framework called Dynamic Edge Fabric environmenT (DEFT) that is capable of using multiple machine learning techniques for task allocation between resources at the edge of IoT network and the cloud. This framework is implemented and tested using heterogeneous IoT devices. The network design is plug and play, allowing new nodes to enter the fabric and automatically distribute tasks originating from any nodes in the networks to nodes that are either fastest for task completion or the most energy-efficient. The allocation is dynamic and uses local machine learning and data storage.

We have applied various machine learning techniques for finding target (receiver) nodes to perform tasks. Given that the DEFT framework uses task-centric machine learning model development, ensemble regression methods performed well for accurate task deployment compared to simple or regularised regression methods in most cases. We have shown the potential benefits of factors such as ongoing cross-validation and the use of lower confidence bounds to manage the trade-off between exploration and exploitation in choosing target (receiver) nodes. These advanced methods require compute time and energy and their value for a given learning task depends on both the availability of these resources and the likely improvements from better models. Prediction models for long, highly variable tasks will benefit most from factors such as ongoing cross-validation.

Tuning hyper parameters and encapsulating various machine learning methods into one for task deployment in the DEFT framework is left to further work, along with online learning to speed up the fitting process. Another interesting avenue for exploration is to allow the user to specify task-specific factors that may influence predictions, such as the size of an image to be processed. Decision tree extraction and heuristic logic based algorithms are in development.

References

[1] F. Xia, F. Ding, J. Li, X. Kong, L. T. Yang, and J. Ma, "Phone2cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing," *Information Systems Frontiers*, vol. 16, no. 1, Mar 2014.

[2] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*, March 2012, pp. 945–953.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *ACM MobiSys 2010*. Association for Computing Machinery, Inc., June 2010.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 301–314.

[5] H. Qian and D. Andresen, "Jade: An efficient energy-aware computation offloading system with heterogeneous network interface bonding for ad-hoc networked mobile devices," in *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, June 2014, pp. 1–8.

[6] F. Jalali, O. J. Smith, T. Lynar, and F. Suits, "Cognitive iot gateways: Automatic task sharing and switching between cloud and edge/fog computing," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos '17, 2017, pp. 121–123.

[7] A. Yousefpour, G. Ishigaki, R. Gour, and J. P. Jue, "On reducing iot service delay via fog offloading," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 998–1010, April 2018.

[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct 2009.

[9] Y. Li, L. Sun, and W. Wang, "Exploring device-to-device communication for mobile cloud computing," in *2014 IEEE International Conference on Communications (ICC)*, June 2014, pp. 2239–2244.

[10] M. Jo, T. Maksymyuk, B. Strykhalyuk, and C. H. Cho, "Device-to-device-based heterogeneous radio access network architecture for mobile cloud computing," *IEEE Wireless Communications*, vol. 22, no. 3, pp. 50–58, June 2015.

[11] Y. Geng and G. Cao, "Peer-assisted computation offloading in wireless networks," *IEEE Transactions on Wireless Communications*, 2018.

[12] N. T. Ti and L. B. Le, "Computation offloading leveraging computing resources from edge cloud and mobile peers," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–6.

[13] R. Hasan, M. Hossain, and R. Khan, "Aura: An incentive-driven ad-hoc iot cloud framework for proximal mobile computation offloading," *Future Generation Computer Systems*, 2017.

[14] T. Lynar, F. Jalali, R. R. Kolluri, O. Smith, and F. Suits, "Machine learning assisted heuristic approach for optimal task deployment in hybrid cloud/edge environments," in *AI4IoT - Workshop on AI for Internet of Things, IJCAI*, July 2018.

[15] "Sysbench," https://github.com/akopytov/sysbench, accessed: 2018-08-17.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python ," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[17] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012, vol. 329.

[18] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs," *Journal of Machine Learning Research*, vol. 3, pp. 397–422, 2002.

[19] N. Meinshausen, "Quantile regression forests," *Journal of Machine Learning Research*, vol. 7, no. Jun, pp. 983–999, 2006.