TNSM-2022-05733.R2

Edge/Cloud Infinite-time Horizon Resource Allocation for Distributed Machine Learning and General Tasks

Ippokratis Sartzetakis, Polyzois Soumplis, Panagiotis Pantazopoulos, Konstantinos V. Katsaros, Vasilis Sourlas, Emmanouel Varvarigos

***Abstract*—Edge computing has emerged as a computing paradigm where the application and data processing takes place close to the end devices. It decreases the distances over which data transfers are made, offering reduced delay and fast speed of action for general data processing and store/retrieve jobs. The benefits of edge computing can also be reaped for distributed computation algorithms, where the cloud also plays an assistive role. In this context, an important challenge is to allocate the required resources at both edge and cloud to carry out the processing of data that are generated over a continuous ("infinite") time horizon. This is a complex problem due to the variety of requirements (resource needs, accuracy, delay, etc.) that may be posed by each computation algorithm, as well as the heterogeneous resources' features (e.g., processing, bandwidth). In this work, we develop a solution for serving weakly coupled general distributed algorithms, with emphasis on machine learning algorithms, at the edge and/or the cloud. We present a dual-objective Integer Linear Programming formulation that optimizes monetary cost and computation accuracy. We also introduce efficient heuristics to perform the resource allocation. We examine various distributed ML allocation scenarios using realistic parameters from actual vendors. We quantify trade-offs related to accuracy, performance and cost of edge/cloud bandwidth and processing resources. Our results indicate that among the many parameters of interest, the processing costs seem to play the most important role for the allocation decisions. Finally, we explore interesting interactions between target accuracy, monetary cost and delay.**

***Index Terms*—cloud and edge computing, distributed computing, distributed machine learning, inference, training, resource allocation.**

I. INTRODUCTION

MOBILE phones, intelligent vehicles, energy meters and other Internet of Things (IoT) devices are spreading into many areas of social activity, empowering new digital services. The abundant edge devices offer numerous applications requiring prompt processing. They also generate enormous amounts of data through which useful and actionable analytics can be obtained. The application processing and the transfer of these data all the way to the cloud can be challenging, undesired and unnecessary, due to time and bandwidth limitations. To tackle these developments, new computing paradigms are developed, such as edge computing. A combined edge-cloud processing infrastructure promises to render inexpensive the distributed data processing, including machine learning (ML) training and inference [1]-[5], thus "commoditizing" the related services. The response time of the observe (IoT, monitoring) - decide (algorithm) - act (actuators or reconfiguration) control loop present in many applications will also be accelerated. Edge computing is more appropriate than cloud computing to process time-sensitive data, as it avoids the time needed to relay the information to a centralized datacenter. Thus, the decision-making process is accelerated. The communication load is also reduced as the paths used have fewer hops. Cloud computing is better for processing delay-tolerant data, because of the economies of scale that central datacenters achieve.

In this context, the challenge of resource allocation for distributed computation algorithms over a continuous (infinite) time horizon arises. The processing of a distributed algorithm (the terms algorithm and algorithmic instance will be used interchangeably) can be divided into a number of tasks that are executed in parallel on different equipment. In distributed computation performed over the edge and cloud, an important challenge is to allocate the most appropriate resources to serve the tasks comprising an algorithm with a certain objective (e.g., minimize the monetary cost). This is similar to computation offloading [6], but presents additional complications due to the requirements of distributed computation. First, a decision has to be made on whether an algorithm will be served at the edge or at the cloud. The decision mainly depends on the algorithmic instance's acceptable cost and delay requirements. It also depends on the number of tasks each algorithm consists of, i.e., the degree of parallelization. The (processing, storage and network) capacity parameters of the edge and cloud resources influence the decision. Then, the suitable number and type of

Manuscript submitted October 6 2022; revised April 21 and August 7 2023. This work was supported in part by the Horizon 2020 5G-IANA project (grant agreement: 101016427), and the Horizon 2020 SERRANO project (grant agreement: 101017168).

Ippokratis Sartzetakis, Polyzois Soumplis and Emmanouel Varvarigos are with the School of Electrical and Computer Engineering, National Technical University of Athens, Zografou, Athens 15773, Greece. They are also with the Institute of Communication and Computer Systems, Zografou, Athens 15773, Greece (e-mail: isartz, soumplis, vmanos @mail.ntua.gr). Panagiotis Pantazopoulos, Konstantinos V. Katsaros, Vasilis Sourlas are with the Institute of Communication and Computer Systems (e-mail: ppantaz, k.katsaros, v.sourlas @iccs.gr).

resources should be allocated according to the needs of the algorithmic instance (e.g., amount or streaming rate of data involved, required accuracy, type of algorithm). The problem requires modeling of the various contributors to the (bandwidth, storage and processing) cost of the algorithmic instance, that could be different at the edge and at the cloud. Moreover, there are different types of algorithmic architectures (e.g., different parallelism paradigms, communication architectures, and synchronization requirements). All these factors and parameters have to be taken into account to formulate and solve the resource allocation problem.

In this work Distributed Machine Learning (DML) training or inference is considered explicitly. DML is a special case of distributed computation, and it is expected to constitute a large part of future computations on data. It fits perfectly the model we consider: it involves the processing of data samples that are generated continuously and require resources for prolonged durations of time (equal to the lifetime of the ML application). This model is opposed to the case where computation jobs are generated once and have to be scheduled individually. Distributed ML differs from federated learning, since in the latter the training is typically performed on the user devices themselves. The topic of resource allocation for federated learning has been studied extensively. More specifically, a joint learning, wireless resource allocation, and user selection problem have been formulated specifically for federated learning [7][8]. Moreover, the total energy consumption of the system under a latency constraint can be served as the objective for the resource allocation [8].

In the following we describe some use cases in IoT and in Internet of Vehicles (IoV) that serve as motivation for our work.

A. Internet of Things

In the recent years, IoT has spread across numerous applications and domains. Billions of devices are already connected to the internet, gathering data from their sensors and communicating with other devices. IoT applications range from smartphones, smart homes and smart manufacturing to smart agriculture and drones. IoT devices use a variety of sensors: image (e.g., camera), voice (e.g., microphone), environmental (e.g., temperature of moisture sensor), mechanical (e.g., vibration sensor). Depending on the type of sensor and the application, the data could be generated either very sporadically, or in high volume and streaming fashion. The total IoT data volume of a large smart city that is fed to a given application can be enormous and will tend to be rather smooth due to its aggregated nature.

The data from IoT devices is used in a large range of applications, the majority of which is expected to be ML-based: image recognition (e.g., security related-face recognition, farming related - grapes disease detection), event detection, and other sectors, such as smart electricity grids and healthcare. Also, in industrial IoT, data from a manufacturing environment is used to predict and prevent mechanical failures or to coordinate robot movement. In all these scenarios, processing of the continuously generated data is important in order for the model to be adaptable to the environment and to other changes. Similarly, smartphone data may feed an ML based voice recognition algorithm for digital assistants. Other examples of ML algorithms may use data from accelerometers and gyro, e.g., for activity recognition.

B. Internet of Vehicles

The automotive industry is on the eve of a major transformation fueled by the developments in autonomous driving and the involvement of IT companies. At the same time, Intelligent Transportation Systems (ITS) continuously evolve and apply novel communication protocols to provide safer and more efficient transportation. IoV involves a network of "smart" vehicles that exchange data not only to enhance traffic safety and efficiency (e.g., mitigate traffic) but also to provide commercial infotainment (e.g., in the form of video and game streaming). IoV pose significant processing and communication requirements, that should be supported by the edge and cloud resources.

An autonomous vehicle is equipped with different sensors and cameras, such as Light Detection and Ranging (LIDAR), sonar and high-definition cameras. These sensors produce enormous amounts of data: a vehicle can generate 4 TB of data per day [9], which will put significant stress on the network resources. The transfer of such amount of data to the cloud is almost prohibited. Therefore, edge computing is expected to play an important role in IoV.

The data of autonomous vehicles can be leveraged in many distributed learning scenarios. In particular, distributed learning vehicle routing algorithms can adjust vehicle routing in real time and reduce traffic congestion [10]. Another example is object detection and classification based on acquired charge-coupled device (CCD) and LIDAR data [11]. There have been several instances where an autonomous vehicle has misinterpreted an object in the environment. Continuous training is important for object detection, given the variety and dynamicity of the environments autonomous vehicles will navigate in when they become widely available. And fast response times can in many cases be delivered only through edge computing.

In this paper we investigate the resource allocation problem for general distributed algorithms and ML applications that process continuously generated data. We develop a general framework, where the different ML and other algorithmic instances that run on the infrastructure are modelled by a vector of requirements that depend on the algorithm. More specifically, we consider a scenario where various devices are located close to the edge of the network. The devices produce data continuously, over an infinite time horizon, that are processed by an algorithm that runs in edge and/or cloud resources. The goal is to assign the required resources (processing, memory, storage, bandwidth) for each algorithm while optimizing certain metrics. We consider different scenarios related to edge/cloud costs, delay and accuracy. We introduce an Integer Linear Programming (ILP) algorithm that solves the resource allocation problem. We also investigate low complexity heuristic resource allocation algorithms. Finally, we compare the results, and extract interesting insights on the

allocation decisions.

The main contributions of our work are:

1) We present a comprehensive solution that can serve distributed algorithms processing continuously generated data. The resulting infinite time horizon model represents an IoT infrastructure (e.g., a smart city) continuously generating data (sequence of samples), according to a deterministic or random process of a certain average rate. These data have to be allocated adequate processing/ storage/bandwidth resources so that the system is stable and the delays are kept under control. This model is in contrast to the one-time problem usually considered, where there is a finite number of tasks that have to be scheduled on certain resources, once and for all. We appropriately design resource allocation schemes that account for the different types (e.g., GPU models, RAMs) of edge and cloud resources, their computational power, their bandwidth, storage and processing monetary costs, their communication delay and the achievable accuracy. The joint consideration of these optimization parameters provides a realistic modeling of the problem, while allowing for interesting insights on the potential benefits and trade-offs involved.
2) We present an ILP algorithm that solves the edge/cloud joint resource allocation problem. The objectives are to minimize the resources' monetary cost (subject to delay requirements) and to maximize the resulting accuracy of the algorithmic instance. A weight parameter is used to control the relative importance of the two objectives. The formulation is versatile and can be used to allocate resources for different variants of distributed applications, including ML training (e.g., all-reduce or aggregation servers) and inference. We also develop efficient heuristic algorithms that can solve faster large instances of the problem.
3) We perform realistic simulation-based experiments for a distributed ML training scenario. We quantify the trade-offs between edge and cloud resources for various accuracy options, bandwidth and processing costs of the edge vs cloud, and the corresponding delays. Finally, we compare the results of the different algorithms and allocation schemes.

Our model assumes the continuous (over an infinite time horizon) generation of data (samples) that are processed by a general distributed (namely ML) application. The application characteristics are captured through a resource requirements vector of parameters, so that the formulation is kept as general as possible. We also use accuracy as a parameter that can be traded off to save resources. This leads to the concept of *approximation as a resource*, in the same way that Demand Response is used as a resource in smart energy grids, when there is a mismatch between demand and supply of energy [12].

The rest of the paper is organized as follows. In Section II we present the related work. In Section III we describe the network scenario considered and the related assumptions. We also introduce the resource allocation problem with formal notation and present an ILP algorithm to solve it. Afterwards, we present the heuristic algorithms. In Section IV we evaluate the proposed framework, present the performance results obtained and explore the various trade-offs. Finally, Section V summarizes the paper and discusses future work.

II. Related Work

Our work mainly relates to two topics: distributed computation (as in DML) of continuously generated data and computation offloading. Distributed computation is not a recent subject [13]. Over the last few years, it regained attention. It has been applied in the context of machine learning [14][15], communication networks [16], power systems [17] and primal-dual algorithms [18], among others. In general, research on distributed computation can refer to several topics, such as the design of suitable algorithms, machines and programming languages. Related issues include the partition of a workload to smaller tasks, the communication of the tasks' results, the synchronization of the computations and the allocation of the suitable resources to perform the computations. The specific characteristics of each application have to be taken into account in order to design a robust resource allocation framework that can be applied to each case.

Distributed ML *training* is an active research topic and a large number of associated methods have been investigated. There are three main taxonomies of distributed ML training [14][19] based on: i) the type of parallelism, ii) the communication architecture, and iii) the computation timing. As far as parallelism is concerned, there is *model parallelism* and *data parallelism*. In model parallelism, the ML model is divided into a certain number of segments (tasks) that are executed in a respective number of workers. Each worker node runs different code (in parallel computation this model is also called MIMD – multiple instruction, multiple data). In data parallelism, the model is common to all workers, but the training data are different (in parallel computation this model is called SIMD – single instruction, multiple data). Each worker (task) computes locally its model weights (parameters) and communicates its values to the rest of the workers that aggregate the results and update the common model. Regarding the communication architecture in distributed training, perhaps the most prominent variant is the *parameter (aggregation) server* [20]. In this case, the workers communicate their local computations to one or more centralized server(s), the parameter server. The server aggregates the weights of the workers and returns the results to the workers to initiate the next training round(s). A different communication architecture, known as *all-reduc*e, does not use any centralized server. Instead, the workers communicate directly with each other, in a peer-to-peer manner, to share the model weights.

Concerning computation timing, there are two main approaches: synchronous and asynchronous training. In *synchronous* training, the aggregation of the workers' model weights is performed synchronously: they proceed to the next execution round only when the previous round and the exchange of the new weights has been completed for all the workers. This can incur certain inefficiencies, commonly known as synchronization penalties, when some workers (stragglers) are progressing slower than others. In asynchronous training, the workers are allowed to perform at their own pace

(so they may be at a different round), thus eliminating the synchronization overhead. When a worker finishes a computation round, the parameters are updated. The rest of the workers will fetch the updated parameters asynchronously [13]. There are certain conditions under which synchronous convergence also implies asynchronous convergence [13]. Finally, a pipelined architecture can be considered, to improve the training throughput [21]. It allows the overlapping of the communication with the computation time, while also reducing the amount of communication required. On a separate issue, distributed ML inference [3][4] typically has less processing requirements than training. However, the workload can still be significant for a user device. Also, the timing requirements are often very stringent; e.g., an autonomous vehicle requesting an image classification task. A large number of works have researched the offloading of inference ML jobs using various strategies.

Computation offloading initially referred to moving compute intensive tasks at the cloud where powerful and relatively abundant resources are available. As technology evolved, new applications required low latency and high bandwidth, which could not be satisfied by the cloud. As a result, Mobile Edge Computing or Multi-access Edge Computing (MEC) emerged. Even though the research on computation offloading is vast [6][23], it cannot directly be applied to our ML case study. The reason is the specific resource allocation requirements of distributed computation algorithms that that are generated continuously and can vary depending on the type of algorithm, the accuracy, the time constraints, and the architecture.

TABLE I
IMPORTANT NOTATIONS

Symbol	Description
J	Set of all algorithmic instances on infrastructure
$\boldsymbol{j_x}, \boldsymbol{j_i}$	An ML training or inference algorithm
$\boldsymbol{T_j}$	Set of all tasks of algorithm j
$\boldsymbol{t_{jk}}$	A task of algorithm j
$\boldsymbol{y_{jk}}$	Set of devices feeding data to the k^{th} task of algorithm j^{th}
$\boldsymbol{\lambda_y}$	The data (samples) generation rate of device y
N	The set of nodes of the edge network
n	A node of the edge network
$\boldsymbol{P_o}$	Period every which tasks are completed
$\boldsymbol{P_{comm}}$	Communication time required at end of period to transfer the ML model weights
$\boldsymbol{P_{comp}}$	Computation time within a period
$\boldsymbol{S_{jk}}$	Total samples that a task has to process within period $\boldsymbol{P_o}$
Q	The set of all the available GPU models
q	A specific GPU model
$\boldsymbol{H_j}$	Required number of training epochs for ML algorithm j
$\boldsymbol{C_E^q}, \boldsymbol{C_C^q}$	The cost for an ML job to use model q processing edge or cloud resources
$\boldsymbol{C_E^w}, \boldsymbol{C_C^w}$	The b/w cost of edge or cloud
$\boldsymbol{\Delta_E}, \boldsymbol{\Delta_C}$	The propagation delay of a job if served to the edge or cloud
$\boldsymbol{\delta_j}$	The maximum acceptable propagation delay of a job
$\boldsymbol{A}$	The set of possible target accuracies of the jobs
$\boldsymbol{a_j}$	An accuracy of a job ranging from 0 to 1
$\boldsymbol{\alpha_j^{min}}$	The minimum required accuracy of job j
$\boldsymbol{R^j} = [\boldsymbol{G^{jka}}, \boldsymbol{M^{jka}}, \boldsymbol{V^{jka}}, \boldsymbol{B^{jka}}, \boldsymbol{\Theta^{jka}}]$	Vector of required GPU, Memory, Storage, B/w resources for *each sample* of job j
$\boldsymbol{R_n} = [\boldsymbol{R_n^{Gq}}, \boldsymbol{R_n^M}, \boldsymbol{R_n^V}, \boldsymbol{R_n^B}, \boldsymbol{R_n^\Theta}]$	GPU, memory, storage, incoming b/w, aggregator resources units in node n
$\boldsymbol{\xi_n^{jkqa}}, \boldsymbol{\xi_C^{jkqa}}$	Binary ILP variable equal to 1 if task $\boldsymbol{t_{jk}}$ uses resource units and GPU model q with accuracy a at edge node n, or it is served at the cloud
W	Weight to control the importance of cost minimization vs accuracy maximization

A recent research topic is the intersection of distributed ML and computation offloading, which is an important subcase of this work as well. In federated learning over wireless networks, a previously studied challenge is to allocate the resources by considering the wireless channel characteristics and the convergence rate of the federated learning algorithm [7][8]. In our work, we consider distributed computations and distributed ML and the modeling is agnostic to the physical layer characteristics. We focus on the trade-offs related to accuracy, delay and the processing/bandwidth costs. Regarding distributed ML training at the edge, the network resources can be efficiently utilized by analyzing the convergence rate of the distributed gradient descent algorithm [24]. The ML training of data from augmented reality edge devices has also been considered [25]. Due to the limited computing power of these devices, backend "helpers" at the edge or cloud can be leveraged. Moreover, the training model can be incrementally offloaded at the edge devices [26]. This strategy accelerates the training since the edge servers are used in a timely manner. A more specialized topic is the offloading of IoT deep learning applications in an edge computing environment [29]. Regarding the performance of training, there are certain differences of federated learning compared to variants of edge and centralized learning [27]. Also, there are certain approaches, such as the joint data collection and resource allocation, to maximize the distributed learning throughput [28]. The problem can be formulated as a mixed-integer non-linear program and an approximation algorithm can be used. Finally, another topic is the job scheduling problem for distributed ML. For example, a scheduling algorithm can be employed to decide the execution time window and the number and type of workers and parameter servers aiming to minimize the weighted average completion time [30]. It is a problem similar to ours, but also differs as it is mainly a one-time scheduling problem. In contrast, we assume continuous generation of data.

To the best of our knowledge there is no previous work that combines realistic modeling of the data collection and resource allocation problems of distributed computation / distributed ML applied on continuously generated data, using both edge and cloud resources and accounting for the different architectures. Moreover, an analytic comparison of the various trade-offs between accuracy, delay, bandwidth/processing costs of the edge and cloud seems to be missing from related work. In this paper, we attempt to tackle these important issues. We should note that this paper is an extension of a previous work [31]. We significantly expanded the work by: i) adding more details on the background of the problem, ii) examining the allocation of

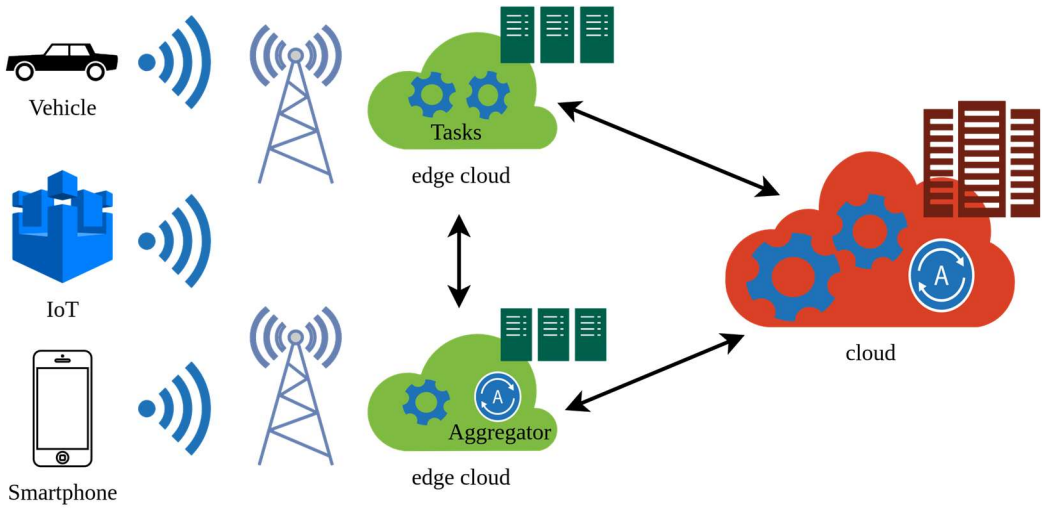

Fig. 1 The abstract architecture considered

inference jobs in addition to training, iii) adding accuracy at the objective and studying trade-offs between accuracy and resource requirements, iv) adding low complexity heuristic algorithms for the resource allocation problem, and v) expanding the simulation experiments to include additional scenarios.

III. Problem Statement

We consider various distributed processing and ML scenarios involving IoT or IoV data generating devices (Fig. 1). Each scenario may correspond to a different processing algorithm, e.g., image recognition, anomaly or event detection, etc. There is an edge network close to the devices, and a more distant cloud. Computation jobs are completely offloaded to the network's resources, and are not processed at all on the devices. In the future, we plan to extend the problem statement to include the possibility of (partial) execution of a job on user devices.

We will now formally define the edge-cloud resource allocation problem for the aforementioned distributed processing scenarios. The formulation is generic in that it can be used to model in a unified way any of the aforementioned scenarios and many different distributed architectures. We consider a number of user devices near the edge that continuously produce data, over a prolonged ("infinite") time horizon. These data are fed for processing to the network's edge or cloud resources. Table I contains all the important notations introduced in the following.

Each device y continuously produces data at an average rate of λ_y samples/sec. Depending on the application, the sample may be a number, a picture, a voice or video segment, etc. Each algorithmic instance j (e.g., a specific ML application) receives data (i.e., samples) generated at a set of devices Y_j that generate a total data set that is denoted by D_j. An ML algorithm can be further characterized as j_x for training and j_i for inference. The set J contains all the algorithmic instances to be supported by a given infrastructure. The processing of algorithm j is divided into a set of distributed tasks $T_j = \{t_{j1}, t_{j2}, \dots, t_{jK}\}$ (or even one task, in case of, e.g., lightweight inference) that are executed in parallel at respective workers. Each task t_{jk} is responsible for processing a subset d_{jk} of the entire dataset D_j of algorithm j. In particular, task t_{jk} processes the data samples in d_{jk} that are generated by a subset of devices y_{jk} belonging to the set of devices Y_j feeding algorithm j. Thus, a subset of devices and their generated data form a (sub)task of an algorithm. In the case of a training ML task t_{jk}, the loss function that the task aims to minimize on its respective dataset is of the form:

$$f_{jk}(w) = \frac{1}{|d_{jk}|} \sum_{l \in d_{jk}} f_l(w) \quad (1)$$

where f is a per-sample loss function, $|\cdot|$ denotes the size of a set, w is the model parameter vector and l a training data sample. The overall training of ML algorithm j takes into account all of its tasks, and aims to optimize:

$$\min_w F_j(w) = \frac{\sum_k d_{jk} f_{jk}(w)}{|D_j|} \quad (2)$$

over the entire data set D_j. The idea in Eq. (2) is that all the parallel tasks t_{jk} in which algorithmic instance j has been decomposed contribute to its loss function proportionally to their respective data sizes (i.e., with weight $d_{jk}/|D_j|$).

We assume that the worker nodes running algorithm J exchange intermediate results (the weights, in the case of an ML algorithm) every P_o *seconds*, where the symbol o represents classes of algorithms with different time scales. For example, we could have three different classes ($o \in \{1,2,3\}$), one for training requiring extensive time that is not time critical, one for time-critical inference, and one for generic inference. Therefore, the time axis can be viewed as divided in time periods of duration P_o (Fig. 2) during which a certain number of samples are processed at the resource (worker) where they are gathered. Note that, for stability, the average number of samples that are processed at a resource during a period should

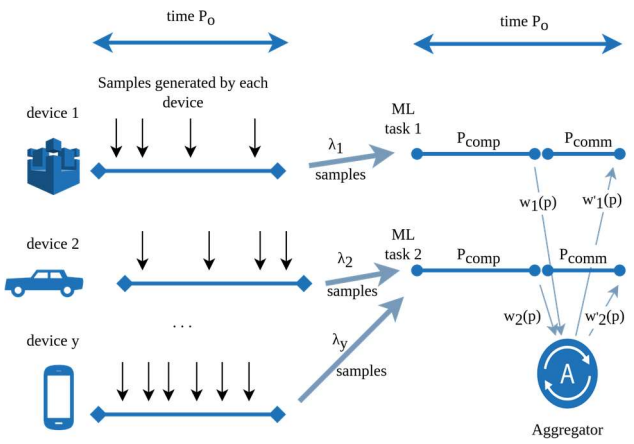

Fig. 2 An example of a distributed ML algorithm processing samples at various worker nodes, who exchange model parameters through an aggregator server. Device y sends either continuously at rate λ_y or in batches of average size $P_o\lambda_y$ every period P_o data for processing at a worker node assigned to it.

be equal to the average number of samples generated within a (previous) interval of duration P_o at the corresponding devices that send their samples to that resource. The periods are asynchronously defined for each resource, in the sense that their start (and finish) times are not generally aligned. The devices feed an algorithm either continuously (streaming) or in batches (at the end of a period) by uploading their data to the appropriate processing, storage, memory and network resources. During a period, a resource unit performs computations by processing a sample upon its arrival or a batch of data received at the previous period (i.e., until the current period begins). At the end of a period the worker sends the new values of its parameters w to an aggregation node or to the rest of the workers, depending on the architecture. The parameters are updated by minimizing the overall loss function, e.g., given by Eq. (2), and the new values of the parameter vector w are communicated to the worker nodes. Note that in the case of synchronous training, the aggregation server has to wait until it has received all the new set of parameters, before starting a new period. In asynchronous training, the weights of the parameters can be incorporated asynchronously as they arrive. Therefore, in synchronous training there is a notion of an iteration (period) that all worker nodes operate in. In asynchronous training the worker nodes may be at a different iteration, or use different period durations. In the synchronous case, the worker nodes have to send updates every *fixed* (or upper-bounded) amount of time. In the asynchronous case they do so every some amount of time *on average*. Also note that the number of a DML model parameters can be pruned [32], if desired, to reduce model size and the related communication and computation requirements.

A. Communication and Computation phase

The total time to complete the processing of a batch is

$$P_o = P_o^{pull} + P_o^{comp} + P_o^{push} + P_o^{upd} \quad (3)$$

where P_o^{pull} is the time required for a worker to pull the parameters that other workers have computed, P_o^{comp} the total computation time required to finish the training, P_o^{push} the time required for a worker to push the parameters it has computed and P_o^{upd} the time required to update the parameters to the current values. In the case of asynchronous iterations, the terms in Eq. (3) should be interpreted as mean values. P_o^{upd} is roughly proportional to the size of parameters that have to be updated, over the computational power of the resource that performs the update (aggregation). It is generally of small importance compared to the other variables and we will ignore it for the rest of the paper. Depending on the architecture of each algorithm, there could be different definitions and relationships between the timings. We denote the aggregated roundtrip communication time by P_o^{comm}, with

$$P_o^{comm} = P_o^{pull} + P_o^{push} \quad (4)$$

The data involved in both pulling and pushing are largely the same in type and quantity. Therefore, the two timings are considered equal, yielding

$$P_o^{comm} = 2\left(\frac{\mu}{\beta} + \delta\right) \quad (5)$$

where μ the size (in bits) of the model parameters, β the available bandwidth (in bits per second – bps) and δ the propagation delay (in sec). The computation time is equal to:

$$P_o^{comp} = \frac{sH}{p} \quad (6)$$

where s is the number of samples that have to be processed, H is the number of times a sample is processed (number of epochs in the case of DML), and p is the processing power in samples processed per sec. In the evaluation section we will present a more detailed version of the above equation.

Depending on the architecture of the distributed algorithm, there could be different relationships between the durations P_o^{comm} and P_o^{comp}. Assuming that the workers first complete a computation round and then send the model weights for aggregation/averaging (i.e., no overlapping of communication and computation times), we have:

$$P_o = P_o^{comp} + P_o^{comm} \quad (7)$$

Thus, $P_o^{comm} = \omega P_o^{comp}$, where $\omega = \frac{2p\left(\frac{\mu}{\beta}+\delta\right)}{sH}$.

The scalar ω represents the percentage communication overhead. It depends on various parameters, including the specific model, the communication architecture (e.g., all reduce or aggregation server), and we expect ω <1 in most if not all cases of interest. For example, in very large-scale distributed ML, the communication time can take nearly half of the time of training [22]. In the case of DML, if we assume pipelining the communication overhead can be reduced up to 95%, and a perfect overlap of computation and communication can be achieved [21], where $P_o \approx P_o^{comp}$. In the case of inference, the communication overhead is either zero (i.e., inference executed in just one resource unit) or generally much lower than that for the case of training (there is no communication of weights over many epochs like in training). Moreover, the efficient layer partitioning (model parallelism) of a Deep Neural Network results in low communication overhead [3].

At each time period P_o, device y produces and sends for training $s_y = P_o\lambda_y$ samples. Each ML task t_{jk} has to process a total number of samples equal to

$$s_{jk} = \sum_{y \in Y_{jk}} P_o \lambda_y = P_o \lambda_{jk} \quad (8)$$

(where $\lambda_{jk} := \sum_{y \in Y_{jk}} \lambda_y$) within time P_o for the system to be stable, as we assume continuous arrivals over an infinite time horizon. The computation power assigned to a worker node should be enough to process the received samples (for a given number of epochs H_j of an ML *training* algorithm) within time P_o. Otherwise, the work generated would be more than the work that can be completed within a period and the system would be unstable. The required number of epochs H_j depends on the type of the ML algorithm *j* (e.g., layers of a Neural Network), and the desired convergence accuracy (as determined by prior profiling-experimentation). In the simulation section we provide more details about how the number of epochs affects the processing requirements and its impact on the whole monetary processing and bandwidth costs and trade-offs.

The communication resources allocated for transferring the data from a device to the worker node assigned to it, should also be sufficient. The communication resources needed between the worker nodes and the aggregator are generally small. These parameters are of small size (numbers) when compared to that of the data samples (e.g., images). Therefore, the bandwidth requirements for transferring the weights *w* of the model can be ignored in the resource allocation problem and the main communication parameter of interest is the propagation delay (from the device to the edge or to the cloud).

B. The Vector of Resource Requirements

In order to perform the updates, each task needs certain computation, memory, storage and network resources during each time period. In particular, our assumption (valid in the practical use cases mentioned in Section I) is that each task t_{jk} running on a worker node has processing (CPU or GPU based), memory, storage, ingress bandwidth requirements that are *roughly proportional* to the number S_{jk} of samples it receives and processes. If the communication architecture is a variant of parameter server, the related resources include a number of aggregators that also have to be allocated. Note also that this proportionality assumption accounts for both model and data parallelism. Moreover, the degree of parallelization of each algorithm, is captured by the number of different tasks within each algorithm. These are also notable differences of our work compared to pre-existing offloading algorithms for generic single tasks. The requirements are thus described by a *vector of resource requirement proportionality coefficients*

$$R^{jka} = [G^{jka}, M^{jka}, V^{jka}, B^{jka}, \Theta^{jka}]$$

where *G*, *M*, *V*, *B*, and Θ are parameters that reflect the amount of processing (G^{jka} in e.g. Floating Point Operations – FLOP), memory (M^{jka}, in bytes), storage (V^{jka}, in bytes), number of bits (B^{jka}) communicated to the nodes, and processing (Θ^{jka} in FLOP) for weight aggregating purposes that *each sample* requires for the specific ML task t_{jk} of algorithm *j*, and for a specific accuracy level α. The rationale for introducing the resource requirements vector R^{jka} is that each sample (e.g., jpeg image) in the task requires some specific processing in order to be handled, some specific memory and storage, and it has a given size to be communicated, for a given accuracy α. It is worth noting that we consider here accuracy as a kind of resource that can be tapped to reduce processing and bandwidth requirements ("approximation as a resource"), with higher accuracy typically requiring more resources. So, depending on the availability or not of the physical resources, the accuracy can be adjusted accordingly. The accuracy of an ML algorithm depends on many factors, some of which can controlled/ adjusted, such as the number of epochs *H* used, weight regularization, depth of the Neural Network, and mini-batch size. For example, accuracy can be improved for small mini-batch sizes. However, this increases the training time as more frequent calls for expensive multiplications are required [33]. The entries of vector R^{jka} depend on accuracy α in different ways: e.g., if accuracy in a Neural Network algorithm *j* is controlled through the number of epochs H_j used (i.e., the number of times each sample is used in computations), then G^{jka} will depend linearly on H_j, while the other parameters of R^{jka} will be mostly independent of H_j. If accuracy is controlled through the number of precision bits used in encoding a sample, then the dependence of B^{jka} on it is linear, while the impact on the other vector parameters will be different. We assume that the dependence of the requirements vector R^{jka} on the desired accuracy level α is a known function of α. In practice, the dependence is complicated and not fully known. Therefore, we consider a finite and coarse granularity of accuracy levels. For example, we could have a finite set of options A={α_{good}, α_{medium}, α_{low}}, in which case we assume we know R^{jka} for $\alpha \in A$. The exact resource requirements for each task t_{jk} and accuracy level α can be determined by profiling or through ML benchmarks, such as MLPERF [34]. Certain strategies can be employed to determine the requirements for a specific set of parameters such as the number of epochs or batch sizes (that affect the accuracy) for a certain application and execution environment [35]. The process involves modified two inputs of the algorithm, and monitoring their impact on accuracy. For example, for a given set of parameters (e.g., three different batch sizes or number of epochs) we can increase the number of GPUs for a given task and keep all the other parameters the same. Then we monitor the resulting accuracy as a function of the processing power for the different model parameters. Additionally, we can scale the batch sizes without significantly reducing the accuracy. Depending on the number of data samples, we can use different functions of scaling (e.g., linear, cubic) based on the number of GPUs and monitor the resulting accuracy. A small resource overprovisioning can be used to ensure that the allocated resources will always be adequate for a given accuracy.

In our modeling, it was natural to assume that the amount of the different type of required resources proportionally depends on the number (or rate) of the samples. The proportionality constants that convert samples to requirements are given by the entries in R^{jka}. Of course, the R^{jka} values depend on the type of algorithm *j* (e.g., parameters *G, M*, *V*, *B* will be different for DML image or voice recognition), on the compression techniques used to encode an image or voice sample, etc. More specifically, a task t_{jk} has processing workload $G^{jka} S_{jk}$

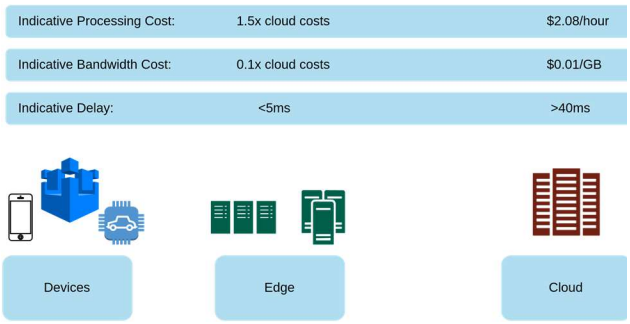

Fig. 3 Indicative values of edge and cloud delay, processing and bandwidth costs

(measured, e.g., in number of FLOP) that has to be executed within a time period P_o. Assuming perfect pipelining of computation and communication, the task requires processing rate $G^{jka}S_{jk}/P_o = G^{jka}\sum_{y\in Y_{jk}}\lambda_y$ in, e.g., FLOP per sec, or samples processed per sec. If pipelining is not used, the required processing rate should be increased by $1+\omega$, to account for the communication overhead (the same work has to be completed within time $P_0/(1+\omega)$). Similarly, a task requires memory $M^{jka}S_{jk}$, storage $V^{jka}S_{jk}$ (as the samples have to be stored for the entire duration P_o) and aggregation processing $\Theta^{jka}S_{jk}$. The required ingress bandwidth constant of proportionality B^{jka} can be set equal to the number of bits needed to represent a sample (representing a measurement, an image, a sentence, etc.). The total ingress data required for the worker node running task t_{jk} will then be $B^{jka}S_{jk}$, and the ingress rate $B^{jka}S_{jk}/P_o$. When assigning tasks to resources, we will use this total ingress bandwidth requirement as an allocation criterion (ignoring the number of links on the path connecting the worker with the specific devices that feed data to it).

C. Edge-Cloud Resource Infrastructure Model

The infrastructure on which the tasks will be executed, consists of an edge and a cloud network. The edge network includes a set of nodes N that can be used by the tasks. More specifically, each edge node n has finite R_n^{Gq} processing (GPU or CPU) model q capacity (e.g., in FLOP per sec or samples/sec), R_n^M memory, R_n^V storage, R_n^B incoming b/w to receive the data from the devices and R_n^Θ aggregator processing capacity. Obviously, the sum of the required resources of all the tasks that are assigned to a node should not exceed the node's capacity. The cloud network, on the other hand, is assumed to have infinite resources. One difference between the edge and the cloud are the respective monetary processing and bandwidth costs. The cost to use a model q processing unit is defined as C_E^q at the edge and C_C^q at the cloud. The cost of ingress b/w is defined as C_E^{bw} at the edge and C_C^{bw} at the cloud. Finally, another differentiator between the edge and the cloud is propagation delay. Since the edge network is much closer to the devices, the propagation delay to the edge, denoted by Δ_E, is expected to be significantly lower than the respective delay to the cloud, denoted by Δ_C. Certain algorithms j (e.g., ML inference jobs) may have stringent constraints on the maximum acceptable propagation delay, denoted by δ_j ($\delta_j = \infty$ for delay insensitive jobs). This should be accounted for by the resource allocation algorithm. The propagation delay is less important for DML training scenarios, where computation requirements and processing times are generally larger. Nevertheless, a similar constraint can be introduced in the formulation in a straightforward way for the training scenarios as well. Typical values for the delay and the monetary costs of the edge and the cloud [36][37][38][39] are given in Fig. 3. More details for these values are provided in Section IV.

The goal of the resource allocation algorithm is to reserve the appropriate number of resources for the tasks (including the specific edge node where each task will be processed), while minimizing certain objectives and satisfying all the constraints.

IV. Resource Allocation Algorithms

A. ILP Algorithm

In this subsection we present an ILP algorithm for assigning edge and cloud resources to the tasks. The algorithm receives certain inputs, and using some related constraints aims to allocate the network's resources (the variables of the algorithm), while satisfying the objective. The formulation assumes one aggregation (parameter) server for each ML training job, but can be modified in a straightforward way to account for other algorithmic instances, e.g., DML with multiple aggregation servers or for all-reduce architectures. The algorithm provides a solution that remains valid for as long as the input parameters are valid. Whenever the parameters change significantly (e.g., the generation rates λ_y in samples/sec for device y) in a way that renders current resources inadequate to finish the tasks on time, the algorithm is re-executed to yield a new solution. During this re-execution of the algorithm (either of the ILP or of the heuristic to be given in this or following subsection, respectively), one may opt to treat the variables indicating the allocation of the other tasks as fixed, so that the other tasks are unaffected. Another possibility is to penalize the difference between the current and the new solution so that these differences are minimized.

The objective of the resource allocation problem is to minimize the total cost to serve the jobs and to maximize their accuracy, subject to delay constraints and also constraints on the available edge resources. Note that the two individual objectives are contradictory. To maximize accuracy, additional resources are required to finish the computations within the training period P_o^{comp}, also increasing the cost. We employ a weight W to control the importance of each individual objective (cost, accuracy) in the objective function of the ILP algorithm.

Inputs:
$N, R_n^{Gq}, R_n^M, R_n^V, R_n^B, R_n^\Theta, J, T_j, Q, C_E^q, C_E^{bw}, C_C^q, C_C^{bw}, \delta_C, \Delta_j, W, A, A_j, \lambda_j$
Variables:
$\xi_n^{jkqa}, \xi_C^{jkqa}$

The symbolism in binary variable ξ_n^{jkqa} means that there is a different variable ξ for every different node n, for every different algorithmic job j, for every different task k, for every different GPU model q, and for every different accuracy α.

Objective:

The objective is to **minimize** the total cost for serving the jobs and to **maximize** the accuracy. The relative importance of each individual objective is controlled by a weight W. The cost of each job depends on the amount of b/w, the model of GPU and whether it is served at the edge or the cloud:

$$min\left(\begin{array}{c}(1-W)\sum_j\sum_q\sum_a\sum_{T_j}\left(\begin{array}{l}\sum_n \xi_n^{jkqa}\lambda_{\mathrm{jk}}\left(C_E^{bw}B^{jka}+C_E^{q}G^{jkqa}\right)\\ +\xi_c^{jkqa}\lambda_{\mathrm{jk}}\left(C_C^{bw}B^{jka}+C_C^{q}G^{jkqa}\right)\end{array}\right)\\ -W\sum_j\sum_{T_j}\left(\begin{array}{l}\sum_n \xi_n^{jkqa}a_j\\ +\xi_c^{jkqa}a_j\end{array}\right)\end{array}\right) \quad (9)$$

Subject to the following constraints:

- Each task of a job should be served once with one accuracy option, at the edge or at the cloud and each task should use only one model of GPU:

$$\forall j,\forall t_{jk}: \sum_{n\in N}\sum_q\sum_a \xi_n^{jkqa}+\sum_q\sum_a \xi_c^{jkqa}=1 \quad (10)$$

- Each edge node should have enough (#GPUs, memory, storage, bandwidth, aggregator) capacity to serve the assigned tasks. So, for all nodes we sum all the resources that a job uses (determined by variables ξ_n^{jkqa}), and this sum should be less than the capacity of that node:

$$\begin{aligned}\forall n\in N: \sum_j\sum_{t_{jk}}\sum_q\sum_a \xi_n^{jkqa}G^{jka}\lambda_{\mathrm{jk}} &\le R_n^{Gq}\\ \forall n\in N: \sum_j\sum_{t_{jk}}\sum_q\sum_a \xi_n^{jkqa}M^{jka} &\le R_n^{M}\\ \forall n\in N: \sum_j\sum_{t_{jk}}\sum_q\sum_a \xi_n^{jkqa}V^{jka}\lambda_{\mathrm{jk}} &\le R_n^{V}\\ \forall n\in N: \sum_j\sum_{t_{jk}}\sum_q\sum_a \xi_n^{jkqa}B^{jka}\lambda_{\mathrm{jk}} &\le R_n^{B}\\ \forall n\in N: \sum_{jx}\sum_{t_{jk}}\sum_q\sum_a \xi_n^{jkqa} &\le R_n^{\Theta}\end{aligned} \quad (11)$$

- In order for a delay sensitive (e.g., inference) job to be served to the cloud, its maximum acceptable delay should be respected:

$$\forall j_i,\forall t_{jk}: \sum_q \xi_c^{jkq}\,\delta_j \le \Delta_C \quad (12)$$

- The minimum required accuracy of the related jobs should be respected:

$$\forall j_x,\forall t_{jk}: \sum_{n\in N}\sum_q\sum_a \xi_n^{jkqa}a_j+\sum_q\sum_a \xi_c^{jkqa}a_j \ge \alpha_j^{min} \quad (13)$$

In Eq. (9), the objective is to obtain a low total cost of serving all the jobs (first part) with high accuracy (second part). The first part of the right-hand side of the equation refers to the cost of a job if it is served at an edge node n, while the second part corresponds to the cost of a job served at the cloud. The edge (or cloud) cost consists of the b/w $B^{jka}\lambda_{\mathrm{jk}}$ required by each task, times the per unit cost of b/w at the edge C_E^{bw} (or at the cloud C_C^{bw}), plus the model q processing units $G^{jkqa}\lambda_{jk}$ of each task, times the cost of each processing unit C_E^q at the edge (or C_C^q at the cloud respectively). In the second part of Eq. (9) we subtract (thus maximize) the (weighted) accuracy of the tasks that are served at the edge and at the cloud. Equation (10) ensures that all tasks of all jobs will be served at the edge or at the cloud. The set of equations in (11) constrains the sum of resources (processing, memory, storage, b/w and aggregators in case of training jobs) used by the tasks at an edge node, to be less or equal than the respective capacity of that edge node. In the case of all-reduce or other architecture, the last equation is removed. Equation (12) guarantees that if a delay-sensitive job is served at the cloud, then the maximum acceptable delay of the job is less than the delay of the cloud. Finally, Eq. (13) ensures that if a job has a minimum required accuracy, this will be respected.

The solution of the algorithm consists of the binary values of all ξ_n^{jkqa}, ξ_C^{jkqa} variables. So, if for $n=1, j=1, t=1, q=1, a=1$, the related ξ_1^{1111} is equal to 1, this means that at the first node of the edge the first task of the first job is served, using the first option of the available GPU models, and the first option of the available accuracy options. To infer related statistics such as the percentage use of a node's resources or the mean accuracy of the tasks, we can sum over the set of decision variables in the following way. For a specific node n we sum all ξ variables for all the jobs, tasks, etc., each multiplied by the related resources that the specific task requires. The result is the resources of node n that are occupied by tasks assigned to it. To calculate the mean accuracy of the tasks, we sum all ξ_n^{jkqa} and ξ_C^{jkqa}, again for all possible combinations of the parameters, with the ξ values being weighted by the % accuracy that accuracy options a correspond to, and dividing by the number of tasks, to obtain the mean accuracy achieved by the tasks. Finally, to calculate the total monetary cost of all the tasks, we again sum over all possible ξ values, and we employ the first part of Eq. (9) that corresponds to the monetary cost.

The ILP algorithm can provide an optimal solution for the resource allocation problem. However, in certain instances the complexity of the problem may result in unacceptable running times. For example, in the case where we have a large number of edge nodes, jobs, tasks, GPU models and accuracy options, the number of binary variables will be large. This will result in a substantial amount of time to create the constraint equations, prepare the ILP solver and find a solution. In these cases, an efficient heuristic algorithm can provide faster a solution that, under certain circumstances, can also be near-optimal.

B. Heuristic algorithms

We considered two kinds of heuristic algorithms: a greedy one, and one based on simulated annealing [40]. The greedy algorithm is described in the respective listing. Its input is

Greedy resource allocation algorithm
Inputs: $T_j, Q, A, N, R^{jk}, R_n^{Gq}, R_n^M, R_n^V, R_n^B, R_n^\Theta$
Output: The allocation of all jobs at the edge/cloud, respective metrics such as cost, usage statistics etc.
Procedure:

```
Allocate tasks with strict constraints (accuracy, delay)
Find min task resource requirements (minR) among all tasks
for task t_jk
    success=0
    for a_gpu_model ∈ Q
        for an_accuracy ∈ A
            Calcul cost (cost+acc) to serve at cloud or edge
            Select least expensive option (location, gpu, accuracy)
        end for
    end for
    if location==edge
        for node n∈ N
            if R_n^x <R^jk
                Allocate t_jk to n
                Update the current resources of node n
                success = 1
                break
            end if
            if R_n^x < minR
                Remove n from N
            end if
        end for
    end if
    if success == 0
        Allocate t_jk to cloud
    end if
end for
```

similar to the ILP algorithm. The difference is that the greedy version receives as input the order in which the tasks will be allocated. The algorithm first pre-calculates the minimum amount of resources required by a task; this will be used at later steps of the algorithm. Then the objective cost to allocate each task at either the edge or the cloud is pre-calculated. Afterwards, for the given ordering of the jobs, the algorithm tries sequentially to assign each task to the appropriate location. If the location with the least cost is the cloud, then it simply assigns it there, since the cloud has infinite resources. If the location with the least cost is the edge, then it starts searching the edge nodes iteratively, until it finds a node with adequate resources. If such a node is found, the task is allocated, and the node's available resources are decreased by the number of resources the task requires. If the node does not have enough resources to serve any other task, the node is removed from the respective list. In the case that there are no available edge resources to serve the task, then it is served by the cloud.

The order in which the heuristic algorithm (sequentially) considers the tasks, affects the solution obtained. A simulated annealing algorithm can be used to search for better solutions (different serving orders) according to the total objective cost. Simulated annealing iteratively searches for solutions to approximate the global optimum of a function. At each step the algorithm considers some neighboring states/solutions according to a parameter called "temperature". At the beginning of the iterations the temperature is set to a high value. The algorithm randomly considers a new state at some distance from the current solution. The higher the temperature, the higher the distance of the new candidate state. As the iterations increase, the temperature decreases, and the algorithm converges to the final solution. There are additional nuances in the application of the algorithm [40]. In our case, for a given number of iterations, the algorithm randomly changes the serving order of a certain number of tasks. The higher the temperature, the larger number of tasks change order. The candidate solution is then considered to be served according to the aforementioned greedy algorithm. When the iterations end, the algorithm returns the best objective cost it has found, and the specific allocation of the tasks at the edge and at the cloud.

C. Optimality and complexity of the algorithms

The resource allocation problem at hand is a combinatorial optimization problem. We are given a set of tasks. We must figure if, where, how many, and under what options can the tasks be allocated at the edge (or at the cloud). A simpler allocation problem would be bin packing. In this case, items of different sizes (in our case tasks with only one resource requirement) must be packed into a finite number of bins, each of a fixed given capacity (in our case one type of edge node resource). This is a known NP-complete problem, implying that our problem is also NP-complete, since even its simplified version is NP-complete.

The ILP formulation given in Section IV.A provides the optimal solution, when tractable and solved with an exact method (e.g., branch and bound, cutting planes). However, the required time may be non-polynomial in general.

The complexity of the greedy algorithm mainly depends on the total number of tasks that have to be allocated. The algorithm first computes the cost of serving each task at the edge or the cloud for all possible CPU/GPU models q and accuracies a. If a task has to be served at the edge, a total of n nodes have to be searched in the worst case. When a node does not have enough resources to serve any task, it is removed from the list. Also, the algorithm selects the first node that has enough resources to serve the task. In any case, the complexity of the greedy algorithm is $O(|T_j|\cdot|Q|\cdot|A| + |T_j|\cdot|N|)$. The simulated annealing, tries a number of different orderings ψ, so its complexity is $O(|T_j|\cdot|Q|\cdot|A| + |T_j|\cdot|N|\cdot\psi)$. Regarding the optimality, if the edge resources are enough to serve all the tasks that have to be served at the edge according to the objective cost, a simple greedy algorithm provides a near-optimal solution. Otherwise, the solution is (highly) likely to be suboptimal, depending on the deficit of the edge resources. Note that in many cases, the simulated annealing cannot find the optimal solution, regardless of the number of iterations. The reason is that in each step, the algorithm decides to allocate a task by minimizing the *individual* objective cost of the task. This cost depends on the chosen accuracy and location (edge or cloud). However, the overall optimal policy may sometimes involve making an *individually* suboptimal allocation decision for certain tasks. For, example, a task may have to be served at the edge, and with the best possible accuracy (meaning a large number of GPUs) in order to minimize its individual objective

cost. However, the overall optimal allocation may actually be to serve it at the edge with a little worse accuracy, thus less required GPUs, so that certain edge resources are left for other tasks to also be served at the edge. Note that if a task has to be definitely served with a specific targeted accuracy, this is taken into account by the algorithm. These scenarios are difficult and computationally expensive to be considered by a heuristic algorithm.

D. Implementation of the algorithms

To implement the resource allocation algorithms, the following inputs must be calculated: the vector of resource requirements R^{jk} of each task and the vector of available resources R_n of each node. In Section III.B.1 we mentioned the use of profiling to determine the vector of resource requirements. As far as vector R_n is concerned, we assume that before each resource assignment takes place, all edge nodes will communicate their current resource availability, so that the allocation algorithm will employ an updated figure.

E. Convergence of the algorithms

Regarding the ML convergence properties, they depend on the type of algorithm used, the specific architecture, its parameters, etc. [33]. We don't look into the convergence properties in any detail, as it is outside the scope of our paper. Convergence properties are, however, modeled and accounted for in our work through the resource requirement coefficient vectors. These vectors characterize the ML task at hand, as they define the resource usage (number of resources, time used, epochs, etc.) required to achieve a specific accuracy.

Regarding the ILP convergence, note that it is not guaranteed to find the optimal solution in polynomial time. However, in our experiments we noticed that the ILP solver was able to find optimal solutions (as demonstrated by the optimality gap of the solution given by the ILP solver) in a few seconds or minutes (depending on the scenario) as describe in the evaluation section. Whether or not this time is acceptable, depends on the specific use case. If it is not acceptable, the optimality requirement can be relaxed.

The aforementioned algorithms allocate resources at the edge or the cloud with the objective to minimize the total cost and maximize the accuracy, while satisfying the constraints and the delay requirements of the jobs. Since there is a large number of optimization parameters, the solution to the problem is not trivial. In the following section we examine various scenarios and evaluate the trade-offs in each case.

V. Evaluation Results

To evaluate our resource allocation framework and quantify the edge-cloud cost relationships, we performed a number of simulation experiments. We used Python and the Pyomo [41] optimization software to code the ILP, and IBM CPLEX [42] to solve the problem on a desktop computer with a quad-core CPU at 4 GHz with 16 GB RAM.

A. Simulation parameters

To demonstrate the running times of the algorithms in resource demanding circumstances, we initially assumed a large, 60-node edge network with finite resources. Each edge node has 5 racks, 1 rack has 10 servers, and 1 server has 4 low cost and 2 higher cost GPUs, for a total of 200 low cost and 100 higher cost GPUs per node. Each edge node is also considered to have the following resources available exclusively for distributed computation purposes: 25 GB RAM, 10 TB of storage, 10 Tbps incoming bandwidth and 6000 CPU physical cores (that could correspond to approximately 100 CPUs). We also assumed a cloud network with infinite resources.

We considered a scenario consisting of a total of 100 training image recognition ML jobs. The size B_j of each sample (image) of a job j is chosen uniformly from the following set of values: [0.4, 0.8, 1.2, 1.6, 2, 2.4] MBs / sample. The available GPU models q were NVIDIA DGX-1 with 1 (low cost) or 8 (higher cost) GPU V100 16G. The respective cost of these GPUs at the cloud is \$2.08/hour and \$16.7/hour [36]. The b/w cost to transfer data to the cloud is \$0.01/GB [36]. More specifically, the respective processing instance name in Amazon is p3.2xlarge or p3.16xlarge, and the pricing corresponds to reserved instances. The required b/w of each task t_{jk} is derived by multiplying the generation rate λ_{jk} of samples/sec by the size B_j in MBs/sample and by the duration of period P_o in seconds. This figure equals to the amount of data that have to be transferred within one period. The calculation of the required storage and memory is relatively trivial and does not play a significant (monetary) role in the resource allocation problem, so it will be ignored in the performance results.

We assume that the jobs do not have a minimum required accuracy. This allows for clear evaluation of the allocation trade-offs between resource costs and accuracy. We examined a set of different parameters to evaluate the trade-offs between processing and b/w cost at the edge and at the cloud. More specifically, we assumed different: i) edge vs cloud bandwidth costs, ii) edge vs cloud processing costs, iii) number of epochs. According to [37] the edge's b/w costs can be approximately 0.1 times the cloud's. We therefore assumed that the edge b/w cost C_E^{bw} could be $\rho_{bw} \in [0.5, 0.1]$ times the cost C_C^{bw} to transfer the data to the cloud, that is, $C_E^{bw}=\rho_{bw} C_C^{bw}$. Moreover, according to [38], the edge processing costs C_E^q can be approximately ρ_q=1.5 times the cloud processing costs C_C^q, that is, $C_E^q=\rho_q C_C^q$. We therefore assumed that the processing costs at the edge could be $\rho_q \in [1.5, 2]$ times more than that of the cloud.

Each of the 100 training jobs consists of either 3, 4,…, 7 ML image recognition tasks, uniformly distributed. The sum of the

TABLE II
Important Simulation Parameters

Symbol	Value	Symbol	Value
N	60 nodes	P	30 sec
J	100 jobs	$\lvert T_{jx} \rvert$	3, 4,…,7 tasks
λ	15 samples/sec	Π_q	166, 566 samples/sec
R_n^G	300 GPUs	R_n^B	10 Tbps
C_C^{bw}	\$0.01/GB	C_C^G	\$2.08, \$16.7/hour

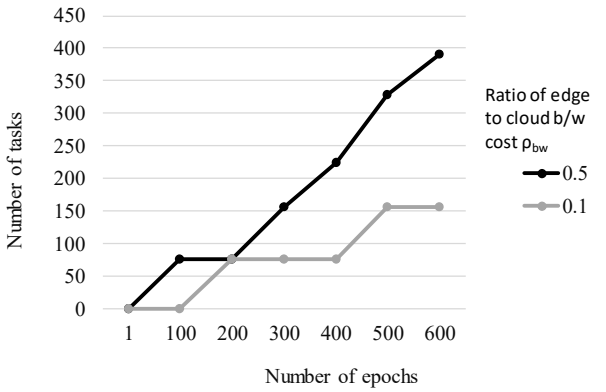

Fig. 4a Number of tasks allocated at the cloud for ratio of edge to cloud processing costs ρ_q=1.5

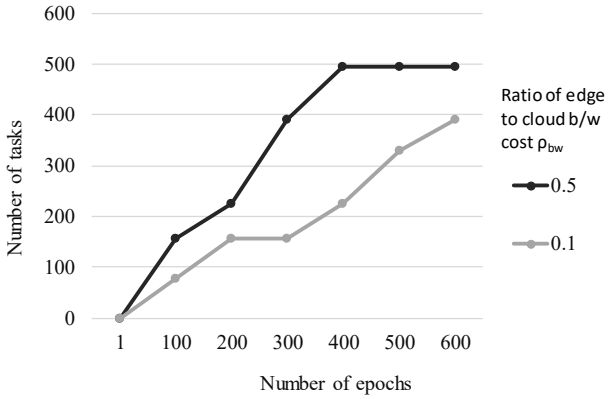

Fig. 4b Number of tasks allocated at the cloud for ratio of edge to cloud processing costs ρ_q=2

sample production rates of the devices providing data to task t_{jk} is $\lambda_{jk} = \sum_{y \in Y_{jk}} \lambda_y$ =15 samples/sec. We consider that the duration of the training period is $P_o = 30$ seconds, yielding $S_{jk} = 150$ samples processed in each period. The exact number of the ML tasks per job, the S_{jk} and the P_o do not play an important role to the simulation and the resulting trade-offs. They mainly affect the magnitude of the problem (i.e., how many processing, b/w, etc., resources each task requires, and not the decision for the allocation of the jobs at the edge or cloud). We also assume that each ML task could be served with two different accuracies $A = \{\alpha^{good}, \alpha^{low}\}$. Next, we will use realistic GPU performance figures to determine the number of aforementioned NVIDIA GPU units U^{jkqa} required per task and per period, based on the number of samples S_{jk} of each task t_{jk} processed per period and the accuracy α_j. To do this, we have to take into account the total samples that have to be processed during each period P_o, which is equal to the number of samples S_{jk} multiplied by the number of epochs H_{jk} (i.e., iterations over the same number of samples). This figure has to be processed by the GPU units (with the performance measured in samples/sec[44]), within P_o seconds. The computational performance Π_q^a (of q model and a accuracy) of 1 GPU V100 16G unit for image recognition training according to [44] is $\Pi_q^a = 166$ or 566 samples/sec for single-precision floating-point math – FP32 or mixed precision accuracy respectively. We assume that the training is fully pipelined, i.e., the computation and communication times fully overlap. The training performance of a resource q consisting of 8 GPU V100 units is $\Pi_q^a = 1210$ or 4160 samples/sec respectively. If a sample has to be processed H_{jk} times (epochs), the total number of GPU units required for the ML task is:

$$U^{jkqa} = \left\lceil \frac{S_{jk} H_{jk}}{P_o \Pi_q^a} \right\rceil = \left\lceil \frac{\lambda_{jk} H_{jk}}{\Pi_q^a} \right\rceil \quad (14)$$

(rounded above so that it is integer).

The number of epochs H required for certain ML benchmarks to reach the required accuracy varies from 5 to approximately 50 epochs [34]. In other cases, a larger number of epochs (e.g., 2000) may be required. Since we assume continuous learning with many training datasets, a low number of epochs can be employed. On a long enough timeline, the accuracy of each ML model will converge to the required. We assume that the number of epochs H can be [1, 100, 200, 300, 400, 500, 600]. The total running time of the ILP algorithm for these realistic parameters was 11 seconds. In Section V.B.3 we provide more details about the running time of the ILP and the heuristics for an even larger problem.

Finally, we do not assume any inference jobs for the evaluation due to space limitations, although we conducted related simulations. The results are similar to the training case. The main difference is that the GPU performance in the case of inference depends on the batch size. The bigger the batch size (e.g., number of consecutive images that we want to perform image recognition), the better is the performance of the GPU. Thus, the less important are the processing costs for the allocation of inference jobs.

B. Simulation Results

1) Edge vs Cloud allocation decisions

First, we examined how the different values of some assumed parameters affect the allocation of a training task at the edge or at the cloud. In Fig. 4 we show the number of training tasks (496 in total) allocated at the cloud as a function of the number of epochs and for different edge/cloud processing and b/w costs. For simplicity reasons, we do not depict the allocation of the remaining tasks at the edge. In Fig. 4a, the processing cost ratio of edge to cloud is ρ_q=1.5. When the number of epochs is small (thus, relatively little processing is required), all (or most) tasks are served at the edge, since the b/w costs are lower than the processing costs. As the number of epochs increases, some tasks are served at the cloud, depending on the b/w cost ratio. The increased number of epochs means that the total processing cost of a task play a more important role than the b/w cost to the allocation of the tasks. For the b/w edge/cloud cost ratio of ρ_{bw}=0.1, the allocation does not change between 200 and 400 epochs. This is due to the greater importance the b/w costs have compared to the processing, under these circumstances. Also, even though the difference of edge and cloud processing costs is small, for large number of epochs the processing costs are so much greater than the bandwidth costs, that many tasks are served at the cloud. Note that as we will see in Fig. 5, the cloud

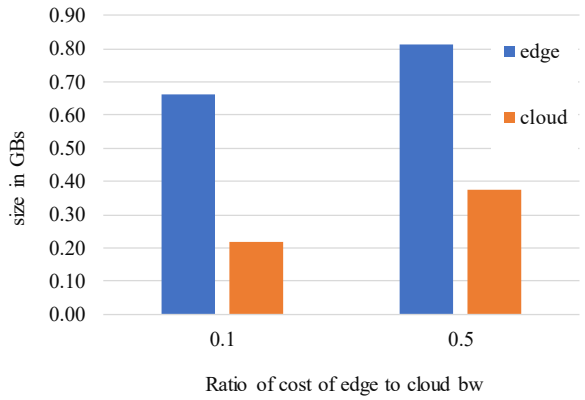

Fig. 5 Mean size of a job allocated in edge or cloud for 500 epochs and for different b/w cost ρ_{bw} ratios of edge to cloud and processing ratio ρ_q=1.5

serves smaller (in terms of Mbytes) tasks. In Fig. 4b the edge processing costs are even more expensive than the cloud's (ρ_q=2). We notice that the allocation of tasks tips towards the cloud more quickly. The different b/w costs still seem to play a relatively significant role for the allocation of the tasks. Despite the edge processing costs being twice the cloud's, the edge is still more preferable until at least 200 epochs. Overall, from Fig. 4 we can conclude that the edge is more preferable to serve tasks with relatively low processing requirements. The different b/w cost ratios in some cases can play a sizeable difference in the allocation of the tasks. In Fig. 4b for example, the cloud can serve in some cases more than two times more tasks when the ratio is ρ_{bw} is 0.5 as opposed to 0.1.

Fig. 5 depicts the mean size in GBs for 500 epochs of a job's task that is served at either the edge or the cloud when the edge's processing costs are ρ_q=1.5 times the cloud's. Similar results can be drawn for different epochs and processing costs (as long as some tasks are served at the edge and others at the cloud). The size of a task depends on the generation rate λ_y in samples/sec of its related devices, the duration of P_o, and the size of each task's sample. The first two variables are the same for all the jobs we considered. Thus, the differentiating factor is the size of a task's sample. Note also that we have assumed a random number of tasks per job, implying that the definite size of a job depends also on the exact number of the tasks. However, this does not significantly affect the decision on the allocated location of a job. The increased number of tasks not only means more data to transfer (hence increased b/w costs), but also more samples to calculate (hence analogous increase on the processing requirements). Since we have assumed that the performance of a GPU in samples/sec is constant regardless of the size of a sample, the differentiating factor in whether a task will be served at the edge or at the cloud is the size of its samples. We notice that the edge tends to serve tasks with large size. It seems that in order for a task to be served at the cloud, it has to be significantly smaller than the tasks that are typically served at the edge. The trends are similar for different edge processing costs. Also, when the number of epochs increases, the contribution of the processing costs also increases. Therefore, a task has to be larger to be served at the edge. Note also, that we considered an image recognition training scenario, where samples are relatively large. In different applications the size of the samples can be smaller, leading to less required b/w, and different job distribution at the edge and cloud. For example, we also evaluated Automated Speech Recognition training. In this scenario, the performance of the low-cost GPU is around 600 sequences/sec, while the size of a sample (word) is very small, and the b/w costs are insignificant. In this scenario, all jobs were served at the cloud when the edge processing costs were greater than the cloud's, regardless of any other parameter.

2) Monetary cost evaluation

In this subsection we investigate the effect the different parameters have on the monetary cost of the jobs served at the edge and at the cloud. Fig. 6 presents the total cost to serve all jobs for one training round, decomposed to edge and cloud b/w and processing costs and for different number of epochs. In Fig. 6a we assumed edge to cloud b/w cost ratio ρ_{bw}=0.1, and edge to cloud processing costs of ρ_q=1.5. The main cost contributor is the processing. As the number of epochs increases, the edge (and later cloud) processing costs play more important role in the total cost of the jobs. The (inexpensive) edge b/w costs are actually a sizeable part of the total edge costs for 1 and 100

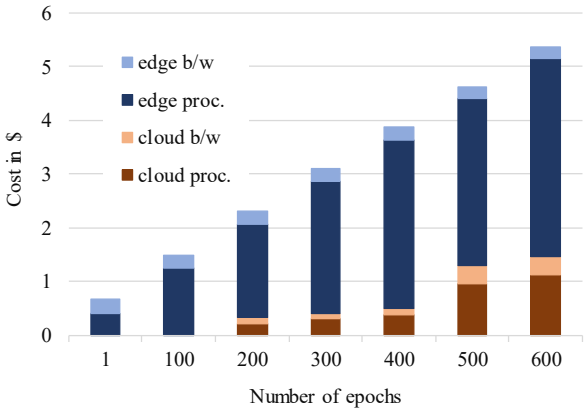

Fig. 6a Total decomposed cost of jobs for edge/cloud cost b/w ratio ρ_{bw}=0.1 and processing ratio ρ_q=1.5

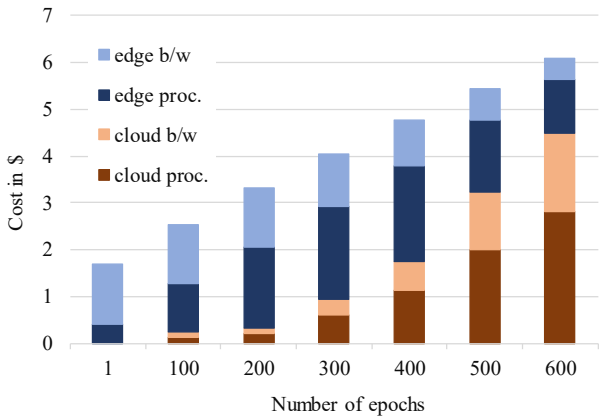

Fig. 6b Total decomposed cost of jobs for ρ_{bw}=0.5, ρ_q=1.5

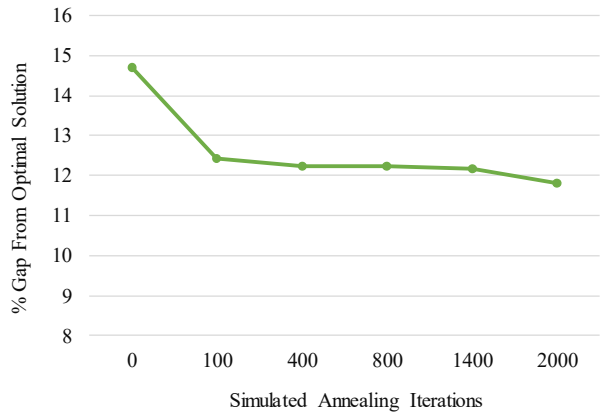

Fig. 7 Effect of the number of simulated annealing iterations to the gap from the optimal solution

epochs (38.5% and 17.2%, respectively). For 600 epochs, the *cloud* processing costs are a considerable fraction (76.8%) of the overall cloud costs. So, in lower number of epochs the edge is more preferable. When the edge to cloud b/w cost ratio is 0.5 (Fig. 6b), the edge b/w costs play an even more important role. For the case of 1 and 100 epochs, the b/w costs are 75.88% and 54.2%, respectively, of the total edge costs. As the number of epochs increases, so does the contribution of processing costs (both edge and cloud). For 600 epochs, the cloud processing costs are approximately 62.5% of the overall cloud costs.

3) Algorithm Comparisons

In this subsection we will first compare the optimality of the heuristic algorithms to the ILP algorithm. We will then compare the performance of each algorithm in terms of monetary cost to achieve a certain accuracy. As we mentioned in the previous section, when the edge has enough resources to serve all the appropriate tasks, then a simple greedy algorithm provides the optimal solution. So, in this subsection we limit the amount of edge resources (assumed only one edge node), to clearly compare the optimality of the algorithms.

In Fig. 7 we can see the gap of the greedy and the simulated annealing algorithm from the optimal solution that the ILP algorithm provides. For zero iterations, the simulated annealing is a greedy algorithm. In this case, the gap to the optimal solution is 14.7%. For 100 iterations, the gap quickly reduces to 12.4%. After that, the reduction is much slower. For 2000 iterations the gap is 11.8%. The gap does not reduce very much with the increased iterations for the reasons we had mentioned in the previous section (about the optimality of the heuristics).

In Fig. 8 we compare the performance of the ILP algorithm to the simulated annealing and the greedy algorithm. In this scenario, we assumed a fixed edge to cloud GPU and b/w cost ratio of 1.5 and 0.1 respectively, and a range of different values for W: $[0.2, \ldots, 0.4]$. For this range of values, we plot the relationship between the mean accuracy of all the served ML tasks and their total monetary cost at the edge and cloud. Again, the edge resources are not enough to serve all the tasks. There is no standard way to predict the exact trade-off between accuracy and monetary cost for different values of W. For example, for larger amount of training jobs the total monetary cost is larger. Therefore, a smaller value of W may be required to achieve the same accuracy. From this graph, one can also see the monetary costs of ML apps with different accuracy requirements. For example, certain security related ML apps typically have high accuracy requirements. The effect and the trade-off between accuracy and monetary cost of different ML apps can be quantified using such a graph.

As far as the comparison of the three algorithms, the ILP algorithm can achieve the best accuracy with the lowest monetary cost. For the baseline 80% accuracy, the related costs are \$2, \$2.44 and \$2.55 for the ILP, simulated annealing and greedy respectively. So, the ILP algorithm can serve the tasks with the same accuracy, but at 18% and 21.5% lower cost respectively. Moreover, for roughly the same cost (\$2.7) the ILP algorithm can achieve 87% percent mean accuracy, compared to the 80% of both heuristics. To achieve the best accuracy all the algorithms have as expected similar monetary cost, since this case is the most expensive option and the margins for economic decisions are very low. Using this case, we can also compare the monetary cost of an algorithm that optimizes only the accuracy, versus the alternatives that also take into account the monetary cost at the objective. For example, to achieve 90% accuracy, the monetary cost is \$3.15.

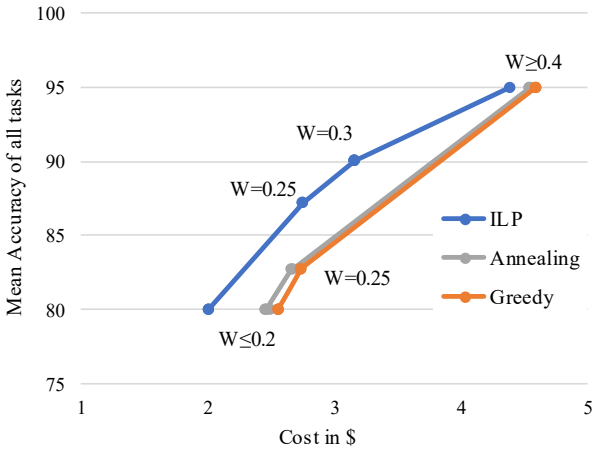

Fig. 8 Monetary cost and Accuracy performance comparison

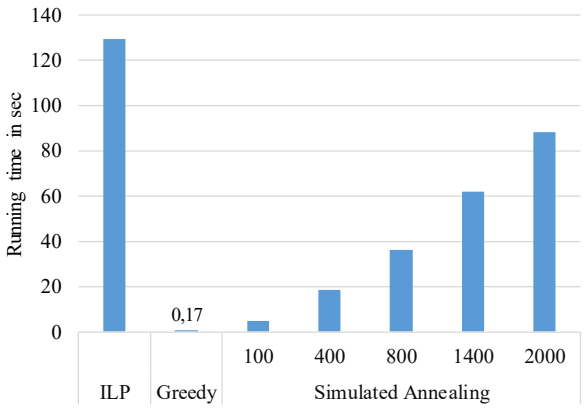

Fig. 9 Running time comparison

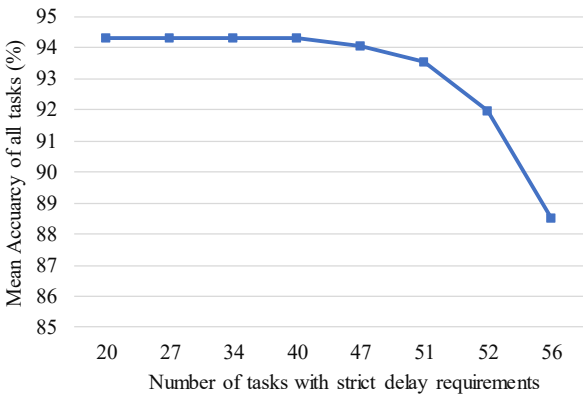

Fig. 10 Delay vs accuracy trade-offs

The algorithm that achieves the best accuracy requires significant additional costs ($4.38).

In Fig. 9 we compare the running times of the ILP, the greedy and the simulated annealing algorithms. To demonstrate the suitability of the algorithms in larger problem instances, we assumed a total of 1000 jobs while keeping all other simulation parameters the same as defined in section V.A. The running time of the ILP algorithm includes the time to prepare all the necessary equations and constraints, the time to prepare the solver, and the time to actually solve the problem (which is the time reported by the solver). The latter is actually a (small) fraction of the total required time described above. As we can see in the figure, the ILP requires the largest amount of time to find a solution. The greedy algorithm can provide very quickly a suboptimal solution. As the iterations of the simulated annealing increase, a proportionally larger amount of time is required to provide a solution. After a certain point, the additional iterations and increased running time do not offer significant benefits in terms of the objective cost (Fig. 7). In conclusion, the ILP algorithm can provide a solution in a rather short amount of time even for large problem instances. However, in cases of time-critical jobs, even this time could be unacceptable. In even larger instances with increased number of jobs, large pool of available GPU models, accuracy options, etc., the ILP running time may be prohibitively large. In these cases, a greedy algorithm can be used to provide a faster solution that is usually near-optimal. If the optimality gap is significant, a simulated annealing algorithm can be used to reduce it, while keeping the total running time relatively low. Generally, for each given problem, its dimensionality and its timing constraints, a suitable solution algorithm should be chosen.

4) Accuracy vs delay trade-offs

In this subsection we focus on the trade-off between accuracy and delay requirements. More specifically, we assume an increasing number of tasks with strict delay requirements. These tasks have to definitely be served at the edge, thus occupying the limited edge resources, leaving less resources available for the rest of the tasks that could benefit (in terms of monetary cost) from the edge. We assume that a ratio $r_j = [0.1, 0.2, \dots, 0.8]$ of the jobs are inference jobs requiring 1 processing unit and having strict delay requirements. We also limited the number of edge nodes to one, so that the edge resources are not enough to accommodate all tasks that would otherwise have been served at the edge. In Fig. 10 we see that as the number of delay-critical tasks increases, the mean accuracy of the tasks decreases. The allocation algorithm decides to lower the accuracy of certain tasks (if their constraints allow it), to decrease their resource requirements so that the edge can serve more tasks. Overall, we can see that there are significant trade-offs between accuracy, edge/cloud monetary costs, and delay requirements that play significant role in the allocation of the tasks.

VI. Conclusions

In this paper we examined the problem of resource allocation for distributed computation applications. We proposed a framework to allocate resources for jobs at the edge–cloud continuum. The objective was to optimize the required monetary cost and accuracy to serve the jobs, while respecting possible stringent timing constraints. We examined various optimization parameters pertained to processing/bandwidth costs, accuracy and delay in both edge and cloud resources. We proposed an ILP algorithm and also examined certain heuristics to solve the resource allocation problem. We evaluated the framework using realistic simulation parameters for a DML scenario. The results indicate that the processing costs play an important role in the allocation of a job at the edge or at the cloud. The cloud bandwidth costs and the delay constraints can also be significant in certain scenarios. The heuristic algorithms can provide a quick solution that is close to that of the ILP (indicative gap to optimal solution ~12%). Nevertheless, the allocation optimality of the ILP can provide significant monetary and accuracy benefits. Future work includes the possibility that a job can be served (partially) at the edge devices. Also, future work includes the modeling of energy consumption as well as prediction of the future workload to better manage the available network resources.

References

[1] R. Shokri, V. Shmatikov, “Privacy-Preserving Deep Learning,” ACM SIGSAC Confe. Computer and Communications Security (CCS), 2015.

[2] J. Konečný, B. McMahan, D. Ramage, “Federated optimization: Distributed optimization beyond the datacenter,” arXiv preprint arXiv:1511.03575 (2015).

[3] T. Mohammed, C. Joe-Wong, R. Babbar, M. D. Francesco, “Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading,” IEEE INFOCOM 2020.

[4] X. Chen, J. Zhang, B. Lin, Z. Chen, K. Wolter, G. Min, “Energy-Efficient Offloading for DNN-Based Smart IoT Systems in Cloud-Edge Environments,” IEEE Transactions on Parallel and Distributed Systems, 33(3), pp. 683-697, March 2022.

[5] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M.’A. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng, “Large scale distributed deep networks,” NIPS, pp. 1223–1231, 2012.

[6] P. Mach, Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” IEEE Communications Surveys & Tutorials, 19(3), 1628-1656, 2017.

[7] M. Chen, Z. Yang, W. Saad, C. Yin, H. V. Poor, S. Cui, “A joint learning and communications framework for federated learning over wireless

networks," IEEE Transactions on Wireless Communications, 20(1), pp. 269-283, 2020.

[8] Z. Yang, M. Chen, W. Saad, C. S. Hong, M. Shikh-Bahaei, "Energy efficient federated learning over wireless communication networks," IEEE Transactions on Wireless Communications, 20(3), pp. 1935-1949, 2020.

[9] J. Zhang, L. Khaled, "Mobile edge intelligence and computing for the internet of vehicles," Proceedings of the IEEE 108.2, 2019.

[10] K. Lin, C. Li, Y. Li, C. Savaglio, G. Fortino, "Distributed Learning for Vehicle Routing Decision in Software Defined Internet of Vehicles," IEEE Transactions on intelligent transportation systems, 22(6), 2021.

[11] X. Zhao, P. Sun, Z. Xu, H. Min, H. Yu, "Fusion of 3D LIDAR and Camera Data for Object Detection in Autonomous Vehicle Applications," IEEE Sensors Journal, 20(9), pp. 4901-4913, 2020.

[12] F. Rahimi, A. Ipakchi, "Demand response as a m arket resource under the smart grid paradigm," IEEE Transactions on smart grid, 1(1), pp. 82-88, 2010.

[13] D. Bertsekas, J. Tsitsiklis, "Parallel and distributed computation: numerical methods," Athena Scientific, 2015.

[14] M. Langer, Z. He, W. Rahayu, Y. Xue, "Distributed Training of Deep Learning Models: A Taxonomic Perspective," IEEE Transactions on Parallel and Distributed Systems, 31(12), pp. 2802-2818, Dec. 2020.

[15] A. Sayed, "Adaptation, learning, and optimization over networks," Foundations and Trends in Machine Learning, 7 (4–5), pp. 311-801, 2014.

[16] A. Nedić, A. Olshevsky, M.G. Rabbat, "Network topology and communication-computation tradeoffs in decentralized optimization," Proceedings of the IEEE, 106(5), pp. 953-976, 2018.

[17] T. Yang, X. Yi, J. Wu, Y. Yuan, D. Wu, Z. Meng, Y. Hong, H. Wang, Z. Lin, K. H. Johansson, "A survey of distributed optimization," Annual Reviews in Control, 47, pp. 278-305, 2019.

[18] C. Ma, J. Konečný, M. Jaggi, V. Smith, M. I. Jordan, P. Richtárik, M. Takáč, "Distributed optimization with arbitrary local solvers," Optimization Methods and Software, 32(4), pp. 813-848, 2017.

[19] T. Ben-Nun, T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis," ACM Comput. Surv. 52, 4, Article 65, 2019.

[20] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," 11th USENIX Symposium on Operating Systems Design and Implementation, pp. 583-598, 2014.

[21] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, "PipeDream: Fast and Efficient Pipeline Parallel DNN Training," arXiv:1806.03377v1, 2018.

[22] I. Thangakrishnan, D. Cavdar, C. Karakus, P. Ghai, Y. Selivonchyk, C. Pruce, "Herring: Rethinking the parameter server at scale for the cloud," SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 2022.

[23] F. Saeik, M. Avgeris, D. Spatharakis, N. Santi, D. Dechouniotis, J. Violos, A. Leivadeas, N. Athanasopoulos, N. Mitton, S. Papavassiliou, "Task offloading in Edge and Cloud Computing: A survey on mathematical, artificial intelligence and control theory solutions," Computer Networks, 195, 2021.

[24] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, K. Chan, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," IEEE INFOCOM, 2018.

[25] X. Ran, H. Chen, X. Zhu, Z. Liu, J. Chen, "DeepDecision: A mobile deep learning framework for edge video analytics," IEEE INFOCOM, 2018.

[26] H.-J. Jeong, H.-J. Lee, C. H. Shin, S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," ACM Symp. Cloud Comput. (SoCC), 2018, pp. 401–411.

[27] G. Drainakis, P. Pantazopoulos, K. V. Katsaros, V. Sourlas, A. Amditis, "On the Distribution of ML Workloads to the Network Edge and Beyond", IEEE INFOCOM 2021.

[28] M. Chen, H. Wang, Z. Meng, H. Xu, Y. Xu, J. Liu, H. Huang, "Joint Data Collection and Resource Allocation for Distributed Machine Learning at the Edge," IEEE Transactions on Mobile Computing 2020.

[29] H. Li, K. Ota, M. Dong, "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing," IEEE network 32.1, 2018.

[30] R. Zhou, J. Pang, Q. Zhang, C. Wu, L. Jiao, Y. Zhong, Z. Li, "Online Scheduling Algorithm for Heterogeneous Distributed Machine Learning Jobs," IEEE Transactions on Cloud Computing, doi: 10.1109/TCC.2022.3143153.

[31] I. Sartzetakis, P. Soumplis, P. Pantazopoulos, K. V. Katsaros, V. Sourlas, E. Varvarigos, "Resource Allocation for Distributed Machine Learning at the Edge-Cloud Continuum," ICC, 2022.

[32] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W. H. Lee, K. K. Leung, L. Tassiulas, "Model pruning enables efficient federated learning on edge devices," IEEE Transactions on Neural Networks and Learning Systems, early access, DOI: 10.1109/TNNLS.2022.3166101, 2022.

[33] S. Gupta, W. Zhang, F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," IEEE ICDM, 2016.

[34] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, A. Ike, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, T. Tabaru, C.-J. Wu, L. Xu, M. Yamazaki, C. Young, M. Zaharia, "MLPerf Training Benchmark," ArXiv abs/1910.01500 (2020).

[35] X. Wu, V. Taylor, J.M. Wozniak, R. Stevens, T. Brettin, F. Xia, "Performance, energy, and scalability analysis and improvement of parallel cancer deep learning candle benchmarks," 48th Intl. Confe. on Parallel Processing, Aug. 2019.

[36] "Amazon ec2 pricing," available online: https://aws.amazon.com/ec2/instance-types/p3/

[37] "Edge computing and transmission costs," available online: https://www.datacenterdynamics.com/en/opinions/edge-computing-and-transmission-costs/

[38] "The economics of edge computing," available online: https://edgecomputing-news.com/2020/10/29/analysis-economics-of-edge-computing

[39] O. Ascigil, T. K. Phan, A. G. Tasiopoulos, V. Sourlas, I. Psaras, G. Pavlou, "On uncoordinated service placement in edge-clouds," IEEE Intl. Confe. on Cloud Computing Technology and Science (CloudCom), 2017.

[40] S. Kirkpatrick, C. D. Gelatt Jr, M. P. Vecchi, "Optimization by simulated annealing," science 220, no. 4598, pp. 671-680, 1983.

[41] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, J. D. Siirola, "Pyomo–Optimization Modeling in Python," Mathematical Programming Computation, 3, 219-260, Springer, 2017.

[42] "IBM CPLEX optimization studio," available online: https://www.ibm.com/products/ilog-cplex-optimization-studio

[43] "Mapping out edge computing: How dense is it?," available online: https://www.lightreading.com/the-edge/mapping-out-edge-computing-how-dense-is-it/d/d-id/771128.

[44] "Nvidia resnext performance," available online: https://ngc.nvidia.com/catalog/resources/nvidia:resnext_for_tensorflow/performance