IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 7, 2021, accepted April 11, 2021, date of publication April 19, 2021, date of current version April 27, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3073902

A Deep Reinforcement Learning-Based Dynamic Computational Offloading Method for Cloud Robotics

MANOJ PENMETCHA, (Graduate Student Member, IEEE), AND BYUNG-CHEOL MIN, (Member, IEEE)

SMART Laboratory, Department of Computer and Information Technology, Purdue University, West Lafayette, IN 47907, USA

Corresponding author: Manoj Penmetcha (mpenmetc@purdue.edu)

This work was supported in part by the Purdue University Libraries Open Access Publishing Fund.

ABSTRACT Robots come with a variety of computing capabilities, and running computationally-intense applications on robots is sometimes challenging on account of limited onboard computing, storage, and power capabilities. Meanwhile, cloud computing provides on-demand computing capabilities, and thus combining robots with cloud computing can overcome the resource constraints robots face. The key to effectively offloading tasks is an application solution that does not underutilize the robot's own computational capabilities and makes decisions based on crucial cost parameters such as latency and CPU availability. In this paper, we formulate the application offloading problem as a Markovian decision process and propose a deep reinforcement learning-based deep Q-network (DQN) approach. The state-space is formulated with the assumption that input data size directly impacts application execution time. The proposed algorithm is designed as a continuous task problem with discrete action space; i.e., we apply a choice of action at each time step and use the corresponding outcome to train the DQN to acquire the maximum rewards possible. To validate the proposed algorithm, we designed and implemented a robot navigation testbed. The results demonstrated that for the given state-space values, the proposed algorithm learned to take appropriate actions to reduce application latency and also learned a policy that takes actions based on input data size. Finally, we compared the proposed DQN algorithm with a long short-term memory (LSTM) algorithm in terms of accuracy. When trained and validated on the same dataset, the proposed DQN algorithm obtained at least 9 percentage points greater accuracy than the LSTM algorithm.

INDEX TERMS Cloud robotics, deep reinforcement learning, deep Q-networks (DQN), AWS, neural networks, application offloading, robot navigation.

I. INTRODUCTION

The field of robotics is growing at a great pace [1], [2], and one factor driving its growth is the widening range of robotic applications [3]. Recent advancements in machine learning [4] are being used to develop smarter robots, and so most of these robotic applications require high-performing computational capabilities to attain a satisfactory level of performance [5]. However, it is often the case that extant robots are equipped with limited computational capabilities, and once a robot is assembled, changing its hardware configuration is not easy [6]. Providing robot access to the on-demand computing resources offered by cloud service providers can be an effective means of solving this problem [7]. Namely, by taking advantage of the computing power and data storage options provided by cloud services [8], cloud-enabled robots can rely less on their local computation resources.

The area of study enabling robots to utilize cloud computing is termed cloud robotics, which was first coined by Kuffner [9] in 2010. Cloud robotics algorithms are designed on the basic premise that when robots have insufficient computational resources for local execution of an application, using cloud resources should improve the performance of the application in terms of execution time and energy efficiency [10]. If a robot has the bare minimum of computational capability, full application offloading will be an obvious choice; however, many robots presently being produced are computationally capable, and so dynamic computational

The associate editor coordinating the review of this manuscript and approving it for publication was Wai-Keung Fung.

offloading algorithms are a wiser choice as they take into account both the robot's computational capability and the application's computational requirements [5].

Robots are equipped with a wide range of sensors. Usually, a robotic application gets input from these sensors and processes that input to provide an output action for the robot [11]. The amount of sensory data that the application needs to process at a given time significantly affects its computational consumption [12]; if the application requires more computational resources than the robot can accommodate, its onboard execution might be extended to a degree that degrades the robot's performance. Hence, we designed the offloading problem to consider application input data size and used a deep reinforcement learning (DRL)-based deep Q-network (DQN) for dynamic application offloading. Deep reinforcement learning [13] is a subfield in machine learning that combines neural networks with reinforcement learning (RL), and has opened up avenues for solving problems that were difficult to solve otherwise [14]. DRL has been applied in a wide range of robotic applications related to navigation, social robotics, motion control, manipulation, and more [14]–[16]. DRL-based algorithms have been studied for application offloading in mobile devices [17]–[19], but only a few studies have used DRL for application offloading from a robotics perspective [10]. Most extant offloading algorithms for mobile devices use mobile edge computing (MEC), and by design cater to mobile-specific applications. To our knowledge, we are the first to provide a DRL-based dynamic application offloading solution for cloud robotics that considers input data size and is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step t and use the corresponding outcome to train the DQN and learn a policy to acquire maximum rewards at a given time step t'. We validated the algorithm on a robotic path planning application running on the Robot Operating System (ROS).

The area of dynamic application offloading for cloud robotics is still in the early stages. In this paper, we propose a novel dynamic application offloading algorithm for cloud robotics; moreover, the proposed architecture includes several novel functionalities that make it the first of its kind.

The main contributions of this paper are as follows:

- We formulate the computational offloading problem as a Markovian decision process (MDP) and propose a deep reinforcement learning-based deep Q-network (DQN) approach for its solution.
- We formulate the state space based on the assumption that input data size directly impacts application execution time and define a variable reward function that helps training converge more quickly and learning of a robust policy.
- We analyze the proposed algorithm using a robot navigation application, evaluating it on real data and also generating synthetic data to analyze whether the network is learning the appropriate policy with respect to all possible outcomes.

The remainder of the paper is organized as follows: In Section II, we describe related work on dynamic application offloading and DQN fundamentals. In Section III, we define the problem formulation based on DQN. In Section IV, we introduce the algorithm's application with a robot navigation framework. In Section V, we provide experiment results and their analysis. Finally, we conclude our work and present our future directions in Section VI.

II. BACKGROUND

This section presents related work concerning application offloading in cloud robotics and also introduces the basic DQN fundamentals that our algorithm uses as a design foundation.

A. RELATED WORK

From 2009 on, several architectures have been proposed that facilitate application offloading for cloud robotics [20]–[29]. The aforementioned works focus on catering to application-specific solutions like navigation, object detection, etc. Most of the architectures propose full offloading without any consideration for the robot's local computing capabilities and the costs associated with offloading. Namely, communication between robots and cloud services, including for complete offloading, has to consider costs such as latency, energy, CPU utilization, and security.

To our knowledge, there are only a few studies in cloud robotics that have focused on dynamic application offloading that account for the cost parameters involved when making an offloading decision [30]. One prior study based its offloading decision on network connectivity and robot mobility; it used a genetic algorithm and concluded that motion- and connectivity-aware offloading leads to more efficient performance in terms of Quality of Service (QoS) and minimum resource consumption [31]. In 2017, Wang *et al.* [32] proposed a resource allocation strategy based on a hierarchical auction mechanism, namely a link quality matrix (LQM) auction; the algorithm was designed and demonstrated for firm real-time applications and outperformed other state-of-the-art algorithms. Later, Hong *et al.* [33] proposed a QoS-aware cooperative computational offloading solution for robot swarms to minimize latency and maximize energy efficiency; they formulated the optimization problem as a multi-hop cooperative computation-offloading game.

Some other proposed offloading solutions were based on edge computing [34], [35]. Shuja *et al.* [36], Ahmed *et al.* [37] have comprehensively surveyed machine learning-based approaches for in-network caching in edge networks for mobile devices. These solutions allow robots to offload computationally-intense applications to the computing infrastructure in their vicinity. However, none of these papers considered deep reinforcement learning-based techniques for decision–making regarding offloading.

Though RL is a well-studied field in the area of robotics, it has some shortcomings in the form of scalability [38]. Meanwhile, deep learning is known for making low-level

categorizations of data and using that information for higher-level categorization [39]. The combination of deep learning and RL helps to address some of the shortcomings of RL, i.e., problems that require higher-level categorization and action spaces [40].

Application offloading for mobile devices is a well-studied area, and several such studies have proposed dynamic application offloading solutions using DRL. As a case in point, Qiu *et al.* [41] proposed a collective and distributed DRL algorithm that considered experience from distributed systems to obtain an optimum offloading policy using MEC. Later, Qiu *et al.* [42] also proposed a DRL-based offloading solution for computationally-intense mining applications that employed multi-hop multi-user blockchain-empowered MEC. Meanwhile, Tang and Wong [43] proposed a distributed DRL solution to minimize the long-term cost for non-divisible and delay-sensitive tasks using MEC. Wang *et al.* [44] proposed a meta reinforcement learning-based algorithm that leveraged recurrent neural networks for faster loss convergence, and represented mobile applications as directed acyclic graphs for validation of the algorithm. Finally, Dai *et al.* [45] proposed a DRL-based computation offloading and resource allocation algorithm to reduce overall energy consumption; it used a multi-user end-edge-cloud orchestrated network. Notably, the aforementioned algorithms were all designed for mobile devices and mobile-specific applications using MEC, whereas our proposed algorithm is designed from the robotics perspective and validated with a robotic application using cloud computing.

Some researchers have applied DRL in application offloading solutions in cloud robotics; for example, Chinchali *et al.* [46] formulated the offloading problem as a sequential decision-making problem and used deep reinforcement learning for object detection applications, and their findings suggest that RL is likely an effective choice for optimizing offloading decision policies. Another prior study proposed a resource allocation scheme based on RL that allowed the cloud to decide whether a request should be accepted and the amount of resources to be allocated to the application; this work also demonstrated better performance of RL algorithms relative to other greedy resource allocation scheme [47]. Finally, Peng *et al.* [48] proposed an online resource scheduling framework based on DQN that implemented a tradeoff between energy consumption and task makespan by prioritizing the rewards. However, most of these DRL-based works were only published in the last couple of years, and there remains a lot of room for improvement.

The area of dynamic application offloading is still in its early stages, and most of the work we present here is the first of its kind in cloud robotics. Our proposed algorithm broadly diverges from existing solutions in two important ways. First, we introduce a novel offloading strategy based on DQN; the state space is built with the assumption that the size of the input data for an applications directly impacts its execution time. Second, we use a variable reward rather than a fixed reward, which led the algorithm to converge faster and to learn an optimal policy efficiently and quickly.

B. DEEP Q NETWORK

Conventional RL algorithms do not scale well with increasing numbers of applications and robots, as this leads to an explosion in state space [49] and becomes an NP-hard problem [50]. DQN is an off-policy, model-free RL algorithm [51] that overcomes several drawbacks faced by traditional RL algorithms [52]–[55]. Using DQN, agents are able to continuously learn and optimize their decision-making through trial and error. DQN models work on the principle of selecting those actions that maximize overall reward in the long term. The agent receives a reward r for change in state s when action a is performed. Observing these rewards, the agent forms a consensus about a policy π that helps it in achieving the maximum reward. Hence, we have modeled the proposed algorithm as a MDP and used DQN to derive an optimal policy π for offloading decision-making.

DQN uses the Q-function [56], [57] as its foundation for calculating expected reward values. $Q(s, a)$ is the reward of the current state-action pair, represented as the summation of the reward for the current state and the maximum reward value expected in the future. Mathematically, $Q(s, a)$ is represented as,

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a') \tag{1}$$

which can be further generalized using Bellman's equation [58], resulting in the following,

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{2}$$

where γ represents the discounting rate, α is the learning rate, and the variables s, a, r respectively denote the state space, action taken, and associated reward for a state-action pair. If s and a denote the current state and action, then s' and a' denote the next state and action.

Traditional Q-learning requires a lot of memory to generate a Q-table, and if the application space and number of robots are large, Q-learning also becomes computationally intense. As DQN provides a way to learn a Q-function using a deep neural network, it is a better choice than traditional Q-learning for the application-offloading scenario. As with other machine learning models, DQN has a well-defined cost function that the network tries to minimize. This cost function is given as,

$$L(\theta) = \left[Q(s, a|\theta) - (r(s, a) + \gamma \max_{a'} Q(s', a'|\theta^{-})) \right]^2 \tag{3}$$

where θ represents the trainable weights of the network.

We will use the foundations of the DRL and DQN explained in this subsection for problem formulation. The loss function in Eq. (3) is what the neural network tries to minimize using the state space, action space, and reward defined in the next section.

III. PROBLEM FORMULATION

As stated earlier, we modeled the dynamic application offloading algorithm using DQN. DQN is based on MDP, which is usually defined in terms of (s, a, r), where s is the state space, a is the action space, and r is the reward. In this section, we talk about how we defined each of these key parameters for the proposed offloading algorithm.

A. STATE SPACE

The time of execution for an application is proportional to the size of the input data [59]. Hence, we designed the state space to reflect the data sizes of the applications running at any given time, along with other system parameters, namely CPU availability and round-trip time for communicating with the cloud. The state space of the model includes the full observation of the system and is defined for every time step as,

$$s_t = \{d_t, u_t, b_t\} \tag{4}$$

where the meaning of each variable at state initialization is as follows:

- $t = 1, 2, 3 \ldots, T$ denotes the time steps for the given episode.
- d_t denotes the input data size of application.
- u_t denotes the CPU availability at state initialization.
- b_t denotes the network latency for data making a round trip between the robot and the cloud.

B. ACTION SPACE

The action space defines what actions an agent can perform in the environment. Our model is designed for binary decision-making, hence our discrete action space is defined as $a_t = (a_0, a_1, \ldots, a_T)|a_t \in \{0, 1\}$. This definition implies that an application can be executed either locally ($a = 0$) or on the cloud service ($a = 1$).

C. REWARD

In the following section, we define variable reward (r_t) and its associated variables for a given state-action pair (s_t, a_t).

When the robot performs on-board computation of an navigation application task, the associated local time of execution (l_t^{local}) is derived from

$$l_t^{local} = l_t^{lcompletion} - l_t^{start}. \tag{5}$$

Similarly, when a robot delegates the computation of an application to the cloud, the associated time of execution (l_t^{cloud}) is derived from

$$l_t^{cloud} = l_t^{ccompletion} - l_t^{start} \tag{6}$$

where $l_t^{lcompletion}$ represents the timestamp of application completion and l_t^{start} represents the time stamp when the task was first assigned to the robot.

By incorporating tuning parameters α and β, the robotic operator has the opportunity to prioritize between offloading and local computation. These parameters are set according to the relative importance of executing the application on the cloud or on the robot, where $\alpha + \beta = 1$ and $\alpha, \beta \in [0, 1]$. When there is no predefined priority between offloading and local computation, $\alpha = \beta$; when offloading is prioritized over local computation, $\alpha > \beta$; and when local computation is prioritized over offloading, $\alpha < \beta$. In our experiment, we do not define a predefined priority, thus always set $\alpha = \beta$.

Combining the tuning parameter α with local execution time (l_t^{local}) gives us total computational cost on the local machine,

$$c_t^{local} = \alpha l_t^{local}. \tag{7}$$

Similarly, combining the tuning parameter β with cloud execution time (l_t^{cloud}) gives us total computational cost on the cloud,

$$c_t^{cloud} = \beta l_t^{cloud}. \tag{8}$$

During its learning phase, the proposed algorithm randomly chooses between local or cloud execution. This algorithm-determined action $c_t^{algorithm}$ can be either c_t^{local} or c_t^{cloud}, and we can represent the algorithm-determined action as,

$$c_t^{algorithm} = c_t^{local} \vee c_t^{cloud}. \tag{9}$$

Then, we define the variable reward r_t for (s_t, a_t) as,

$$r_t = -(c_t^{algorithm} - c_t^{local})/(c_t^{local} + c_t^{cloud}) \tag{10}$$

where r_t will always be in the range between -1 and 1. When $c_t^{algorithm} = c_t^{local}$, the reward assigned will be zero, and when $c_t^{algorithm} = c_t^{cloud}$, the reward values will be either positive or negative. The agent is given a positive reward for choosing to offload when cloud execution time (c_t^{cloud}) is less than local execution time (c_t^{local}), and a negative reward for choosing to offload when cloud execution time (c_t^{cloud}) is greater than local execution time (c_t^{local}).

D. DQN ALGORITHM FOR DYNAMIC OFFLOADING

Now that we have clear definitions of the state space s, action space a, and reward r, we can define our DQN-based offloading algorithm as given in Algorithm 1. The DQN architecture used for validating the proposed algorithm in robot navigation is illustrated in Fig. 1. Our problem formulation is based on acquiring maximum rewards and not on the end goal success criterion. As such, the algorithm is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step t and use the corresponding outcome to train the DQN and learn a policy to acquire maximum rewards at a given time step t'.

IV. EXPERIMENTAL SETUP WITH ROBOT NAVIGATION APPLICATION

To validate the proposed algorithm, we used it with a robot navigation application. The simulation environment employed in the experimental setup is shown in Fig. 3; on the

Algorithm 1 Proposed DQN-Based Offloading Algorithm

Initialize replay memory R with capacity N;
Initialize action-value Q function with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$;
Input: State space $s_t = \{d_t, u_t, b_t\}$
Output: Q value for state-action pair
for episode $= 1$ and $t = 1$ **do**
    Receive initial state observation;
    repeat
        Load the state values s_t;
        Choose a random action a_t from action space A;
        Calculate reward $r_t(s_t, a_t)$ by Eq. (10);
        Load the next state values s_{t+1};
        Store the experience (s_t, a_t, r_t, s_{t+1}) in replay memory R;
        Select a random minibatch (s_j, a_j, r_j, s_{j+1}) from R;
        Set $y_j = r_j + \gamma \max_{a_{j+1}} \hat{Q}(s_{j+1}, a_{j+1}|\theta^{-})$ from Eq. (1);
        Perform a gradient descent to minimize loss using equation $(y_j - Q(s_j, a_j|\theta))^2$ with network parameter θ from Eq. (3);
        Every few steps, copy weights from Q to $\hat{Q}$;
        Set $t = t + 1$;
    until A predefined stopping condition is reached, i.e., loss function reached convergence;
end for

left is a graphical representation of the simulation environment, and on the right the global cost map of the navigation environment along with obstacles, the Husky robot, and the camera view from the robot. The simulation environment was constructed using ROS with a gazebo simulator, which we will briefly talk about in the next paragraphs.

The navigation application utilized here is built on top of the ROS framework for writing robotics software, which can be used across most robotic platforms [60] and helps researchers and developers to create software that is modular, concurrent, open-source, and supportive of code reuse. ROS *messages* are structures that contain data of various types, and these messages are transmitted using ROS *nodes*. A brief overview of the ROS application framework is shown in the ROS Layer of Fig. 2.

Gazebo is a 3D dynamic simulator that can accurately and efficiently simulate the real-world behaviors of robots, environments, and their interactions [61]. It can be easily integrated with ROS, thereby allowing robots to avail themselves of the hundreds of open-source ROS tools and packages while within the gazebo environment. Finally, we used the open-source Husky robot [62] as the navigation vehicle in the environment.

A. NAVIGATION APPLICATION FRAMEWORK

The gazebo simulation environment provides access to a Husky robot, designed by Clearpath Robotics [62], which was equipped with sensors such as a camera, LiDaR, and wheel encoders, and programmed to perform path planning. The main goal of this experiment was to determine if the robot could learn a policy from state space values regarding when to offload the path planning application to the cloud.

The navigation application framework used for validating the proposed algorithm is depicted in Fig. 2. The framework can be broadly categorized into two layers: ROS and DQN. In the ROS layer, the robot uses the simulation environment to generate state space data, local path-planning execution time (l_t^{local}), and cloud path-planning execution time (l_t^{cloud}). In every time step, random goal coordinates are assigned to the robot for path planning. For simplicity's sake, the robot does not actually navigate to the destination, but just calculates the shortest path from the origin to the assigned destination using the Dijkstra algorithm [63].

The values from the ROS layer are recorded in a Rosbag file, and that file is then used to train the DQN network and generate Q estimates for the state-action pair.

As mentioned earlier, the state space is built on the assumption that the size of input data (d_t) provided to an application directly impacts its execution time. In this experiment, we consider path planning for robot navigation and demonstrate how we derive the input data sizes for this application (d_t^p).

B. PATH PLANNING

Path planning is a well-studied research area in robotics. The main objective of path-planning algorithms is to provide the shortest obstacle-free path between origin and destination. Among the well-known and well-studied algorithms for path planning are the Dijkstra [63], A* [64], breadth-first search [65], and depth-first search algorithms [65]. To ensure faster training and data collection, we limited the task in the present study to only plotting a path between the current position and the goal position, without actually moving the robot. Random goal coordinates were assigned inside the global map, and the Dijkstra algorithm was used to compute the shortest path.

A proper representation of computational size (d_t^p) is vital for the DQN algorithm to learn optimal policy. The objective in our experiment was not only to determine the shortest obstacle-free path but to do so in the least possible execution time. The inputs for the path planner are the current position of the robot (A), the goal position (B), and the global map in the form of an image. Its output is an obstacle-free global path from the current position to the goal position, as depicted in Fig. 4. Dijkstra's algorithm uses a node-based approach to calculate the shortest path, and input size can be represented by $n\log n$ [66], where n represents the nodes. As is evident in Fig. 4, the number of nodes in the dotted rectangular space

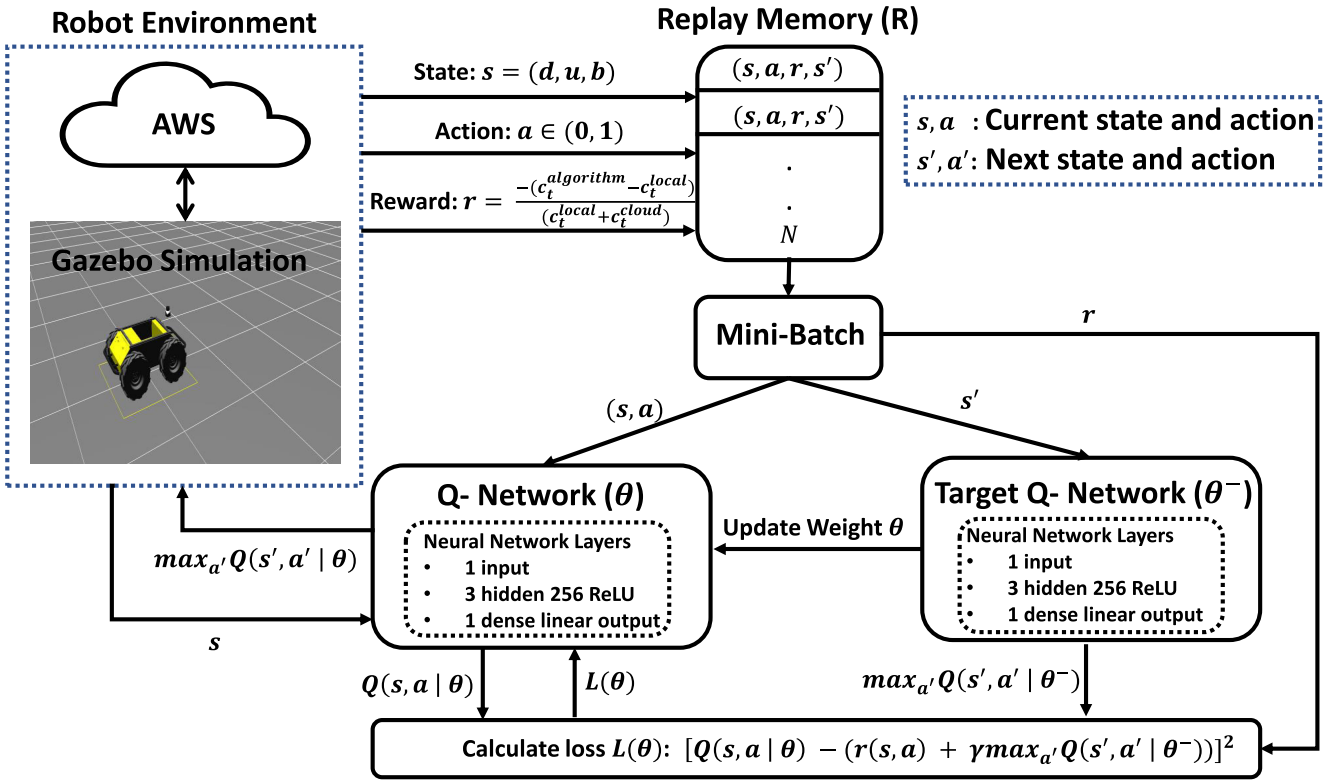


FIGURE 1. Dynamic computational offloading framework based on DQN. The robot environment provides input in the form of state space, action performed, and reward acquired. The DQN learns from these inputs and sends back a response in the form of a Q-value for the state and action pair. The neural network used for the navigation application has one input layer, three hidden ReLU layers with 256 neurons each, and one dense linear output layer.

gives us a good estimate of the number of nodes (n) that the algorithm needs to explore before reaching the goal position. The algorithm will traverse these nodes several times to find the shortest path and hence the input size is represented as $n\log n$. The Euclidean distance from A to B is the diagonal for the rectangle. We can obtain the number of nodes in the rectangle by dividing the area of the rectangle by the area of each node inside it. Hence, for path planning we define the number of nodes n as,

$$n = (\frac{1}{2}x^2)/r^2 \tag{11}$$

where the numerator represents the area of the rectangle with diagonal $\bar{AB}$ and the denominator represents the area of each node. The variable x is the Euclidean distance from A to B, and r is the length of the each side in a node.

When ROS launches a navigation module, the granularity of the occupancy grid (r) is usually set to 0.05 meters [67], but can be manually changed as required by the application. A granularity of 0.05 means that each side of the square grid is 0.05 meters and the area of each square is $(0.05)^2$. Hence, the resolution of each node in the occupancy grid map can be represented as r and area of each node as r^2.

We can use the number of nodes n derived from Eq. (11) to determine the computational cost of path planning as,

$$d_t^p = n\log n. \tag{12}$$

We demonstrate in Fig. 3 an example of how we calculated n and d_t^p. The starting position of the Husky was (0.001, −0.001) and the assigned goal position was (22.951, 39.054). The Euclidian distance (x) between those two positions can be calculated as approximately 45.09. The area of the occupancy grid r^2 as stated above was 0.0025. By substituting the values of x and r^2 in Eq. (11), we get the number of nodes (n) as being approximately 410,320. Substituting n in Eq. (12), we get d_t^p as approximately 5,303,251; the Husky will need to explore the nodes several times in order to obtain a shortest path. This implies that for the Husky to reach its goal position, it will need to traverse around 5,303,251 nodes ($n\log n$) and the number of nodes in the in the rectangular area that husky will traverse several times to find the shortest path, can be represented as 410,320. In the experiment, we normalized the value of d_t^p by dividing by the global map size, which yielded values in the range of 0 to 5.

C. AWS AND LATENCY

In this experiment, we utilized Amazon web services (AWS) as the cloud service provider. There are several means by

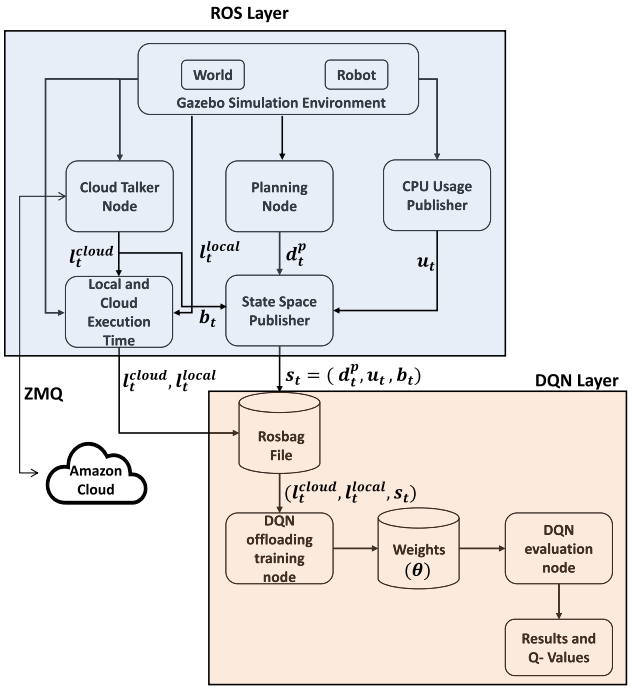


FIGURE 2. Two-layered navigation framework for algorithm validation. ROS Layer (top): A robot interacts with a Gazebo world and AWS to generate state-space values along with the local and cloud execution times for path-planning. DQN Layer (bottom): Derives an optimal offloading policy using inputs from the ROS layer.

which a robot can communicate with AWS; for this application, we chose ZeroMQ (ZMQ) [68]. ZMQ is a high-performance asynchronous messaging library that can be used in both distributed and concurrent applications, and furthermore is known for its excellent performance scalability and low latency. ZMQ provides a ROS-like publisher-subscriber messaging that supports several communication protocols, including WebSockets; in addition, the broker-less framework provided by ZMQ is faster than the inherent ROS communication framework. Given all these exciting features, ZMQ makes an excellent framework for establishing a communication protocol between robots and AWS. For the experiment, we used the Amazon Ohio instance with a static IP to ensure easier communication with the cloud service. As depicted in Fig. 5, the local machine sends obstacle information, the current robot position, and the goal position to the cloud. The cloud sends back a response in the form of a planned path.

The state space includes round trip latency (b_t) for each time step, which was calculated in real-time by pinging the cloud. The simulation was carried out on a computer that had a stable internet connection, and latency was constant at around 30 milliseconds with less than 5 percent variation for 99 percent of cases. This can be considered a drawback in the current experiment, as the latency values had minimal variation. In any case, the latency parameter did not play a meaningful role in deriving policy.

TABLE 1. DQN network parameters used for training.

Parameters	Values
Number of hidden layers	3
Number of nodes	256; 256; 256
Mini-batch size	128
Learning rate	0.01
Exploration rate	0.1
Discount factor	0.9

D. DQN NETWORK

We used a neural network consisting of one input layer, three hidden ReLU layers (each having 256 neurons), and one dense linear output layer. The configuration of hidden layers and the number of neurons can be altered based on convergence, training time, or any other performance criteria [69]. The choice of neural network can also be adapted to fit any specific learning problem, such as using a convolution neural network (CNN), long short term memory (LSTM), etc. [70]. Using the network parameters given in Table 1, we obtained a satisfactory convergence for the results.

V. RESULTS AND ANALYSIS

In supervised learning, algorithm model evaluation is reasonably straightforward: the data is labeled and an evaluation set is used to assess the accuracy of the results [71]. However, model evaluation is tricky for algorithms based on DRL as they do not have a labeled dataset to be used as ground truth to validate the performance of the algorithm. As our problem formulation is based on acquiring maximum rewards and not the end-goal success criterion, our algorithm is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step and use the corresponding outcome to train the DQN. When replay memory reaches a threshold of 1,000 experiences, experiences are replaced according to the first-in first-out (FIFO) approach.

In the first part of the results, we analyze the network performance with reference to actual data collected from the cloud and robot. Unlike in episodic problems, it is difficult to judge an agent's performance in a continuous problem as there is no terminal state that defines if the action was a success or a failure. Hence, we need to somehow generate a dataset that can be used to verify the agent's expected behavior. In addition, a synthetic dataset can be used to analyze whether the network is learning appropriate policy with respect to all possible outcomes. Accordingly, we generated three synthetic datasets where for each dataset we had a policy in mind that the agent should learn. Finally, we evaluated these datasets using the loss function and rewards acquired to determine if the agent performed as expected. The hardware configuration of the AWS (p2.xlarge) instance and robot are given in Table 2.

As our model is designed for binary decision-making, to efficiently validate the proposed algorithm we need to

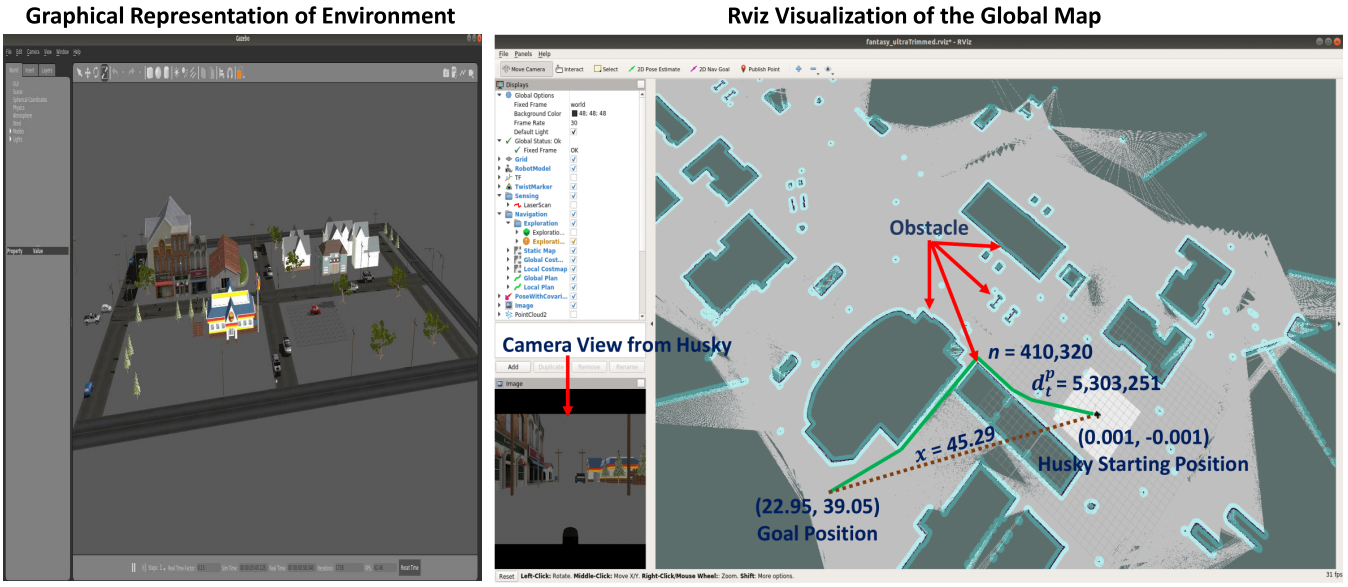


FIGURE 3. Visualization of Gazebo world and global cost map. The left panel shows the graphical simulation environment that was used to validate the proposed algorithm. The right panel shows a global map of the same environment with obstacles indicated (red arrows). In this instance, the Husky's starting position is (0.001, -0.001) and the goal position is (22.95, 39.05). Between the starting position and goal, the shortest path is represented by a green line and the Euclidean distance is $x = 45.29$. The number of nodes to traverse is $n = 410{,}320$, and the input data size is $d_t^p = 5{,}303{,}251$.

TABLE 2. Hardware configuration of the robot and AWS.

	Robot - Local	AWS (p2.xlarge) - Cloud
CPU	Intel Core i7-6700 CPU @ 3.40GHz	2.7 GHz (turbo) Intel Xeon E5-2686 v4
GPU	1 GeForce GTX 1050 - 768 processing cores and 4 gb of GPU memory	1 NVIDIA K80 - 2496 parallel processing cores and 12 gb of GPU memory
RAM	16 gb	61 gb
Cores	8	4
OS	Ubuntu-18.04	Ubuntu-18.04

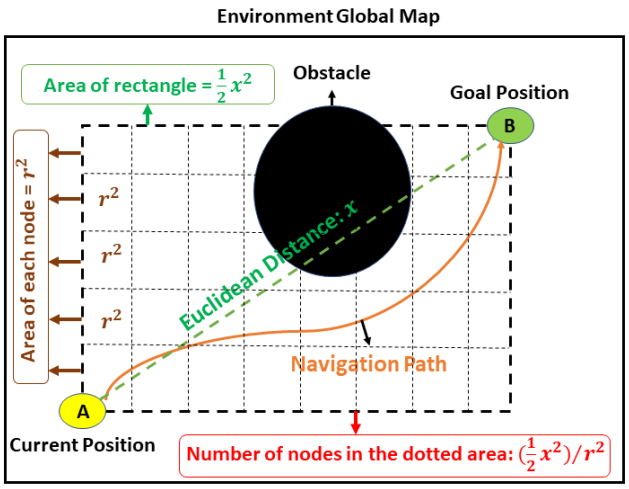


FIGURE 4. Schematic representation for calculating the number of nodes that the robot needs to explore before reaching its destination.

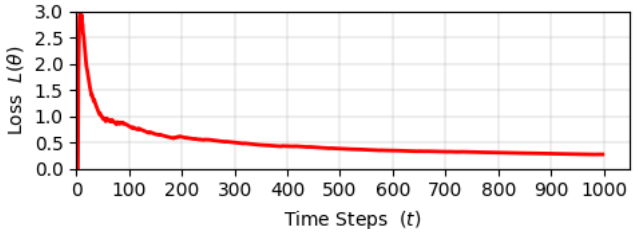
have data distributed between both action space choices; i.e., both cases $l_t^{cloud} < l_t^{local}$ and $l_t^{cloud} > l_t^{local}$ need be reasonably represented. Using the hardware configuration

FIGURE 5. Robot-AWS communication framework. The robot forwards LiDaR data (obstacles) and the current and goal positions to AWS. AWS calculates and forwards the shortest path between those positions to the robot.

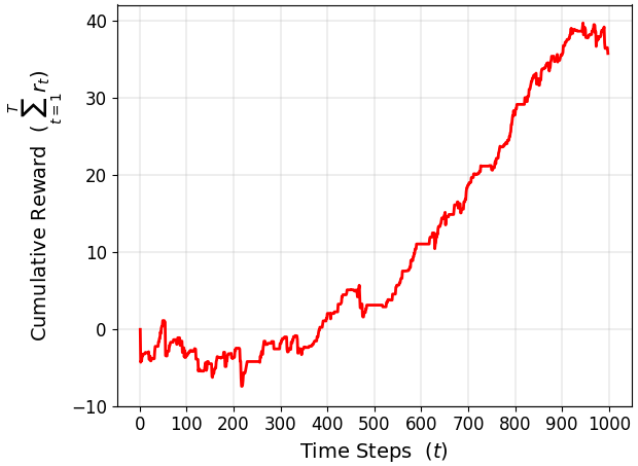
seen in Table 2, we obtained a dataset with a split of about 60:40; i.e., 60 percent of decisions were for cloud computation ($l_t^{cloud} < l_t^{local}$) and 40 percent for local computation ($l_t^{cloud} > l_t^{local}$). Comparatively, if we choose a less capable hardware configuration, the dataset will be skewed towards cloud computation ($l_t^{cloud} < l_t^{local}$); similarly, if we choose a more powerful hardware configuration, the dataset will be skewed towards local computation ($l_t^{cloud} > l_t^{local}$) due to the additional round-trip time required for cloud communication.

A. REAL DATASET

In this part of the experiment, we collected a real dataset where the robot and cloud were connected through ZMQ. We also implemented a cloud timeout functionality (5 s) to handle any network failures. This feature was added to punish the agent for choosing cloud computation if the cloud did not respond in a given timeframe. For every time step (t), the path-planning execution time was collected for both cloud execution (l_t^{cloud}) and local execution (l_t^{local}), along with



(a) Loss value plotted against time step ($t = 1000$); plot shows average loss decreasing and convergence.



(b) Rewards plotted against time step ($t = 1000$); plot shows rewards increasing over time, implying that the network learned a policy for acquiring maximum rewards.

FIGURE 6. Plots showing loss convergence and rewards acquired over time for the real dataset.

state-space values (s_t) and the action (a_t) performed. These values were used to train the DQN network and evaluate the performance of our proposed algorithm.

In Fig. 6a, we plot the average loss ($L(\theta)$) against time step (t) using the loss function defined in Eq. (3). The plot demonstrates convergence and shows average loss as decreasing over time. This can be interpreted as the weight (θ) parameters of the network being optimized by gradient descent over time [72] and also as the network learning a more efficient policy. In Fig. 6b, cumulative reward (r_t) is plotted against time step (t). As our algorithm is based on the foundation of acquiring maximum rewards rather than on the end-goal success criterion, DQN learning occurs over one single episode with t time steps, where $t = 1, 2, 3, \ldots, T$. The episode reward plot shows rewards increasing over time, implying that the network learned the policy for acquiring maximum rewards, i.e., the action (a_t) to take for the given state space (s_t) in order to acquire maximum rewards ($\max(r_t)$). Rewards decreased at some time steps, mainly due to the network performing exploration [71], which helps it to form a better policy. The exploration rate is the probability that our agent will explore the environment rather than exploiting the original policy consensus; we set the exploration rate (epsilon-greedy) [71] value to 0.1.

We also assessed the accuracy of the entire dataset by evaluating the correctness of the action taken in context of the respective execution times. That is, the correct action should result in less execution time. We achieved a final accuracy of 84 percent on the dataset, suggesting that the algorithm learned to take correct actions with respect to the input state space over time. The overall mean execution time of actions selected by the algorithm was 71.28 milliseconds, while the respective means for local and cloud execution were 88.38 and 73.46 milliseconds. Thus, the proposed algorithm achieved execution time savings of almost 23 percent and 3 percent when compared with wholly local or wholly cloud execution. The cumulative execution times were 140.88, 146.09, and 175.81 seconds for algorithm-selected, cloud, and local execution, respectively. Hence, dynamic offloading using this algorithm reduced the latency of the application.

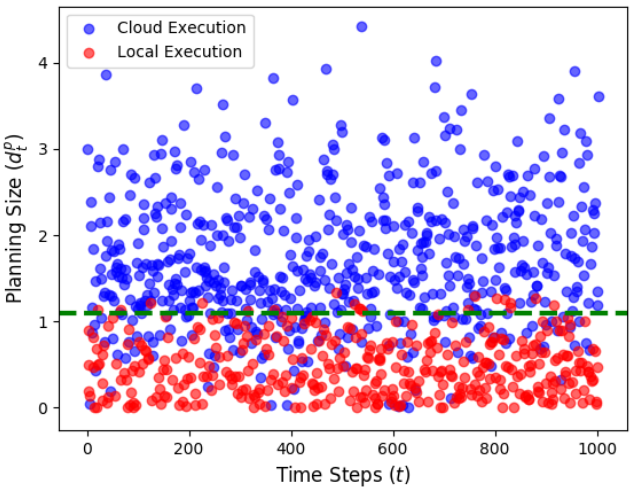
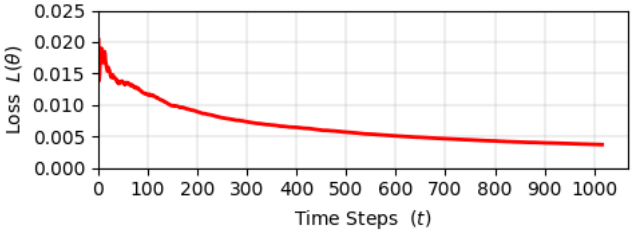


FIGURE 7. Choice analysis plot for the size of path planning data input (d_t^p). For normalized values $d_t^p > 1.1$, cloud computing was the preferred choice of the policy.

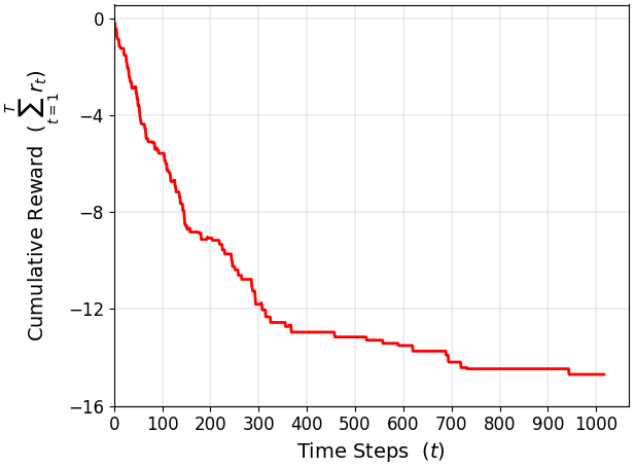
Even though the DQN network performed well with respect to average loss and rewards acquired, it is hard to intuitively ascertain what the agent learned. We plotted action (a_t) with respect to the size of path input data (d_t^p) and time step sequence (t) (Fig. 7) and observed that when input data size surpassed 1.1, the agent chose in the majority of cases to offload the application to the cloud. Hence, data size impacted offloading choice, and when the planning problem data had a size greater than 1.1, executing the application on the cloud was the better choice; the application took less time to execute even with network latency. This observation also strengthens our hypothesis that the size of a problem is directly proportional to execution time. Thus, we conclude that the policy learned by the network is to choose cloud computation for path planning over larger areas.

B. SYNTHETIC DATASET

In the previous section, we saw how the algorithm performed on a real data set. To further evaluate the algorithm's performance and behavior, we generated three different synthetic datasets with three different goals. We wanted to see

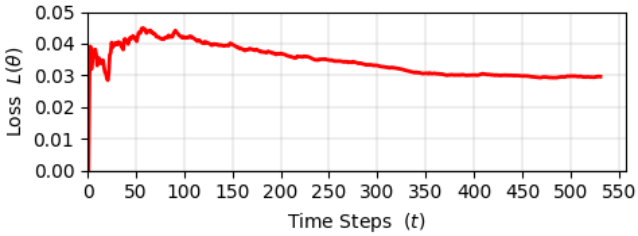


(a) Loss value plotted against time step ($t = 1020$); plot shows average loss decreasing and convergence.

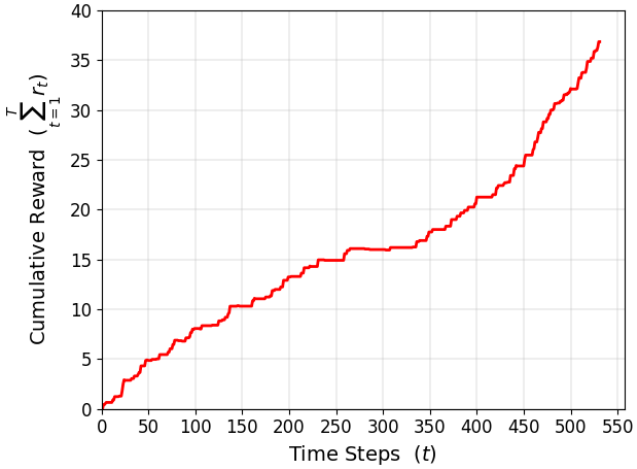


(b) Rewards plotted against the time step ($t = 1020$); plot shows agent learning a policy to not acquire any further negative rewards.

FIGURE 8. Loss convergence and rewards acquired over time for a synthetic dataset. This dataset was generated to observe if the network could learn to do onboard computation for the given state.



(a) Loss value plotted against time step ($t = 530$); plot shows average loss decreasing and convergence.



(b) Rewards plotted against the time step ($t = 530$); plot shows agent learned a policy that choosing cloud computing over local computing gives it a positive reward.

FIGURE 9. Loss convergence and rewards acquired over time for a synthetic dataset. This dataset was generated to observe if the network could learn to do cloud computation for the given state.

if first, the network could learn to do onboard computation for a given state (i.e., local computation); second, if the network could learn to offload an application for a given state (i.e., cloud computation); and finally, if the network could learn a constant CPU availability value and use that as the basis for offloading decisions (i.e., learning a CPU availability at which to offload). Additionally, this section also provides insights into how reward (r_t) assignment varies for local and cloud computation.

1) LOCAL COMPUTATION

To generate the synthetic dataset for local computation, we first obtained the local execution time (l_t^{local}) for path planning at each time step, then multiplied it with a random number from 0.9 to 1.9 to obtain the cloud execution time (l_t^{cloud}),

$$l_t^{cloud} = (0.90 + rand()percent10/10.00) * l_t^{local}. \quad (13)$$

Fig. 8a shows that average loss decreases and converges over time, implying that the algorithm has learned the policy to acquire maximum rewards. Fig. 8b plots cumulative rewards against time step. One important observation is that in this scenario, the possible reward r_t ranges from -1 to 0 inclusive, i.e., $r_t \in [-1, 0]$. While learning, the algorithm can choose to either execute the application onboard or to offload it; that is, in Eq. (10), the value for $c_t^{algorithm}$ can be either c_t^{local} or c_t^{cloud}. In this particular dataset, when $c_t^{algorithm}$ is c_t^{cloud}, the reward is always negative as $l_t^{cloud} > l_t^{local}$. In contrast, when $c_t^{algorithm}$ is c_t^{local}, the reward is zero.

The reward for the network (Fig. 8b) started to stabilize at around time step 300, with no further negative rewards being gained; the algorithm had by that point learned the best possible action is not to gain any further negative rewards and to always choose local computation instead of cloud computation.

2) CLOUD COMPUTATION

To generate the cloud computation synthetic dataset, we first obtained the local execution time (l_t^{local}) for path-planning at each time step, then multiplied that with a random number from 0.1 to 1.1 to obtain the cloud execution time (l_t^{cloud}),

$$l_t^{cloud} = (0.1 + rand()percent10/10.00) * l_t^{local}. \quad (14)$$

Fig. 9a shows the average loss with this dataset as converging over time, implying that the algorithm learned the policy to acquire maximum rewards. Fig. 9b plots cumulative rewards against time steps. In this scenario, the synthetic data is skewed to favor cloud computation and the possible reward r_t for the agent ranges from 0 to 1 inclusive, i.e., $r_t \in [0, 1]$. Thus, when $c_t^{algorithm}$ is c_t^{cloud}, the reward is always positive as

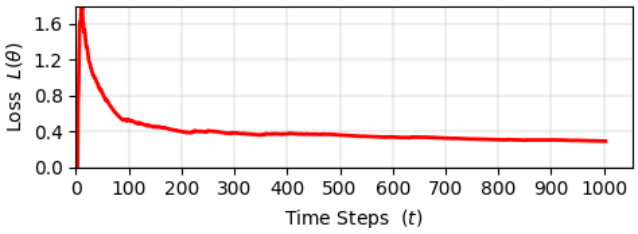
$l_t^{cloud} < l_t^{local}$, and when $c_t^{algorithm}$ is c_t^{local}, the reward is zero. That the reward accumulated was always increasing indicates the agent learned that choosing cloud computing over local computing gives it a positive reward, and thus always chose cloud computing instead of local computing.

3) LEARNING A CPU VALUE TO OFFLOAD

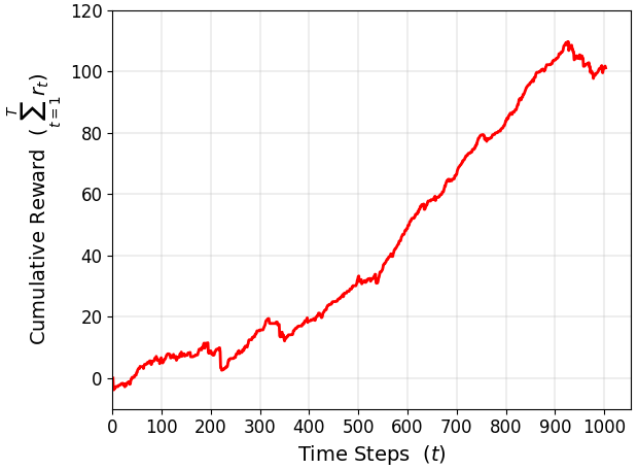
In the previous sections, we observed how data input size (d_t^p) affects the decision to offload. In this section, we wanted to see if the network can learn about a CPU value $u_t = x$ and use this value to decide whether to offload or do local computation. We generated a synthetic dataset where execution times l_t^{cloud} and l_t^{local} were set as follows,

$$\begin{aligned} l_t^{cloud} &< l_t^{local} \quad when \quad u_t < x \\ l_t^{cloud} &> l_t^{local} \quad when \quad u_t > x. \end{aligned} \tag{15}$$

When collecting the synthetic data, we observed CPU availability on the local machine to hover between 50 and 70 percent, and hence set the threshold value x as 60 percent. This allowed us to collect data that was distributed on both sides of x.



(a) Loss value plotted against time step ($t = 1050$); plot shows average loss decreasing and convergence.



(b) Rewards plotted against the time step ($t = 1050$); plot shows agent learning a policy to acquire positive rewards.

FIGURE 10. Loss convergence and rewards acquired over time for a synthetic dataset. This dataset was generated to observe if the network could learn to use CPU availability as the basis for deciding whether to offload the application.

Fig. 10b plots cumulative reward against time step. It is difficult to interpret from this plot what the agent has learned; accordingly, we also rendered a scatter plot as illustrated in Fig. 11 and a confusion matrix as given in Fig. 12 in order to decode what the agent learned from the dataset. The goal of the policy was to extract the CPU availability value (y) from the synthetic data and use it as the basis for deciding between offloading and local computation. The learned threshold value (y) was around 0.61 (Fig. 11), while the pre-set x value for generating l_t^{cloud} and l_t^{local} was 0.6; we can therefore conclude that the network was within a reasonable margin of error. This is further supported by the training accuracy (Fig. 12), with the network having achieved 94 percent accuracy; most wrong actions took place when c_t was around 0.6. Finally, the average loss decreased and converged over time (Fig. 10a), implying that the algorithm learned the policy of offloading the application whenever CPU availability was less than 61 percent and of doing local computation when it was greater than 61 percent.

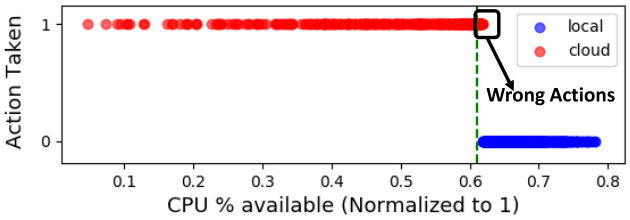


FIGURE 11. Choice analysis plot illustrating the CPU threshold value learned by the offloading decision policy, which was 0.61.

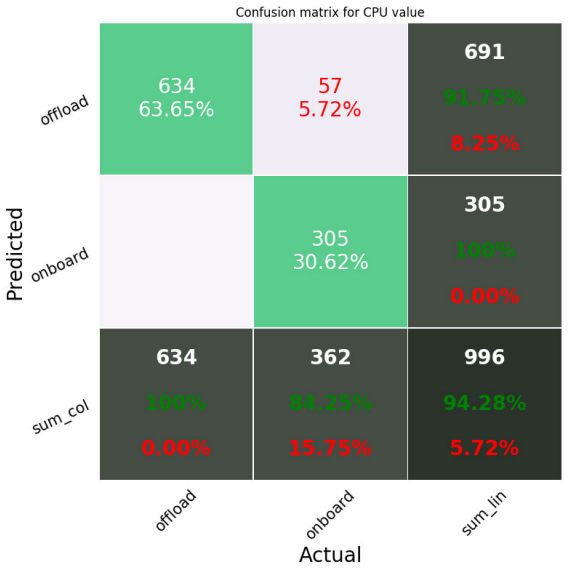


FIGURE 12. Confusion matrix used to determine the accuracy of the network; over 94 percent accuracy was achieved.

C. COMPARATIVE EVALUATION WITH LONG SHORT-TERM MEMORY ALGORITHM

We further evaluate the proposed DRL algorithm by comparing it with another state-of-the-art machine learning model. One approach for such evaluation is to compare the algorithm with other DRL models such as DoubleDQN [73] or

DuelingDoubleDQN [74]. These models are quite similar to each other as they all learn value functions and act greedily based on those values; the major difference between them lies in performance [54], and hence, comparing these models will imply comparing performance parameters such as how loss convergence varies for various batch sizes, learning rates, discounted factors, etc. However, the major emphasis of this paper is to understand the policy learned by the agent and the final accuracy of the algorithm. As such, rather than comparing performance among various DRL models, a more appropriate evaluation is to compare the accuracy of the proposed algorithm with other state-of-the-art machine learning models.

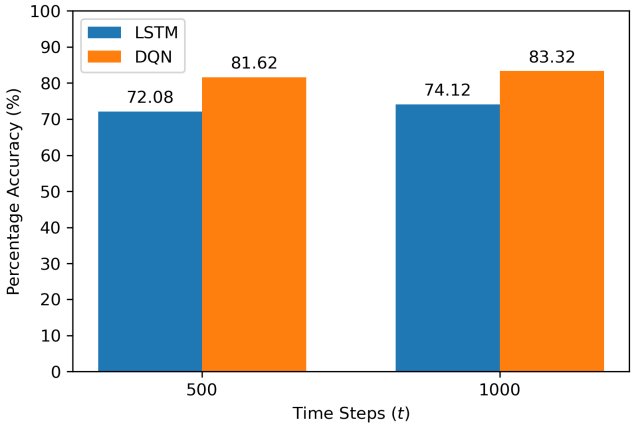


FIGURE 13. Bar chart comparing application offloading prediction accuracy for LSTM and proposed DRL algorithms. Prediction accuracy was calculated by training the models with the first 500 time steps (t) and with the entire dataset of 1000 time steps. Our proposed DRL algorithm achieved better accuracy in both cases.

The literature has predominantly suggested using long short-term memory (LSTM) models for predicting execution time [75]. Hence, we implemented a LSTM model based on [76] (hidden layers = 4, batch size = 50, steps/batch = 8) to predict application execution time from input data size. Fig. 6a and Fig. 6b demonstrate that for the DRL model, loss starts to converge and cumulative rewards to steadily increase from around $t = 400$; this implies that by that step, the agent has figured out a policy to acquire positive rewards. Thus, we used actual data from the first 500 time steps to train the LSTM model, with an 80:20 training:validation split. We then used the trained LSTM to predict execution times for both cloud and local execution of the application and compared resulting predicted actions with actual actions obtained from the proposed DRL algorithm. As shown in Fig. 13, while the LSTM model achieved a final accuracy of 72.08 percent, the proposed DRL algorithm achieved a greater accuracy of 81.62 percent. Similarly, when using the entire 1000-time step dataset with an 80:20 split to train the LSTM model, it had a final accuracy of 74.12 percent, whereas the proposed DRL algorithm achieved 83.32 percent. Hence, we can conclude that our proposed DRL algorithm is comparable to or better than LSTM in predicting appropriate actions to take concerning application offloading with reference to the application data input size.

D. DISCUSSION

The results demonstrated that the agent was able to learn a policy that maximized reward and reduced overall application execution latency. We also observed that the size of application input data from the state space played an important role in the policy forming a consensus. In this section, we will present some of the limitations and key observations of the present study. Notably, the overall accuracy after training was greater than 80 percent for all experimental cases we tested. Considering we only carried out the simulation for around 1000 time steps, this accuracy gives us confidence in the policy learned by the agent. We are also certain that the accuracy can be increased by increasing the time steps (t) taken to train the network.

One important observation from the results is the algorithm convergence. From the loss plots, we can discern that the algorithm started to converge before 500 time steps in all cases. Such rapid convergence is vital in reducing the amount of time that needs to be spent on training and enables the algorithm to start taking correct actions (a_t) more consistently in less time.

One limitation of this study is that there is no one true guideline that defines how to correctly represent the size of input data for an application, and sometimes it falls to the personal intuition of the human operator to correctly represent this metric. Representations of input data size can vary based on factors such as the algorithm implemented, the computational model used, and also on the application; for example, we represented the size of path-planning input data as $n\log n$. Similarly, if we want to search n randomly ordered elements, the input data size is n; binary search is $\log n$; and sorting is $n\log n$ [77]. For image detection based algorithms, we can assume that frames per second multiplied by the data size of each frame would be a good representation of the input data [78]. Ultimately, the accuracy of the proposed algorithm depends on the correctness of the input data size determination, and so it's not a foolproof system.

One of the other limitations of this study is that the Gazebo-based experimental setup might not offer a perfect representation of the network latency in a robotic environment. The experiment was carried out on a computer that was connected to the internet through a LAN cable with capacity greater than 400 Mbps. The latency between the computer and AWS on this setup was constant at around 30 milliseconds. In a real-world scenario, this case might not hold as robots are mostly connected through a wireless connection, and so might experience fluctuating internet speeds along with network dropouts depending on the environment. Even though we introduced a latency parameter (b_t) in the state space, the network never truly learned anything meaningful about connection latency as the parameter had minimal variation.

Finally, even though we verified the algorithm for a robot navigation application, it can be generalized to a majority of robotic applications. This research also verifies that there is a relationship between input data size and the time needed for execution. Hence, if we model our state space to capture information concerning data size, the algorithm will converge and learn a policy for offloading decision-making. A video on the paper is available for reference at: https://youtu.be/JAwxaOH9BFk.

VI. CONCLUSION

A dynamic computational offloading solution based on DQN has been proposed in this paper. The proposed algorithm was able to learn an optimal policy on when to offload an application based on state-space values. The state space was built on the assumption that the size of input data submitted to an application directly impacts its execution time, and we successfully validated this assumption using the size of path input data (d_t^p) in a navigation application. The algorithm was designed as a continuous task problem with discrete action space for every time step (t), and at each step the execution time for path planning in robot navigation was collected for both cloud execution and local execution, as were state-space values and the action performed. These values were used to train the DQN network and evaluate the performance of our proposed algorithm.

The effectiveness of the algorithm was evaluated by observing the loss and rewards over time steps. The results showed convergence and the agent learning a policy to maximize rewards over time. To further evaluate the algorithm, we also generated three synthetic datasets, each designed around a particular policy that the agent should learn. To compare the performance of the proposed algorithm with another state-of-the-art algorithm, we trained and validated an LSTM model on the same dataset, using either 500 or 1000 time steps. In both cases, the proposed algorithm achieved 9 percent greater accuracy over the LSTM algorithm. The network successfully extracted the key features from these datasets, and the agent learned the policy that we intended. All told, the results have validated the effectiveness of our proposed algorithm.

Further work to accommodate a multi-robot scenario is already underway. In future work, we will also focus on adding additional cost parameters such as energy usage. Finally, we are exploring various application prioritization mechanisms that can help prioritize mission-critical applications, and we plan to apply mechanism-specific cost parameters to the applications. For example, applications that are critical for robot functioning will have reducing latency as the first priority, while applications that are not mission-critical will prioritize reducing their own energy use.

REFERENCES

[1] W. Wang and K. Siau, "Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda," *J. Database Manage.*, vol. 30, no. 1, pp. 61–79, Jan. 2019.

[2] S. H. Alsamhi, O. Ma, and M. S. Ansari, "Convergence of machine learning and robotics communication in collaborative assembly: Mobility, connectivity and future perspectives," *J. Intell. Robot. Syst.*, vol. 98, nos. 3–4, pp. 541–566, Jun. 2020.

[3] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Trans. Autom. Sci. Eng.*, vol. 12, no. 2, pp. 398–409, Apr. 2015.

[4] P. C. Sen, M. Hajra, and M. Ghosh, "Supervised classification algorithms in machine learning: A survey and review," in *Emerging Technology in Modelling and Graphics*, J. K. Mandal and D. Bhattacharya, Eds. Singapore: Springer, 2020, pp. 99–111.

[5] W. Chen, Y. Yaguchi, K. Naruse, Y. Watanobe, K. Nakamura, and J. Ogawa, "A study of robotic cooperation in cloud robotics: Architecture and challenges," *IEEE Access*, vol. 6, pp. 36662–36682, 2018.

[6] M. Foughali, B. Berthomieu, S. D. Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "Formal verification of complex robotic systems on resource-constrained platforms," in *Proc. 6th Conf. Formal Methods Softw. Eng.*, Jun. 2018, pp. 2–9.

[7] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

[8] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Gener. Comput. Syst.*, vol. 79, pp. 849–861, Feb. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17302224

[9] J. Kuffner, "Cloud-enabled humanoid robots," in *Proc. 10th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)*, Nashville, TN, USA, Dec. 2010, pp. 19–23. [Online]. Available: https://ci.nii.ac.jp/naid/10031099795/en/

[10] O. Saha and P. Dasgupta, "A comprehensive survey of recent trends in cloud robotics architectures and applications," *Robotics*, vol. 7, no. 3, p. 47, Aug. 2018. [Online]. Available: https://www.mdpi.com/2218-6581/7/3/47

[11] H. Everett, *Sensors for Mobile Robots*. Boca Raton, FL, USA: CRC Press, 1995.

[12] J. Abella, M. Padilla, J. D. Castillo, and F. J. Cazorla, "Measurement-based worst-case execution time estimation using the coefficient of variation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 22, no. 4, pp. 1–29, Jul. 2017, doi: 10.1145/3065924.

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019, *arXiv:1509.02971*. [Online]. Available: https://arxiv.org/abs/1509.02971

[14] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017.

[15] H. Jiang, H. Wang, W.-Y. Yau, and K.-W. Wan, "A brief survey: Deep reinforcement learning in mobile robot navigation," in *Proc. 15th IEEE Conf. Ind. Electron. Appl. (ICIEA)*, Nov. 2020, pp. 592–597.

[16] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Dec. 2020, pp. 737–744.

[17] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart., 2017.

[18] H. Wu, "Multi-objective decision-making for mobile cloud offloading: A survey," *IEEE Access*, vol. 6, pp. 3962–3976, 2018.

[19] C. Jiang, X. Cheng, H. Gao, X. Zhou, and J. Wan, "Toward computation offloading in edge computing: A survey," *IEEE Access*, vol. 7, pp. 131543–131558, 2019.

[20] R. Arumugam, V. R. Enti, L. Bingbing, W. Xiaojun, K. Baskaran, F. F. Kong, A. S. Kumar, K. Dee Meng, and G. W. Kit, "DAvinCi: A cloud computing framework for service robots," in *Proc. IEEE Int. Conf. Robot. Automat.*, May 2010, pp. 3084–3089.

[21] M. Inaba, S. Kagami, F. Kanehiro, Y. Hoshino, and H. Inoue, "A platform for robotics research based on the remote-brained robot approach," *Int. J. Robot. Res.*, vol. 19, no. 10, pp. 933–954, 2000, doi: 10.1177/02783640022067878.

[22] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Gálvez-López, K. Häussermann, R. Janssen, J. M. M. Montiel, A. Perzylo, B. Schießle, M. Tenorth, O. Zweigle, and R. De Molengraft, "RoboEarth," *IEEE Robot. Autom. Mag.*, vol. 18, no. 2, pp. 69–82, Jun. 2011.

[23] G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel, "Rapyuta: A cloud robotics platform," *IEEE Trans. Autom. Sci. Eng.*, vol. 12, no. 2, pp. 481–493, Apr. 2015.

[24] R. Doriya, P. Chakraborty, and G. C. Nandi, "'Robot-cloud': A framework to assist heterogeneous low cost robots," in *Proc. Int. Conf. Commun., Inf. Comput. Technol. (ICCICT)*, Oct. 2012, pp. 1–5.

[25] Y. Li, H. Wang, B. Ding, and W. Zhou, "RoboCloud: Augmenting robotic visions for open environment modeling using Internet knowledge," *Sci. China Inf. Sci.*, vol. 61, no. 5, Apr. 2018, Art. no. 050102, doi: 10.1007/s11432-017-9380-5.

[26] L. Riazuelo, J. Civera, and J. M. M. Montiel, "C2TAM: A cloud framework for cooperative tracking and mapping," *Robot. Auton. Syst.*, vol. 62, no. 4, pp. 401–413, Apr. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889013002248

[27] M. Penmetcha, S. Sundar Kannan, and B.-C. Min, "Smart cloud: Scalable cloud robotic architecture for Web-powered multi-robot applications," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2020, pp. 2397–2402.

[28] L. Muratore, B. Lennox, and N. Tsagarakis, "Xbotcloud: A scalable cloud computing infrastructure for xbot powered robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2018, pp. 1–9.

[29] N. Tian, M. Matl, J. Mahler, Y. X. Zhou, S. Staszak, C. Correa, S. Zheng, Q. Li, R. Zhang, and K. Goldberg, "A cloud robot system using the dexterity network and berkeley robotics and automation as a service (Brass)," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2017, pp. 1615–1622.

[30] G. Mohanarajah, V. Usenko, M. Singh, R. D'Andrea, and M. Waibel, "Cloud-based collaborative 3D mapping in real-time with low-cost robots," *IEEE Trans. Autom. Sci. Eng.*, vol. 12, no. 2, pp. 423–431, Apr. 2015.

[31] A. Rahman, J. Jin, A. Cricenti, A. Rahman, and M. Panda, "Motion and connectivity aware offloading in cloud robotics via genetic algorithm," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2017, pp. 1–6.

[32] L. Wang, M. Liu, and M. Q.-H. Meng, "A hierarchical auction-based mechanism for real-time resource allocation in cloud robotic systems," *IEEE Trans. Cybern.*, vol. 47, no. 2, pp. 473–484, Feb. 2017.

[33] Z. Hong, H. Huang, S. Guo, W. Chen, and Z. Zheng, "QoS-aware cooperative computation offloading for robot swarms in cloud robotics," *IEEE Trans. Veh. Technol.*, vol. 68, no. 4, pp. 4027–4041, Apr. 2019.

[34] D. Spatharakis, M. Avgeris, N. Athanasopoulos, D. Dechouniotis, and S. Papavassiliou, "A switching offloading mechanism for path planning and localization in robotic applications," in *Proc. Int. Conf. Internet Things (iThings), IEEE Green Comput. Commun. (GreenCom), IEEE Cyber, Phys. Social Comput. (CPSCom), IEEE Smart Data (SmartData), IEEE Congr. Cybermatics (Cybermatics)*, Nov. 2020, pp. 77–84.

[35] Q.-V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, L. B. Le, W.-J. Hwang, and Z. Ding, "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art," *IEEE Access*, vol. 8, pp. 116974–117017, 2020.

[36] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 181, May 2021, Art. no. 103005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521000321

[37] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran, and M. Shoaib, "Bringing computation closer toward the user network: Is edge computing the solution?" *IEEE Commun. Mag.*, vol. 55, no. 11, pp. 138–144, Nov. 2017.

[38] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, vol. 135. Cambridge, MA, USA: MIT Press, 1998.

[39] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[40] S. S. Mousavi, M. Schukat, and E. Howley, "Deep reinforcement learning: An overview," in *Proc. SAI Intell. Syst. Conf. (IntelliSys)*, Y. Bi, S. Kapoor, and R. Bhatia, Eds. Cham, Switzerland: Springer, 2018, pp. 426–440.

[41] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1085–1101, May 2021.

[42] X. Qiu, L. Liu, W. Chen, Z. Hong, and Z. Zheng, "Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing," *IEEE Trans. Veh. Technol.*, vol. 68, no. 8, pp. 8050–8062, Aug. 2019.

[43] M. Tang and V. W. S. Wong, "Deep reinforcement learning for task offloading in mobile edge computing systems," *IEEE Trans. Mobile Comput.*, early access, Nov. 10, 2020, doi: 10.1109/TMC.2020.3036871.

[44] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, Jan. 2021.

[45] Y. Dai, K. Zhang, S. Maharjan, and Y. Zhang, "Edge intelligence for energy-efficient computation offloading and resource allocation in 5G beyond," *IEEE Trans. Veh. Technol.*, vol. 69, no. 10, pp. 12175–12186, Oct. 2020.

[46] S. Chinchali, A. Sharma, J. Harrison, A. Elhafsi, D. Kang, E. Pergament, E. Cidon, S. Katti, and M. Pavone, "Network offloading policies for cloud robotics: A learning-based approach," 2019, *arXiv:1902.05703*. [Online]. Available: http://arxiv.org/abs/1902.05703

[47] H. Liu, S. Liu, and K. Zheng, "A reinforcement learning-based resource allocation scheme for cloud robotics," *IEEE Access*, vol. 6, pp. 17215–17222, 2018.

[48] Z. Peng, J. Lin, D. Cui, Q. Li, and J. He, "A multi-objective trade-off framework for cloud resource scheduling based on the deep Q-network algorithm," *Cluster Comput.*, vol. 23, pp. 1–15, Jan. 2020.

[49] A. Valmari, *The State Explosion Problem*. Berlin, Germany: Springer, 1998, pp. 429–528, doi: 10.1007/3-540-65306-6_21.

[50] D. S. Hochba, "Approximation algorithms for np-hard problems," *SIGACT News*, vol. 28, no. 2, pp. 40–52, Jun. 1997, doi: 10.1145/261342.571216.

[51] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013, *arXiv:1312.5602*. [Online]. Available: http://arxiv.org/abs/1312.5602

[52] K. Cho, Y. Sung, and K. Um, "A production technique for a Q-table with an influence map for speeding up Q-learning," in *Proc. Int. Conf. Intell. Pervas. Comput. (IPC)*, Oct. 2007, pp. 72–75.

[53] H. Zhu, I. C. Paschalidis, and M. E. Hasselmo, "Feature extraction in Q-learning using neural networks," in *Proc. IEEE 56th Annu. Conf. Decis. Control (CDC)*, Dec. 2017, pp. 3330–3335.

[54] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017, doi: 10.1109/MSP.2017.2743240.

[55] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep Q-learning with model-based acceleration," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, vol. 48, 2016, pp. 2829–2838.

[56] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3, pp. 279–292, May 1992.

[57] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[58] C. Chicone, *Stability Theory of Ordinary Differential Equations*. New York, NY, USA: Springer, 2011, pp. 1653–1671, doi: 10.1007/978-1-4614-1806-1_106.

[59] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for MEC," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2018, pp. 1–6.

[60] M. Quigley, B. P. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, Kobe, Japan, May 2009, vol. 3, no. 3.2, p. 5.

[61] *Gazebo*. Accessed: Jan. 7, 2021. [Online]. Available: http://gazebosim.org/

[62] *Simulating Jackal—Jackal Tutorials 0.5.4 Documentation*. Accessed: Jan. 5, 2021. [Online]. Available: https://www.clearpathrobotics.com/assets/guides/kinetic/jackal/simulation.html

[63] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[64] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968, doi: 10.1109/tssc.1968.300136.

[65] P. Raja, "Optimal path planning of mobile robots: A review," *Int. J. Phys. Sci.*, vol. 7, no. 9, pp. 1314–1320, Feb. 2012.

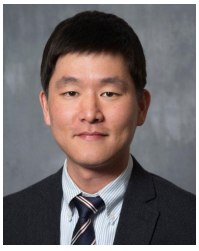
[66] A. V. Goldberg and R. E. Tarjan, "Expected performance of Dijkstra's shortest path algorithm," NEC Res., Tech. Rep. TR-96-062, 1996.

[67] *ROS Notes: Map Resolution—New Screwdriver*. Accessed: Jan. 12, 2021. [Online]. Available: https://newscrewdriver.com/2018/09/21/ros-notes-map-resolution/

[68] *Zeromq*. Accessed: Jan. 13, 2021. [Online]. Available: https://zeromq.org/
[69] L. K. Hansen and P. Salamon, "Neural network ensembles," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 10, pp. 993–1001, Oct. 1990.
[70] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, Nov. 2018, Art. no. e00938. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405844018332067
[71] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Dec. 2016, pp. 1–8.
[72] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, *arXiv:1609.04747*. [Online]. Available: http://arxiv.org/abs/1609.04747
[73] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," 2015, *arXiv:1509.06461*. [Online]. Available: https://arxiv.org/abs/1509.06461
[74] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016, *arXiv:1511.06581*. [Online]. Available: https://arxiv.org/abs/1511.06581
[75] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu, "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning," *Future Gener. Comput. Syst.*, vol. 102, pp. 847–861, Jan. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19308209
[76] Y. Miao, G. Wu, M. Li, A. Ghoneim, M. Al-Rakhami, and M. S. Hossain, "Intelligent task prediction and computation offloading based on mobile-edge cloud computing," *Future Gener. Comput. Syst.*, vol. 102, pp. 925–931, Jan. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19320862
[77] *The Fundamentals of the Big-O Notation | by Ruben Winastwan | Towards Data Science*. Accessed: Feb. 15, 2021. [Online]. Available: https://towardsdatascience.com/the-fundamentals-of-the-big-o-notation-7fe14210b675
[78] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767

MANOJ PENMETCHA (Graduate Student Member, IEEE) received the B.S. degree in information technology from Osmania University, India, in 2010, and the M.S. degree in computer and information technology from Purdue University, West Lafayette, IN, USA, in 2012, where he is currently pursuing the Ph.D. degree in technology.

His research interests include machine learning, multi-robot systems, cloud robotics, cloud computing, edge computing, and wireless networks.



BYUNG-CHEOL MIN (Member, IEEE) received the B.S. degree in electronics engineering and the M.S. degree in electronics and radio engineering from Kyung Hee University, Yongin, South Korea, in 2008 and 2010, respectively, and the Ph.D. degree in technology with a specialization in robotics from Purdue University, West Lafayette, IN, USA, in 2014.

He was a Postdoctoral Fellow with the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA. He is currently an Associate Professor with the Department of Computer and Information Technology and the Director of the SMART Laboratory, Purdue University. His research interests include multi-robot systems, human–robot interaction, robot design and control, with applications in field robotics, and assistive technology and robotics.

Dr. Min was a recipient of the NSF CAREER Award, in 2019; the Purdue PPI Outstanding Faculty in Discovery Award, in 2019; the Purdue CIT Outstanding Graduate Mentor Award, in 2019; and the Purdue Focus Award, in 2020.