2019

A deep recurrent Q network towards self-adapting distributed microservice architecture

Basel Magableh

Muder Almiani

Received: 11 December 2018 | Revised: 14 October 2019 | Accepted: 16 October 2019

DOI: 10.1002/spe.2778

RESEARCH ARTICLE

WILEY

A deep recurrent Q network towards self-adapting distributed microservice architecture

Basel Magableh[1] | **Muder Almiani**[2]

[1]Technological University of Dublin, Dublin, Ireland

[2]Al-Hussein Bin Talal University, Ma'an, Jordan

Correspondence
Basel Magableh, School of Computer Science, Technological University, Dublin, Ireland.
Email: basel.magableh@tudublin.ie

Muder Almiani, School of Computer Science, Al-Hussein Bin Talal University, Ma'an 71111, Jordan.
Email: malmiani@my.bridgeport.edu

Summary

One desired aspect of microservice architecture is the ability to self-adapt its own architecture and behavior in response to changes in the operational environment. To achieve the desired high levels of self-adaptability, this research implements distributed microservice architecture model running a swarm cluster, as informed by the Monitor, Analyze, Plan, and Execute over a shared Knowledge (MAPE-K) model. The proposed architecture employs multiadaptation agents supported by a centralized controller, which can observe the environment and execute a suitable adaptation action. The adaptation planning is managed by a deep recurrent Q-learning network (DRQN). It is argued that such integration between DRQN and Markov decision process (MDP) agents in a MAPE-K model offers distributed microservice architecture with self-adaptability and high levels of availability and scalability. Integrating DRQN into the adaptation process improves the effectiveness of the adaptation and reduces any adaptation risks, including resource overprovisioning and thrashing. The performance of DRQN is evaluated against deep Q-learning and policy gradient algorithms, including (1) a deep Q-learning network (DQN), (2) a dueling DQN (DDQN), (3) a policy gradient neural network, and (4) deep deterministic policy gradient. The DRQN implementation in this paper manages to outperform the aforementioned algorithms in terms of total reward, less adaptation time, lower error rates, plus faster convergence and training time. We strongly believe that DRQN is more suitable for driving the adaptation in distributed services-oriented architecture and offers better performance than other dynamic decision-making algorithms.

KEYWORDS

deep Q-learning networks, multiagent environment, policy approximation, Q-learning algorithms, recurrent Q-learning networks, reinforcement learning, self-adaptive architectures, service-oriented architecture

1 | INTRODUCTION

Self-adaptability refers to the ability of software architecture to adjust its behavior and structure in response to contextual changes found in its operating environment. Supporting microservices with self-adaptability faces the challenge of identifying the best adaptation action among a set of possible adaptation actions. The goal is to identify the adaptation action that yield the highest reward and preserve the architecture state. Alternatively, reinforcement learning (RL) provides software architecture with the possibility to learn a specific policy that can be used to take decisions among a set of actions and maximizing the cumulative rewards yielded from executing a specific action.[1]

The RL algorithm can be used for planning the adaptation and learning the most profitable rewards to be gained from performing adaptation actions in microservice architecture. However, supporting the adaptation planning requires: (1) sequential decision-making in a continuous domain of states and actions due to the problem of unforeseen contextual changes that can be found at runtime in microservice architecture and (2) a mechanism to estimate the reward of the adaptation action. Unfortunately, the reward will be unknown and delayed until the adaptation action is completed and the architecture enters its new state. The adaptation manager has no idea what the transition probabilities are between the system's states, and it does not know what the rewards are going to be either once it moves from one state to another. This problem is referred to as partially observable Markov decision processes (POMDPs). In a POMDP environment, the agent has only a partial view of the full environment's state and action space, which leads to poor performance of the reinforcement algorithms.

For this aim, this paper studies the possibility to support microservice architecture with self-adaptability supported by an RL algorithm that enables the microservice architecture to adapt the unforeseen changes in partially observable environment. In this paper, the microservice architecture is implemented as a Monitor, Analyze, Plan, and Execute over a shared Knowledge (MAPE-K) model using a Docker swarm as a cluster management framework. Implementing the microservice architecture using Docker swarm limits this research to study the possibility of supporting the adaptation in centralized multiagent microservice architecture. In Docker swarm,* only the elected leader makes all swarm management and orchestration decisions for the cluster. Once the leader node dies unexpectedly, other managers can be elected to serve as a leader for the swarm, and then the leader restores the cluster state. This particular implementation of Docker swarm enforces this research to use multiple agents with centralized controller (ie, swarm leader). At each node of the swarm, the agent is collecting and observing the computational resources and feeds them to the adaptation manager running at the swarm leader. Once all observations are collected by the adaptation manager, a deep Q-learning network (DQN) is used to calculate the best adaptation action that will yield the highest reward. Because of the problem of partial observability and the need to increase the performance of the DQN, we propose in this paper integrating the DQN with a recurrent neural network (RNN) architecture. Integrating DQN with an RNN is found to improve the performance of the adaptation manager and improves the adaptability of the architecture. In addition, it manages to lower the training time and reduces the adaptation time. This integration is called deep recurrent Q-learning network (DRQN) algorithm.

More importantly, by employing an RNN and gated recurrent units (GRUs), the DRQN is able to reveal the patterns between the collected observations and the cumulative yielded rewards for each pair of state-action. The DRQN has the ability to perform backpropagation through time, to find the derivatives of the error with respect to the calculated weights of the observations, which enables the DRQN to improve the gradient descent, and subsequently minimizes the loss and error rate of the predicted Q-value. A GRU enables the DRQN to possess a memory of all the yielded rewards from each pair of state-action pairs, so that the DRQN will be able to identify which action sequences return the maximum reward and consequently DRQN reaches a terminal state in less time than DQN. The problematic issues of credit assignment and temporal difference are solved by the ability of the DRQN to feed the current state output as an input for the next state combined with the collected observations, which improve its accuracy and significantly reduces the required training time.

The objectives of this research are: to provide a test bed of self-adapting distributed microservice architecture, which can be used by other researchers to experiment with various types of adaptation techniques, to evaluate which RL algorithm is more suitable to support dynamic adaptation in microservices cluster, to propose a service stack that can be used to implement a distributed microservice architecture that confirms to the MAPE-K model, and to propose adaptation manager that implements a Markov decision process (MDP) agents.[2] Those MDP agents are able to collect observations about the environment and can execute adaptation actions elected by the DRQN algorithm running at the adaptation manager. Also, this paper evaluates the effectiveness of DRQN algorithms against DQNs,[3] DRQN,[4] and policy gradient (PG) algorithms such as a PG neural network (PGNN),[5] and deep deterministic PG (DDPG).[6]

This paper is structured as follows. Section 2 provides an overview of self-adaptation techniques and surveys the approaches used for context sensing, adaptation planning, and RL. Section 3 presents a model of a distributed microservice architecture that can continuously observe and adapt the architecture. The proposed method of adaptation planning and execution is discussed in Section 3.2. Section 3.3 discusses different approaches for implementing RL algorithms.

*https://docs.docker.com/engine/swarm/

The calculation of the reward function is explained in Section 3.4. Section 4 addresses the implementation of this model and gives more details about the architecture of the implementation of the five RL algorithms. Section 5 evaluates the effectiveness of the five RL algorithms (DQN, DDQN, DRQN, PGNN, and DDPG) in adaptation planning and execution. Section 6 summarizes this research, highlighting its contribution and setting out potential future work.

2 | RELATED WORK

Self-adaptability refers to the ability of service-oriented architecture (SOA) to modify its own structure and behavior in response to changes in the operating environment.[7] High levels of self-adaptability present the challenges of self-organizing, self-tuning, and self-healing the architecture against an interruption. Moreover, because of the service pervasiveness and the need to make any adaptation strategy effective and successful, adaptation actions must be considered in conjunction with service availability,[†] and reliability by providing an intelligent selection of the adaptation actions, so that the performed action meets the adaptation goals, objectives, and the desired architecture quality attributes.[8-10] Thus, adaptation planning requires mechanisms that are able to learn how to choose adaptation actions from continuous actions space. The adaptation strategy must able to optimize the adaptation actions to guarantee that the architecture will reach the adaptation objectives.[3] On the other hand, RL provides software agents with the possibility to learn a specific policy that can be used to take decisions among a set of actions by maximizing the cumulative rewards yielded from executing a specific action.[1]

2.1 | Self-adaptive microservice architecture

Self-adaptive architecture is characterized by a number of properties best referred to as autonomic.[11] These properties are grouped under the "self-* properties" heading; they include self-organization, self-healing, self-optimization, and self-protection.[12] Self-adapting architecture refers to the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems and, accordingly, take suitable actions to prevent a system failure.[12] Self-adapting aspects of SOA require a decision-making strategy that can work in real time. This is essential for the architecture to reason about their own state and its surrounding environment in a closed control loop model and act appropriately.[13]

Typically, a self-adapting system follows the MAPE-K model. An efficient self-adaptive system should implements the MAPE-K model, which will include: (1) gathering of data related to the surrounding context (context sensing), (2) context observation and detection, (3) dynamic decision-making, (3) adaptation execution to achieve the adaptation objectives defined as quality of service (QoS), and (4) verification and validation of the applied adaptation actions in terms of the ability of the taken action to meet the adaptation objectives and meet the desired QoS.

However, many approaches are used for achieving high levels of self-adaptability though context reasoning; a model that involves context collection, observation, and detection of contextual changes in the operational environment.[14] Also, the ability of the system to dynamically adjust its behavior can be achieved using parameter-tuning,[15] component-based composition,[16] or middleware-based approaches.[17] Another important aspect of self-adaptive system is related to its ability to validate and verify the adaptation action at runtime based on game theory,[18] utility theory,[19,20] or a model-driven approach.[21]

Context information refers to any information that is computationally accessible and upon which behavioral variations depend.[22] *Context observation and detection approaches* are used to detect abnormal behavior within the architecture at run time. Related work in context modeling, context detection, and engineering self-adaptive software systems are discussed in previous studies.[13,14,23,24] In *dynamic decision-making* and *context reasoning* the architecture should be able to monitor and detect normal/abnormal behavior by continuously monitoring the contextual changes found in the operational environment.

In distributed SOA, the performance of the cluster's nodes could fluctuate around the demand to accommodate scalability, orchestration, and load balancing issued by cluster's leader. This behavior requires a model that is able to detect anomalies in real time and which can generate a high rate of accuracy in detecting any anomalies and a low rate of false alarms. In addition, there will be a set of variations that can be used by the system to adapt the changes in its operational environment. This adaptation requires dynamic decision-making process that can maximizes the cumulative rewards gained from executing a specific action. Such a problem is solved by RL algorithms[3] as discussed in the following section.

[†]Software dependability refers to the degree to which a software system or component is operational and accessible when required for use.

2.2 | Reinforcement learning in continuous state and action spaces

Assuming that the SOA has a finite number of states and actions, and the SOA design confirms to the MAPE-K model.[25] The RL algorithm can be used for planning the adaptation and learning the rewards to be gained from performing each action. However, supporting the adaptation planning requires (1) sequential decision-making in a continuous domain of states and actions and (2) a mechanism to calculate the result reward of the adaptation. Unfortunately, the reward will be unknown and delayed until the adaptation action is completed and the architecture enters its new state.

In this context, the adaptation execution follows Markov decision process (MDP). MDP defined as set of states $s \in S$ and actions $a \in A$. The transition model from state s to state $\tilde{s}$ is defined as a function $T(s, a, \tilde{s})$ and the reward of this action in the new state $\tilde{s}$ is defined by $R(s, a, \tilde{s})$, which return a real value every time the system moves from one state to another. Adaptation agent has no idea what the transition probabilities are! It does not know $T(s, a, \tilde{s})$, and it does not know what the rewards are going to be either (it does not know $R(s, a, \tilde{s})$) once it moves from one state to another.

This research implements SOA as a distributed microservice architecture running in Docker swarm[‡] as distributed cluster of workers and managers nodes. Docker swarm enforces the cluster to have one single leader following the implementation of Raft consensus algorithm.[26] Consequently, we consider the problem of sequential decision-making in a continuous domain with delayed reward signals. The problem of delayed noisy rewards can be found in distributed microservice architecture. The full problem requires an algorithm to learn how to choose an action from infinitely large action space, in order to optimize a noisy delayed cumulative reward in a large state space, where the outcome of a single action can be stochastic,[§] because each node in the cluster could have different reward value as result of the selected action. The challenges of distributed microservice architecture are: (1) the context model is unknown at runtime, (2) each node might have different values of the observation space as they are naturally distributed, and (3) each node could calculates different reward for each pair of state-action.

This paper studies the problem of supporting dynamic adaptation by multiagent centralized RL, which can yield the highest rewards during a dynamic adaptation process. In RL, this problem is solved using (1) PG, (2) Q value, and (3) Q-learning[3]; however, it is unknown for us which approach is more suited to the domains of self-adaptive distributed SOA with a centralized controller.

2.2.1 | Q-value:

Bellman[2] found an algorithm to estimate the optimal state-action values called Q-values. The Q-value of a state-action pair is noted by $Q(s, a)$. The $Q(s, a)$ refers to the sum of discounted future rewards that the action expects to reach in a state s after selecting the action α. Q-value estimation is not applicable in environments with large sets of states and actions. Alternatively, for large state and action spaces two popular approaches were proposed (1) Q-learning and (2) PG.[3]

2.2.2 | Q-learning:

In Q-learning, neural networks are used to estimate the Q-value by defining approximation function and train the model in DQNs; an approach is called deep Q-learning. Deep Q-learning is a multilayered neural networks that, for a given state s output, a vector of action values using Markov's decision process,[2] which uses approximation function to estimate the Q-value $Q(s, a)$. The goal of DQN is to learn a state-action value function (Q), which is given by the deep networks, by minimizing temporal-difference errors.[27]

Several efforts were made to employ neural networks in the implementation of RL algorithms.[3] The idea is to use the DQN to identify the mathematical relationship between the input data and to identify the maximum reward function of finding the output. Such efforts can be found in the work of Lample and Chaplot,[27] where RL and a neural network were used to play Atari games or Go games as in the work of Sutton and Barto.[28] Also, several methods were proposed to solve computer vision problems,[29] such as object localization[30] or action recognition,[31] by employing the deep RL algorithms. Based on the DQN algorithm,[3] various neural networks such as double DQN[32] and dueling DQN (DDQN)[33] were proposed to improve performance and keep stability. Also, RNNs were employed to overcome the problem of partial observability as proposed by Hausknecht and Stone.[4] A combination between long-short–term memory (LSTM)[34] and RNN is used to enhance the knowledge of the agents about the observable environment, which is called POMDPs. In POMDP environment, the agent has only a partial view of the full environment's state space and action space, which leads to poor performance of the Q-learning algorithms. The use of deep recurrent Q-learning combined with LSTM enables

[‡]https://docs.docker.com/engine/swarm/
[§]Stochastic: having a random probability distribution or pattern that may be analyzed statistically but may not be predicted precisely.

the agent to build more knowledge about the transformation model and the cumulative rewards yielded from moving between states. The use of DRQN was found to be a very promising approach for planning the adaptation in microservice architecture.

RNN offers many features that suit the nature of self-adapting architecture. Self-adaptive SOA can be described as a POMDP environment at state s as the adaptation agent has no full view of the unforeseen context changes in the next future state. Also, the DQN uses forward propagation to estimate the Q-value; then, DQN checks the error between the predicted Q-value and the true actual yielded Q-value from each state-action pair, which leads to poor performance in real-world applications and slow convergence.[4]

On the other hand, the DRQN by employing RNN and LSTM it is able to reveal the patterns between the collected observations and the cumulative yielded rewards for each pair of state-action. The DRQN has the ability to perform back-propagation through time, to find the derivatives of the error with respect to the calculated weights of the observations, which enables the DRQN to improve the gradient descent, and subsequently minimizes the loss and error rate of the predicted Q-value. The LSTM enables the DRQN to possess a memory of all the yielded rewards from each pair of state-action pairs, so that the DRQN will be able to identify which action sequences return the maximum reward, and consequently, the DRQN reaches a terminal state in less time than DQN. The problematic issues of credit assignment and temporal difference are solved by the ability of the DRQN to feed the current state output as an input for the next state combined with the collected observations which improve its accuracy and significantly reduces the required training time.

2.2.3 | Policy gradient

On the other hand, the idea of a PG algorithm is to update the adaptation policy with gradient ascent on the cumulative expected Q-value. So, the algorithm learns directly the policy function $\Pi(\alpha, s)$ without worrying about the Q-value.[35] If the gradient is known, the algorithm will update the policy parameters with probability that the agent will take action α in state s. However, we use a deep neural network to find the policy $\pi(s, a, \theta)$, given the state S with parameters θ. The network takes state as an input and produces the probability distribution of actions. Such approach of PG can be found in PG[5] and DDPG.[6]

PG methods directly learn the policy by optimizing the deep policy networks with respect to the expected future reward using gradient descent. Williams[35] proposed the REINFORCE algorithm that simply uses the immediate reward to estimate the value of the policy. Silver et al[6] proposed a deterministic algorithm to improve the performance and effectiveness of the PG in high-dimensional action space. Silver et al[6] showed that pretraining the policy networks with supervised learning before employing PG can improve the performance of PG algorithms.

2.3 | Reinforcement learning in self-adaptive architecture

The employment of RL for dynamic action selection is not new to the domain of self-adaptive architecture. Several approaches were proposed including parameter tuning as can be found in previous studies.[36-39] Model-based adaptation that involve architecture-based configurations was proposed by Kim and Park.[40] Kim and Park[40] demonstrated the use of Q-learning to perform architectural changes in a simulated robot. This early work of Kim and Park uses a Q-table for all state-action pairs that can be used to calculate the Q-value and dynamically the algorithm selects an adaptation action.

However, the aforementioned approaches faces many limitations, including that they attempt to apply RL in a discreet set of states and actions limited to a specific domain, which in turn limits the generalization of those approaches as presented by Salehie and Tahvildari[37] and Tesauro.[38] More recently, RL was proposed as a method to adjust resource allocation in cloud infrastructure.[41,42] Finally, Wu et al[43] employed RL to handle context uncertainty in self-adaptive systems.

Handling the unforeseen contextual changes in dynamic software systems is a complex problem and a challenging task that exceeds the assumption of having a discrete set of states and actions as specified by Wu et al.[43] Uncertainty of context changes requires a mechanism to handle continuous state and/or action spaces, a situation that requires a mechanism to represent the context model at runtime to reflect the recent changes in the operational environments. In addition to that, context uncertainty in distributed self-adaptive systems requires multidecentralized adaptation agents that could possibly adapt to changes in distributed system. Decentralized multiagent deep RL (DMARL) is an ongoing research field as described in several studies.[44-47] Learning in decentralized multiagent environment is fundamentally more challenging than the employment of a single agent. DMARL faces serious problem like having nonstationary state, high dimensionality of the observation space, multiagent credit assignment, robustness, and scalability,[48] so employing DMARL in self-adaptive architecture is a challenging task, which could be investigated in future work.

Mnih et al[49] proposed asynchronous advantage actor-critic (A3C) algorithm to overcome the issue of multiagent credit assignment. Multiple workers are allowed to estimate the Q-value for each state-action pair, and a single global network decided which policy is more suitable for the state. It is a combination between PG and Q-learning that allows parallel DQNs to work asynchronously. A2C is a synchronous, deterministic version of A3C.[49] A3C could be investigated in the future, but the problem of A3C is that some workers will estimate the Q-value based on an older version of the parameters θ; this might add more complexity in top of the well-known issue of eventual versus strong consistency found in distributed systems.

2.4 | Summary

In Docker swarm,[¶] manager nodes are used to handle the management of the cluster's tasks and maintaining cluster's state, scheduling the microservices, and provide access to the microservices endpoints in global mode. Meaning, microservices running in worker nodes can be accessible via the managers' endpoints.[50] The challenge of Docker swarm cluster is that the manager's quorum are stored in all manager nodes, but only the elected leader makes all swarm management and orchestration decisions for the swarm. The quorum are used to store information about the cluster states, and the consistency of information is achieved through consensus via the Raft consensus algorithm.[26] Once the leader node dies unexpectedly, other managers can be elected to serve as a leader for the swarm, and then the leader restores the cluster state. This particular implementation of Raft in Docker swarm enforces this research to use multiple agents with a centralized controller (ie, swarm leader); therefore, decentralized multiagent RL is beyond the scope of this paper.

Microservice architecture often features incomplete and noisy state information, which leads to partial observability and uncertainty of the context model. DRQN is proposed in this paper to overcome the problem of partial observability.[4] In this paper, we considered combining the DRQN with GRU cells[51] instead of using LSTM.[4] The use of GRU cell offers DRQN better convergence and needs less training time in comparison to the use of LSTM cell in RNNs as demonstrated by Jozefowicz et al.[52] In this paper, we decided to conduct our research in a centralized multiagent microservice architecture, where all agents are able to observe the environment, and only the manager nodes will execute the selected action specified by DRQN algorithms. Keeping in mind that Raft consensus algorithm[26] is a central component of the implementation of a Docker swarm cluster, therefore the selected actions by the deep RL algorithms are subjected to the voting process of Raft consensus algorithm.[26] Multiple agents are observing the environment, but the execution of the adaptation actions are limited to manager nodes, supported by one single leader at a time.

Our proposed approach differs from the related work described in the above sections, as it does integrate the RNNs with a Q-leaning algorithm to estimate the reward of each action and to overcome the problems of vanishing gradients by employing GRU cells in the input layer of the observation. Also, we overcome the problem of nonstationary state found in a microservice cluster by stacking the observation instead of dealing with a single observation at a time. Then memory is used to replay the last collected set of observation to the DRQN, which gives the DRQN more insights about the rate of change, bias, and gradient decent of the observations. Finally, we overcome the problem of having an older version of the Q-value by allowing only the leader node to update the Q value at the end of each iteration.

3 | DESIGN AND METHODOLOGY

3.1 | Self-healing microservice architecture

One important aspect of a self-adapting microservice architecture is its ability to continuously monitor the operational environment and to detect and observe contextual changes. By so monitoring it will also detect and provide a reasonable policy for self-scaling, self-healing, and self-tuning the computational resources so that those resources can dynamically be adapted to any sudden changes in its operational environment.

To validate the ideas presented in this paper, a working prototype of microservice architecture is implemented in Docker swarm,[#] as shown in Figure 1. The cluster consisted of one leader and many manager and worker nodes. To meet scalability and availability, the cluster leader distributed the workload between the workers based on a Raft consensus algorithm[26]; as a result each service could be executed by assigning multiple containers across the cluster.

[¶]https://docs.docker.com/engine/swarm/
[#] https://docs.docker.com/engine/swarm/

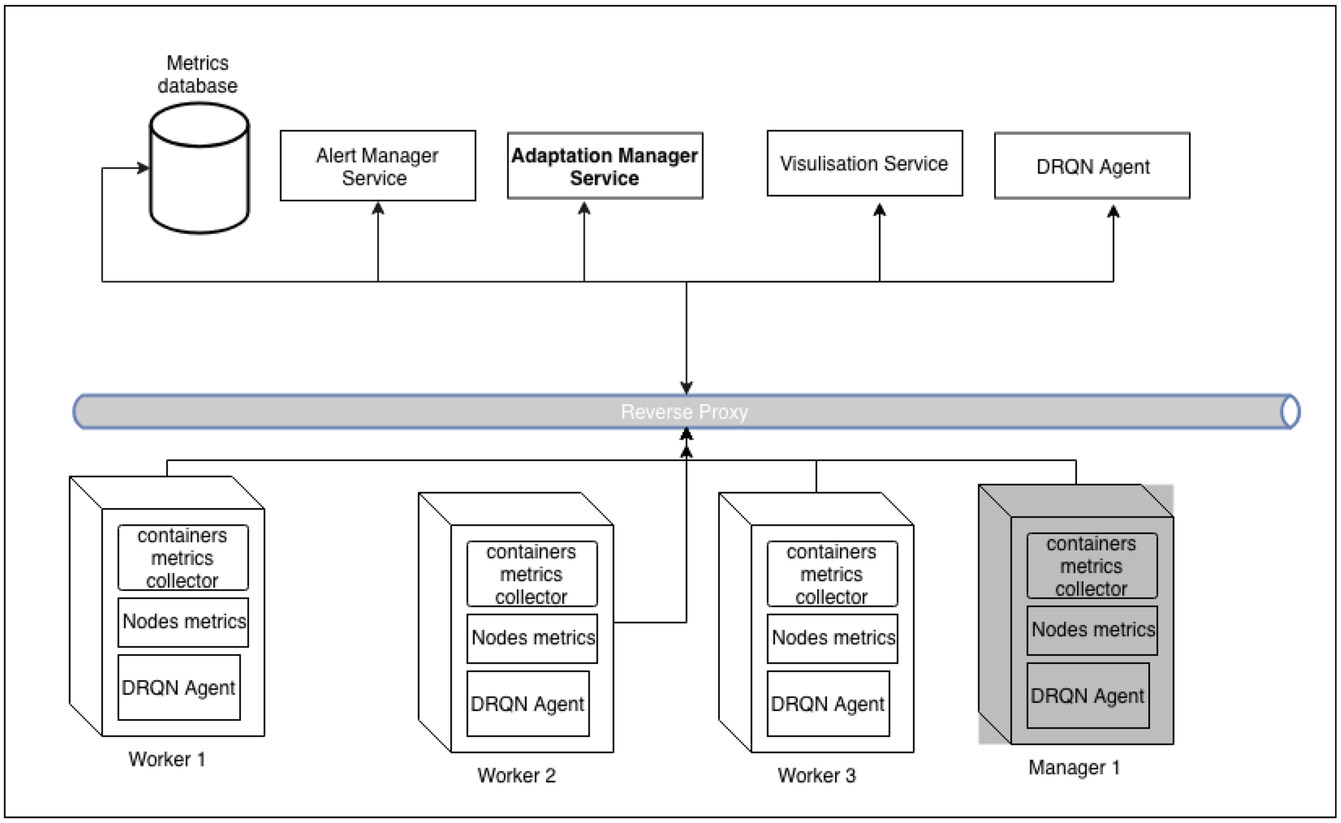


FIGURE 1 Microservice architecture implemented in Docker swarm. DQRN, deep recurrent Q network [Colour figure can be viewed at wileyonlinelibrary.com]

The microservices' architecture is shown in Figure 3. The architecture was designed according to the MAPE-K model.[25] So the architecture implements (1) a service for monitoring the environment, (2) a service for analyzing the metric values, (3) a service for planning and executing the adaptation, and (4) a service for calculating the reward gained from executing a specific adaptation action. This microservice architecture model offers the following services:

1. **Architecture monitoring:** This service provides the continuous collection of fine-grained metrics about cluster nodes, services, and containers including: (1) CPU usage, (2) memory, (3) Disk reads (bytes per second), (3) Network read per second, (5) network write per second, and disk writes (bytes per second).
2. **Adaptation planning:** This service is responsible for reading the collected observations and calculates the estimated reward from selecting adaptation actions. The DRQN algorithm predicts the architecture's performance based on the collected observation and selects a suitable adaptation action. What happens next is that (1) the alert manager notifies the adaptation manager about the contextual changes that has been detected, and (2) the adaptation manager selects the adaptation action(s) after calculating the Q value for all actions as explained in Section 3.2.
3. **Adaptation election:** The adaptation manager executes the action based on the aggregated value of the Q-value returned by the DRQN. Once the adaptation action is completed by the adaptation manager. To avoid conflicts between multiple adaptation policies, the adapter allow the adaptation actions to be fully completed and verified by the cluster leader according to the consensus performed by the leader of the cluster.
4. **Adaptation verification:** The cluster leader and all managers in the cluster will vote on the adaptation action based on the consensus algorithm.[26] The vote results are used to validate and verify the possibility of deploying the adaptation action. If the adaptation action won the voting, the adaptation action will be executed by the cluster leader, and the adaptation manager records the adaptation attempt as successful and collects the associated reward value. If the adaptation action lost the voting process, then the adaptation manager maintains the current state of the cluster and records the adaptation attempt as failed, which results in getting a negative reward value.

3.2 | Adaptation manager

Supporting microservices architecture in distributed swarm cluster with self-adaptability, make such environment follows the theory of POMDPs. POMDPs can be defined as a set of states $s \in S$ and actions $a \in A$. The transition model from state s to state $\tilde{s}$ is defined as a function $T(s, a, \tilde{s})$ and the reward of this action in the new state $\tilde{s}$ is defined by $R(s, a, \tilde{s})$, which returns a real value every time the system moves from one state to another. In microservices architecture, the set of possible adaptation actions is identified (1) based on the observation of the current state of the architecture and (2) by proposing a suitable action that can adapt to the contextual changes. The adaptation requires the identification of a set of sequential actions to adapt to changes by executing an adaptation action that enables the architecture to reach the desired objectives or QoS. As a result, the adaptation agent will be rewarded with positive value once it reaches the adaptation objectives (ie, service convergence) or negative reward value for every failed adaptation action. Having a noisy delayed cumulative reward value prevents the agents from identifying the best action to take. The problem of noisy reward is solved by estimating the Q-value using a RL approach as discussed in the following section.

3.3 | Reinforcement learning in continuous state and action spaces

The adaptation agents has a set of discrete adaptation actions and has no idea what the transition probabilities are! The adaptation agent does not know $T(s, a, \tilde{s})$, and it does not know what the rewards are going to be either (it does not know $R(s, a, \tilde{s})$) once it moves from one state to another. The agents must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities. Knowing the optimal state value is very useful to identify the best adaptation action(s). Bellman[2] found an algorithm to estimate the optimal state-action values called the Q-values. The Q-value of a state-action pair is noted by $Q(s, a)$. The $Q(s, a)$ refers to the sum of discounted future rewards that the adaptation action expects to reach in a state s after selecting the adaptation action α. The Q-value estimation is not applicable in environment with a large set of states and actions. Alternatively, there are two more popular approaches: Q-learning and PG algorithms.

In Q-learning, neural networks are used to estimate the Q-value by defining an approximation function and training the model in a deep neural network; this approach is called deep approximate learning (Deep Q-learning). Deep Q-learning is a multilayered neural network that, for a given state s, outputs a vector of action values, and it uses approximation function to estimate the Q-value $Q(s, a)$.

On the other hand, the idea of a PG algorithm is to update the adaptation policy with gradient ascent on the cumulative expected Q-value. So, the algorithm learns directly the policy function $\Pi(\alpha, s)$ without worrying about the Q-value.[35] If the gradient is known, the algorithm will update the policy parameters with the probability that the agents will take action α in state s. However, we use a deep neural network to find the policy $\pi(s, a, \theta)$, given the state s with parameters θ. θ refers to a selected well-crafted features of the observation space.[3] The network takes "state" as an input and produces the probability distribution of an action. The proposed adaptation planning will implement both approaches of Q-learning and PG; then, their effectiveness in driving the adaptation process will be evaluated in this study. To be able to understand the process of the adaptation, it is vital to understand the reward function used in each approach and how it is calculated; these are issues that will be described in the following.

3.4 | Reward function

The adaptation agents will be using one deep Q-learning/PG approaches for identifying the best adaptation action that can return the highest reward once it reaches the desired adaptation objective. For this aim, we need to define the reward function ie, Q-value $Q(s, a)$. To achieve this, we need to look back at the state s of the service architecture (see Figure 2). At each state $s0$ in Figure 2, there is a set of observations $c \in C$ measuring the matrices of operating environment such as CPU, memory, disk I/O, and network as shown in Figure 2.

The target is to provide the deep Q-learning or PG algorithms with a scalable value that can be used to assign a weight $W(s, c)$ for all context values c found in state s. The value $w(s, c)$ is calculated using the neural network integrated with deep Q-learning/PG algorithms. The deep Q-learning approach is a regression algorithm, so it uses mean squared error as a cost function and minimizes the loss during the training between the predicted/estimated Q-value and the actual Q-value. This makes the mean squared error a good criterion to estimate the performance of DQN, DDQN, and DRQN algorithms. On the other hand, the PG approach is a classification algorithm, so we used cross entropy to calculate the optimal policy. So, the probability that a given sequence of actions occurs is equal to the probability that the corresponding sequence of states and actions occurs with the given policy. This makes the cross entropy measures the probability of executing a

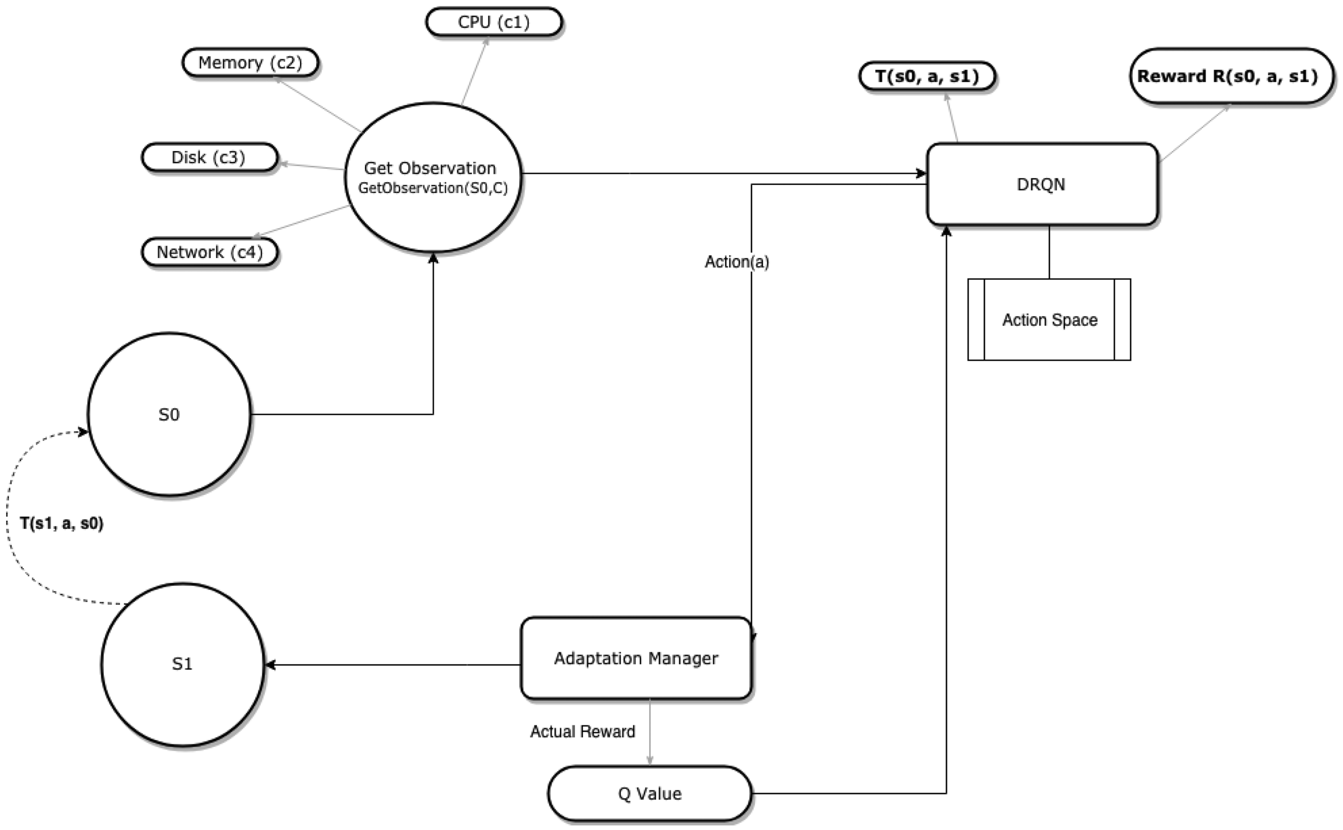


FIGURE 2 Deep Q network and transition model from S0 to S1

random selected action, and it will return the loss between the calculated probability and the actual probability of the executed action. This makes the Softmax of cross entropy loss a good measurement of PG algorithms.

The following section explains the experimental setup and the structure of neural networks used in the implementation of self-adaptive microservice architecture.

4 | EXPERIMENT SETUP

4.1 | Distributed microservice architecture implementation

To validate the ideas presented in this paper, we developed a working prototype of microservice architecture running in Docker swarm.[‖] The cluster has only one single leader at a time supported by manager and worker nodes. Only the leader/manager node(s) executes the proposed adaptation actions, then all manager nodes will vote on the proposed action. The outcome of the vote is stored by the leader node's logs. The action will be executed after a successful result according to the mechanism explained by Raft consensus algorithm.[26] The leader node and all managers will be running the adaptation agents for observation and adaptation execution. The adaptation planning will be running at the leader node, where one single algorithm of Q-learning or PG will be used to drive the adaptation, and its evaluation measurements will be collected in a separate experiment.

To comply with the MAPE-K model, the microservice architecture implements the following services:

- **Monitoring:** A time series metric database for context collection and monitoring is implemented using the Prometheus framework.[**] Metrics of all node in the cluster is collected using node exporter service.[††] Collecting fine-grained metrics about all running containers running in the swarm cluster is performed using Cadvisor service[‡‡]

[‖] https://docs.docker.com/engine/swarm/
[**] https://prometheus.io
[††] https://prometheus.io/docs/guides/node-exporter/
[‡‡] https://github.com/google/cadvisor

- **Analyzing:** The collected metric is analyzed and processed using an alert and notification managers, which is used to notify the adaptation manager about contextual changes and sudden spike in the monitored context.[§§] Also, a reverse proxy for routing traffic between all services in the cluster is implemented using caddy server.[¶¶] Time series analytic and visualization dashboard for observing the behavior of the microservices cluster by the end users.[##]
- **Planning:** MDP adaptation agents deployed in all nodes can observe the microservice architecture and execute the selected adaptation action(s). Adaptation planning service running on the leader node, which allows the DQNs to have a consistent value for all parameters.

  The adaptation planning service is implemented via DQN, DDQN, DRQN, PGNN, or DDPG. The five algorithms were implemented using the Keras-rl framework[53] and Tensorflow framework.[‖‖]
- **Execution:** Each algorithm will be executed separately, and the algorithm will collect the observation via the multi-adaptation agents running in all nodes. The algorithm processes the observation and proposes an adaptation action according to their implementation of Q-learning or PG. The selected action policy will be returned to the adaptation agent for execution. The leader node initiates the voting process over the selected action. Winning the vote will results of orchestrating the selected action to all nodes following the mechanism of Raft consensus algorithm.[26] The agents run the adaptation and return the new observation and the cumulative reward value yielded in the new state.
- **Shared knowledge:** One benefit of using RL for adaptation planning and execution is that the RL algorithm is able to learn the adaptation actions that yield the highest reward, and it leads to keep the swarm cluster in optimal state.

To understand the implementation of the five RL algorithms, we need to study the features of each algorithm as well as the structure of the DQN used to estimate the Q-value/PG. The implementation of the five algorithms is discussed in the following sections. A full code of the adaptation agents, the five algorithms implementation and services stack used in this experiment can be found in GitHub.[***]

4.2 | Deep Q-learning network (DQN)

In our implementation of DQN, it takes a state *s* observation as input and outputs a Q-value estimate per action, which will be executed by the adaptation agent. The neural network architecture shown in Figure 3 consisted of an input layer equals the size of the observation space. In this experiment, the adaptation agent collects CPU usage, memory usage, disk space, and network I/O. The observation at each state is the input for the DQN first flatten layer (see Figure 3). The observation space is multidimensional box; for this reason, we used a flatten layer that flattens the observations into one-dimensional input so they can be used by following dense layer. The second layer is a dense hidden layer of 20 units. The RELU activation function[54] is used to activate the output for the next dense layer. The last layer in the DQN in Figure 3 is a dense layer with linear activation function. The output of dense-4 layer in Figure 3 equals the estimated Q-value of the associated action value of the action space defined by the adaptation agent. Our adaptation agent implements a discrete action space of ten possible adaptation policies that can be used (1) to scale the microservice architecture horizontally or vertically, (2) to perform service composition, (3) to preserve the cluster state, and (4) to perform autorecovery.

However, the action space can be extended using requirements reflections,[55] dynamic services compositions, or model-driven variability management as in the work of Medvidovic et al[56] and Paspallis.[57] Keeping that in mind, the Docker swarm is supported particularly with the features of (1) service discovery, (2) service composition, (3) load balancing, and (4) orchestration. All features are managed under Raft consensus algorithm,[26] which makes extending the action space an easy task by means of tailoring different variations of Docker-composed files driven by business requirements or goal driven adaptation.[17]

4.3 | Dueling deep Q-learning network (DDQN)

The architecture of DDQN is very similar to DQN illustrated in Figure 3. The modifications that DDQN algorithm offers are as follows. First, it separates the representation of observation state from the representation of action space. Second, DDQN calculates an estimation of the advantage and Q-value for each state-action pair separately in each layer, after which it will combines them back into a single Q-value at the final layer as shown in Figure 4. The DDQN calculates

[§§]https://prometheus.io/docs/alerting/alertmanager/
[¶¶]https://caddyserver.com/docs/proxy
[##]https://grafana.com
[‖‖]https://www.tensorflow.org
[***]https://github.com/baselm/cmarl.git

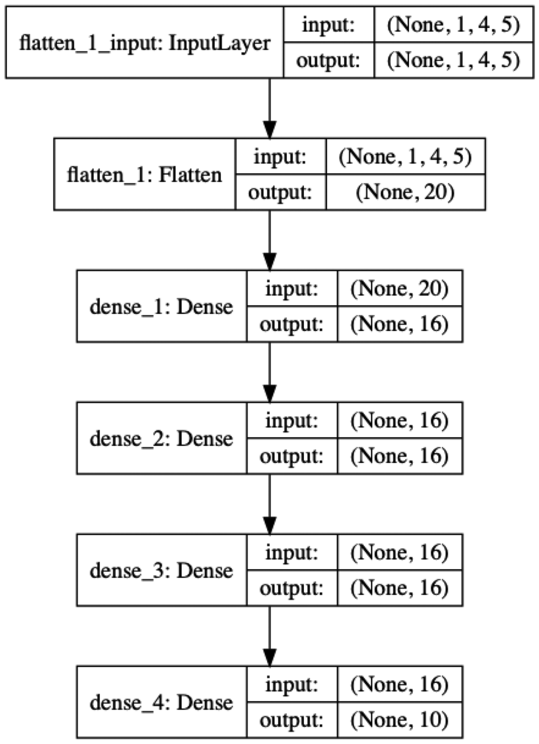


FIGURE 3 Deep Q-learning network

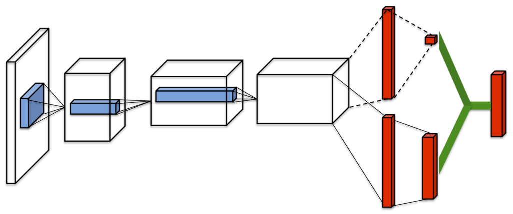


FIGURE 4 Dueling deep Q-learning network architecture [Colour figure can be viewed at wileyonlinelibrary.com]

the Q-value $Q(s, a)$ from the estimated Q-value $V(s)$ and the calculated advantage of state-action pair $A(s, \alpha)$; this result of calculating the Q-value is $Q(s, a) = V(s) + A(a)$. Wang et al[33] claims that the DDQN speeds up the convergence better than the DQN when the action space is very large.

4.4 | Deep recurrent Q-learning network (DRQN)

Considering the implementation of the microservice architecture proposed in this paper, you will find that the adaptation agent can observe the architecture, but it has only a partial view of the full environment observations as it does not know the probabilities of $T(s, a, \tilde{s})$ and the $R(s, a, \tilde{s})$ in the new state $\tilde{s}$. This problem in MDP is referred to as POMDPs,[58] which implies that the agents have no idea what is the transformation or the reward after executing a specific action. The solution of POMDPs in RL involves the use of temporal difference algorithms[3] to provide the agent with better knowledge about the environment, which leads to a better estimation of the Q-value. This can be achieved by stacking the observation instead of dealing with a single observation at a time. Then, a memory is used to replay the last collected set of observation to the DRQN, which gives the DRQN more insights about the rate of change, bias, and gradient descent of the observations. This method works well in many real-world examples as demonstrated by Mou et al[59] and Murad and Pyun.[60] Microservice architecture often features incomplete and noisy states' information, which leads to partial observability and uncertainty of the context model. DRQN was proposed to overcome the problem of partial observability.[4] In this paper, we considered combining the DRQN with GRU cells[51] instead of using LSTM.[4] The use of GRU cells offers DRQN better convergence and needs less training time in comparison to the use of LSTM cells in RNNs as demonstrated by Jozefowicz et al.[52]

The implementation of DRQN is presented in Figure 5. The DQN consisted of GRU cells that take the observation space as an input. The GRU is configured to return the last output in the output sequence as an input for the next dense layer. As shown in Figure 5, the following layers are very similar to the architecture of DQN, where RELU used as an activation

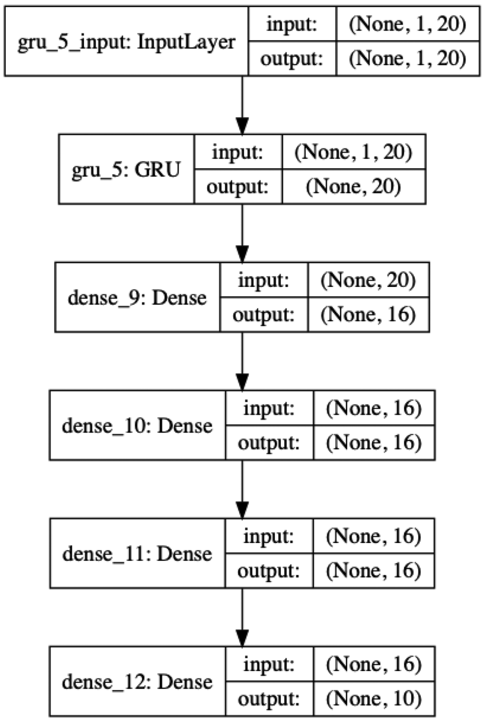


FIGURE 5 Deep recurrent Q-learning network

function until the output layer of the estimated Q-value is returned, a process which is associated with the action policy to be executed by the adaptation agent.

4.5 | Policy gradient with deep neural network (PGNN)

As was discussed above, the PG optimizes the policy parameters by following the gradient ascent towards the highest reward. So, the neural network policy will play several adaptation actions and compute the gradients at each step that would make the chosen action more likely without applying this adaptation in the architecture. If an action's score is positive, it means that the action was good and the agent would apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the score is negative, it means the action was bad and the agent would apply the opposite gradients to make this action slightly less likely in the future. The solution is simply to multiply each gradient vector by the corresponding action's score. Finally, the PG calculates the mean of resulting gradient vectors and uses it to perform a gradient descent step.

Our implementation of the PGNN algorithm involves the definition of the neural network shown in Figure 6. A simple neural network is used to predict the probability of taking an action using a cross entropy between the estimated probability and the target probability. The network shown in Figure 6 consisted of a simple input layer and one single layer that outputs the action to be taken by the agent. The problem of having this simple neural network is that the rewards are

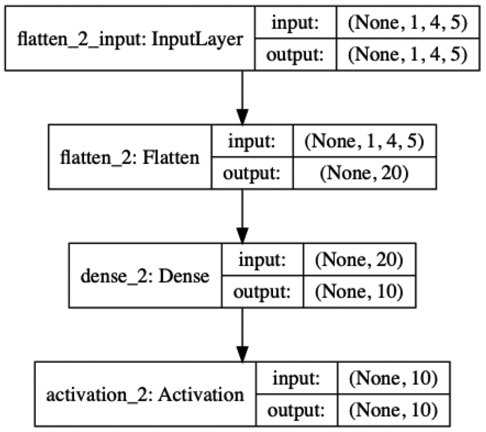


FIGURE 6 Policy gradient with deep neural network

sparse and delayed, so the network will only have the reward value to know whether an action was good or bad. In other words, the agent will fail to justify which action was responsible for positive/negative reward after executing a sequence of actions. To solve this problem, a common approach is to use a discount rate to evaluate an action based on the sum of all cumulative rewards that come after it by applying a discount rate *r* at each step; then, this reward will be normalized across the many iterations of the execution.

4.6 | Deep deterministic policy gradient (DDPG)

The problem of having an environment with continuous state space or continuous action space is a challenging task for RL. Having a continuous action space requires an algorithm that could provide better estimations of the best action to take, considering the partial observation problem found in the POMDP environment. The DDPG algorithm was proposed by Lillicrap et al[61] to overcome the issue of continuous action space. The idea of the DDPG algorithm is to have two neural networks, one acting as the critic and the other acting as the actor. The critic DQN is used to estimate the Q-value of a state-action pair. The actor is a PGNN that controls how the agent is behaving; judgment is based on the estimation it received from the critic network. Lillicrap et al[61] claimed that having a hybrid approach between Q-learning and PG would overcome the problem of handling continuous actions space and the partial observability of the environment.

The architecture of DDPG is shown in Figure 3, and the actor network is DQN similar to the architecture in Figure 3. The first layer is an input flatten the observation space followed by three dense layers output the probability of an action. This output rather being sent to the agent is feed into the critic network. The critic network in Figure 7B concatenates the output of the actor network and the observation space as a single input, then it feeds them into three consecutive dense layers, so it can identify the best probability for the state-action pair. We implement the critic network to use the Ornstein Uhlenbeck process for policy approximation as it offers better performance over epsilon-greedy approximation as stated by Daprato and Lunardi.[62]

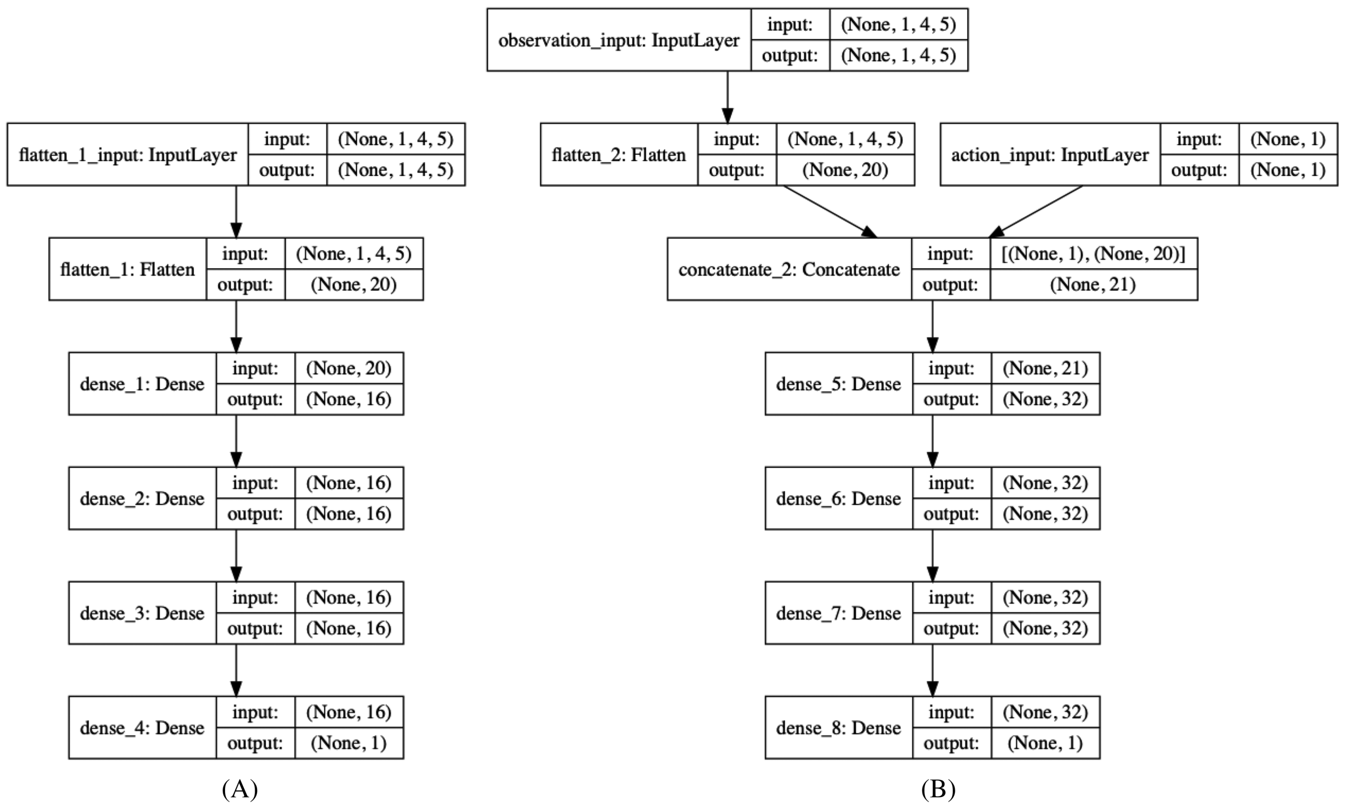


FIGURE 7 Deep deterministic policy gradient. A, Actor network; B, Critic network

5 | RESULTS

Five different experiments were executed to evaluate the effectiveness of using the five algorithms in dynamic adaptation. From each experiment, the following metrics are used to measure the algorithms' performance: mean of the Q-value, the total reward yielded per single episode, and the adaptation time in seconds, which measures the time needed for each algorithm to complete action selection and the total number of steps taken by the algorithm at each episode to reach a terminal state. This measurement is very important to observe as some algorithms will become trapped in a state continuously believing it yields the maximum reward, which extended the adaptation time and delayed the transition to an optimal sate. The other metrics include the mean absolute error (MAE), which captures the error rate between the predicted Q-value and the actual Q-value obtained after executing the actions, and the number of steps per episode needed to reach a terminal result, which means that the adaptation agent manages to achieve full convergence of all services and cluster nodes.

Figure 8 depicts the mean of the Q-value for the five algorithms (DQN, DDQN, DRQN, PGNN, and DDPG). From the figure, it is clear that DRQN and DDQN both returned the height Q-value. DRQN achieved the highest Q-value in 120 episodes, but DDQN achieved the highest Q-value in 155 episodes. The DDPG started by getting a huge negative reward, and then it converged to a positive value around 130 episodes as shown in Figure 8. The performance of PGNN was very poor as the mean Q-value was close to zero. However, the DQN performance was better than DDPG, but it required a longer time to converge, and yet it did not yield better Q-value after long training time.

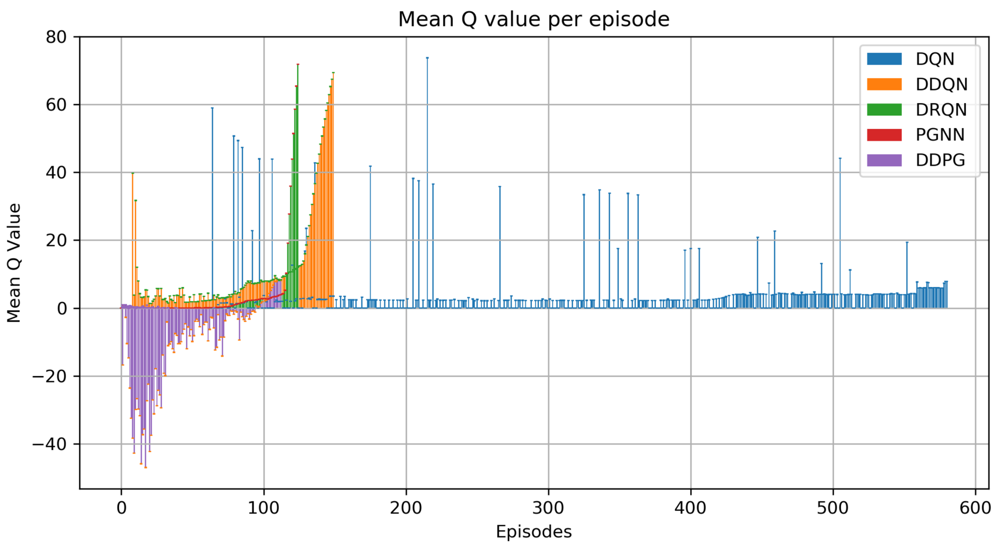


FIGURE 8 Mean Q-learning value per episode for deep Q-learning network (DQN), dueling DQN (DDQN), deep recurrent Q-learning network (DRQN), policy gradient neural network (PGNN), and deep deterministic policy gradient (DDPG) [Colour figure can be viewed at wileyonlinelibrary.com]

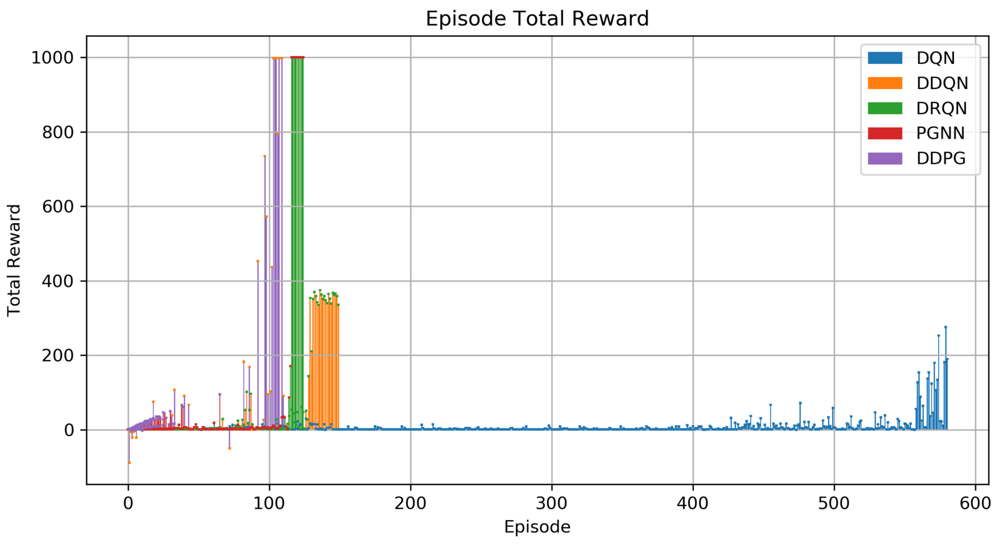


FIGURE 9 Total reward episode for deep Q-learning network (DQN), dueling DQN (DDQN), deep recurrent Q-learning network (DRQN), policy gradient neural network (PGNN), and deep deterministic policy gradient (DDPG) [Colour figure can be viewed at wileyonlinelibrary.com]

The divergence of the mean Q-values between the five algorithms can be justified by inspecting the total reward obtained by them, as shown in Figure 9. Looking at Figure 9, it was found that DRQN and DDPG managed to score the highest total rewards in 130 episodes. However, the DDQN came third in terms of the total obtained rewards, although DDQN takes longer time and more episodes of 600 to achieve the total rewards. However, the DQN outperform the PGNN in term of the total reward obtained but it takes the DQN the longest time to achieve this result (see Figure 9).

On the other hand, the MAE of the five algorithms is illustrated in Figure 10. Figure 10 confirms the result of mean Q-value and total rewards described above. This figure shows that the DRQN, DDQN, and DQN algorithms produced fewer errors rate than the other two, as DRQN, DDQN, and DQN performed better in predicting the Q-value for each pair of state-action. The DDPG produces high error rate, and it failed to return an acceptable value of total rewards. This outcome is justified by how DDPG is working in the gradient ascent until it reaches a high negative reward value so it would work on the opposite direction—gradient descent—as this can be illustrated in Figure 8. However, the DQN produces the same MAE as DRQN and DDQN, but it requires more training and episodes than the other four algorithms to achieve this performance. This can be explained by looking at the adaptation time/duration of the execution of all actions per episode in Figure 11. The DRQN, DDPG, and DDQN manage to converge faster and perform the adaptation in less time than DQN and PGNN. The PGNN requires longer time to complete the adaptation action, as it requires more time to adapt to the changes found in the observation. On the other hand, it was found that DRQN responded fastest to the changes in the observation space and managed to adapt quickly to the contextual changes. Also, the DQN and DDQN

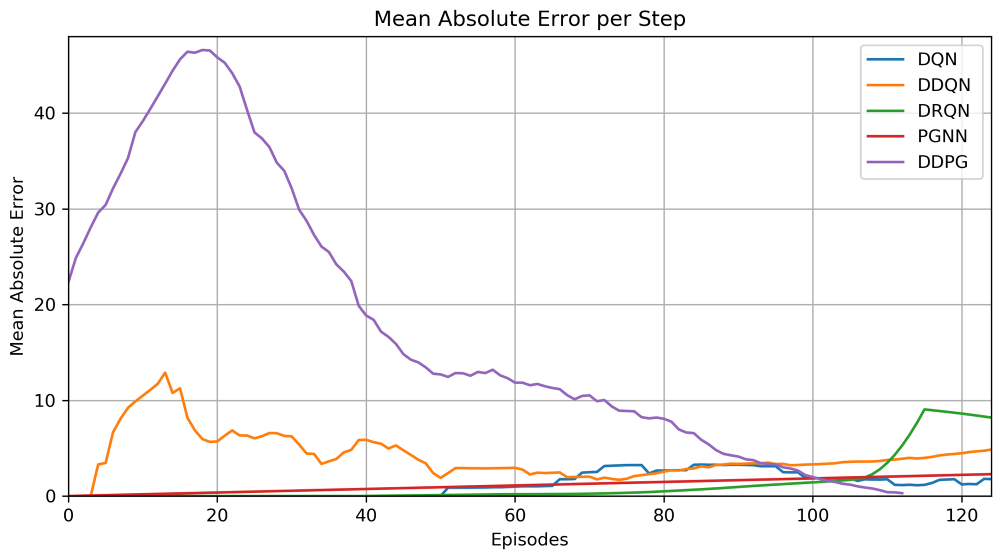


FIGURE 10 Mean absolute error for deep Q-learning network (DQN), dueling DQN (DDQN), deep recurrent Q-learning network (DRQN), policy gradient neural network (PGNN), and deep deterministic policy gradient (DDPG) [Colour figure can be viewed at wileyonlinelibrary.com]

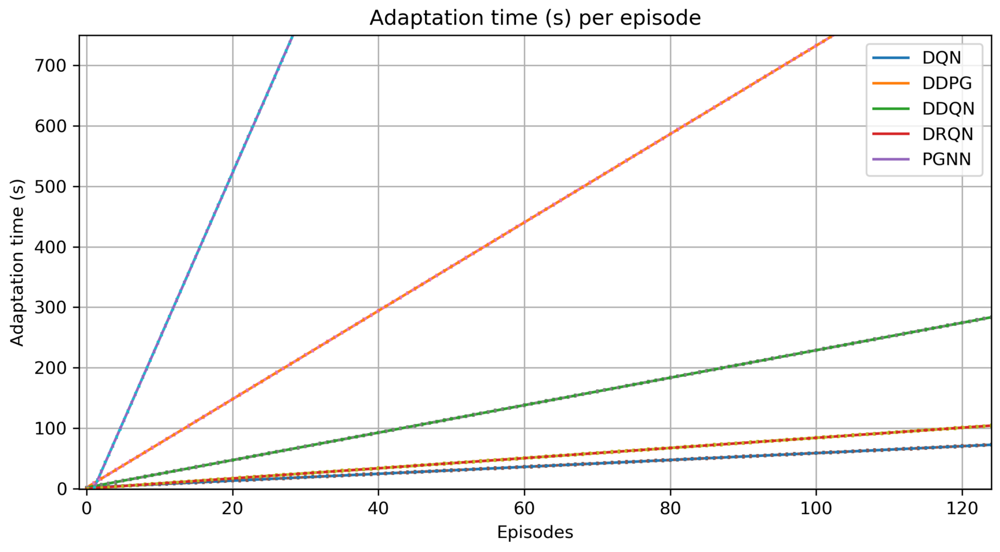


FIGURE 11 Adaptation time (s) action duration for mean absolute error for deep Q-learning network (DQN), dueling DQN (DDQN), deep recurrent Q-learning network (DRQN), policy gradient neural network (PGNN), and deep deterministic policy gradient (DDPG) [Colour figure can be viewed at wileyonlinelibrary.com]

algorithms showed high levels of adaptation to changes, but it takes both of them longer time to execute the adaptation with more number of episodes and steps.

The excellent performance of DRQN in terms of the total rewards, maximum returned Q-value, less adaptation time, and MAE is supported by the calculated loss for the five algorithms in Figure 12. The DDPG scores the highest loss,

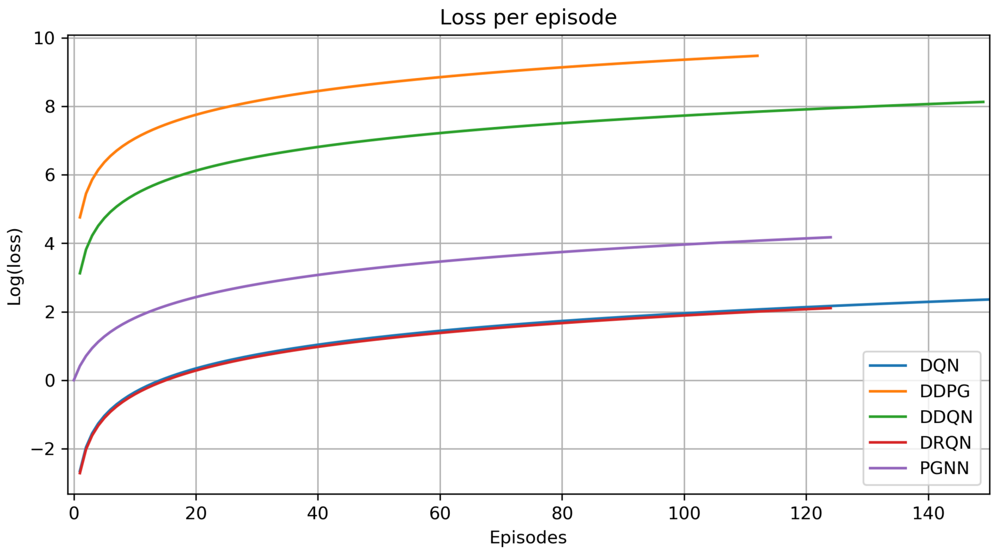
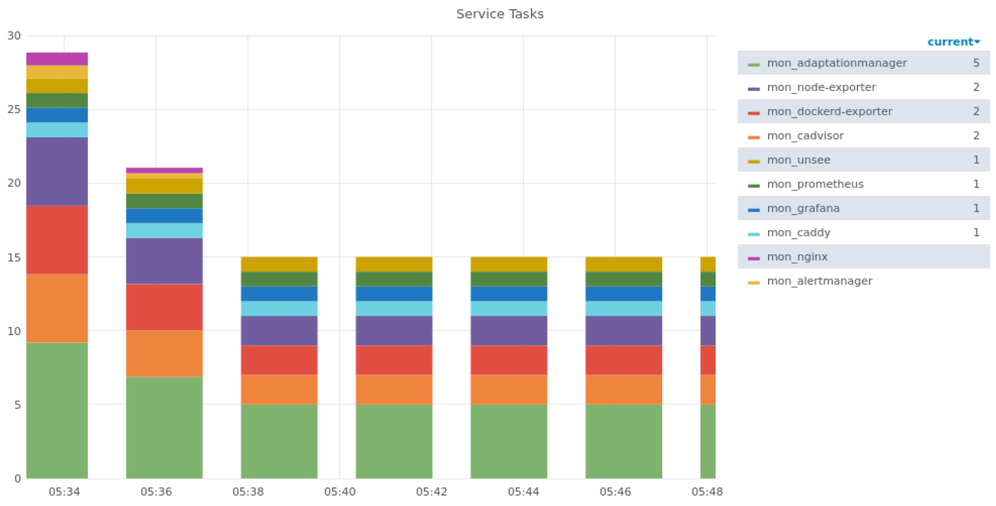
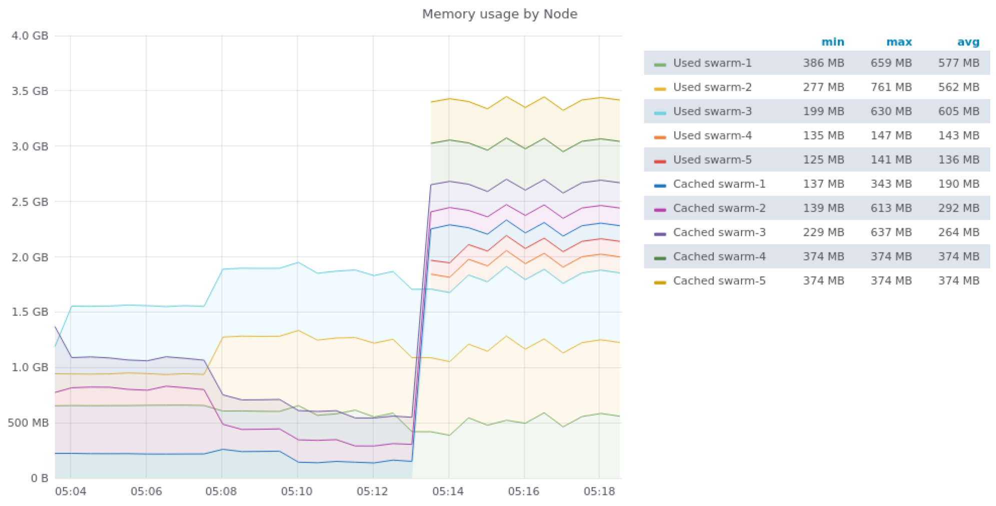


FIGURE 12 Loss per episode for mean absolute error for deep Q-learning network (DQN), dueling DQN (DDQN), deep recurrent Q-learning network (DRQN), policy gradient neural network (PGNN), and deep deterministic policy gradient (DDPG) [Colour figure can be viewed at wileyonlinelibrary.com]



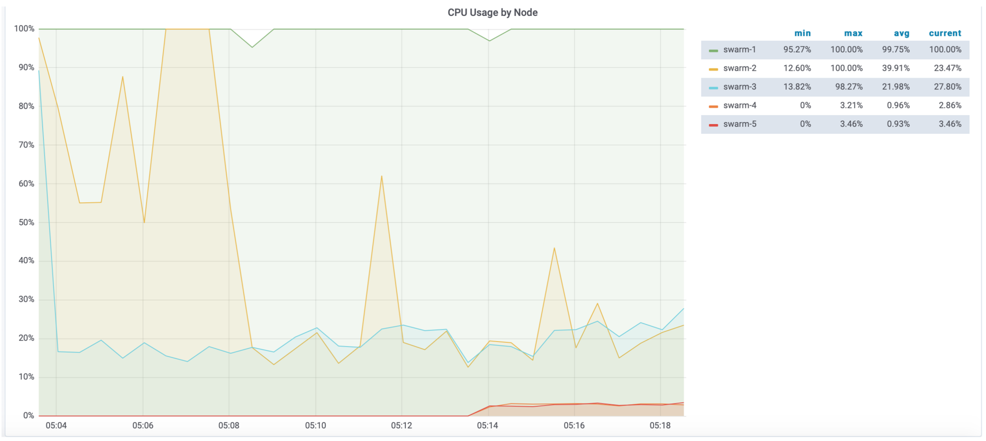
	current▾
🟩 mon_adaptationmanager	5
🟪 mon_node-exporter	2
🟥 mon_dockerd-exporter	2
🟧 mon_cadvisor	2
🟨 mon_unsee	1
🟩 mon_prometheus	1
🟦 mon_grafana	1
🟦 mon_caddy	1
🟪 mon_nginx	
🟨 mon_alertmanager	

FIGURE 13 CPU usage by services of a swarm cluster running deep recurrent Q-learning network [Colour figure can be viewed at wileyonlinelibrary.com]



	min	max	avg
🟩 Used swarm-1	386 MB	659 MB	577 MB
🟨 Used swarm-2	277 MB	761 MB	562 MB
🟦 Used swarm-3	199 MB	630 MB	605 MB
🟧 Used swarm-4	135 MB	147 MB	143 MB
🟥 Used swarm-5	125 MB	141 MB	136 MB
🟦 Cached swarm-1	137 MB	343 MB	190 MB
🟪 Cached swarm-2	139 MB	613 MB	292 MB
🟪 Cached swarm-3	229 MB	637 MB	264 MB
🟩 Cached swarm-4	374 MB	374 MB	374 MB
🟨 Cached swarm-5	374 MB	374 MB	374 MB

FIGURE 14 Memory usage by nodes of a swarm cluster running deep recurrent Q-learning network [Colour figure can be viewed at wileyonlinelibrary.com]



	min	max	avg	current
🟩 swarm-1	95.27%	100.00%	99.75%	100.00%
🟨 swarm-2	12.60%	100.00%	39.91%	23.47%
🟦 swarm-3	13.82%	98.27%	21.98%	27.80%
🟧 swarm-4	0%	3.21%	0.96%	2.86%
🟥 swarm-5	0%	3.46%	0.93%	3.46%

FIGURE 15 CPU usage by nodes of a swarm cluster running deep recurrent Q-learning network [Colour figure can be viewed at wileyonlinelibrary.com]

followed by PGNN. However, the DRQN comes fourth in term of loss. The DQN scores the best in terms of loss as the DQN algorithm takes a longer time to train, and the DQN collects more information about the environment, which results in a better loss that the other. However, the DQN has higher MAE as demonstrated in Figure 10. Also, the performance of the DRQN is supported by looking at the CPU usage by services shown in Figure 13. It is clear from Figure 13, that the adaptation manager has less impact on the cluster's nodes and maintaining steady performance. In addition to this, we find that the DRQN has less impact over the memory usage as shown in Figure 14. Finally, Figure 15 is showing the CPU's of all cluster node during the adaptation performed by DRQN 15.

6 | CONCLUSIONS AND FUTURE WORK

This paper presents a microservice architecture model that has continuous monitoring and continuous analysis of the observation space, and provides the architecture with dynamic decision-making based on the employment of deep Q-learning/PG algorithms. The MDP adaptation agents that can be used to observe the architecture's state spaces and executes an adaptation actions selected according to the outcome of the deep Q-learning/PG algorithms. An evaluation of five deep Q-learning/PG algorithms including DQNs, DDQNs, DRQN, PGNN, and DDPG. The evaluation in this research shows that DRQN is more suitable for driving the adaptation in POMDPs environment such as distributed microservice architecture. The DRQN shows (1) faster training and convergence time, (2) the highest maximum total reward in shortest number of episodes, (3) faster adaptation time, and (4) fewer errors rate and loss between the predicted and actual Q-value.

This excellent performance of DRQN, when compared to the other four algorithms, is justified by its ability: (1) to reveal the patterns between the collected observations and the cumulative yielded rewards for each pair of state-action; (2) to perform backpropagation through time, which minimizes the loss and error rate of the predicted Q-value; (3) to use GRU cells, so enabling the DRQN to possess a memory of all the yielded rewards from each pair of state-action pairs, which helps the DRQN to maximize the reward value quickly and reaches an optimal state faster than the other algorithms; (4) to overcome the problem of credit assignment and temporal-difference by stacking the collected observation instead of dealing with a single observation at a time, after which it uses its own memory to replay the last collected set of observation to the DRQN. This ability, in turn, gives the DRQN more insights about the rate of change, bias, and gradient decent of the observations. The uses of Q-learning/PG algorithms enable the architecture to dynamically elect a reasoning approach based on the highest reward gained from each action state pair. The self-adapting property is achieved by parameter tuning of the running services and dynamic composition/adjustment of the swarm cluster. We believe integrating RL in the decision-making process improves the effectiveness of the adaptation and reduces the adaptation risk, including the possibility of resources overprovisioning and thrashing. Also, our model preserves the cluster state by preventing multiple adaptations from taking place at the same time, as well as eliminating the actions that would return the lowest negative reward. Currently, this model can be extended by adding new actions to the action space implemented in the MDB agents, which will allow other researchers to run different types of experiments over, and informed by, this model. Our implementation is limited to a centralized multiagent adaptation due to the limitation of a Docker swarm and the complexity to implement decentralized multiagent RL algorithms. A Docker swarm enables the cluster to have one single leader, which prevents us from testing this model in multi agents'/leaders' environments. Also, the current implementation of an adaptation agent is limited to discrete action space, which can be extended by adding more action policies to the action space of the MDP agent. Finally, we strongly believe in the ability of the SOA to reach high levels of self-adaptability by integrating MDP agents and DRQNs in a well-designed MAPE-K architecture.

ORCID

Basel Magableh https://orcid.org/0000-0003-2337-637X

REFERENCES

1. Silver D, Huang A, Maddison CJ, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016;529(7587):484.
2. Bellman R. A Markovian decision process. *J Math Mech*. 1957;6:679-684.
3. van Hasselt H. Reinforcement learning in continuous state and action spaces. In: *Reinforcement Learning*. Berlin, Germany: Springer; 2012;207-251.
4. Hausknecht M, Stone P. Deep recurrent Q-learning for partially observable MDPs. In: Proceedings of the AAAI Fall Symposium Series; 2015; Arlington, VA.
5. Sutton RS, McAllester DA, Singh SP, Mansour Y. Policy gradient methods for reinforcement learning with function approximation. In: Proceedings of the Advances in Neural Information Processing Systems; 2000; Denver, CO.
6. Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M. Deterministic policy gradient algorithms. In: Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML'14); 2014; Beijing, China.
7. Oreizy P, Gorlick MM, Taylor RN, et al. An architecture-based approach to self-adaptive software. *Int Syst Their Appl*. 1999;14(3):54-62.
8. Cheng BHC, de Lemos R, Giese H, et al. Software engineering for self-adaptive systems: A research roadmap. In: *Software Engineering for Self-Adaptive Systems*. Berlin, Germany: Springer; 2009:1-26.
9. Bailey C, Chadwick DW, de Lemos R. Self-adaptive authorization framework for policy based RBAC/ABAC models. In: Proceedings of the IEEE 9th International Conference on Dependable, Autonomic and Secure Computing (DASC); 2011; Sydney, Australia.
10. Barbacci M. Software quality attributes: modifiability and usability. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University; 2004.
11. Babaoglu O, Jelasity M, Montresor A, et al. *Self-Star Properties in Complex Information Systems*. Berlin, Germany: Springer; 2005.
12. Horn P. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Technical Report. 2001.
13. Cheng BHC, de Lemos R, Giese H, et al. Software engineering for self-adaptive systems: a research road map (draft version). Dagstuhl Seminar Proc. 08031. 2008.
14. Strang T, Linnhoff-Popien C. A context modeling survey. In: Proceedings of the International Workshop on Advanced Context Modelling, Reasoning And Management at UbiComp 2004; 2004; Nottingham, UK.
15. Cheng S-W, Garlan D, Schmerl B. Evaluating the effectiveness of the Rainbow self-adaptive system. In: Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09); 2009; Vancouver, Canada.
16. Mikalsen M, Paspallis N, Floch J, Stav E, Papadopoulos GA, Chimaris A. Distributed context management in a mobility and adaptation enabling middleware (madam). In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06); 2006; Dijon, France.
17. Cheung-Foo-Wo D, Tigli J-Y, Lavirotte S, Riveill M. Self-adaptation of event-driven component-oriented middleware using aspects of assembly. In: Proceedings of the 5th International Workshop on Middleware for Pervasive and Ad-Hoc Computing: Held at the ACM/IFIP/USENIX 8th International Middleware Conference; 2007; Newport Beach, CA.
18. Wei W, Fan X, Song H, Fan X, Yang J. Imperfect information dynamic Stackelberg game based resource allocation using hidden Markov for cloud computing. *IEEE Trans Serv Comput*. 2016;11(1):78-89.
19. Menascé DA, Dubey V. Utility-based QoS brokering in service oriented architectures. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2007); 2007; Salt Lake City, UT.
20. Kakousis K, Paspallis N, Papadopoulos GA. Optimizing the utility function-based self-adaptive behavior of context-aware systems using user feedback. In: *On the Move to Meaningful Internet Systems: OTM 2008: OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*. Berlin, Germany: Springer; 2008:657-674.
21. Sama M, Rosenblum DS, Wang Z, Elbaum S. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2008; Atlanta, GA.
22. Hirschfeld R, Costanza P, Nierstrasz OM. Context-oriented programming. *J Object Technol*. 2008;7(3):125-151.
23. Salehie M, Tahvildari L. Self-adaptive software: landscape and research challenges. *Trans Autonom Adapt Syst*. 2009;4(2).
24. de Lemos R, Giese H, Müller HA, et al. Software engineering for self-adaptive systems: A second research roadmap. In: *Software Engineering for Self-Adaptive Systems II*. Berlin, Germany: Springer; 2013:1-32.
25. Computing A. An architectural blueprint for autonomic computing. *IBM White Paper*. 2006;31:1-6.
26. Ongaro D, Ousterhout JK. In search of an understandable consensus algorithm. In: Proceedings of the USENIX Annual Technical Conference; 2014; Philadelphia, PA.
27. Lample G, Chaplot DS. Playing FPS games with deep reinforcement learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence; 2017; San Francisco, CA.
28. Sutton RS, Barto AG. *Introduction to Reinforcement Learning*. Cambridge, MA: MIT Press; 1998.

29. Yoo S, Yun K, Choi JY, Yun K, Choi JY. Action-decision networks for visual tracking with deep reinforcement learning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017); 2017; Honolulu, HI.
30. Caicedo JC, Lazebnik S. Active object localization with deep reinforcement learning. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV); 2015; Santiago, Chile.
31. Jayaraman D, Grauman K. Look-ahead before you leap: end-to-end active recognition by forecasting the effect of motion. In: *Computer Vision - ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part V*. Cham, Switzerland: Springer; 2016:489-505.
32. Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double Q-learning. In: Proceedings of the AAAI conference on Artificial Intelligence; 2016; Phoenix, AZ.
33. Wang Z, Schaul T, Hessel M, Van Hasselt H, Lanctot M, de Freitas N. Dueling network architectures for deep reinforcement learning. In: Proceedings of the 33rd International Conference on Machine Learning; 2016; New York, NY.
34. Hochreiter S, Schmidhuber J. Long short-term memory. *Neural Computation*. 1997;9(8):1735-1780.
35. Williams RJ. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*. 1992;8(3-4):229-256.
36. Dowling J, Cahill V. Self-managed decentralised systems using K-components and collaborative reinforcement learning. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems; 2004; Newport Beach, CA.
37. Salehie M, Tahvildari L. A policy-based decision making approach for orchestrating autonomic elements. In: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice; 2005; Budapest, Hungary.
38. Tesauro G. Reinforcement learning in autonomic computing: a manifesto and case studies. *IEEE Internet Comput*. 2007;11(1):22-30.
39. Amoui M, Salehie M, Mirarab S, Tahvildari L. Adaptive action selection in autonomic software using reinforcement learning. In: Proceedings of the 4th International Conference on Autonomic and Autonomous Systems (ICAS'08); Gosier, Guadeloupe; 2008.
40. Kim D, Park S. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In: Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09); 2009; Vancouver, Canada.
41. Dutreilh X, Kirgizov S, Melekhova O, Malenfant J, Rivierre N, Truck I. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In: Proceedings of the 7th International Conference on Autonomic and Autonomous Systems (ICAS 2011); 2011; Venice, Italy.
42. Jamshidi P, Sharifloo A, Pahl C, Arabnejad H, Metzger A, Estrada G. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In: Proceedings of the 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA); 2016; Venice, Italy.
43. Wu T, Li Q, Wang L, He L, Li Y. Using reinforcement learning to handle the runtime uncertainties in self-adaptive software. In: *Software Technologies: Applications and Foundations: STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*. Cham, Switzerland: Springer; 2018:387-393.
44. Busoniu L, Babuska R, De Schutter B. A comprehensive survey of multiagent reinforcement learning. *IEEE Trans Syst Man Cybern C Appl Rev*. 2008;38(2):156-172.
45. Omidshafiei S, Pazis J, Amato C, How JP, Vian J. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In: Proceedings of the 34th International Conference on Machine Learning; 2017; Sydney, Australia.
46. Lowe R, Wu Y, Tamar A, Harb J, Abbeel OP, Mordatch I. Multi-agent actor-critic for mixed cooperative-competitive environments. In: Proceedings of the Advances in Neural Information Processing Systems; 2017; Long Beach, CA.
47. Wai H-T, Yang Z, Wang PZ, Hong M. Multi-agent reinforcement learning via double averaging primal-dual optimization. In: Proceedings of the Advances in Neural Information Processing Systems; 2018; Montreal, Canada.
48. Gupta JK, Egorov M, Kochenderfer M. Cooperative multi-agent control using deep reinforcement learning. In: *Autonomous Agents and Multiagent Systems AAMAS 2017 Workshops, Best Papers, Sao Paulo, Brazil, May 8-12, 2017, Revised Selected Papers*. Cham, Switzerland: Springer; 2017:66-83.
49. Mnih V, Badia AP, Mirza M, et al. Asynchronous methods for deep reinforcement learning. In: Proceedings of the International Conference on Machine Learning; 2016; New York, NY.
50. Kiss T, Kacsuk P, Kovacs J, et al. MiCADO—microservice-based cloud application-level dynamic orchestrator. *Future Gener Comput Syst*. 2017;94:937-946.
51. Chung J, Gulcehre C, Cho K, Bengio Y. Gated feedback recurrent neural networks. In: Proceedings of the International Conference on Machine Learning; 2015; Lille, France.
52. Jozefowicz R, Zaremba W, Sutskever I. An empirical exploration of recurrent network architectures. In: Proceedings of the International Conference on Machine Learning; 2015; Lille, France.
53. Plappert M. keras-rl. GitHub. 2016. https://github.com/keras-rl/keras-rl
54. Nair V, Hinton GE. Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10); 2010; Haifa, Israel.
55. Bencomo N, Whittle J, Sawyer P, Finkelstein A, Letier E. Requirements reflection: requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering; 2010; Cape Town, South Africa.
56. Medvidovic N, Oreizy P, Robbins JE, Taylor RN. Using object-oriented typing to support architectural design in the C2 style. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering; 1996; San Francisco, CA.

57. Paspallis N. *Middleware-Based Development of Context-Aware Applications with Reusable Components*. Nicosia, Cyprus: University of Cyprus; 2009.
58. Monahan GE. State of the art—a survey of partially observable Markov decision processes: theory, models, and algorithms. *Management Science*. 1982;28(1):1-16.
59. Mou L, Ghamisi P, Zhu XX. Deep recurrent neural networks for hyperspectral image classification. *IEEE Trans Geosci Remote Sens*. 2017;55(7):3639-3655.
60. Murad A, Pyun J-Y. Deep recurrent neural networks for human activity recognition. *Sensors*. 2017;17(11):2556.
61. Lillicrap TP, Hunt JJ, Pritzel A, et al. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*. 2015.
62. Daprato G, Lunardi A. On the Ornstein-Uhlenbeck operator in spaces of continuous functions. *J Funct Anal*. 1995;131(1):94-114.

How to cite this article: Magableh B, Almiani M. A deep recurrent Q network towards self-adapting distributed microservice architecture. *Softw: Pract Exper*. 2020;50:116–135. https://doi.org/10.1002/spe.2778