To be, or not to be: Offloading in edge computing

Abhishek Verma
New York University
New York, NY, USA
av2783@nyu.edu

Ishita Jaisia
New York University
New York, NY, USA
ij2056@nyu.edu

Utkarsh Kumar
New York University
New York, NY, USA
uk2012@nyu.edu

***Abstract*—Several wireless devices can offload their computation tasks to an edge server in mobile edge computing (MEC) networks. Optimizing joint offloading and bandwidth allocation is formulated as a mixed-integer programming problem to conserve energy and maintain quality of service for WDs. Unfortunately, the problem cannot be solved by general optimization tools in an effective and efficient manner because of the curse of dimensionality, especially for large-scale WDs. This paper proposes an offloading algorithm for MEC networks based on distributed deep learning (DDLO). DNNs are further trained and improved with the help of a shared replay memory. DDLO seems to be able to generate near-optimal offloading decisions in a reasonable amount of time, based on extensive numerical results. Furthermore, this paper examines the algorithm's performance on devices with different configurations. It was found that the algorithm works efficiently even on devices with low computational power - simulating how it would work on edge devices.**

***Index Terms*—Edge computing · Offloading · Deep learnisng · Distributed learning**

I. Introduction

Edge Computing is a distributed information technology architecture. In Edge Computing, the data of the client is processed as close to the originating source as possible. Some of the storage and computing resources are moved out of the central data center and placed near the source data unlike traditional computing where the data is moved to the compute resource and is processed by an application and then sent back to the client endpoint. As the volume of data increases day by day it has become highly expensive to carry out the process involving traditional data centers and cloud as it would increase data congestion over the network which affects a lot of situations which are time sensitive and disruption sensitive. The focus has now shifted from central data centers to the edge infrastructure.

Edge computing has become relevant nowadays because it provides us with an effective solution to the emerging network problems due to high increase in data volumes which are produced and consumed by numerous organizations on a daily basis. Other than the increasing volume, it also produces efficient results for time-sensitive applications which demand fast and responsive networks. Edge computing techniques are primarily used to collect, filter, process and analyze data near the edge of the network or "in-place". This makes the wireless networks more flexible and cost-effective. It also includes the ability to perform on-site big data analytics which allows us to make near real-time decision making. It further helps in enforcing security practices and meeting regulatory policies as the risk of sensitive data being exposed is highly reduced by keeping all the computing power local. The cost of bandwidth is reduced immensely as the data doesn't have to sent back and forth between the central and regional sites.

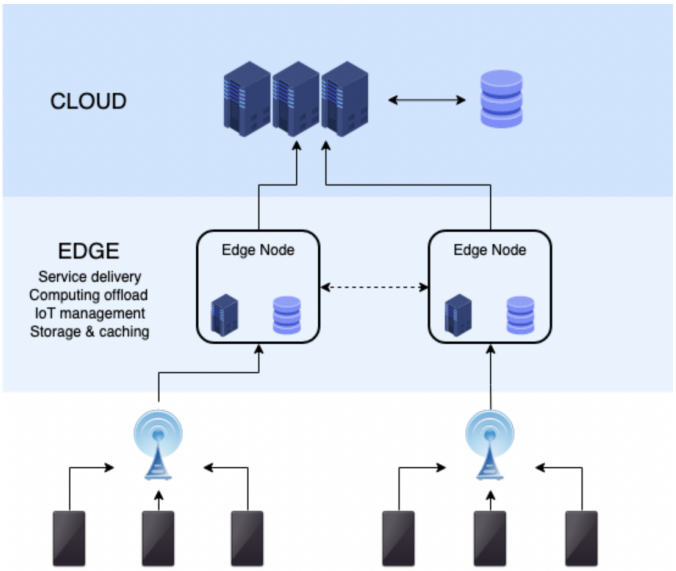

Fig. 1. Edge Computing Infrastructure

Edge computation empahsizes on real-time computation and data collection which greatly contributes to the success of data-intensive intelligent applications. There are innumerable use cases of edge technology. Edge technology helps autonomous vehicles to interact and respond efficiently in order to assist safely, quickly and accurately. It also benefits the health industry as it provides better monitoring for chronic conditions which can alert caregivers when help is required. If the devices used for monitoring and making decision rely on cloud then there could be a delay in transmitting the message which could result in fatal results. Security surveillance systems use edge technology to respond to potential threats and to alert the users regarding any unusual activity encountered in real-time. IoT devices uses this technology to gain the ability to interpret voice instructions and carry out the required actions. The quality of the video can be improved by placing the server-side video of conferencing closer to the participants. It also helps service providers to ameliorate the user experience in

the mobile devices and the tasks performed including online gaming, AR/VR, etc.

The advancement of wireless communication technology has made it possible to transmit compute-intensive tasks from wireless devices to nearby internet access points or base stations, which has led to the creation of meaningful cloud-computing applications, such as online gaming, virtual/augmented reality, and real-time streaming media. In order to bridge the gap between users and edge servers, mobile edge computing (MEC) [1] [2] [3] enables compute servers to be deployed on the user's side, avoiding the need to backhaul traffic generated by applications to a remote data center. For delay-sensitive cloud applications, it reduces computation delays and reduces energy consumption.

If WDs offload their computation tasks to MEC servers, this is something that should be carefully considered. The uplink wireless channels will experience severe congestion if computation tasks are aggressively offloaded to the edge server, resulting in a significant delay in computing computation tasks. Due to this, offloading decisions and associated radio bandwidth allocation need to be managed jointly to exploit the computation offloading. The binary nature of offloading decisions, however, means that it is computationally impossible to directly enumerate every possible solution.

II. Literature Review

In the literature [13] [14], a number of low-complexity algorithms have been proposed to meet the binary computation offloading problem. [13] proposes a game-theoretic distributed algorithm for the MEC system, which requires multiple rounds of communication between edge servers and WDs. A further iterative algorithm for solving joint task offloading and resource allocation in MEC networks is to iteratively update the binary offloading decision, in which the traditional resource allocation problem is simplified when the binary offloading decision is involved. An eDors algorithm for MEC systems is proposed in [15] by relaxing the binary constraints to real variables. While all those algorithms are optimal, the trade-off between optimality and computational complexity does not apply under time-varying MEC networks with real-time offloading.

The use of deep neural networks (DNNs) with multiple processing layers for deep learning has enabled a number of breakthroughs in areas such as robot control [16], natural language processing [17], and gaming [18]. The use of a DNN to approximate the relationship between state and action for systems with large state-action spaces can result in near-optimal performance. Several computationally expensive problems in wireless communications have also been solved with deep learning, for instance, resource allocation, signal detections, interference alignment, and caching.

Our project proposes an offloading framework for MEC networks that is efficient and effective by utilizing deep learning. Assume that we have a MEC network with one edge server and multiple wireless devices (WDs), and that each WD can choose to offload its computation tasks to the edge server. The following results are obtained when WDs are conserved and quality service is preserved.

In our model, MEC networks are considered as the sum of their energy consumption and delay in completing tasks for all workers. Specifically, we propose an optimizing approach for offloading tasks of wireless devices as well as bandwidth allocation for MEC networks that optimizes both offloading decisions and transmission bandwidth for each wireless device simultaneously.

The proposed algorithm for the offloading of MEC networks is based on deep learning and offloading action generation. To generate offloading decisions effectively and efficiently, the algorithm uses multiple parallel deep neural networks (DNNs). DNNs are further trained and improved by storing the generated offloading decisions in shared memory.

When two or more DNNs are used, the proposed algorithm converges to the optimal solution. It is possible to provide near-optimal offloading decisions in under a second with a wide range of parameter settings. Deep learning-based offloading is a method to generate effective and efficient offloading decisions for MEC networks based on a distributed method which is based on deep learning. Even though our methodology solves offloading decisions and bandwidth allocation simultaneously, the proposed algorithm can easily be extended to other bandwidth allocation problems, including transmission power allocation at wireless devices and computing resource allocation at edge servers.

III. Our Contribution

We design and implement a deep learning algorithm which when given an input task and its characteristics produces an output which is a binary offloading decision. The output 0 indicates that the task is to be computed locally and 1 means that the the task is to be offloaded to the edge server. This algorithm reduces the gap between total computation time and accuracy of the system. We compare the performance of the algorithm with the greedy algorithm, which was computed in 2^{NM} time complexity, on different machine configurations. We also profile the training time and inference time on bare-metal machines, virtual machines, Kubernetes cluster and NYU HPC Cluster. We then analyse our algorithm on different low-level machine configurations because the inference will run on real-world edge device which doesn't have a lot of high computation power. We also observe how the model scales on different machines.

IV. Problem formulation and system model

A. System Model

As shown in Fig. 1, we consider a Mobile Edge Computing network consisting of one edge server, one wireless access point, and N wireless devices. Access points and edge servers are connected by optical fiber, whose delays can be ignored. WDs can perform multiple tasks either locally or through the

access point to the edge server. To retain generality, we assume each wireless device is able to accomplish M unique tasks, denoted by $m \in \{1, 2, ..., M\}$. Assign d_{nm} to the workload associated with the m^{th} task of user n. In each device n determines whether or not to offload its m^{th} task to the edge server; this offloading decision is expressed by $x_{nm} \in \{0, 1\}$. A xnm value of 1 means the task m is offloaded to the edge server, and a x_{nm} value of 0 indicates the task m is executed locally by user n. Here are more details about edge computing and local computing.

B. Edge Server Computing

1) Energy: An edge server gets its workload by transmitting it wirelessly to an access point, which in turn sends it to the edge server for processing. Because feedback information is generally small, we ignore the energy consumption and delays when the edge server transmits the results back to wireless devices. As a function of workload d_{nm} and energy cost for data processing at the edge server, we denote E_{nm} as the energy consumed by wireless devices during uploading to the edge server.

Weight of energy consumption at edge servers is represented by α. When $\alpha = 0$, only wireless devices are considered. This cost includes both the energy consumption for sending this task and the utility costs associated with executing it on the server. It is represented by E^c_{nm},

$$E^c_{nm} = E^t_{nm} + \alpha d_{nm} \tag{1}$$

2) Time delay: As a next step, we model the delay in offloading computations. In particular, c_n represents the bandwidth allotted to user n for transferring its offload task to the edge server. Consequently, when user n offloads task m to the edge server, the transmission delay will be as follows:

$$T^t_{nm} = \frac{d_{nm}}{c_n} \tag{2}$$

Also, the edge processing delay where f^c is the edge processing rate is given by,

$$T^c_{nm} = \frac{d_{nm}}{f^c} \tag{3}$$

The user n will experience the following total delay based on the offloading decisions x_{nm}:

$$T^c_n = \sum_{n=1}^{M} (T^t_{nm} + T^c_{nm}) x_{nm} \tag{4}$$

C. Edge Device Computing

The following scenario is when the user n decides to execute its tasks on the edge device. e^l_n is the energy the user n will consume per data bit to execute its task m locally. Accordingly, the energy consumption of user n when it is performing its task m locally is as follows:

$$E^l_{nm} = d_{nm} e^l_n \tag{5}$$

The local processing time per bit is denoted by t^l, which is the local processing time per user n. This means that the total processing time required for user n to execute its task m was as follows:

$$T^l_{nm} = d_{nm} t^l \tag{6}$$

$\{x_{nm}\}$ being the offloading decision, running the tasks of n users locally requires total time of,

$$T^l_n = \sum_{m=1}^{M} T^l_{nm} (1 - x_{nm}) \tag{7}$$

D. Problem formulation

We define a system utility $Q(d, x, c)$ that represents the weighted sum of energy consumption and task completion delay, in order to minimize both the total delay in completing all users' tasks as well as the energy consumption associated with it.

$$Q(d, x, c) = \textstyle\sum_{n=1}^{N} (\sum_{m=1}^{M} (E^c_{nm} x_{nm} + E^l_{nm} (1 - x_{nm}))$$

$$+\beta \max(T^l_n, T^c_n))$$

Here, $d = \{d_{nm} | n \in \{1, 2, 3, ..., N\}, m \in \{1, 2, 3, ..., M\}\}$, $x \in \{x_{nm}\}, c \in \{c_n | n \in \{1, 2, 3, ..., N\}\}$, β denotes the weight of energy consumption and task completion.

To minimize $Q(d, x, c)$ by jointly optimizing each user n's offloading decisions $\{x_{nm}\}$ and the bandwidth allocations c_n for user n's task transmission, we formulate an optimization problem `(P1)` in Fig. 2.

$$\begin{aligned} \textbf{(P1)}: \boldsymbol{Q^*(d)} = \underset{\boldsymbol{x}, \boldsymbol{c}}{\text{minimize}} \quad & \boldsymbol{Q(d, x, c)} \\ \text{subject to:} \quad & \sum_{n=1}^{N} \boldsymbol{c_n} \leq \boldsymbol{C}, \\ & \boldsymbol{c_n} \geq 0, \forall \boldsymbol{n} \in \mathcal{N}, \\ & \boldsymbol{x_{nm}} \in \{0, 1\}. \end{aligned}$$

Fig. 2.

The total bandwidth allocated to all users cannot exceed the maximum bandwidth. For each user c_n, the allocated bandwidth is either 0 or positive. Optimizing `(P1)` entails the solution of a mixed-integer programming problem, which is in general difficult. To solve `(P1)` efficiently and effectively, we study an approximate algorithm based on deep learning.

V. Algorithm Design

Here, we present an algorithm for MEC networks, where K parallel multiple DNNs generate binary offloading decisions.

Given the sizes of output data and input data of all users, denoted as d, our goal is to find an offloading policy function π to generate the optimal offloading action $x^* \in \{0,1\}^{NM}$ for (P1) as

$$\pi : d \longrightarrow x^*$$

Taking into account N users in MEC networks and M tasks per user, the size of target binary offloading decision set $\{\mathrm{x}\}$ is 2^{NM}. As finding the best offloading strategy is NP-hard, we approximate π in this paper by a parameterized DNN-based function.

The proposed algorithm for MEC networks is composed of action generation offloading and deep learning, as shown in Fig. 3. The specific implementation generates K offloading actions for each input d using K distributed offloading actors $\{x_k|k \in \{1,2,...,K\}\}$. The offloading action with the lowest system utility is selected as the output, denoted as x. Lastly, the data entry (d, x^*) is stored in a memory structure to train these K distributed offloading actors. Detailed procedures for offloading action generation and deep learning are provided below.

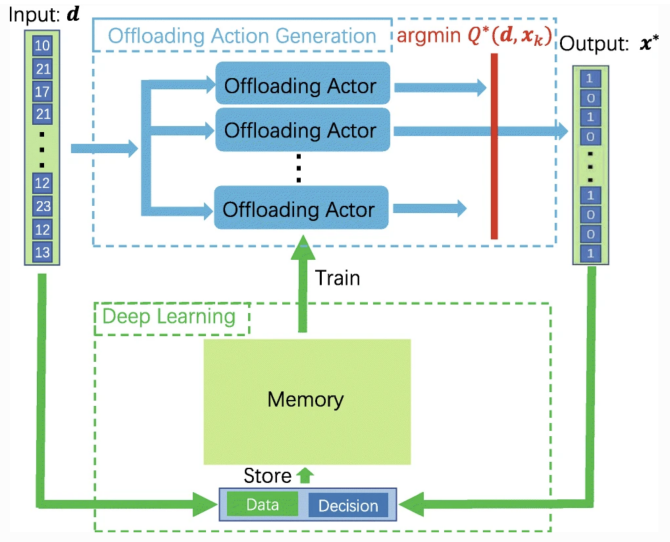

Fig. 3. Architecture of distributed deep learning-based partial offloading algorithm

A. Offloading action generation

Using K offloading actors for each input d, we produce K candidate offloading actions with one action for each actor, as shown in Fig. 4. The offloading actor uses a DNN to generate binary offloading action x_k, which is a parametrizable function f_{θ_k} defined as

$$f_{\theta_k} : d \longrightarrow x_k$$

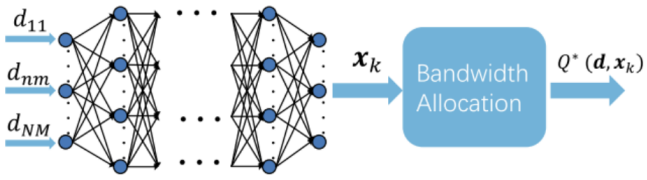

Fig. 4. An offloading actor

Here, θ_k denotes the parameters of the k-th DNN. These K DNNs all have the same structure, but they all have different parameter values θ_k.

In the presence of a binary offloading decision x_k, the original optimization problem (P1) becomes a bandwidth assignment problem (P2) as shown in Fig. 5.

$$\begin{aligned} \text{(P2)}: Q^*(\boldsymbol{d},\boldsymbol{x}) = \underset{\boldsymbol{c}}{\text{minimize}} \quad & Q(\boldsymbol{d},\boldsymbol{x},\boldsymbol{c}) \\ \text{subject to:} \quad & \sum_{n=1}^{N} c_n \le C, \\ & c_n \ge 0, \forall n \in \mathcal{N}. \end{aligned}$$

Fig. 5.

In the literature, numerous studies have been done on how to solve bandwidth allocation problems efficiently [5] [6]. We solve (P2) by using a standard optimization tool available in Scipy [24].

Following the solution of K bandwidth allocation problems (P2), the offloading action with the least $Q^*(d, x_k)$ is selected among all K candidates as

$$\arg \min_{k \in 0,1,...,K} Q^*(d, x_k) \tag{8}$$

Based on the input d, this selected x^* will be output as a binary offloading decision.

B. Deep learning

We save the best offloading decision as a new entry of labeled data, (d, x^*), in the finite-size memory structure, where the oldest entry is discarded when the memory is full. Following the generation of the labeled data, all K DNNs are trained as illustrated in Fig. 6.

As opposed to training DNNs with the entire memory of data, the proposed algorithm uses an experience relay technique [4] [7] [8]. Each DNN extracts a batch of data samples from the same memory and all DNNs share the same memory. Gradient descent is used to optimize the parameter values θ_k of each DNN by minimizing cross-entropy loss, as follows:

$$L(\theta_k) = -x^T log(f_{\theta_k}(d)) - (1-x)^T log(1 - f_{\theta_k}(d)) \tag{9}$$

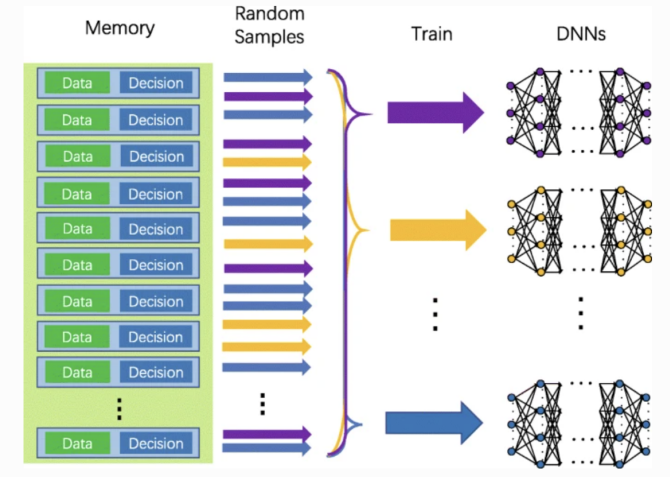

Fig. 6. K DNNs are separately trained with randomly sampled data entries

As a result of the finite-size design of the memory structure, new data entries are preferred over older ones, which improves data efficiency. To further speed up the training process, one can use advanced techniques such as distributed importance sampling [9] [10] and prioritized experience replay [8] [11].

The proposed algorithm 7 for MEC networks is shown in Algorithm 1, which is implemented using TensorFlow [12]. Initially, K DNNs are initialized with random parameter values θ_k and the memory is empty. By choosing a $K \geq 2$ value, the algorithm should converge to the optimal offloading actions.

Input: Input different workloads $\boldsymbol{d}_t$ at time t
Output: Optimal offloading decision $\boldsymbol{x}_t^*$ at time t
Initialization:
  Initialize the K DNNs with random parameters θ_k, $k \in \mathcal{K}$;
  Empty the memory structure
for $t = 1, 2, \ldots, G$ **do**
    Replicate different workloads $\boldsymbol{d}_t$ to all K DNNs.
    Generate k-th offloading action candidate $\boldsymbol{x}_k$ from the k-th DNN in a parallel way, as $\boldsymbol{x}_k = f_{\theta_{k,t}}(\boldsymbol{d}_t)$;
    Solve all K bandwidth allocation optimization (P2) in a parallel way when the offloading action candidates $\{\boldsymbol{x}_k\}$ are present;
    Select the best offloading decision as the output $\boldsymbol{x}_t^* = \arg\min_{k \in \mathcal{K}} Q^*(\boldsymbol{d}_t, \boldsymbol{x}_k)$;
    Store $(\boldsymbol{d}_t, \boldsymbol{x}_t^*)$ into the memory structure;
    Randomly Sample K batches of training data from the memory structure;
    Train the DNNs and update $\theta_{k,t}$;
end for

Fig. 7. Algorithm

VI. Cross Platform Profiling

In this section, we discuss the platform (Bare metal machines, VMs, Kubernetes cluster) best suited for carrying out the model training. Additionally, in order to test the algorithm in real life scenarios, we discuss the inference performance in different low resource settings. This simulates the edge devices which will run the inference model locally. We discuss our experimental findings in the subsection B of section VIII.

A brief introduction to the different platforms is presented below:

- Bare metal machine: In this case, instructions are executed directly on the logic hardware without an intervening operating system. From elementary systems to complex, highly sensitive systems incorporating many services, modern operating systems have evolved through various stages. After the development of programmable computers (which did not require physical changes to run different programs) but before the development of operating systems, sequential instructions were executed by machine language directly on the hardware without any system software layer. [19]
- Vitual machine: Virtual machines, commonly called VMs, are no different than physical computers like laptops, smartphones, and servers. If necessary, it can connect to the internet and is equipped with a CPU, memory, and disks to store your files. In contrast to the actual parts of your computer (called hardware), virtual machines (VMs) exist only as code and are considered virtual computers or software-defined computers within physical servers. In the virtualization process, a computer is placed into a software-based "virtual" environment where CPU, memory, and disk space are "borrowed" from a physical host computer -like your personal computer - and/or a remote server -such as a datacenter server of a cloud provider. Usually referred to as an image, a virtual machine is a computer file that behaves like a real computer. Many work computers run in a separate window, often for the purpose of running an entirely different operating system-or even to serve as the entire computer experience for the user. A virtual machine is partitioned from the rest of the system, so the software running inside it can't interfere with the computer's primary operating system. In spite of the fact that virtual machines run like individual computers with individual operating systems and applications, they remain completely independent of each other and of the physical host machine. Different operating systems can be run on different virtual machines simultaneously using a software called a hypervisor, or virtual machine manager. For instance, this makes it possible to run Linux VMs on a Windows OS, or to run an earlier version of Windows on a more recent Windows OS. Furthermore, because virtual machines are independent of one another, they're extremely portable. The ability to move a VM on a hypervisor to a different hypervisor on a completely

different machine is almost instantaneous. [20]

- Kubernetes cluster: Google created Kubernetes, which is a platform/tool that is open-source. It is written in GO-Lang. Currently, Kubernetes is an open-source project licensed under the Apache 2.0 license. Sometimes in the industry, Kubernetes is also known as "K8s". One can run any Linux container across public, private, and hybrid clouds with Kubernetes. Besides load balancers, service discovery, and role-based access control, Kubernetes also provides some edge functions. Container management is the answer. In a production environment that uses a microservice pattern with many containers, a number of things need to be taken care of. For example, health checks, version control, scaling, and rollback mechanisms. The process of making sure all of these things still work will be very frustrating. With Kubernetes, you can orchestrate and manage containers at scale. Using Kubernetes orchestration, you can schedule containers across multiple clusters, scale up and down those containers, and manage the health of those containers over time. It is manager's responsibility to ensure that subordinates do what they are supposed to do. In order to deploy a new microservice using Kubernetes, you must prepare your infrastructure first. A large team would need to script deployment workflows manually without Kubernetes. The use of Kubernetes reduces the amount of time and resources spent on DevOps by removing the need to write deployment scripts manually.

A. Model Training

We have witnessed a paradigm shift in infrastructure technologies from physical servers to virtual machines (VMs) to containers. VMs have done a great job in the last decade or so, but containers offer some inherent advantages over VMs. Containers are also an excellent choice for running edge workloads.

Hypervisors (software or firmware layers) run each virtual machine's operating system individually, leading to 'hardware-level virtualization.' Conversely, containers run on top of physical infrastructures and share a kernel, leading to 'OS-level virtualization.

Using a shared OS, containers are kept in MBs, making them extremely "light" and agile, reducing startup time to a few seconds rather than several minutes for a VM. The OS administrator's management tasks (patching, upgrades, etc.) are also reduced since containers share the same OS. Alternatively, kernel exploits will bring down the entire host in the case of a container. Even so, a VM is a better alternative if an attacker routes through the host kernel and hypervisor before reaching the VM kernel.

There is a lot of research being done today to bring the power of bare metal to edge workloads. Packet is one of those organizations that is pursuing this unique proposition of delivering low latency and local processing.

In a recent study, CenturyLink [22] examined the performance of Kubernetes clusters on both bare metal and virtual machines. The network latency of both clusters was measured using an open-source tool called netperf.

Since physical servers do not have hypervisors as overhead, results were as expected. Containers and Kubernetes running on bare metal achieved significantly lower latency; in fact, three times lower than running Kubernetes on virtual machines. Additionally, CPU consumption was considerably higher when a cluster was run on VMs than bare metal.

Databases, analytics, machine learning algorithms, and other data-intensive applications are ideal candidates for running containers on bare metal, but there are some advantages to running containers on virtual machines. If compared with bare-metal computing, VMs provide more out-of-the-box capabilities (such as moving workloads from one host to another, rolling back to a previous configuration in case of an issue, upgrading software, etc.) than bare metal. Therefore, containers, being lightweight and quick to start/stop, are ideal for edge workloads. When running on bare metal or a virtual machine, there is always a tradeoff.

Containers as a Service (CaaS) is provided by most of the public clouds, including Microsoft Azure and Amazon Web Services. Both are built on top of the existing infrastructure layer, based on VMs, thereby delivering portability and agility, which edge computing needs. As part of its Greengrass [23]8 initiative, Amazon has introduced a software layer that extends cloud-like capabilities to edges, enabling local collection and execution of data.

The Greengrass Group consists of two components. The first is the Greengrass core, which runs AWS Lambda, messaging, and security locally. The second is IoT, SDK-enabled devices that connect to Greengrass core through the local network. In the event that Greengrass core lost communication with the cloud, it would still communicate with other devices locally.

Containers are one of the hottest technologies for investigation because of their speed, density, and agility. Enterprises may face challenges adopting edge network solutions workloads using containers due to security concerns. These issues include:

- Denial of service: A single application may consume most of the OS resources, leaving others with insufficient resources and forcing a shutdown of the OS.
- Exploiting the kernel: Containers use the same kernel, so if an attacker were able to gain access to the host OS, they would be able to run all applications on the host.

Thus, it is important to compare the performance of the algorithm on different edge devices while training. Different training environments can lead to different convergence time because of the available resources and the way they are managed.

B. Model Testing

Generally, deep learning algorithms have been known to require a significant amount of computation power. In a real life scenario, a trained model will be used for inference on the edge devices. These devices generally don't have a high computing power, hence it becomes really important to

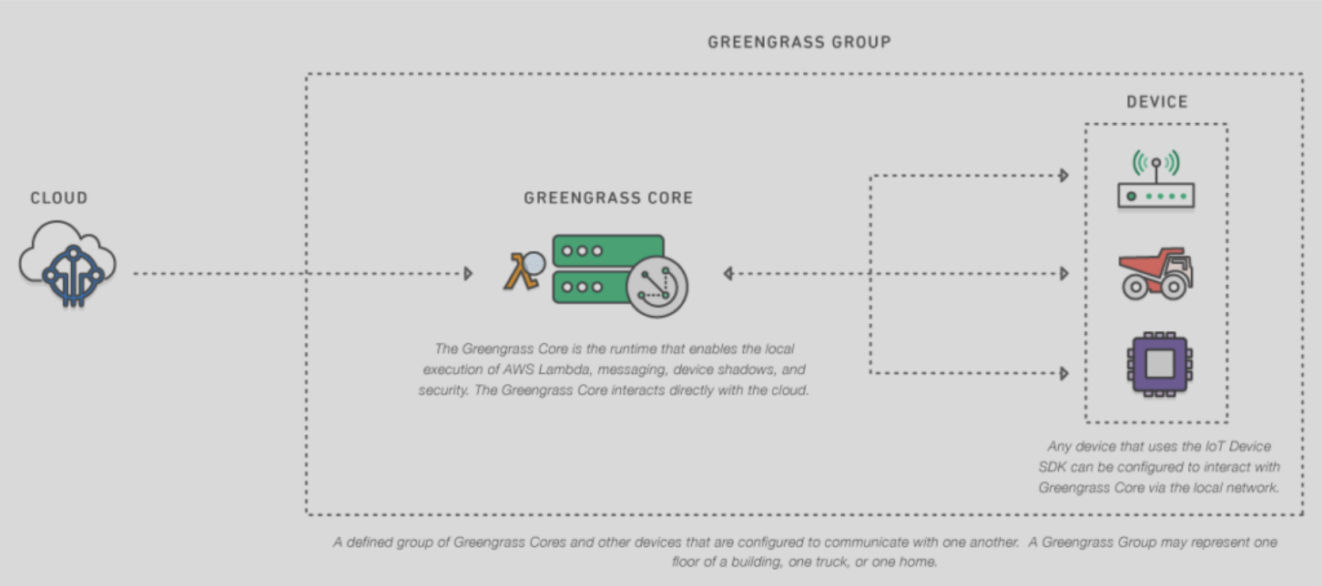

Fig. 8. AWS Greengrass

profile the performance of the inference phase on different devices devices with low computing resources.

VII. Experimental Setup

A. Assumptions

Following are the assumptions of our training model.

Number of wireless devices	3
Number of tasks per device	3
Local Computation time of mobile devices	4.75 x 10^{-7} s/bit
Processing energy consumption	3.25 x 10^{-7} J/bit
Uplink bandwidth limit	150 Mbps
Weight of the energy consumption at edge server (α)	1.5 x 10^{-7} J/bit
Weight of time delay (β)	1 J/sec

These values are assumed for the performance evaluation on the NYU HPC Cluster in VIII-A. We then also change the values of weights α and β and observe the effect it has on the cost.

B. Machine Configurations

- **NYU HPC Cluster**
  Model Name: Intel(R) Xeon(R) Platinum 8268 CPU
  CPU Frequency: 2.90 GHz
  Number of CPUs: 4
  Threads Per Core: 1
- **IBM Virtual Machine**
  Model Name: Intel Xeon Processor (Cascadelake)
  CPU Frequency: 2.40 GHz
  Number of CPUs: 4
  Threads Per Core: 2
- **Bare-Metal**
  Model Name: 11th Gen Intel(R) Core(TM) i5-1135G7
  CPU Frequency: 2.40 GHz
  Number of CPUs: 8
  Threads Per Core: 2
- **Kubernetes**
  Model Name: Intel Xeon Processor (Cascadelake)
  CPU Frequency: 2.40 GHz
  Number of CPUs: 4
  Threads Per Core: 2

We experiment and compare the different training and inference times on the above mentioned configurations.

VIII. Result and Analysis

We have divided this section in two parts. First, we show the numerical results for our proposed algorithm for solving Problem (P1). Second, we analyze the cross platform performance on two aspects: training and inference.

A. Performance evaluation

1) Convergence performance: Fig. 9 shows the effects of number of DNNs on the convergence of our algorithm. Because the system utility `Q(d, x, c)` varies under different input `d`, we introduce a greedy method as a benchmark, which generates the optimal system utility by enumerating all 2^{NM} binary offloading decisions, as:

$$\max_{x' \in \{0,1\}^{NM}} Q^*(d, x_k) \tag{10}$$

In order to compare our algorithm with the greedy method, we plot their gain ratios in Fig. 9. Our algorithm converges to 1 with the increase of learning steps. When 5 DNNs are used,

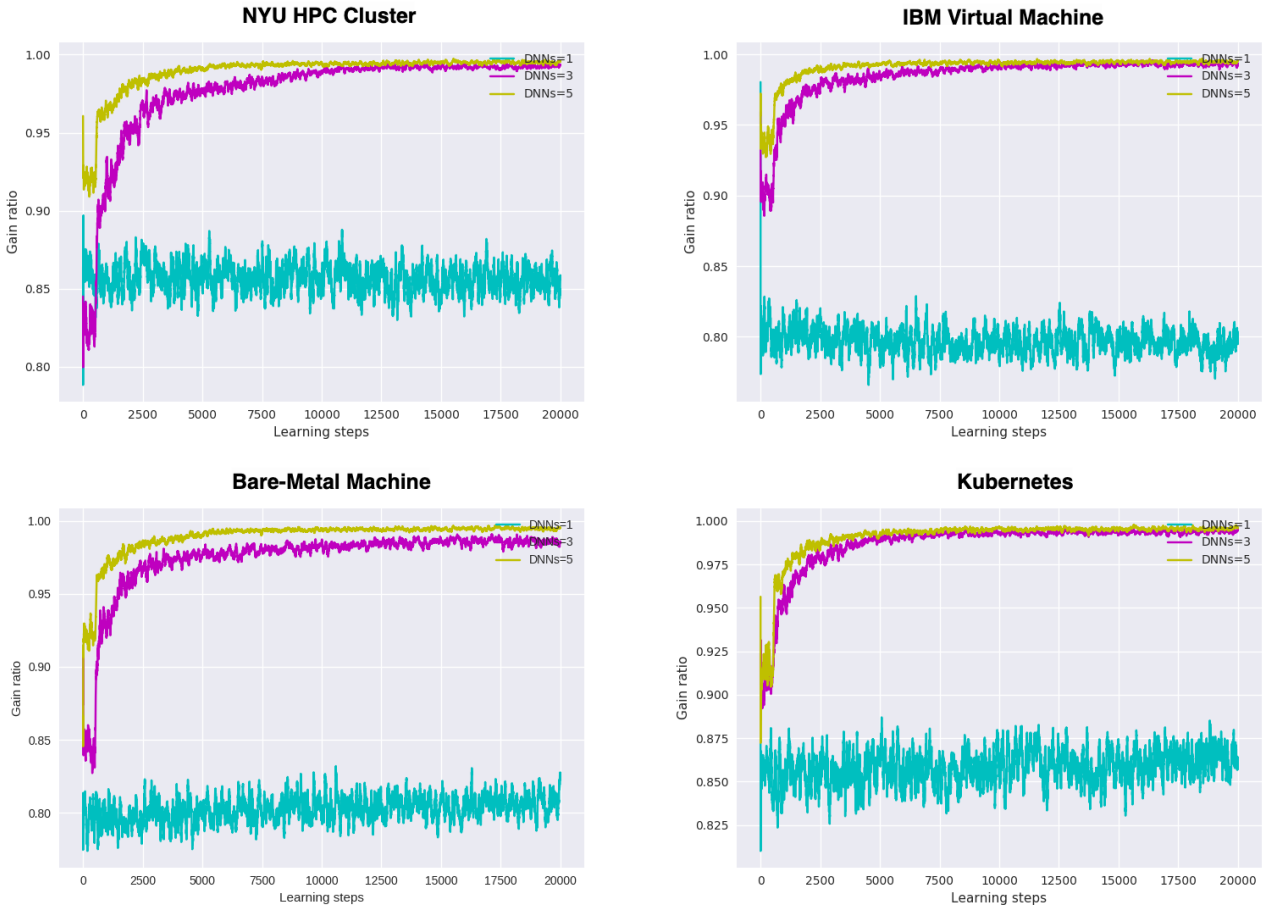

Fig. 9. Convergence performance under different learning rates - on different platforms INCLUDE ALL 4 GRAPHS here

a gain ratio of 0.98 is achieved within 1000 learning steps. The algorithm cannot converge when only one DNN is used, since it does not learn anything from the data it generates. Therefore, at least two DNNs are required. With more DNNs, the algorithm will converge faster.

As expected, the convergence is not dependent on the training environment. The algorithm converges at around the same point in all the training environments.

We study the convergence of our algorithm under different learning rates and different memory sizes in Fig. 10 and Fig. 11, respectively. Faster convergence of the algorithm occurs with a higher learning rate. Due to the higher learning rate, it is possible to obtain a local optimal rather than a global optimal solution. It is thus important to select a learning rate based on the specific situation. We study the effects of different memory sizes in Fig. 11. The smaller the memory size, the faster the convergence, but it may fall into local optimum. As a trade-off between convergence and performance, we select an experience replay memory with a size of 1024 for the considered MEC network.

2) System utility: The performance of our proposed method is compared with that of the other three schemes, specifically

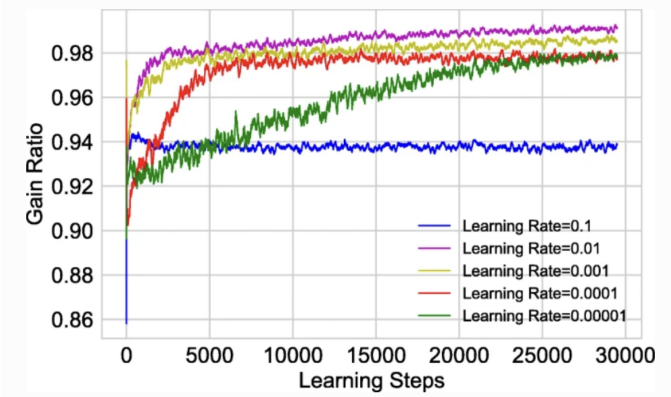

Fig. 10. Convergence performance under different learning rates

the local processing only scheme, the edge processing only scheme, and the greedy scheme. The performance is studied under different α and β parameters in Fig. 12 and Fig. 13, respectively. In the local processing only scheme, all users' tasks are handled locally. In the edge processing only scheme, the edge processes all users' tasks. We select the best

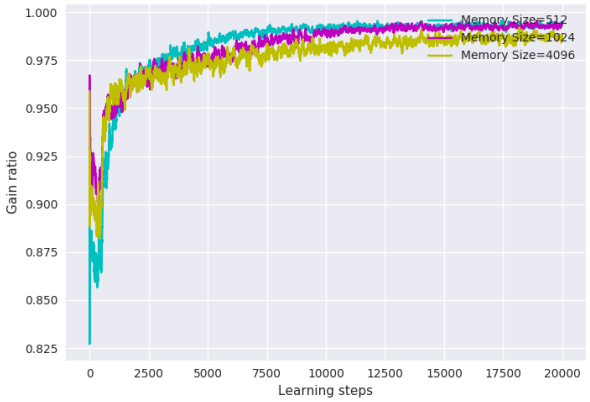

Fig. 11. Convergence performance under different memory sizes

offloading decision combination from all offloading decision combinations for the greedy scheme. The greedy scheme, however, is very time-consuming, especially when lots of users and tasks are involved. As illustrated in Fig. 12 and Fig. 13, as α or β increases, the total energy cost of all schemes increases. The performance of our method is close to the greedy scheme.

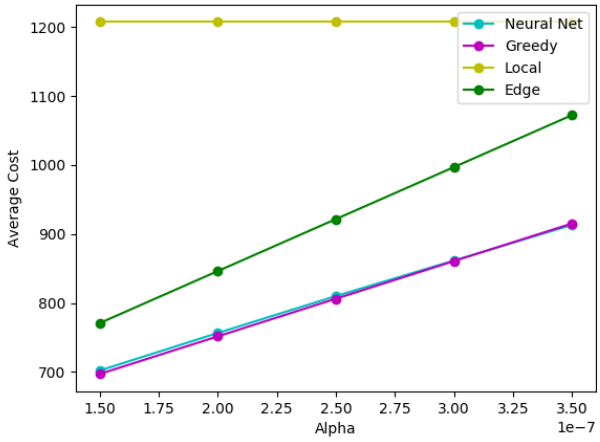

Fig. 12. Energy cost under different α (Joules/bit) for different offloading algorithms

3) Computation Time: Fig. 14 shows the computation time of the algorithm under different numbers of DNNs. With different numbers of DNNs, the inference time for each input is almost identical, around 0.1 second for an MEC network with 3 WDs with 3 tasks. Increasing the number of DNNs improves the convergence performance of the algorithm, as shown in Fig. 9. Based on the computing hardware at the edge server, we can deploy the algorithm with multiple parallel DNNs. Currently, all those server-oriented CPUs support multiple cores, e.g. a single Intel Xeon W-2195 processor has 18 cores. Therefore, the algorithm can be implemented on multiple cores to increase its performance.

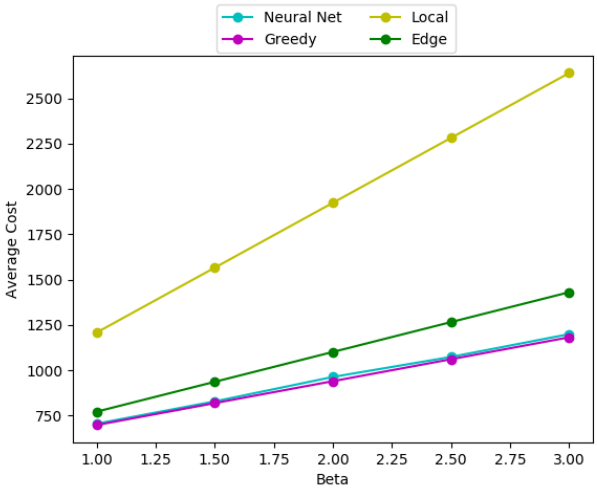

Fig. 13. Energy cost under different β (Joules/sec) for different offloading algorithms

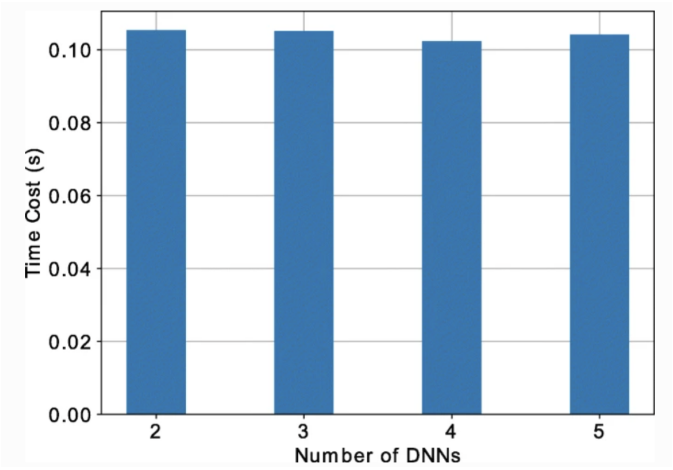

Fig. 14. Inference Time cost for each input under different number of DNNs (Parallel)

B. Cross Platform Performance

In this section, we talk about the practicality of the algorithm. We profile the training and inference times on different environments and present a detailed analysis of the same.

1) Model Training: We compare the training time across different environments, namely, HPC cluster, Kubernetes cluster, Bare metal and Virtual machines in Fig. 15. The observations are:

- As the number of CPUs increase, there is not much decrease in the training time. This implies that there is no point in increasing the number of CPUs above a certain level as it will be a waste of resources.
- Kubernetes is slower than Virtual machines because of the communication and management overhead in Kubernetes.
- The training time for 3 and 5 DNNs is around 1hr15min. Thus, training is relatively faster compared to most of deep learning models as each DNN has just 2 fully-

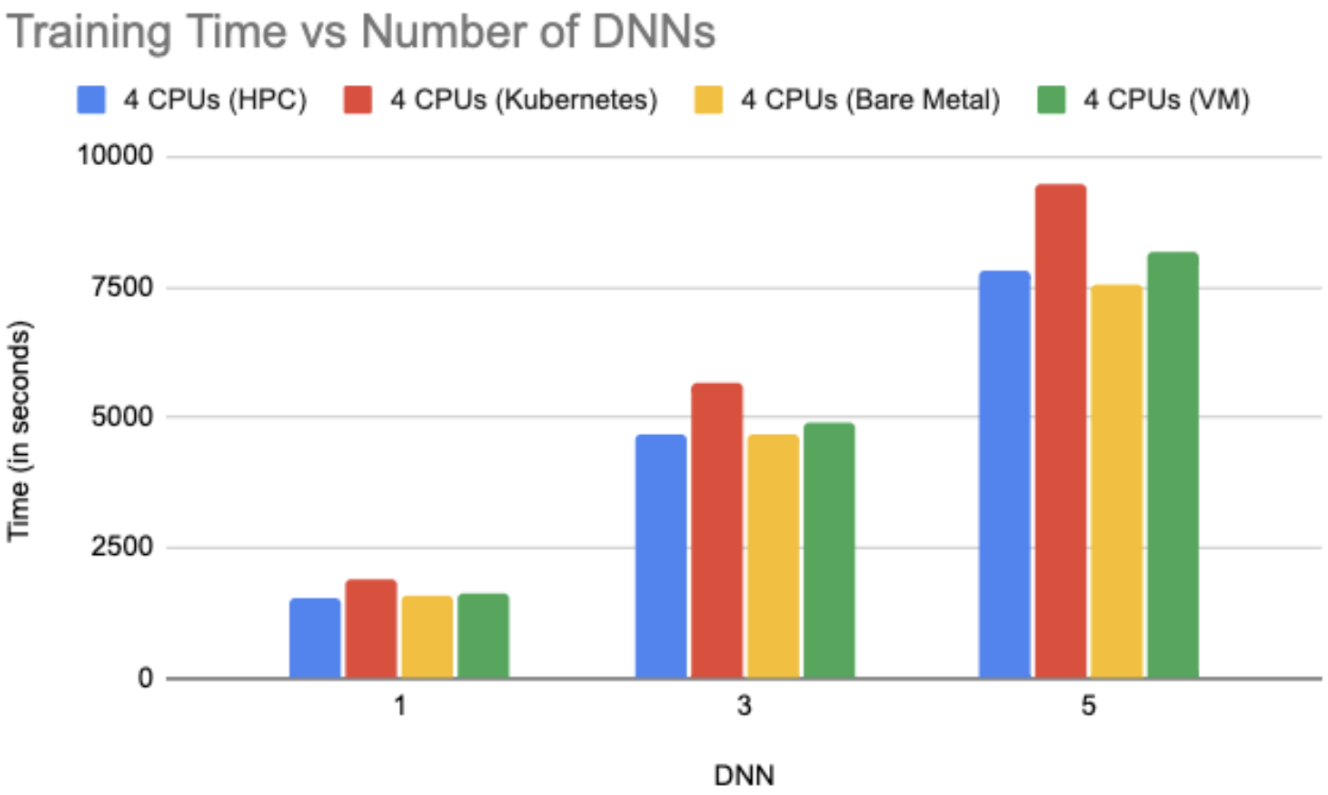

Fig. 15. Training time on different platforms (Serial)

connected hidden layers.

- If there are ample resources available, using more DNNs is a good option as it leads to a significant decrease in training time.
- If the DNNs can be run in parallel, it will bring the training time down by a significant amount thus making the training step scalable with respect to number of DNNs.

2) Model Testing: The offloading decision will be made at the edge device. The edge devices generally don't have a high computational power. Hence it becomes really important to study the inference performance of the algorithm on devices with low computational power. Fig. 16 shows the inference time on different platforms for different number of DNNs. Some of the observations are discussed below.

- With different numbers of DNNs, the inference time for each input increases with an increase in number of DNNs, around 0.1 second for an MEC network with 3 WDs with 3 tasks for 1 DNN, 0.3 seconds for 3 DNNs and 0.5 seconds for 5 DNNs. However, the inference time remains the same for all the DNNs when they're used in parallel - as shown in Fig. 14.
- Increasing the number of DNNs (when in sequential use) increases the accuracy but has a negative effect on the inference time.
- On all the platforms, the algorithm performs almost the same (for a given number of DNNs). This implies that the algorithm can be used on a variety of cloud services without any major difference in the performance. Hence, the solution is quite general.

IX. Challenges

This project involved extensive research. An efficient algorithm has been developed to solve the joint offloading problem in edge networks. Despite that, there is still a lot of work that can be done in order to improve the algorithm's performance in real-life applications. We share some thoughts and observations here.

- Profiling an algorithm, besides its resource consumption, can yield interesting insights into its assumptions.
- Kubeflow [25] is great! It is not necessary to write yaml(s) for every job. Just run your Jupyter notebooks on the Kubeflow cluster.
- The installation of utilities and libraries on vanilla VMs can be a challenge.
- As research on various edge computing costs was involved, it was challenging to come up with a cost function for the DNNs.

X. Conclusion

Using distributed deep learning, we have developed an offloading algorithm for MEC networks to minimize system utility regardless of the amount of power consumed or the

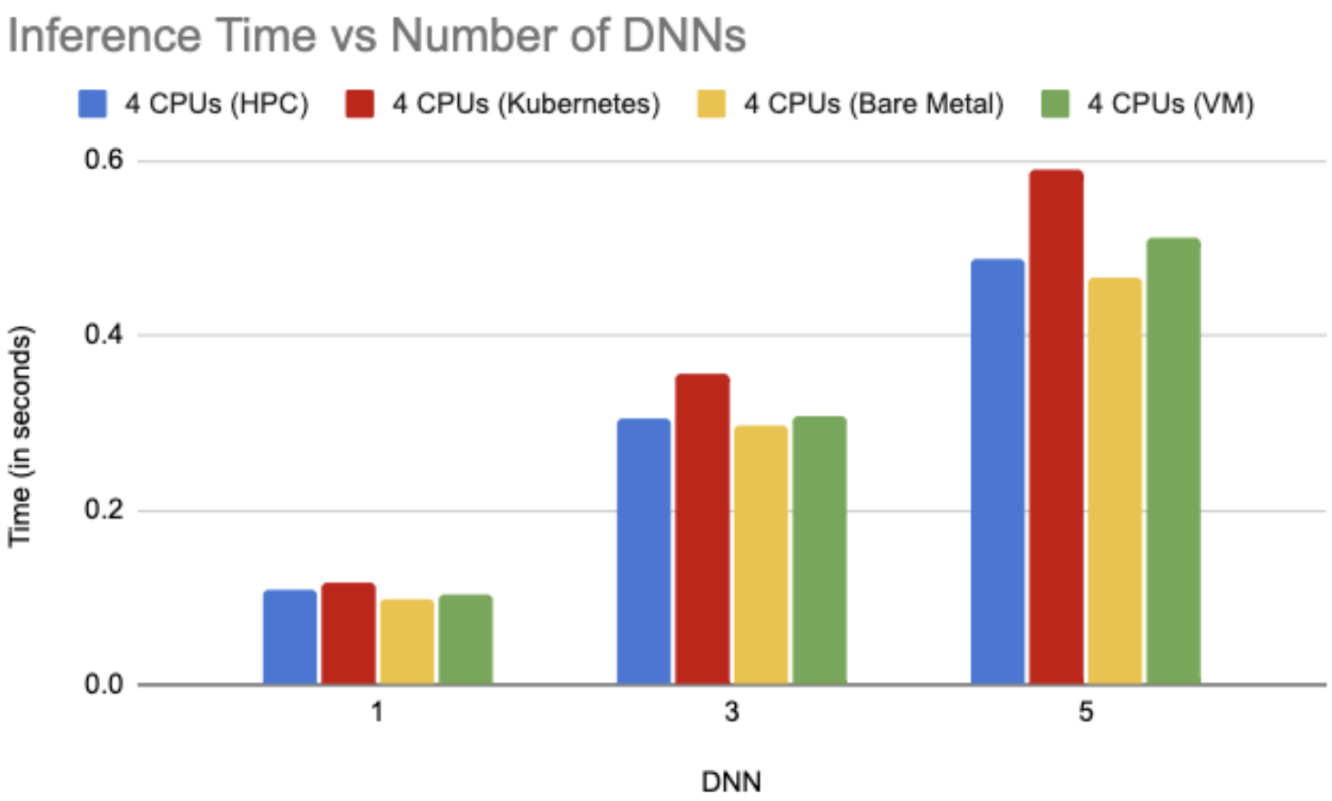

Fig. 16. Inference time on different platforms (Serial)

delay in completing the task. By utilizing multiple DNNs, the algorithm generates close-to-optimal solutions without manually labeled data. The proposed algorithm was validated by numerical results in terms of accuracy and its performance advantage over greedy algorithms. Moreover, the proposed algorithm is capable of generating near-optimal offloading decisions in less than one second, regardless of the number of DNNs in the network. In the future, this distributed deep learning framework could be extended to optimize real-time offloading for MEC networks.

The algorithm works well in a variety of environments. The inference time stays almost the same for all the environments implying that the inference model can be deployed on different kind of environments without any significant performance difference. Scalability of the model with respect to the number of DNNs can be increased by training and inference in parallel. For future work, profiling the inference algorithm on edge devices with minimal computation power will lead to insightful results and observations. Also, running the algorithm on edge simulation frameworks will validate the results of the algorithm in a real-life scenario.

REFERENCES

[1] Chiang M, Zhang T (2016) "Fog and IoT: an overview of research opportunities". IEEE Internet J 3(6):854–864

[2] Mao Y, You C, Zhang J, Huang K, Letaief K (2017) "A survey on mobile edge computing: the communication perspective". IEEE Commun Surv Tutorials 19(4):2322–2358

[3] Abbas N, Zhang Y, Taherkordi A, Skeie T (2018) "Mobile edge computing: a survey". IEEE Internet J 5(1):450–465

[4] Mnih V, Kavukcuoglu K et al (2015) "Human-level control through deep reinforcement learning". Nature 518(7540):529–533

[5] You C, Huang K, Chae H, Kim B (2017) "Energy-Efficient Resource Allocation for Mobile-Edge Computation Offloading". IEEE Trans Wirel Commun 16(3):1397–1411

[6] Zhang H, Liu H, Cheng J, Leung MCV (2018) "Downlink Energy Efficiency of Power Allocation and Wireless Backhaul Bandwidth Allocation in Heterogeneous Small Cell Networks". IEEE Trans Commun 66(4):1705–1716

[7] Lin LJ (1993) "Reinforcement learning for robots using neural networks", Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Technical Report

[8] Horgan D, Quan J, Budden D, Barth-Maron G, Hessel M, van Hasselt H, Silver D (2018) "Distributed prioritized experience replay". arXiv:1803.00933

[9] Loshchilov I, Hutter F (2015) "Online batch selection for faster training of neural networks". arXiv:1511.06343

[10] Alain G, Lamb A, Sankar C, Courville A, Bengio Y (2015) "Variance reduction in SGD by distributed importance sampling". arXiv:1511.06481

[11] Schaul T, Quan J, Antonoglou I, Silver D (2016) "Prioritized experience replay. In International conference on learning representations" (ICLR)

[12] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S (2016) "Tensorflow: large-scale machine learning on heterogeneous distributed systems". arXiv:1603.04467

[13] Chen X, Jiao L, Li W, Fu X (2016) “Efficient multi-user compu- tation offloading for mobile-edge cloud computing”. IEEE/ACM Trans Netw 24(5):2795–2808
[14] Bi S, Zhang Y (2018) “Computation rate maximization for wire- less powered mobile-edge computing with binary computation offloading”. IEEE Trans Wirel Commun 17(6):4177–4190
[15] Guo S, Xiao B, Yang Y, Yang Y (2016) “Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing”. In: IEEE international conference on computer communications, pp 1–9
[16] Phaniteja S, Dewangan P, Guhan P, Sarkar A, Krishna K (2017) “A deep reinforcement learning approach for dynamically stable inverse kinematics of humanoid robots”. In: 2017 IEEE international conference on robotics and biomimetics (ROBIO), Macau, pp 1818–1823
[17] Sharma A, Kaushik P (2017) “Literature survey of statistical, deep and reinforcement learning in natural language processing”. In: International conference on computing, communication and automation, Greater Noida, pp 350–354
[18] Mnih V, Kavukcuoglu K et al (2015) “Human-level control through deep reinforcement learning”. Nature 518(7540):529–533
[19] “Bare Machine”, https://en.wikipedia.org/wiki/Bare_machine
[20] “What is a virtual machine (VM)?”, https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/
[21] “Step by Step Introduction to Basic Concept of Kubernetes”, https://medium.com/easyread/step-by-step-introduction-to-basic-concept-of-kubernetes-e20383bdd118
[22] “Kubernetes on Bare Metal: When Low Network Latency is Key” https://www.ctl.io/developers/blog/post/kubernetes-bare-metal-servers
[23] “AWS IoT Greengrass” https://aws.amazon.com/greengrass/
[24] “Scipy” https://scipy.org/
[25] “Kubeflow” https://www.kubeflow.org/