2019-01-15

A Deep Recurrent Q Network Towards Self-adapting Distributed Microservices Architecture (in press)

Basel Magableh

Technological University Dublin, 453543@tudublin.ie

Recommended Citation

Magableh, B. (2019). A Deep Recurrent Q Network towards Self-adapting Distributed Microservices architecture. CoRR, cs.SE, arXiv:1901.04011. [preprint in press article accepted Jan. 2019.] doi.org/10.21427/c4v9-an45

A Deep Recurrent Q Network towards Self-adapting Distributed Microservices architecture

Basel Magableh
School of Computer Science,
Dublin Institute of Technology,
Technological University
Dublin, Ireland
basel.magableh@dit.ie

Abstract

One desired aspect of microservices architecture is the ability to self-adapt its own architecture and behaviour in response to changes in the operational environment. To achieve the desired high levels of self-adaptability, this research implements the distributed microservices architectures model, as informed by the MAPE-K model. The proposed architecture employs a multi adaptation agents supported by a centralised controller, that can observe the environment and execute a suitable adaptation action. The adaptation planning is managed by a deep recurrent Q-network (DRQN). It is argued that such integration between DRQN and 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 resources over-provisioning and thrashing. The performance of DRQN is evaluated against deep Q-learning and policy gradient algorithms including: i) deep q-network (DQN), ii) dulling deep Q-network (DDQN), iii) a policy gradient neural network (PGNN), and iv) deep deterministic policy gradient (DDPG). The DRQN implementation in this paper manages to outperform the above mentioned algorithms in terms of total reward, less adaptation time, lower error rates, plus faster convergence and training times. 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.

Index Terms

Service oriented architecture, self-adaptive architectures, reinforcement learning, Q-learning algorithms, deep Q-Learning networks, recurrent Q-learning networks, policy approximation, multi agents environment.

I. Introduction

Self-adaptability refers to the ability of service oriented architecture (SOA) to modify its own structure and behaviour in response to changes in the operating environment [1]. High levels of self-adaptability present the challenges of self-organising, self-tuning, and self-healing the architecture against an interruption. Moreover, because of the services' pervasiveness, and in order to make any adaptation strategy effective and successful, adaptation actions must be considered in conjunction with services availability, dependability [1], 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 [2]–[4]. 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 optimise the adaptation actions in order to guarantee that the architecture will reach the adaptation objectives [5]. On the other hand, reinforcement learning (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 maximising the cumulative rewards yielded from executing a specific action [6]

Assuming that the SOA has a finite number of states and actions, and the SOA design confirms to the the MAPE-K model (monitor, analyse, plan, execute over a shared knowledge) [7]. The reinforcement learning (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 i) a sequential decision making in a continuous domain of states and actions, and ii) 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.

This research implements SOA as a distributed microservices architecture running in Docker swarm [2] 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 [8]. 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 microservices architecture. The full problem requires an algorithm to learn how to choose an action from infinitely large action space, in order to optimise a noisy delayed cumulative reward in a large state space, where the outcome of a single action can be stochastic [3], because each

[1]Software dependability refers to the degree to which a software system or component is operational and accessible when required for use.
[2]https://docs.docker.com/engine/swarm/
[3]Stochastic: having a random probability distribution or pattern that may be analysed statistically but may not be predicted precisely

node in the cluster could have different reward value as result of the selected action. The challenges of distributed microservices architecture are: a) the context model is unknown at runtime, b) each node might have different values of the observation space as they are naturally distributed, and c) each node could calculates different reward for each pair of state-action.

This paper studies the problem of supporting dynamic adaptation by multi agents centralised reinforcement learning, that can yield the highest rewards during dynamic adaptation process. In reinforcement learning this problem is solved using: i) policy gradient, ii) Q value, and iii) Q learning [5]; however, it is unknown for us which approach is more suited to the domains of self adaptive distributed service oriented architecture with a centralised controller.

The objectives of this research are: a) to provide a test bed of self-adapting distributed microservices architecture, that can be used by other researchers to experiment with various types of adaptation techniques. b) to evaluate which reinforcement learning algorithm is more suitable to support dynamic adaptation in microservices cluster. c) to propose a service stack that can be used to implement a distributed microservices architecture that confirms to the MAPE-K model. d) to propose adaptation agents that i) implement a Markov decision process (MDP) environment [9], ii) are able to collect observations about the environment and iii) can execute adaptation actions. e) to evaluate the effectiveness of reinforcement learning algorithms in dynamic selection of adaptation actions in distributed microservices architecture.

In this research, five reinforcement learning algorithms are implemented to drive the adaptation process in self-adaptive architecture. This paper will evaluate the following Q-learning algorithms: 1) deep Q networks (DQN) [5], 2) duelling deep Q networks (DDQN) [10], 3) deep recurrent Q networks (DRQN) [11]), as well as policy gradient algorithms: 4) policy gradient neural network (PGNN) [12], 5) deep deterministic policy Gradient (DDPG) [13]). Also, this paper will explain in detail the model that was used for building the microservices cluster, and the implementation of the experiment.

This paper is structured as follows:Section II provides an overview of self-adaptation techniques and surveys the approaches used for context sensing, adaptation planning and reinforcement learning. Section III presents a model of a distributed microservices architecture that can continuously observe and adapt the architecture. The proposed method of adaptation planning and execution is discussed in Section III-A. Section III-B discusses different approaches for implementing reinforcement Learning algorithms. The calculation of the reward function is explained in section III-C. IV addresses the implementation of this model and give more details about the architecture of the implementation of the five reinforcement learning algorithms. Section V evaluates the effectiveness of the five reinforcement learning algorithms (DQN, DDQN, DRQN, PGNN, and DDPG) in adaptation planning and execution. Section VI summarises this research, highlighting its contribution and setting out potential future work.

II. Related work

A. *Self-adaptive service oriented architecture*

Self-adaptive architecture is characterised by a number of properties best referred to as autonomic [14]. These properties are grouped under the ‘self-* properties’ heading; they include self-organisation, self-healing, self-optimisation and self-protection [15]. 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 [15]. Self-adapting aspects of service oriented architecture 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 [16]. Typically, a self-adapting system follows MAPE-K model (monitor-analyse-plan-execute over a shared knowledge). an efficient self-adaptive system should implements MAPE-K model which will include: a) gathering of data related to the surrounding context (context Sensing); b) context observation and detection; c) dynamic decision making; d) adaptation execution to achieve the adaptation objectives defined as QoS; e) 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 sensing; a model that involves context collection, observation and detection of contextual changes in the operational environment [17]. Also, the ability of the system to dynamically adjust its behaviour can be achieved using parameter-tuning [18], component-based composition [19], or middleware-based approaches [20]. 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 [21], utility theory as in [22], [23], or a model driven approach as in [24].

Context information refers to any information that is computationally accessible and upon which behavioural variations depend [25]. *Context observation and detection approaches* are used to detect abnormal behaviour within the architecture at run-time. Related work in context modelling, context detection and engineering self-adaptive software systems are discussed in [16], [17], [26], [27]. In *dynamic decision making* and *context reasoning* the architecture should be able to monitor and detect normal/abnormal behaviour by continuously monitoring the contextual changes found in the operational environment.

In distributed services oriented architecture, 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 behaviour 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 maximises the cumulative rewards gained from executing a specific action. Such problem is solved by reinforcement learning algorithms [5] as discussed in the following section.

B. Reinforcement learning in continuous state and action spaces

1) Q-value:: Bellman [9] 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 i) Q-learning and ii) policy gradient [5].

2) Q-learning:: in Q-learning, neural network are used to estimate the Q-value by defining approximation function and train the model in deep Q-networks; an approach is called deep Q-learning. Deep Q-learning is a multi-layered neural networks that for a given state s outputs a vector of action values using Markov's decision process [9] 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 minimising temporal-difference errors [28].

Several efforts were made to employ neural networks in the implementation of RL algorithms [5]. 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 [28] where RL and NN were used to play Atari games or Go games as in [29]. Also, several methods were proposed to solve computer vision problems [30], such as object localisation [31] or action recognition [32], by employing the deep RL algorithms. Based on the DQN algorithm [5], various neural networks such as double DQN [33] and duelling DQN [10] were proposed to improve performance and keep stability. Also, recurrent neural networks (RNN) were employed to overcome the problem of partial observability as proposed in [11]. A combination between Long Short Term Memory (LSTM) [34] and RNN to enhance the knowledge of the agents about the observable environment, or what so called Partially Observable Markov Decision Processes (POMDPs). In POMDPs 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 the agent to build more knowledge about the transformation model and the cumulative rewards yielded from moving between states. The use of deep recurrent Q-learning network was found to be a very promising approach for planning the adaptation in microservices architecture.

RNN offers many features that suit the nature of self-adapting architecture. Self-adaptive service oriented architecture can be described as POMDPs environment at state s 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 [11].

On the other hand, the DRQN by employing RNN and LSTM it is able to i) reveal the patterns between the collected observations and the cumulative yielded rewards for each pair of state-action. ii) 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. iii) 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 DRQN reaches a terminal state in less time than DQN. iv) 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.

3) Policy gradient: On the other hand, the idea of a policy gradient 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 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 policy gradient can be found in policy gradient (PG) [12], Deep Deterministic Policy Gradient (DDPG) [13]).

Policy gradient methods directly learn the policy by optimizing the deep policy networks with respect to the expected future reward using gradient descent. Williams et al. [35] proposed REINFORCE algorithm which simply uses the immediate reward to estimate the value of the policy. Silver et al. [13] proposed a deterministic algorithm to improve the performance and effectiveness of the policy gradient in high-dimensional action space. Silver et al. [13] showed that pre-training the policy networks with supervised learning before employing policy gradient can improve the performance of PG algorithms.

C. Reinforcement Learning in Self-adaptive Architecture

The employment of reinforcement learning 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 [36]–[39]. Model-based adaptation that involve architecture-based configurations was proposed in [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 above mentioned approaches faces many limitations including that they attempt to apply reinforcement learning in discreet state and action space limited to a specific domain, which in turn limits the generalisation of those approaches as presented in [37], [38]. More recently, reinforcement learning was proposed as a method to adjust resource allocation in cloud infrastructure in [41], [42]. Finally, Tong et al. [43] employs reinforcement learning to handle context uncertainty in self-adaptive systems.

Handling the unforeseen contextual changes in dynamic software systems is a complex problem and a changeling task that exceeds the assumption of having a discrete state and action spaces as specified by [43]. Uncertainty of context changes requires a mechanism to handle continuous state and/or action spaces; a situation which 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 a multi decentralised adaptation agents that could possibly adapt to changes in distributed system. Decentralised multi-agent deep reinforcement learning (DMARL) is an ongoing research filed as described in [44]–[47]. Learning in decentralised multi-agent environment is fundamentally more challenging than the employment of single agent. DMARL faces serious problem like having non-stationary state, high dimensionality of the observation space, multi-agent credit assignment, robustness, and scalability [48], so employing DMARL in self-adaptive architecture is a challenging task, that could be investigated in future work.

Mnih et al. [49] proposes asynchronous advantage actor-critic (A3C) algorithm to overcome the issue of multi-agent credit assignment. Multiple workers are allowed to estimate the Q-value for each state-action pair and single global network decided which policy is more suitable for the state. It is a combination between policy gradient and Q-learning that allows parallel deep Q-networks to work asynchronously. A2C is a synchronous, deterministic version of A3C [49]. This research implements DDPG a similar algorithm to A2C. A3C could be investigated in future, but the problem of A3C 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 vs strong consistency found in distributed systems. So we decided to conduct our research in a centralised multi-agent microservices architecture, where all agents are able to observe the environment, and only the manager nodes will execute the selected action by RL algorithms. Keeping in mind, that RAFT consensus algorithm [8] is a central component of the implementation of Docker swarm cluster. So, the selected actions by the deep RL algorithms are a) subject to the voting process of RAFT consensus algorithm [8]. b) 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.

In Docker swarm [4] 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 [8]. Once the leader node dies unexpectedly, other managers can be elected to serve as a leader for the swarm, then the leader restores the cluster state. This particular implementation of Raft in Docker swarm enforces this research to use multi-agent with centralised controller (i.e. swarm leader), so that decentralised multi agent reinforcement learning is beyond the scope of this paper.

III. Adaptation Strategy

This section focuses on describing a model that can continuously observe and monitor microservices architecture and be able to provide an adaptation action that can maintain the architecture's availability. At the same time, the architecture should be able to respond to unforeseen changes by selecting the best adaptation actions, that optimise the computational resources and achieve the adaptation goals.

The proposed model consisted of: 1) Service oriented architecture implemented as distributed microservices architecture. 2) Adaptation (MDP) multi-agents that observes the execution environment and executes the adaptation action. 3) an adaptation planning component implemented by using one of the Q-learning or policy gradient algorithms. The implementation details of the microservices used in this architecture are explained in Section IV. To understand the proposed model of self-adaptive microservices architecture, it is necessary to understand the functionality of the adaptation agents as described in the following section.

A. Adaptation MDP Agents

The adaptation agents follows Markov decision process (MDP). MDP 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

[4]https://docs.docker.com/engine/swarm/

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 service oriented architecture the set of possible adaptation actions is identified a) based on the observation of the current state of the architecture and b) 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 quality of services. As a result the adaptation agent will be rewarded with positive value once it reach the adaptation objectives (i.e. 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 reinforcement learning approach as discussed in the following section.

B. Reinforcement Learning in Continuous State and Action Spaces

However, the adaptation agents with discrete adaptation actions 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. 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 [9] 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 large set of states and actions. Alternatively, there are two more popular approaches: Q-learning and policy gradient algorithms.

In Q-learning, neural network are used to estimate the Q-value by defining approximation function and train the model in deep neural network, this approach is called Deep Approximate learning (Deep Q-learning). Deep Q-learning is a multi-layered 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 policy gradient 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 deep neural network to 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 [5]. The network takes 'state' as an input and produces the probability distribution of an action. The proposed adaptation planing will implement both approaches of Q-learning and policy gradient, 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; issues that will be described in the following section.

C. Reward Function

The adaptation agents will be using one deep Q-learning/policy gradient 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 i.e. Q-value $Q(s, a)$. To achieve this, we need to look back at the state s of the service architecture (see Figure 1. At each state $s0$ in Figure 1 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 1.

The target is to provide the deep Q-learning or policy gradient 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 neural network integrated with deep q-learning/policy gradient algorithms. The deep Q-learning approach is a regression algorithm so we have used mean squared error as cost function and we minimise that 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 policy gradient approach is a classification algorithm so we use 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 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 policy gradient algorithms.

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

IV. Experiment Setup

A. Microservices Architecture implementation

To validate the ideas presented in this paper, we developed a working prototype of service oriented architecture as microservices running in Docker swarm [5]. The cluster has only one single leader at a time supported by manager and worker nodes.

[5]https://docs.docker.com/engine/swarm/

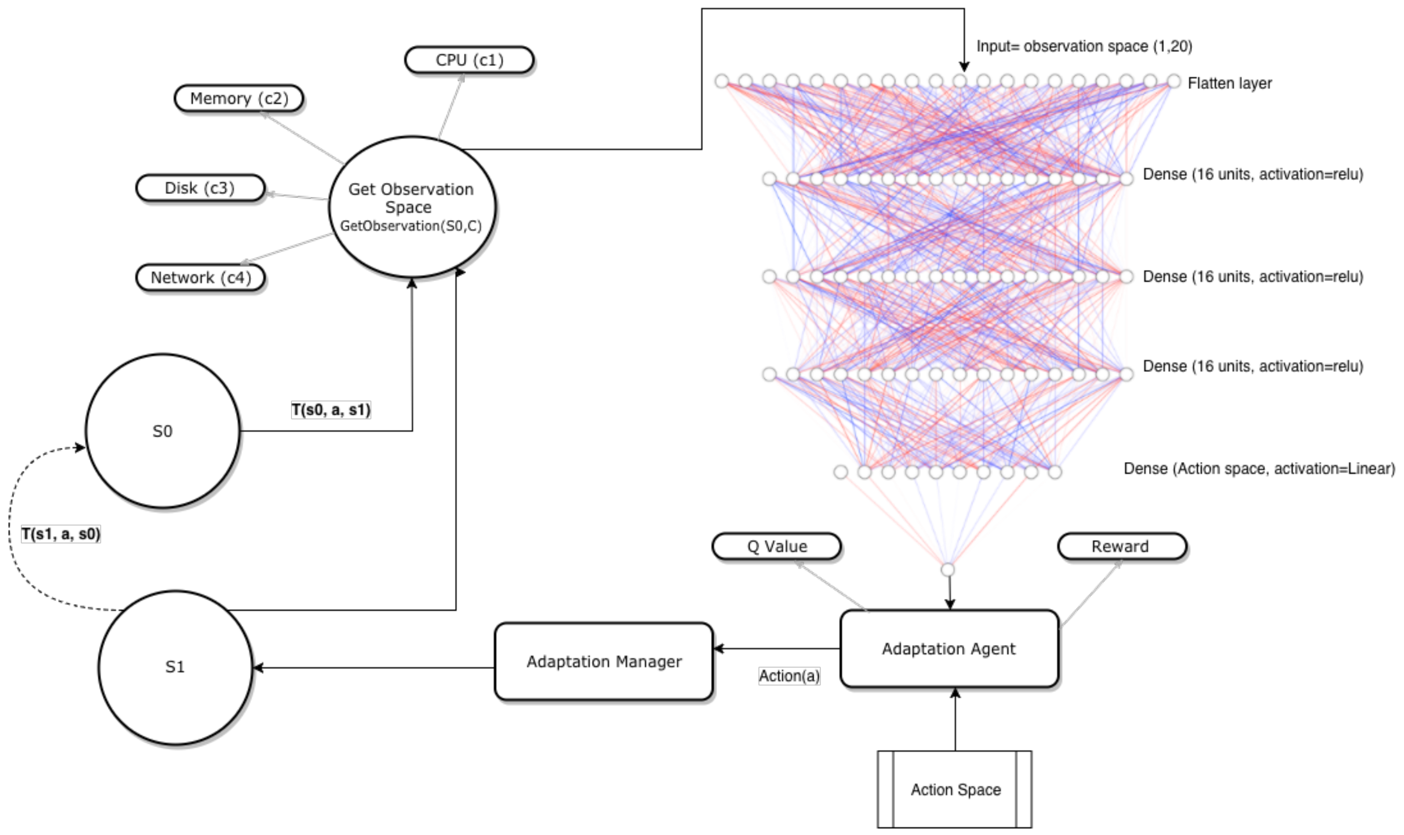

Input= observation space (1,20)
Flatten layer
Dense (16 units, activation=relu)
Dense (16 units, activation=relu)
Dense (16 units, activation=relu)
Dense (Action space, activation=Linear)
CPU (c1)
Memory (c2)
Disk (c3)
Network (c4)
Get Observation Space GetObservation(S0,C)
S0
T(s0, a, s1)
T(s1, a, s0)
S1
Q Value
Reward
Adaptation Manager
Action(a)
Adaptation Agent
Action Space

Fig. 1: Deep Q Network and transition model from S0 to S1

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 [8]. 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 policy gradient will be used to drive the adaptation and its evaluation measurements will be collected in separate experiment.

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

- A time series metrics database for context collection implemented using Prometheus framework [6]
- Nodes metrics used to collect metrics from all nodes in the cluster,
- Docker containers metrics collector for collecting fine-grained metrics about all running containers in all nodes [7]
- Alert and notification managers used to notify the adaptation manager about contextual changes [8].
- Reverse proxy for routing traffic between all services in the cluster [9].
- Time series analytics and visualisation dashboard for observing the behaviour of the microservices cluster by the end users [10].
- MDP adaptation agents deployed in all nodes, that can observes the microservices architecture and executes the selected adaptation action(s).
- Adaptation planning service running on the leader node, which allows the deep Q-networks to have a consistent value for all parameters.

The Adaptation planning service is implemented via (deep Q-learning network (DQN), Duelling Deep Q Learning Network (DDQN), Deep Recurrent Q Learning Network (DRQN), Policy Gradient Neural Network (PGNN), or Deep Deterministic Policy Gradient (DDPG)). The five algorithms were implemented using the Keras-rl framework [51] and Tensorflow framework [11].

[6]https://prometheus.io
[7]https://github.com/google/cadvisor
[8]https://prometheus.io/docs/alerting/alertmanager/
[9]https://caddyserver.com/docs/proxy
[10]https://grafana.com
[11]https://www.tensorflow.org

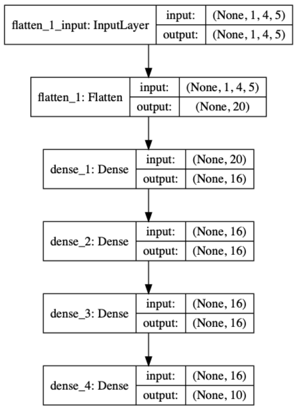


Fig. 2: Deep Q-learning Network

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 policy gradient. The selected action policy will be returned to the adaptation agent for execution. The leader node initiates the voting process over the selected action. Wining the vote will results of orchestrating the selected action to all nodes following the mechanism of Raft consensus algorithm [8]. The agents run the adaptation and return the new observation and the cumulative reward value yielded in the new state.

To understand the implementation of the five reinforcement learning algorithm, we need to study the features of each algorithm as well as the structure of the deep Q-network used to estimate the Q-value/policy gradient. The implementation of the five algorithms are 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 [12]

B. 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 2 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, network I/O. The observation at each state is the input for the DQN first flatten layer (see Figure 2). 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 dense hidden layer of 20 units. The RELU activation function [52] is used to activate the output for the next dense layer. The last layer in the DQN in Figure 2 is a dense layer with linear activation function. The output of dense-4 layer in Figure 2 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 10 possible adaptation policy that can be used to: a) scale the microservices architecture horizontally or vertically, b) perform service composition, c) preserves the cluster state, and d) perform auto recovery.

However, the action space can be extended using requirements reflections [53], dynamic services compositions, or model-driven variability management as in [54], [55]. Keeping in mind, that Docker swarm are supported particularly with the features of: i) services discovery, ii) services composition, iii) load balancing, and iv) orchestration. All features are managed under Raft consensus algorithm [8], which makes extending the actions space an easy task by means of tailoring different variations of docker-compose files driven by business requirements or goal driven adaptation [20].

C. Duelling Deep Q Learning Network (DDQN)

The architecture of duelling deep Q-learning network (DDQN) is very similar to DQN illustrated in Figure 2. The modifications that DDQN algorithm offers are: a) it separates the representation of observation state from the representation of action space. b) 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 3. The DDQN calculates 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 results of calculating the Q-value as $Q(s, a) = V(s) + A(a)$. Wang et al. [10] claims that DDQN speeds up the convergence better than DQN when the action space is very large.

[12]https://github.com/baselm/cmarl.git

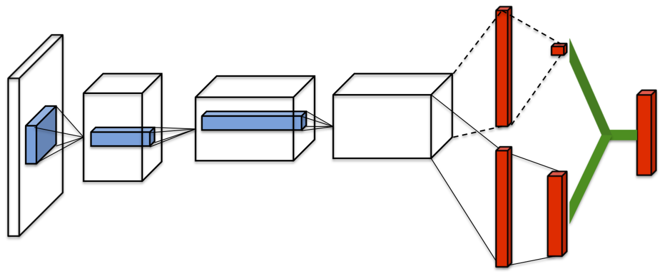


Fig. 3: Duelling Deep Q Learning Network Architecture (DDQN)

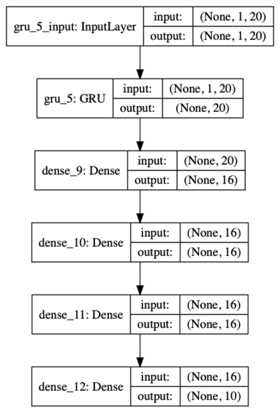


Fig. 4: Deep Recurrent Q Learning Network DRQN

D. Deep Recurrent Q Learning Network (DRQN)

Considering the implementation of the microservices 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 partially observable Markov decision processes (POMDPs) [56], 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 reinforcement learning involves the use of temporal difference algorithms [5] 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 DQN, which gives the DQN more insights about the rate of change, bias, and gradient decent of the observations. This method works well in many real-world examples as demonstrated in [57], [58]. Microservices architecture often features incomplete and noisy states' information, which leads to partial observability and uncertainty of the context model. DQRN was proposed to overcome the problem of partial observability [11]. In this paper, we considered combining the deep recurrent Q-network with Gated Recurrent Unit (GRU) cells [59] instead of using LSTM [11]. The use of GRU cell offers RNNs better convergence and needs less training time in comparison to the use of LSTM cell in RNNs as demonstrated by [60].

The implementation of DRQN is presented in Figure 4. 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 4, the following layers are very similar to the architecture of DQN, where RELU used as an activation 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.

E. Policy Gradient with Deep Neural Network (PGNN)

As was discussed above, the policy gradient optimises the policy parameters by following the gradient assent 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 actions 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

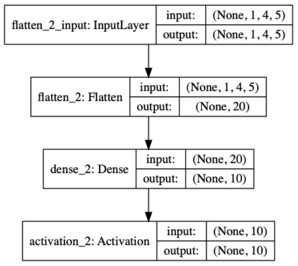
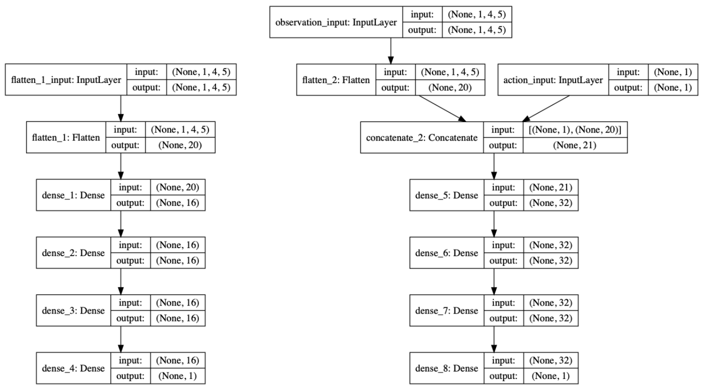


Fig. 5: Policy Gradient with Deep Neural Network (PGNN)



(a) Actor Network (b) Critic Network

Fig. 6: Deep Deterministic Policy Gradient (DDPG)

by the corresponding actions score. Finally, the PG calculates the mean of resulting gradient vectors and uses it to preform a gradient descent step.

Our implementation of the policy gradient neural network (PGNN) algorithm involves the definition of the neural network shown in Figure 5. A simple neural network 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 5 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 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 normalised across the many iterations of the execution.

F. Deep Deterministic Policy Gradient (DDPG)

The problem of having an environment with continuous state space or continuous action space is a challenging task for reinforcement learning. 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 POMDPs environment. The deep deterministic policy gradient (DDPG) algorithm was proposed in [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 policy gradient neural network that controls how the agent is behaving; judgement is based on the estimation it received from the critic network. Lillicrap at al. [61] claim that having a hybrid approach between Q-learning and policy gradient would overcome the problem of handling continuous actions space and the partial observability of the environment.

The architecture of DDPG is shown in Figure 2, the actor network is DQN similar to the architecture in Figure 2. 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 6b 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 in [62].

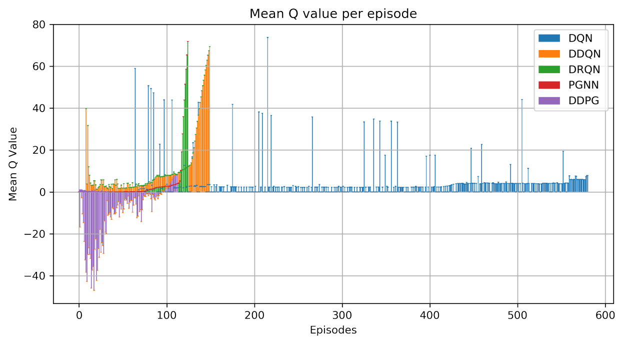


Fig. 7: Mean Q value per episode for (DQN, DDQN, DRQN, PGNN, and DDPG)

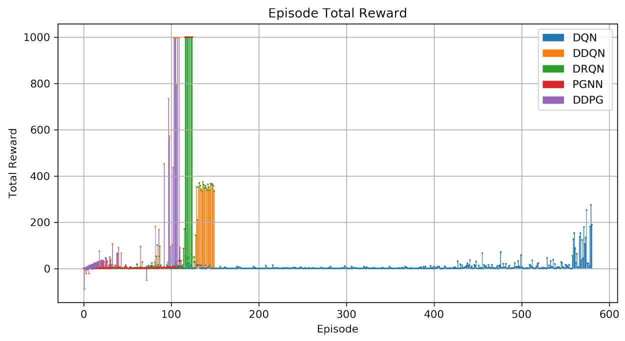


Fig. 8: Total Reward episode for (DQN, DDQN, DRQN, PGNN, and DDPG)

V. 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 including: a) mean of the Q-value, b) total reward yielded per single episode, c) 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. d) 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. e) 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 7 depicts the Mean of the Q-value for the five algorithms (DQN, DDQN, DRQN, PGNN, DDPG). From the figure it is clear that DRQN and DDQN both returned the heights 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 positive value around 130 episodes as shown in Figure 7. 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.

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 8. Looking at Figure 8, 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 8).

On the other hand, the mean absolute error (MAE) of the five algorithms is illustrated in Figure 9. Figure 9 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 errors 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 7. 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 10. The DRQN, DDPG and DDQN manage to converge faster and perform the adaptation in less time than DQN and

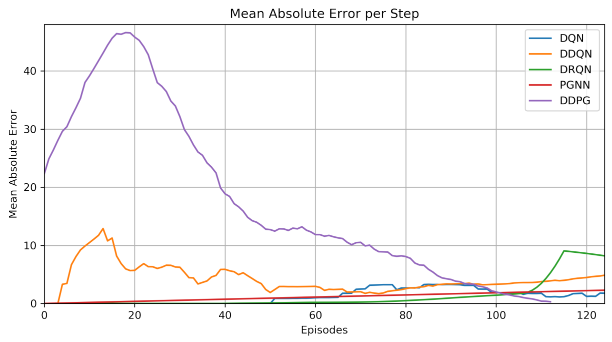


Fig. 9: Mean Absolute Error (MAE) for (DQN, DDQN, DRQN, PGNN, and DDPG)

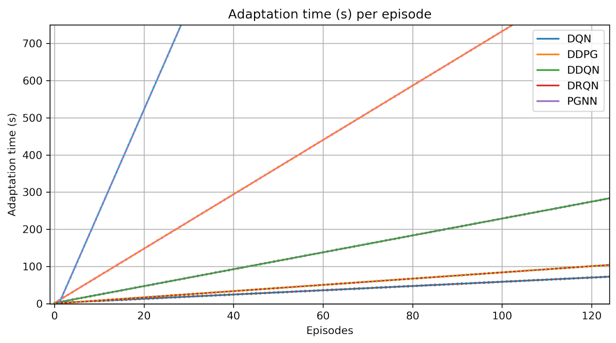


Fig. 10: Adaptation Time (s) Action Duration for (MAE) for (DQN, DDQN, DRQN, PGNN, and DDPG)

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 algorithm 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 11. The DDPG scores the highest loss, 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 9.

VI. Conclusions and Future Work

This paper presents i) a microservices architecture model that has continuous monitoring, continuous analysis of the observation space, and provides the architecture with dynamic decision making based on the employment of deep Q-learning/policy gradient algorithms. ii) 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/policy gradient algorithms. iii) An evaluation of five deep Q-learning/policy gradient algorithms including: deep Q-learning networks (DQN), duelling deep Q-learning networks

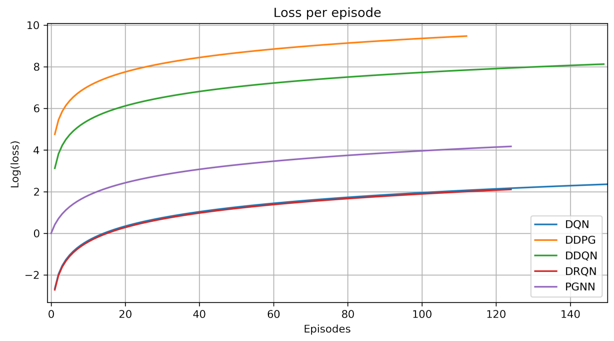


Fig. 11: Loss per episode for (MAE) for (DQN, DDQN, DRQN, PGNN, and DDPG)

(DDQN), deep recurrent Q-learning network (DRQN), policy gradient deep neural networks (PGNN), and deep deterministic policy gradient (DDPG). The evaluation in this research shows that DRQN is more suitable for driving the adaptation in POMDPs environment such as distributed microservices architecture. The DRQN shows a) faster training and convergence time, b) the highest maximum total reward in shortest number of episodes, c) faster adaptation time, and e) 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: i) to reveal the patterns between the collected observations and the cumulative yielded rewards for each pair of state-action; ii) to perform back-propagation through time, which minimises the loss and error rate of the predicted Q-value; iii) 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 maximise the reward value quickly and reaches a optimal state faster than the other algorithms; iv) 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/policy gradient 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 reinforcement learning in the decision making process improves the effectiveness of the adaptation and reduces the adaptation risk, including the possibility of resources over-provisioning 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 centralised multi-agents adaptation due to the limitation of a Docker swarm and the complexity to implement decentralised multi-agent 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 service oriented architecture to reach high levels of self-adaptability by integrating MDP agents and deep recurrent Q-learning networks in a well designed MAPE-K architecture.

References

[1] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *Intelligent Systems and Their Applications, IEEE*, vol. 14, no. 3, pp. 54–62, 1999.

[2] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, and B. Cukic, "Software engineering for self-adaptive systems: A research roadmap," *Software Engineering for Self-Adaptive Systems*, pp. 1–26, 2009.

[3] C. Bailey, D. W. Chadwick, and R. De Lemos, "Self-adaptive authorization framework for policy based rbac/abac models," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*. IEEE, 2011, pp. 37–44.

[4] M. Barbacci, "Software Quality Attributes: Modifiability and Usability," *Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA*, vol. 15213.

[5] H. Van Hasselt, "Reinforcement learning in continuous state and action spaces," in *Reinforcement learning*. Springer, 2012, pp. 207–251.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and others, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, 2016.

[7] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, pp. 1–6, 2006.

[8] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

[9] R. Bellman, "A Markovian decision process," *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[10] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[11] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR, abs/1507.06527*, vol. 7, no. 1, 2015.

[12] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[14] O. B. M. Jelasity, A. M. C. Fetzer, S. L. A. van Moorsel, and M. van Steen, "Self-star properties in complex information systems," 2005.

[15] P. Horn, "Autonomic computing: IBM's Perspective on the State of Information Technology," Tech. Rep., 2001.

[16] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, R. M. Malek, H. Müller, S. Park, M. Shaw, and M. Tichy, "Software Engineering for Self-Adaptive Systems: A Research Road Map (Draft Version)," *Dagstuhl Seminar Proc. 08031*, 2008.

[17] T. Strang and C. Linnhoff-Popien, "A context modeling survey," in *Workshop on advanced context modelling, reasoning and management, UbiComp*, vol. 4, 2004, pp. 34–41.

[18] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*. IEEE, 2009, pp. 132–141.

[19] M. Mikalsen, N. Paspallis, J. Floch, E. Stav, G. A. Papadopoulos, and A. Chimaris, "Distributed context management in a mobility and adaptation enabling middleware (madam)," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 733–734.

[20] D. Cheung-Foo-Wo, J. Y. Tigli, S. Lavirotte, and M. Riveill, "Self-adaptation of event-driven component-oriented middleware using aspects of assembly," *Proceedings of the 5th international workshop on Middleware for pervasive and ad-hoc computing: held at the ACM/IFIP/USENIX 8th International Middleware Conference*, pp. 31–36, 2007.

[21] W. Wei, X. Fan, H. Song, X. Fan, and J. Yang, "Imperfect Information Dynamic Stackelberg Game Based Resource Allocation Using Hidden Markov for Cloud Computing," *IEEE Transactions on Services Computing*, vol. 11, no. 1, pp. 78–89, Feb. 2016.

[22] D. A. Menascé and V. Dubey, "Utility-based qos brokering in service oriented architectures," in *Web Services, 2007. ICWS 2007. IEEE International Conference on*. IEEE, 2007, pp. 422–430.

[23] K. Kakousis, N. Paspallis, and G. A. Papadopoulos, “Optimizing the utility function-based self-adaptive behavior of context-aware systems using user feedback,” in *OTM Confederated International Conferences” On the Move to Meaningful Internet Systems”*. Springer, 2008, pp. 657–674.

[24] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum, “Model-based fault detection in context-aware adaptive applications,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 261–271.

[25] R. Hirschfeld, P. Costanza, and O. M. Nierstrasz, “Context-oriented programming,” *Journal of Object technology*, vol. 7, no. 3, pp. 125–151, 2008.

[26] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *Transactions on Autonomous and Adaptive Systems (TAAS*, vol. 4, no. 2, May 2009.

[27] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, “Software engineering for self-adaptive systems: A second research roadmap,” in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[29] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 135.

[30] S. Yoo, K. Yun, J. Y. Choi, K. Yun, and J. Choi, “Action-decision networks for visual tracking with deep reinforcement learning.” CVPR, 2017.

[31] J. C. Caicedo and S. Lazebnik, “Active object localization with deep reinforcement learning,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2488–2496.

[32] D. Jayaraman and K. Grauman, “Look-ahead before you leap: end-to-end active recognition by forecasting the effect of motion,” in *European Conference on Computer Vision*. Springer, 2016, pp. 489–505.

[33] H. Van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-Learning.” in *AAAI*. Phoenix, AZ, 2016, p. 5.

[34] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[35] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[36] J. Dowling and V. Cahill, “Self-managed decentralised systems using k-components and collaborative reinforcement learning,” in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM, 2004, pp. 39–43.

[37] M. Salehie and L. Tahvildari, “A policy-based decision making approach for orchestrating autonomic elements,” in *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*. IEEE, 2005, pp. 173–181.

[38] G. Tesauro, “Reinforcement learning in autonomic computing: A manifesto and case studies,” *IEEE Internet Computing*, vol. 11, no. 1, 2007.

[39] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari, “Adaptive action selection in autonomic software using reinforcement learning,” in *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*. IEEE, 2008, pp. 175–181.

[40] D. Kim and S. Park, “Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software,” in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS’09. ICSE Workshop on*. IEEE, 2009, pp. 76–85.

[41] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, “Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow,” in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

[42] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada, “Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures,” in *Quality of Software Architectures (QoSA), 2016 12th International ACM SIGSOFT Conference on*. IEEE, 2016, pp. 70–79.

[43] T. Wu, Q. Li, L. Wang, L. He, and Y. Li, “Using reinforcement learning to handle the runtime uncertainties in self-adaptive software,” in *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 2018, pp. 387–393.

[44] L. Busoniu, R. Babuska, and B. De Schutter, “A comprehensive survey of multiagent reinforcement learning,” *IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews, 38 (2), 2008*, 2008.

[45] A. Marinescu, I. Dusparic, A. Taylor, V. Cahill, and S. Clarke, “Decentralised multi-agent reinforcement learning for dynamic and uncertain environments,” *arXiv preprint arXiv:1409.4561*, 2014.

[46] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, “Multi-agent actor-critic for mixed cooperative-competitive environments,” in *Advances in Neural Information Processing Systems*, 2017, pp. 6379–6390.

[47] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Başar, “Fully decentralized multi-agent reinforcement learning with networked agents,” *arXiv preprint arXiv:1802.08757*, 2018.

[48] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, “Is multiagent deep reinforcement learning the answer or the question? a brief survey,” *arXiv preprint arXiv:1810.05587*, 2018.

[49] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International conference on machine learning*, 2016, pp. 1928–1937.

[50] T. Kiss, P. Kacsuk, J. Kovacs, B. Rakoczi, A. Hajnal, A. Farkas, G. Gesmier, and G. Terstyanszky, “Micadomicroservice-based cloud application-level dynamic orchestrator,” *Future Generation Computer Systems*, 2017.

[51] M. Plappert, “keras-rl,” https://github.com/keras-rl/keras-rl, 2016.

[52] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[53] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, “Requirements reflection: requirements as runtime entities,” *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pp. 199–202, 2010.

[54] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, “Using object-oriented typing to support architectural design in the C2 style,” *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 24–32, 1996.

[55] N. Paspallis, “Middleware-based development of context-aware applications with reusable components,” *University of Cyprus*, 2009.

[56] G. E. Monahan, “State of the arta survey of partially observable markov decision processes: theory, models, and algorithms,” *Management Science*, vol. 28, no. 1, pp. 1–16, 1982.

[57] L. Mou, P. Ghamisi, and X. X. Zhu, “Deep recurrent neural networks for hyperspectral image classification,” *IEEE Trans. Geosci. Remote Sens*, vol. 55, no. 7, pp. 3639–3655, 2017.

[58] A. Murad and J.-Y. Pyun, “Deep recurrent neural networks for human activity recognition,” *Sensors*, vol. 17, no. 11, p. 2556, 2017.

[59] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” *arXiv preprint arXiv:1409.1259*, 2014.

[60] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014.

[61] 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,” *arXiv preprint arXiv:1509.02971*, 2015.

[62] G. Daprato and A. Lunardi, “On the ornstein-uhlenbeck operator in spaces of continuous functions,” *Journal of Functional Analysis*, vol. 131, no. 1, pp. 94–114, 1995.