Energy Efficient Computation Offloading with DVFS using Deep Reinforcement Learning for Time-Critical IoT Applications in Edge Computing

Saroj Kumar Panda, Man Lin, and Ti Zhou

***Abstract*—Internet of Things (IoT) is a technology that allows ordinary physical devices to collect, process, and share data with other physical devices and systems over the internet. It provides pervasively connected infrastructures to support innovative applications and services that can automate otherwise intensely laborious manual effort. Edge computing (EC) complements the powerful centralized cloud servers by providing powerful computation capability close to the data source, minimize communication latency, and securing data privacy. The energy consumption problem has continued to receive a lot of attention from the IoT community in applying various techniques to reduce energy consumption while still meeting the computational demand. In this paper, we propose an application-deadline-aware data offloading scheme using deep reinforcement learning and Dynamic Voltage and Frequency Scaling (DVFS) in an edge computing environment to reduce the energy consumption of IoT devices. The proposed scheme learns the optimal data distribution policies and local computation DVFS frequency scaling by interacting with the system environment and learning the device, the network, and edge servers' behavior. The proposed scheme was tested on multiple edge computing environments with different IoT devices. Experimental results show that this scheme can reduce energy consumption while achieving the IoT application and services timing and computational goals. The proposed scheme has substantial energy savings when compared with the native Linux governors.**

***Index Terms*—Offloading, Dynamic Voltage and Frequency Scaling, Edge Computing(EC), Energy Consumption, Edge Server, IoT, Deep Reinforcement Learning.**

I. Introduction

Internet of Things (IoT) is a technology that allows ordinary physical devices to collect, process, and share data with other physical devices and systems over the internet. IoT is changing the way these devices can be used to interact with the environment around them to improve the quality of day-to-day life through many smart systems such as smart cities, smart health care, and smart homes, etc[19, 18]. Traditionally IoT devices offload the data to centralized cloud servers, which have an abundance of computational resources available. The cloud servers perform the computation and send back the results to the IoT devices to take appropriate actions. This approach suffers from several shortcomings such as network latency, higher network bandwidth requirement, privacy and security concerns. To address these challenges, edge computing (EC) [23] moves the computation close to the data source, i.e., edge of the network. Edge servers are placed in close proximity to IoT edge devices. The edge devices can offload data to edge servers using various communication mediums such as Bluetooth, Local Area Network, etc., preventing the data from traversing through any public domain and thereby preserving privacy and reducing the risk of security threat to the data. The computation results from edge servers become available to edge devices faster, enabling them to provide a real-time response experience to the end-user.

With the evolution in the development of compact IoT devices, today's IoT devices have computing power of their own and memory to store data enabling EC to move the computation directly to the data source. However, compared with desktops or servers, IoT devices tend to have limited computation capability. In addition, since IoT devices are deployed in the field, they are normally powered by built-in batteries. Their lifetime, therefore, depends on the computation intensity of the executed applications. With these limitations, running computation-intensive or delay-sensitive applications only on the IoT devices may not meet the application requirement and may reduce the lifetime of the battery.

Offloading is considered as one of the solutions to the above problems, which transfers some of the computation to other nearby systems such as edge servers, Fog Access Points, etc. A variety of different offloading schemes have been proposed [21, 6, 13, 12, 14]. These schemes take advantage of high bandwidth local communication mediums and close proximity resource-rich servers to offload computation, enabling the delay-sensitive applications to complete on time and increase IoT device battery life. But, the core challenge lies in how to jointly optimize computation, communication, energy consumption and make a controlled computation offloading decision, which is affected by many factors, such as application computation characteristics, communication medium condition, offloading energy consumption, and edge server computing resource utilization. If an offloading decision does not take into account the variations in these factors, it will lead to poor performance. For example, poor communication network conditions or heavily used edge server computing resources or both will hurt the benefits of offloading and may lead to more energy consumption and computation time. In this scenario, performing more computation on the local IoT device is a better option.

Deep Learning (DL) using Deep Neural Networks (DNN) has become an emerging technology for IoT applications and systems [17, 11] to extract meaningful information from the application data and assist in taking intelligent actions. One advantage of DL is that these DL models can be trained offline using high computational resources such as cloud servers or edge servers and deployed online on IoT devices to perform inference on real-time application data. This is based on the fact that DNN requires less computational resources during inference than in the training phase. Another advantage of DL is that these DL models are highly data-parallel, i.e. inference can be done in parallel with multiple instances of the DL model deployed on different edge computing devices for separate sets of data. We take advantage of this data parallelism to perform a simpler computation offload by offloading part of the data from the IoT device to nearby edge servers. Both the IoT device and the edge server execute the same computation task on separate sets of data.

In recent years, many Reinforcement Learning (RL) based offloading schemes have been proposed [16, 6], where IoT devices learn the best offloading policy by interacting with the application environment. Instead of solving a complex problem of what to offload, IoT devices adjust the offloading policy continuously in a trial-and-error manner by interacting with the system till the best policy is found without the need for prior system knowledge, which

This work was supported by the Natural Sciences and Engineering Research Council of Canada

Saroj Kumar Panda, Man Lin, and Ti Zhou are with St. Francis Xavier University (mlin@stfx.ca)

is often unavailable in practice. Another effective approach to reduce energy consumption is Dynamic Voltage and Frequency Scaling (DVFS), which is one of the most promising solutions to optimize processor energy consumption and control system temperature where the processor frequency and voltage are adjusted to lower levels to reduce power consumption while still meeting the task completion deadlines [1]. Most of the modern processors and operating systems support DVFS.

In this paper, we present a DVFS aware deep reinforcement learning (DRL) based controlled Data Offloading (DRLDO) scheme in an EC environment that jointly learns the optimal computation and communication characteristics of the whole system to control the computation speed of the local IoT device and minimize energy consumption while still satisfying the computation time constraints of the IoT application. The contributions of this paper are as follows:

- A DVFS aware Deep Reinforcement Learning-based offloading scheme that minimizes IoT device energy consumption by jointly optimizing partial data offloading ratio, and IoT device CPU frequency based on the current network, edge server states, and the IoT application deadline.
- A new method of measuring energy consumption of IoT devices such as Raspberry Pi [4] and Nvidia Jetson Nano [2].
- Comparison of results of this scheme using various devices as IoT device such as Raspberry Pi, Nvidia Jetson Nano, and Linux Laptop etc.

II. RELATED WORK

In recent years, **EC** and DVFS have been gaining popularity; EC with IoT applications that require real-time response and DVFS in modern devices to save energy. Many energy-efficient edge task scheduling and computation offloading approaches have been proposed.

Zhang et al. [26] proposed a deep reinforcement learning-based DVFS technique selection scheme for energy-efficient edge scheduling on edge devices. But, as the computation by edge devices grows extensively and with more stringent IoT application completion time constraints, it is imperative to offload some computation to the servers.

Li et al. [12] proposed a deep reinforcement learning (DRL) offloading approach to jointly optimize the offloading decision and resource allocations for a multi-user MEC system, where multiple user equipment (UEs) can perform computation offloading via wireless channels to a Mobile edge computing (MEC) server to minimize the sum of delay cost and energy consumption for all UEs.

Huang et al. [6] proposed a real-time reinforcement learning (Double Q-learning) based offloading scheme, RRLO, that jointly optimizes task scheduling and offloading for real-time applications in a MEC environment. RRLO offloads part of the computational tasks to MEC servers and selects the proper DVFS scheme dynamically for the tasks executed locally to minimize the energy consumption of the mobile device.

Lin et al. [13] proposed an algorithm for the MCC (Mobile Cloud Computing) task scheduling to minimize the total energy consumption of an application in a mobile device under the hard constraint of application completion time by representing the task schedule as a DAG (Directed Acyclic Graph). They follow a multi-step approach. In the first step, they generate a minimal-delay task scheduling graph. In the second step, energy reduction is performed by migrating tasks towards the cloud or other local cores available that can bring energy reduction without violating the application completion time constraint. They apply the DVFS technique to reduce energy consumption in the third step further.

Ren et al. [21] proposed a deep reinforcement learning (DRL) based scheme to minimize long-term system energy consumption by offloading computation to Fog access points (F-APs) and to the cloud in a multiple IIoT devices scenario. They create a DRL model for each IIoT device that identifies its computation offload serving F-AP based on network and device states. To scale for any growing computation demand, they execute a low complexity greedy algorithm at each F-AP to determine which offloading requests can further be forwarded to the cloud.

Generally, reducing energy consumption using the DVFS technique highly depends on selecting the right CPU core voltage and frequency according to the varying system states. To reduce energy consumption, many reinforcement learning-based methods have been proposed to learn how to switch DVFS dynamically to adapt to different states. There are extensive researches in the literature. Islam and Lin [8] proposed a RL based DVFS algorithm selection approach that integrates a set of DVFS algorithms and Q-learning to reduce Real-Time Systems energy consumption named as Hybrid DVFS Scheduling. Zhang et al. [26] proposed a deep reinforcement learning-based DVFS technique selection scheme for energy-efficient edge scheduling on edge devices. However, as the data generated by IoT edge devices grows extensively and more restricted time constraints for the IoT applications, it is imperative to offload some of the data or computational tasks or both to the edge servers.

Most of the proposed offloading schemes above offload the computation by offloading tasks to servers, which itself requires computation to prepare the task precedence schedule and decision of task allocation. In this work, we propose a simpler computation offloading scheme by offloading only partial data required for computation to the edge servers with the same computation tasks available at both the IoT device and edge servers. We use trained DL models as IoT applications to validate the scheme.

Another difference is that the proposed works above use formula-based energy consumption estimations and simulation results to validate their proposed scheme and often only estimate processor energy consumption, which may differ significantly from the system energy consumption of the IoT device. Apart from the processor energy, other system components such as memory, sensors, etc. also consume energy as well as running other software components on the system. In our proposed scheme, we use real devices and optimize the whole device system energy.

III. SYSTEM MODEL

In this section, we present the details of the communication model, offloading model, and energy consumption model adopted in the system under investigation.

A. Communication Model

Generally, IoT devices and edge servers are connected to the internet either through wireless mode (Wi-Fi) or through the physical mode (Ethernet). The IoT device offloads the data to the edge server using TCP/IP over the Local Area Network (LAN). However, it is also possible to use other communication mediums such as Bluetooth and Wi-Fi Direct, etc. The data transmission time (U_t) depends on the communication medium data transmission rate (r_u) and the number of bits in the data (d_n) as given in the following equation:

$$U_t = \frac{d_n}{r_u} \tag{1}$$

The communication medium data transmission rate may vary over time depending on the communication medium state. Our proposed scheme takes into account the variation of data rates while selecting the best policy.

B. Offloading Model

The data d can be divided into two parts (2); where d_e is the data processed by the IoT device and d_s is the data processed by the edge server.

$$d = d_e + d_s \tag{2}$$

If the data is offloaded to multiple servers then d_s will be sum of the data that is offloaded to all the servers (3), where d_{s_i} is the data offloaded to *ith* server and *n* is the number of edge servers used.

$$d_s = \sum_{i=1}^{n} d_{s_i} \tag{3}$$

Let T_{s_i} be the processing time of the *ith* server for the data d_{s_i}. Using (1) the response time of the *ith* server (R_{s_i}) will be as follows.

$$R_{s_i} = U_{t_i} + T_{s_i} \tag{4}$$

where U_{t_i} is the transmission time of the data d_{s_i} sent to the *ith* edge server.

Let T_e be the processing time of the data d_e processed locally, the response time R of the application can now be computed as maximum of the local processing time and response times of all the servers.

$$R = \max[T_e, R_{s_1}, R_{s_2}...R_{s_n}] \tag{5}$$

where R_{s_1}, R_{s_2}, ... , R_{s_n} are response times of respective edge servers.

If T is the expected response time of the application for the data d, the goal of offloading is to satisfy the condition (6)

$$\max[T_e, R_{s_1}, R_{s_2}...R_{s_n}] \leq T \tag{6}$$

C. Energy Consumption Model

The objective of our proposed approach is to minimize the total energy consumption of the IoT device, which consists of two parts, dynamic energy consumption, and static energy consumption. The static energy consumption is due to the leakage current that happens continuously even if the CPU is idle. The dynamic energy is consumed for computation and communication. Computation energy depends on the CPU core voltage (V), frequency (f), and execution time (t) (8). The communication energy depends on the amount of data transferred (d_n), transmission rate (r_u), and the transmission power (q_t) as given in the equation (7).

$$E_t = \frac{d_n \cdot q_t}{r_u} \tag{7}$$

$$E_c \propto V^2 \cdot f \cdot t \tag{8}$$

The total energy consumed by the application task (E) can be calculated by the following equation.

$$E = E_c + E_l + E_t \tag{9}$$

where E_c, E_l, and E_t are the energy consumed by the IoT device for computation, leakage current and communication respectively. From (9), and (6) the energy optimization problem can be formulated as follows:

$$\begin{aligned} &\text{Minimize} \quad (E_c + E_l + E_t) \\ &\text{s.t.} \qquad\quad \max[T_e, R_{s_1}, R_{s_2}...R_{s_n}] \leq T \end{aligned} \tag{10}$$

IoT device local execution time and computation energy both depend on the CPU frequency and the amount of data processed locally. The response times of servers and communication energy depend on the data distributed to the servers, network conditions, and server resource availability. Therefore finding an appropriate data offloading policy and local CPU frequency based on the current network and server state is critical to meet the application time constraint and energy optimization.

IV. DEEP REINFORCEMENT LEARNING BASED DATA OFFLOADING

In this section, we first present a brief introduction of deep reinforcement learning (DRL) in general. Then, we discuss the steps involved in the proposed offloading scheme **DRLDO**.

A. DEEP REINFORCEMENT LEARNING

Reinforcement Learning is an autonomous learning process where the learning agent learns the optimal policy through knowledge obtained by trial-and-error and continuously interacting with a dynamic environment [20]. The agent periodically takes actions in the environment and gradually improves the strategy of action selection by processing the feedback received from the environment until the optimal policy is achieved. In each period, the agent first observes the environment state s and then takes action a on the environment. For the action, the agent receives either a reward r or penalty p as the feedback from the environment and the environment moves to a different state s' for the next period. The agent repeats the above process to learn the optimal policy π that optimizes the long-term accumulative reward.

Q-learning is one of the simplest model-free RL algorithms that has been applied to various domains and applications [9, 8]. It learns the Q-Value *Q(s,a)* of any action in any particular state by taking that action. Iterative update of the Q-values are continued till the changes to Q-values falls below a threshold value where the Q-table converges and the Q-learning algorithm training can be stopped. However for Q-Learning to work the state-action space should be finite and Q-values for each state-action group should be computed. But, in practical applications, there may be thousands of states. Storing all possible state-action pair Q-values would take up a lot of memory. Therefore, instead of computing and storing each state-action pair Q-values, we use a deep neural network (DNN) to estimate the *Q(s,a)* Q-value. This is the primary idea of DRL. The DNN is trained by giving state as the input and action values as the target. The trained DNN can estimate the action values even for the states it had not seen before. In our proposed offloading scheme, we use energy consumption for any action taken as the corresponding action value.

B. SYSTEM DESIGN

The system design of the proposed offloading scheme is shown in Fig. 1. Given below are the key components and concepts of the system.

1) ***Network Interface:*** The network interface provides access to the communication medium to offload the data to the edge servers and receive responses from them.
2) ***Environment States :*** IoT applications have time constraints on processing the data, also known as the application deadline. The energy consumption action values will vary with respect to the application deadline. The availability of computation results for the data offloaded to the edge server will vary based on the communication network state and the edge server's computational resources availability state. We capture this information by computing the edge server response time. If an edge server has high-end computational resources and is idle

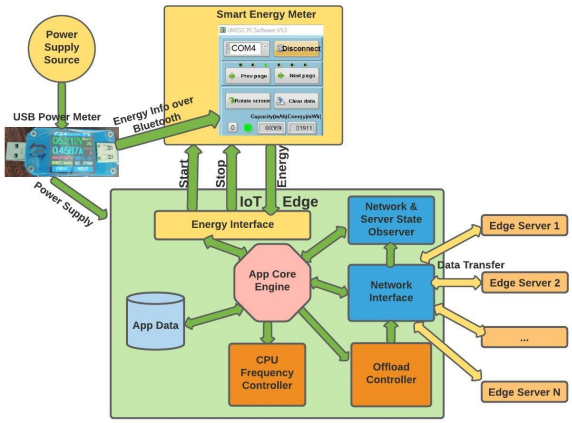

Fig. 1. System Design of Deep Reinforcement Learning based Offloading Scheme

then the processing will be faster and the response time would be smaller. However, if the server is overloaded with other tasks then the processing would be slower and the response time would be higher. Similarly, if the communication medium is idle response would be faster with lower response time compared to communication medium that is busy. Computation of response time is implementation-dependent. It can be computed as an edge server per data response time or for a batch of data response time. We use MNIST [24] classification application as the IoT application and per image edge server processing time as the response time. Therefore the proposed system states consist of server response times and application deadlines. For example, the state *(2, 40000)* where the response time 2 ms (millisecond) means that the edge server takes on an average 2 ms to receive an MNIST image, classify it and send the result back to the IoT device and 40000 ms is the application deadline for all the MNIST images to be processed. For offloading to multiple edge servers, the system state can be extended by adding the response time of additional servers to the state parameter.

3) ***Action Space:*** The proposed system actions consist of the local computation CPU frequency and the data offload distribution among IoT device and edge servers. For example, the action *(800000, 30)* with one edge server means that the local computation on the 30% of the data will be performed with the CPU clock speed set to 800 MHz and 70% of the data to be offloaded to the edge server. If the application tasks are completed before the application deadline, the CPU frequency is set to the lowest frequency for remaining of the time period. For more than one edge server, the data distribution ratios for IoT device and each of the edge servers should be specified separately such that the sum of all the ratios adds up to 100. For each of the system states, we use the same set of actions.
4) ***Network & Server State Observer:*** This component keeps track of the data offloaded to the edge servers and the responses received from them to calculate the response times for each of the servers, which are parameters of the system state.
5) ***Edge Server:*** The system may consist of one or more edge servers. Each edge server performs the same computational task on the data received and sends back the computation result to the IoT device.
6) ***Energy Interface:*** For each action taken, energy consumed by the action is returned as the action value for that action. The ***App Core Engine*** receives the energy consumption through this interface from ***Smart Energy Meter*** application.
7) ***Smart Energy Meter:*** This application computes the energy consumed by the IoT device for taking any action by calculating the difference between the energy readings at the start of the action and at the end of the action. Implementation of ***Smart Energy Meter*** and ***Energy Interface*** varies based on the IoT device used. The details of the implementation is given in section IV-C when Raspberry Pi [4] or Nvidia Jetson Nano [2] are used as IoT device. The ***Smart Energy Meter*** application is replaced by Intel RAPL (Running Average Power Limit) [3] when we use Linux laptop with Intel CPU [7] as the IoT device. We use the Linux Power Capping Framework that provides the *sysfs* interface to directly read the energy counters to measure the energy consumed by the whole system, including the Memory and the SOC (System on Chip). For example reading the file content
/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj
provides the energy consumed by package-0, which includes energy consumption for all the CPU cores, DRAM, and CPU uncore. Similarly reading the file content
/sys/class/powercap/intel-rapl/intel-rapl:1/energy_uj
provides the energy consumed by the SOC. We use the sum of package-0 and the SOC energy as the total system energy for the Linux Laptop.
8) ***Frequency Controller:*** This component is responsible for setting the IoT device CPU frequency to the action frequency.
9) ***App Core Engine:*** This is the central unit of the proposed system, which trains and maintains the trained DRL model, launches the IoT application, and controls all other components of the system.
10) ***Offload Controller:*** This component offloads the data to edge servers as per the distribution ratio of the action being taken by ***App Core Engine***.
11) ***App Data:*** This component holds the data to be processed by the IoT application.

C. Energy Consumption Measurement

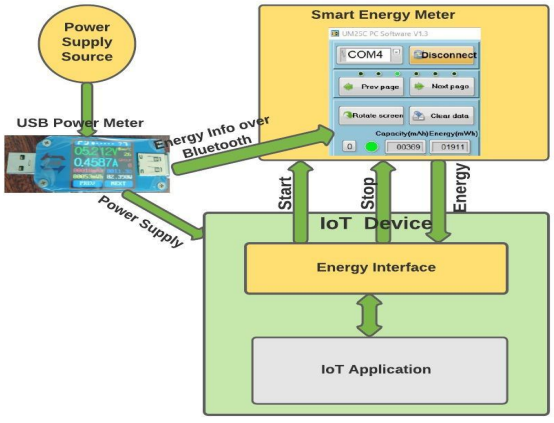

Fig. 2. System Architecture of IoT Device Energy Measurement

We use an external device USB Power meter, UM25C [15] to measure the system energy of Raspberry Pi 4 Model B [4] and Nvidia Jetson Nano 2GB [2]. The System Architecture of the connection and communication is schematically illustrated in Fig. 2 [22, p. 7]. The IoT device is powered through the power meter and the power

meter sends the instantaneous energy consumption information to its Windows Bluetooth application to display it on the screen. We developed two software components, an ***Energy Interface*** and a ***Smart Energy Meter*** application to read the energy from the screen. Using the ***Energy Interface*** the IoT application communicates with the ***Smart Energy Meter*** background application, which captures the Windows machine screen and processes the captured screens through computer vision algorithms to extract the energy information and sends it back to the IoT application through the ***Energy Interface***.

D. Proposed DRLDO Algorithm

The Algorithm 1 depicts the proposed DRLDO algorithm. Application deadlines are set by the user. A set of CPU frequencies and data distribution combinations are defined as the actions possible on the system. Initially, the application sends some data to the edge servers for processing through the ***Network Interface*** and observes the response times, which captures the network and server behavior at that time point. These response times and the application deadline determine the system state. The system deploys an explore and exploit strategy in the training phase. The explore/exploit parameter defines the percentage of the number of iterations the DRL agent would explore. It initially explores the environment by selecting random action and then exploits by selecting the action with the minimum DRL model estimated action value in the observed state. A part of the data is offloaded to the edge server as per the selected action data distribution, and the other part is processed locally in parallel after setting the CPU frequency to the desired frequency. After the action is completed as determined by (5), the ***App Core Engine*** gets the energy consumption information as the reward, trains the model with the new information collected. The new state is determined from the response times of the edge servers for the last action. We use min estimated action values as our goal is to minimize energy consumption. A penalty of higher number is used as reward if the computation completion time exceeds the application deadline. The new state observed may be the same as the previous state if there is no change in the network or server processing capability and would be different, if there is any change in the network condition or the server processing capability. The process of observing the state and executing an action in the observed state is repeated and the trained DRL model is saved after the iteration for future use. The time complexity of the algorithm is *O(N * D)*, where N is the number of iterations and D is the number of deadlines.

Algorithm 1 Proposed DRLDO Algorithm

```
Set list of application deadlines
Set list of actions (frequency and ratio combination)
Set number of iterations
Set explore/exploit parameter
Set penalty = Any higher number than the expected energy consumption
for number of iterations do
    for each application deadline do
        Determine the system state
        if iteration within explore then
            Select a random action
        else
            Select the action with min estimated action value from the DRL model given the state
        end if
        Set local CPU frequency as in selected action
        Perform computation and communication as per selected action data distribution
        Get completion time and energy consumption
        if completion time > application deadline then
            Set reward = penalty
        else
            Set reward = energy consumption
        end if
        Train the DRL model with the data collected
    end for
end for
Save the DRL Model
```

V. EXPERIMENT SETUP AND RESULTS

In this section, we present the details of the experiment setup, the results of the proposed scheme on a set of IoT devices, and an analysis of their results. We also present a comparison of the energy consumption by the proposed method vs. the Linux native ondemand and conservative governors.

We used MNIST [24] classification as the IoT application, developed using TensorFlow *2.2* in Python *3.7*. MNIST is a dataset of handwritten digits *0 to 9* with *60000* training and *10000* testing images. After training on the training images, the trained model is saved and then deployed on both the IoT device and on the edge server. We use UM25C [15], USB Power Meter to measure energy consumption when using Raspberry Pi [4] or Jetson Nano [2] as the IoT device. As milliWatt hours (mWh) is the unit of energy measured by the Power Meter, we use *100,000* MNIST test images in our experiment by classifying the original *10,000* test images *10* times through the MNIST model to observe noticeable differences between the energy consumption readings for the various distribution of the data between the IoT device and the edge server.

We use one IoT device and one edge server in our experiment. We define a set of actions consisting of the IoT device-supported CPU frequencies and data distribution ratios between the IoT device and the edge server. We define one or more application deadlines. To measure the server response time in the beginning, we send 10000 MNIST images to the edge server for classification. After receiving the results we compute per image edge server processing time as the response time. The combination of response time and the application deadline determines the state. After the initial state is determined, we take random action in that state and measure the energy consumption for the action. If the action gets completed within the application deadline then the energy consumption is used as the action value otherwise a penalty of a high number is used as the action value. Subsequent states are determined by observing the response time for the action taken in the previous step. We repeat the steps of determining the state, taking an action in the state, measuring the reward, and training a DRL model as explained in the algorithm 1.

To validate our scheme, we deploy the trained DRL model on the IoT device. We follow the approach described above to find the initial server response time and determine the system state. We feed the state to our trained DRL model to estimate the action values for the actions and find the action with the minimum estimated action value, perform the action, and measure the energy and time consumption. Subsequently, we take the rest of the actions available in that state one after another and measure the corresponding energy and time consumption. We compare the results obtained for each of those actions.

A. Linux Laptop

The Linux Laptop used as IoT device has Intel Core i3-6006U Processor [7], which supports 16 frequencies *2000000, 1900000,*

1800000, 1700000, 1600000, 1500000, 1400000, 1300000, 1100000, 1000000, 900000, 800000, 700000, 600000, 500000, and 400000 measured in unit kHz. The machine supports DVFS with separate voltage for most of the frequencies as given in Fig.3. The mapping was obtained by changing the CPU core frequency and observing the corresponding voltage through the open-source tool **CPU-X** [25].

CPU-X is a Free and Open Source software that gathers information on CPU, motherboard, and much more, including the CPU core voltage for systems running GNU/Linux and FreeBSD. It uses ***libcpuid*** [5], a C library for x86 CPU detection and feature extraction to get the relevant information. Currently, libcpuid does not have support for ARM CPU based systems, which follow ***RISC (Reduced Instruction Set Computing)*** architecture.

Frequency (MHz)	Voltage Lavels (V)
400	0.600
500	0.600
600	0.600 , 0.610
700	0.610 , 0.620
800	0.620
900	0.630
1000	0.640 , 0.650
1100	0.660 , 0.670
1300	0.690 , 0.700
1400	0.700 , 0.710
1500	0.720
1600	0.730 , 0.740
1700	0.760 , 0.770
1800	0.780 , 0.790
1900	0.800 , 0.810
2000	0.820 , 0.830

Fig. 3. Intel Core I3 Processor Frequency to Voltage Mapping

We selected three frequencies *[800000, 1400000, 2000000]* for the local computation and data distribution ratios as *[30, 50, 70]*, which generates *9* actions for the environment as *[(800000, 30), (800000, 50), (800000, 70), (1400000, 30), (1400000, 50), (1400000, 70), (2000000, 30), (2000000, 50), (2000000, 70)]*. We used Legion 5i as the edge server, *40000 ms* and *20000 ms* as two sets of application deadline and *50.0* as the penalty action value whenever the action taken does not meet the application deadline. We use Lenovo Legion 5i laptop (Legion 5i) [10] running Windows 10 as the edge server. We use 50.0 as the penalty whenever any action does not meet the application deadline.

On testing with the application deadline set to *40000 ms*, we obtained response time as *5 ms* and the state as *(5, 40000)*. Passing this state to the trained DRL model, the estimated action values for the above 9 actions were *[26.33 27.39 31.24 27.96 26.51 29.32 27.47 29.97 33.55]* with minimum value as *26.33*, which corresponds to index *0* with the action *(800000, 30)* and that becomes the best action for that state. The results for the best action and other actions are given in Table I. As seen from the table, the best action *(800000, 30)* meets the application deadline and has the lowest energy consumption of *24.45* mWh. The rest of the actions have higher energy consumption than *24.45 mWh*.

On testing with the application deadline set to *20100 ms*, a slight variation of 100 ms from the deadline *20000 ms*, which the DRL agent had seen during training, we observed server response time as *5 ms* and hence the state as *(5, 20100)*. Passing this state to the trained DRL model, the estimated action values for the above 9 actions were *[53.58 49.75 55.27 51.34 18.10 20.84 19.60 22.16 27.13]* with minimum value as *18.10*, which corresponds to index *4* with the action *(1400000, 50)* and that becomes the best action for that state. The results for all the actions are given in Table. II. As seen from the table, the best action *(1400000, 50)* meets the application deadline and has the lowest energy consumption of *19.83* mWh. Other actions either miss the deadline and receive a penalty of *50.0* or meet the deadline but have a higher energy consumption than *19.83 mWh*. The action *(800000, 30)*, which was the best action for the *40000 ms* deadline, is no longer the best action for the *20100 ms* deadline as it misses the deadline for running the computation on a lower frequency. The trained DRL model is able to estimate the action values correctly for some variations in the observed state even if it had not seen that state during training.

TABLE I
LINUX TEST RESULTS FOR ALL ACTIONS WITH DEADLINE 40000 MS

Action	Energy/Penalty	Deadline exceed?
(800000, 30)	24.45	No
(800000, 50)	25.88	No
(800000, 70)	28.87	No
(1400000, 30)	25.94	No
(1400000, 50)	27.60	No
(1400000, 70)	29.86	No
(2000000, 30)	27.61	No
(2000000, 50)	30.69	No
(2000000, 70)	33.18	No

TABLE II
LINUX TEST RESULTS FOR ALL ACTIONS WITH DEADLINE 20100 MS

Action	Energy/Penalty	Deadline exceed?
(800000, 30)	50.00	Yes
(800000, 50)	50.00	Yes
(800000, 70)	50.00	Yes
(1400000, 30)	50.00	Yes
(1400000, 50)	19.83	No
(1400000, 70)	22.40	No
(2000000, 30)	50.00	Yes
(2000000, 50)	22.89	No
(2000000, 70)	26.59	No

We compared energy consumption from the proposed method with the Linux governors *ondemand* and *conservative* where we use the same data distribution ratio found by the proposed method and leave the CPU frequency scaling to the governors. Both *ondemand* and *conservative* governors change the frequency based on the CPU utilization where *conservative* governor scales the frequency up or down gradually whereas *ondemand* can scale up to highest frequency or scale down to lowest frequency more rapidly. The comparison results for various deadlines of 20, 25, 30, and 40 seconds are given in Fig. 4. The proposed method has lowest energy consumption for all the deadline settings. The *ondemand* governor consumes more energy in the range of *3.25 - 5.78%* when compared to the proposed method energy consumption. Similarly, *conservative* governor consumes more energy in the range of *9.68 - 10.35%* when compared to the proposed method energy consumption as shown in Fig. 5.

B. Nvidia Jetson Nano 2GB (Nano 2GB)

Nano 2GB [2] is a small and powerful board designed for AI and robotics applications. It supports 15 CPU frequencies *102000, 204000, 307200, 403200, 518400, 614400, 710400, 825600, 921600, 1036800, 1132800, 1224000, 1326000, 1428000, and 1479000* measured in unit kHz. We selected three frequencies *[307200, 518400, 1479000]* for the local computation and data distribution ratios as *30, 50, 70*, which generates the action space consisting of *9* actions

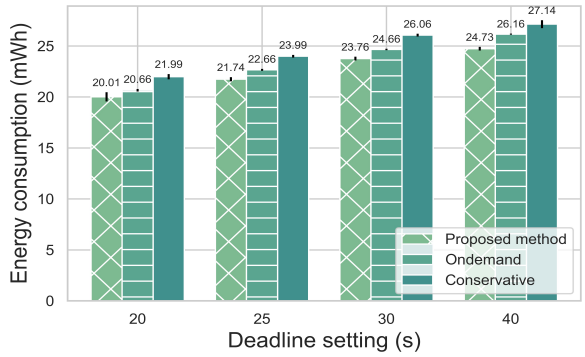

Fig. 4. Linux Laptop Energy Consumption Comparison

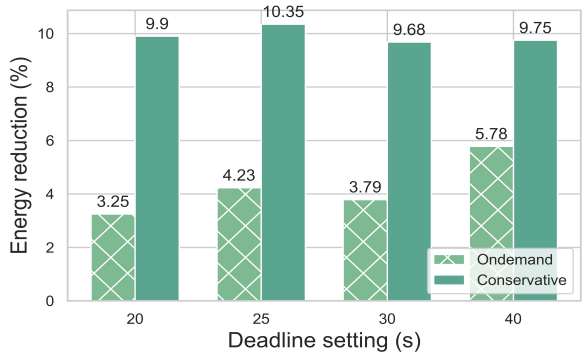

Fig. 5. Proposed Method Energy Reduction Compared to Native Governors on Linux Laptop

[(307200, 30), (307200, 50), (307200, 70), (518400, 30), (518400, 50), (518400, 70)].

We used Lenovo Legion 5i laptop (Legion 5i) [10] running Windows 10 as the edge server, and *80000 ms* as the application deadline. We use *99.0* as the penalty whenever any action does not meet the application deadline.

On deploying the trained DRL model on the Nano 2GB, we obtained server response time as *4 ms* and hence state as *(4, 80000)*. Passing this state to the trained DRL model estimated action values for the above 9 actions were *[34.99 94.11 104.69 47.84 52.89 51.61 42.16 44.99 41.32]* with minimum value as *34.99*, which corresponds to index *0* with the action *(307200, 30)* and that becomes the best action for that state. The results for the best action and other actions are given in Table. III. As seen from the table, the best action *(307200, 30)* meets the application deadline and has the lowest energy consumption of *33.0 mWh*. The rest of the actions either miss the deadline and receive a penalty of *99.0* or meet the deadline but have a higher energy consumption than *33.0 mWh*.

Similar to the Linux Laptop experiments, we compared energy consumption from the proposed method for two deadlines 75 and 80 seconds with the Linux governors *ondemand* and *conservative* and the comparison results are shown in Fig. 6. The *ondemand* governor consumes more energy in the range of *3.07 - 7.72%* while the *conservative* governor consumes more energy in the range of *7.13 - 9.89%* when compared to the proposed method energy consumption as shown in Fig. 7.

C. Raspberry Pi 4 Model B (RPi4B)

RPi4B [4] is the latest release of Raspberry Pi that is widely used as IoT device for various IoT applications. We use the 4GB

TABLE III
JETSON NANO 2GB TEST RESULTS FOR ALL ACTIONS WITH DEADLINE 80000 MS

Action	Energy/Penalty	Deadline exceed?
(307200, 30)	33.0	No
(307200, 50)	99.0	Yes
(307200, 70)	99.0	Yes
(518400, 30)	41.0	No
(518400, 50)	41.0	No
(518400, 70)	99.0	Yes
(1479000, 30)	36.0	No
(1479000, 50)	38.0	No
(1479000, 70)	41.0	No

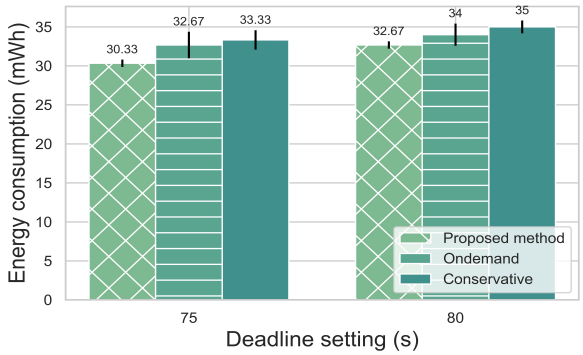

Fig. 6. Nano 2GB Energy Consumption Comparison

version of RPi4B running Raspberry Pi OS 64 bit beta version, which supports 10 frequencies *600000, 700000, 800000, 900000, 1000000, 1100000, 1200000, 1300000, 1400000, and 1500000* measured in unit kHz. Unlike the Linux Laptop, the RPi4B does not have different voltage levels for different frequencies. The command *vcgencmd measure_volts core*, returns the CPU core voltage as ***0.8625V*** for any of the above listed frequencies set as CPU frequency. ***vcgencmd*** is a command-line utility that can get various pieces of information from the Raspberry Pi including the CPU core voltage.

We selected three frequencies *[600000, 1000000, 1500000]* for the local computation and data distribution ratios as *30, 50, 70*, which generates action space of *9* actions *[(600000, 30), (600000, 50), (600000, 70), (1000000, 30), (1000000, 50), (1000000, 70), (1500000, 30), (1500000, 50), (1500000, 70)]*. We used Legion 5i as the edge server, and *60000 ms* as application deadline. We use *99.0* as the penalty whenever any action does not meet the application deadline.

On testing, we observed the response time as *4 ms* and hence the state as *(4, 60000)*. Passing this state to the trained DRL model, the estimated action values for the above 9 actions were *[56.91 55.11 104.98 52.29 60.04 57.99 53.32 58.46 64.20]* with minimum value as *52.29*, which corresponds to index *3* with the action *(1000000, 30)* and that becomes the best action for that state. The results for all the actions are given in Table. IV. As seen from the table, the best action *(1000000, 30)* meets the application deadline and has the lowest energy consumption of *53.0 mWh*. The rest of the actions either miss the deadline and receive a penalty of *99.0* or meet the deadline but have the same or higher energy consumption than *53.0 mWh*.

We compared the results of our proposed method with the native *ondemand* and *conservative* governors but the energy gain was significantly negligible in the range of *1%* as on Raspberry Pi, the CPU core voltage is the same for all the operating frequencies. The

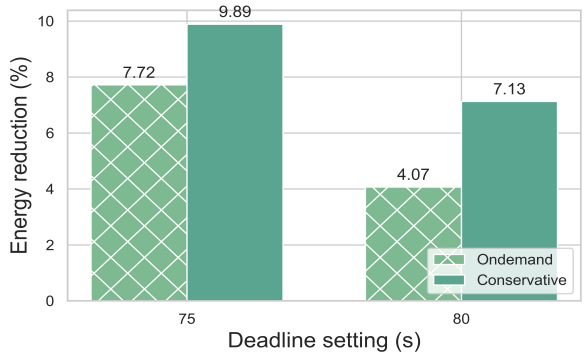

Fig. 7. Proposed Method Energy Reduction Compared to Native Governors on Nano 2GB

same CPU core voltage for all operating frequencies is also reflected in the results shown in Table. IV as the energy consumption for the same data distribution with different frequencies is more or less the same.

TABLE IV
RPI4B TEST RESULTS FOR ALL ACTIONS WITH DEADLINE 60000 MS

Action	Energy/Penalty	Deadline exceed?
(600000, 30)	53.0	No
(600000, 50)	56.0	No
(600000, 70)	99.0	Yes
(1000000, 30)	53.0	No
(1000000, 50)	55.0	No
(1000000, 70)	59.0	No
(1500000, 30)	54.0	No
(1500000, 50)	56.0	No
(1500000, 70)	58.0	No

VI. CONCLUSION

To enable the IoT application running on the IoT device to meet the application time constraint and minimize the energy consumption, this paper proposes a DRL scheme that performs decision-making of *a)* DVFS: computation speed control on the device and *b)* offloading: communication from IoT device to edge servers. The proposed scheme learns the optimal data distribution policies and local computation DVFS frequency scaling by interacting with the system environment and learning the device, the network, and edge servers' behavior. The proposed scheme was tested on multiple edge computing environments with different IoT devices. Experimental results from various IoT devices using one edge server show that the proposed scheme always selects the optimal data distribution and the optimal DVFS frequency scaling that meets the IoT application deadline and minimizes the IoT device energy consumption. The trained DRL model estimates the correct action values even for some variations in the environmental situation, which it had not seen during training. The proposed scheme has significant energy savings when compared with the native Linux governors and will work better by selecting lower frequency if the IoT device supports lower voltage levels for lower frequencies. The experiment can be extended to find optimal data distribution with multiple edge servers available in the system.

REFERENCES

[1] H. Aydin et al. "Power-aware scheduling for periodic real-time tasks". In: *IEEE Transactions on Computers* 53.5 (2004), pp. 584–600.

[2] Nvidia Corporation. *Jetson Nano 2GB Developer Kit*. https://developer.nvidia.com/embedded/jetson-nano-2gb-developer-kit.

[3] The Linux Kernel Documentation. *The Linux Kernel Power Capping Framework*. https://www.kernel.org/doc/html/latest/power/powercap/powercap.html.

[4] Raspberry Pi Foundation. *Raspberry Pi 4 Model B Specifications*. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[5] Veselin Georgiev. *C library for x86 CPU detection and feature extraction*. https://github.com/anrieff/libcpuid.

[6] Hui Huang, Qiang Ye, and Hongwei Du. "Reinforcement Learning Based Offloading for Realtime Applications in Mobile Edge Computing". In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. 2020, pp. 1–6.

[7] Intel. *Intel® Core™ i3-6006U Processor Specification*. https://ark.intel.com/content/www/us/en/ark/products/91157/intel-core-i3-6006u-processor-3m-cache-2-00-ghz.html.

[8] Fakhruddin Muhammad Mahbub ul Islam and Man Lin. "Hybrid DVFS Scheduling for Real-Time Systems Based on Reinforcement Learning". In: *IEEE Systems Journal* 11.2 (2017), pp. 931–940.

[9] Beakcheol Jang et al. "Q-Learning Algorithms: A Comprehensive Classification and Applications". In: *IEEE Access* 7 (2019), pp. 133653–133667.

[10] Lenovo. *Lenovo Legion 5i (17") gaming laptop*. https://www.lenovo.com/ca/en/laptops/legion-laptops/legion-5-series/Legion-5i-17/p/88GMY501441.

[11] He Li, Kaoru Ota, and Mianxiong Dong. "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing". In: *IEEE Network* 32.1 (2018), pp. 96–101. DOI: 10.1109/MNET.2018.1700202.

[12] Ji Li et al. "Deep reinforcement learning based computation offloading and resource allocation for MEC". In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. 2018, pp. 1–6.

[13] Xue Lin et al. "Task Scheduling with Dynamic Voltage and Frequency Scaling for Energy Minimization in the Mobile Cloud Computing Environment". In: *IEEE Transactions on Services Computing* 8.2 (2015), pp. 175–186.

[14] Fagui Liu, Zhenxi Huang, and Liangming Wang. "Energy-Efficient Collaborative Task Computation Offloading in Cloud-Assisted Edge Computing for IoT Sensors". In: *Sensors* 19.5 (2019). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/19/5/1105.

[15] Glen liu. *UM25C Documents and Software*. https://www.mediafire.com/folder/q2b8h079hpywq/UM25.

[16] Nguyen Cong Luong et al. "Applications of Deep Reinforcement Learning in Communications and Networking: A Survey". In: *IEEE Communications Surveys Tutorials* 21.4 (2019), pp. 3133–3174.

[17] Xiaoqiang Ma et al. "A Survey on Deep Learning Empowered IoT Applications". In: *IEEE Access* 7 (2019), pp. 181721–181732. DOI: 10.1109/ACCESS.2019.2958962.

[18] Timothy Malche and Priti Maheshwary. "Internet of Things (IoT) for building smart home system". In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. 2017, pp. 65–70.

[19] R. Porkodi and V. Bhuvaneswari. "The Internet of Things (IoT) Applications and Communication Enabling Technology Standards: An Overview". In: *2014 International Conference on Intelligent Computing Applications*. 2014, pp. 324–329.

[20] Wang Qiang and Zhan Zhongli. “Reinforcement learning model, algorithms and its application”. In: *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*. 2011, pp. 1143–1146.

[21] Yijing Ren, Yaohua Sun, and Mugen Peng. “Deep Reinforcement Learning Based Computation Offloading in Fog Enabled Industrial Internet of Things”. In: *IEEE Transactions on Industrial Informatics* 17.7 (2021), pp. 4978–4987.

[22] Man Lin Saroj Kumar Panda. *IoT Device Energy Consumption Measurement Using A Machine Learning Model*. https://ahsn.committees.comsoc.org/files/2021/07/IOT-AHSN-Newsletter-June-2021-V2.pdf.

[23] Weisong Shi, George Pallis, and Zhiwei Xu. “Edge Computing [Scanning the Issue]”. In: *Proceedings of the IEEE* 107.8 (2019), pp. 1474–1481.

[24] Wikipedia. *MNIST Database*. https://en.wikipedia.org/wiki/MNIST_database.

[25] Weblate (bot) Xorg. *CPU-X Free software*. https://github.com/X0rg/CPU-X.

[26] Qingchen Zhang et al. “A Double Deep Q-Learning Model for Energy-Efficient Edge Scheduling”. In: *IEEE Transactions on Services Computing* 12.5 (2019), pp. 739–749.