NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium | 978-1-6654-7716-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/NOMS56928.2023.10154301

Integrating state prediction into the Deep Reinforcement Learning for the Autoscaling of Core Network Functions

Yoichi Matsuo*, Jatinder Singh†, Shantanu Verma† and Guillaume Fraysse‡
* *NTT Network Service Systems Laboratories, NTT Corporation, Tokyo, Japan*
† *Orange Innovation Networks, Gurgaon, India*
‡ *Orange Innovation Networks, Paris, France*
yoichi.matsuo@ntt.com, {jatinder1.singh, shantanu.verma, guillaume.fraysse}@orange.com

***Abstract*—Reinforcement Learning (RL)-based scaling methods have been proposed to automatically scale in or out Network Functions (NFs) according to their traffic load. However, the scaling operation of NFs in real systems has significant *scaling process delay* which is the time difference between the moment the scaling of a NF is executed and the moment a new instance is ready to handle the incoming traffic, which leads to degrading the performance of basic RL-based approach. In this paper, we propose a Deep Reinforcement Learning (DRL)-based scaling method which integrates state prediction to deal with this delay. The proposed method was evaluated on a testbed based on the Open-Source Magma project that was automated to perform scaling actions of NF. The results show that the proposed method enables to handle a higher load and improves CPU and memory usage by approximately 14% when compared to the baselineDRL-based method.**

***Index Terms*—Prediction method, Deep Reinforcement Learning, Autoscaling, Network Functions**

I. Introduction

Network operators have to increase or decrease capacity of Core Network Functions (NFs) such as Mobility Management Entity (MME), Serving Gateway (SGW), and Packet data network Gateway (PGW), to adapt the NFs to the incoming traffic load by scaling in or out the NFs. This is because excessive capacity wastes resources while limited capacity leads to degradation in Quality of Service (QoS) of NFs. To address this problem, RL-based scaling methods [1]–[4] have been developed for scaling NFs, since RL has the ability to learn when to scale in or out by trial-and-error.

In general, RL-based scaling methods select an action based on the current state such as the number of instances, Central Processing Unit (CPU) usage, or users load to NFs. Here, the actions can be scaling out (resp. in) i.e., increasing (resp. decreasing) instances, and doing nothing (i.e., keeping the current instances). Then, the state transits to new state due to the selected action, and a reward is computed based on the new state which is input to train the RL. Then, RL selects the next action again. This process is then repeated over and over. RL is trained by trial-and-error so that future rewards are maximized.

However, it is possible to experience a *scaling process delay* between the moment the scaling action is executed and the moment the action is activated, i.e. the delay between the decision to create a new NF instance and the moment this new NF instance can handle the traffic. This is because launching the NF instance and initializing process such as loading the necessary configurations and network registrations, have to be executed. In classic RL environment, actions are based on the current state and have instantaneous effects whereas real environments are more stochastic in nature and actions experience the aforementioned delay. While the effect of selected action comes into play, the state for which the action was taken has changed already. As a result, the selected action by RL is less useful in the real system due to this delay.

In this paper, we propose a scaling method using DRL and integrated state prediction into DRL which enables DRL to select an action based on the predicted state at the time the activation of NFs is completed. The proposed method uses Double Dueling Deep Q-network (D3QN) [5] as DRL and Stochastic Gradient Descent (SGD) Regressor as a prediction method for predicting next state which is merged into Q function in D3QN. Since the proposed method enables D3QN to select an action based not only on the current state but also on the next state in Q function, the effect of delay is reduced which leads to improving the performance of D3QN.

For evaluating the proposed method, we used a platform, in which MME, SGW, and PGW are deployed. The proposed method scales in or out these NFs according to the state such as resource usage of NFs and load into NFs. The results show that the proposed method improves the reward by approximately 30% and resources usage by 5% on average compared with basic D3QN.

II. Proposed method

The proposed method, named *state-predicted D3QN method*, consists of D3QN for deciding the scaling and SGD Regressor for adapting the delay by predicting the state of NFs. The overview of the state-predicted D3QN method is

This work is joint collaboration research project between Orange S.A. and NTT Corporation.

The second and third authors are equal contributors.

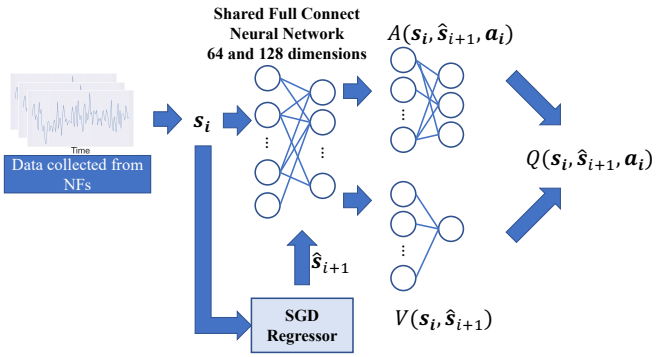


Fig. 1: The overview of D3QN and SGD Regressor integration

illustrated in Figure 1. In this section, first D3QN-based scaling method is explained. Then, we describe how the delay degrades the performance and how to integrate the prediction method into D3QN.

A. D3QN-based scaling method

In D3QN, the Q function, which is the state-action value function, is trained by trial-and-error to decide the action (e.g. scaling in or out) to be taken according to the state of the environment computed from the metrics collected from NFs.

Let $\boldsymbol{s}_i$ and $\boldsymbol{a}_i$ be vectors which represents state and action at iteration i, respectively. In our experiments, states $\boldsymbol{s}_i$ are normalized CPU usage, memory usage of NFs, and load of the NFs (e.g. the number of connected user devices to MME) computed from collected metrics in the NFs or environment at each iteration i. We define the actions a_i as follows.

$$\boldsymbol{a}_i = \begin{cases} 0 & \text{(keep current NFs)} \\ 1 & \text{(scale out NFs)} \\ 2 & \text{(scale in NFs)} \end{cases} \tag{1}$$

In D3QN, the Q function consists of state-value function $V(\boldsymbol{s}_i)$, which represents the state value of $\boldsymbol{s}_i$, and advantage function $A(\boldsymbol{s}_i, \boldsymbol{a}_i)$, which represents how advantageous it is if a certain action is taken at the current state. $V(\boldsymbol{s}_i)$ and $A(\boldsymbol{s}_i, \boldsymbol{a}_i)$ are estimated using Deep Learning (DL), where both functions share the first layers of neural network and have different last layers as illustrated in Figure 1.

In our experiments, we used a fully connected neural network of $V(\boldsymbol{s}_i; \theta, \beta)$ and $A(\boldsymbol{s}_i, \boldsymbol{a}_i; \theta, \alpha)$ where θ is the parameter of shared network, α is the parameter of last layer in $A(\boldsymbol{s}_i, \boldsymbol{a}_i; \theta, \alpha)$, and β is the parameter of last layer in $V(\boldsymbol{s}_i; \theta, \beta)$. Then, Q function is defined as follows.

$$Q(\boldsymbol{s}, \boldsymbol{a}; \theta, \alpha, \beta) = V(\boldsymbol{s}; \theta, \beta) + A(\boldsymbol{s}, \boldsymbol{a}; \theta, \alpha) - \frac{1}{|A|} \sum_{\acute{\boldsymbol{a}}} A(s, \acute{\boldsymbol{a}}; \theta, \alpha). \tag{2}$$

The $Q(\boldsymbol{s}, \boldsymbol{a}; \theta, \alpha, \beta)$ is trained to maximize the sum of the discounted future rewards $R_i = \sum_{l=i}^{\inf} \gamma^{l-i} r_i$, where r_i is the reward function at iteration i and γ is the discount value.

According to the survey paper [3], many kinds of reward functions are developed for the scaling problem based on key performance indicators such as resource usage of instances or convergence time of RL. Solving the scaling problem of NFs allows to maximize the performance, such as the amount of handled traffic load, for QoS or cost (e.g. the energy consumption or the number of billable running instances of NFs). Thus, we define the reward function r_i at iteration i as follows.

$$r_i = \begin{cases} -10 & \text{(invalid actions or crash)} \\ \frac{|\text{UEs}|}{|\text{NFs}|} & \text{(otherwise)}, \end{cases} \tag{3}$$

where $|\text{UEs}|$ is the number of user devices such as smartphone connected to the NFs (i.e. the amount of handled traffic load in NFs) and $|\text{NFs}|$ is the number of running NFs (i.e., the cost). Thus, this reward function represents the performance. Here, since $|\text{UEs}|$ is normalized, $r_i \leq 1$. Also, a penalty of -10 is given when the environment experiences a crash or when an invalid action is selected. Note that, we define the invalid action as scaling in when the current running NF is one, or scaling out when the maximum number of NFs is already running. Crashes of the NFs can occur when the load is too high and scaling out action is not taken in time.

Finally, $Q(\boldsymbol{s}, \boldsymbol{a}; \theta, \alpha, \beta)$ is trained by minimizing the following equation.

$$\mathcal{L} = Y - Q(\boldsymbol{s}_i, \boldsymbol{a}_i; \theta, \alpha, \beta). \tag{4}$$

with

$$Y = R_i + \gamma Q\left(\boldsymbol{s}_{i+1}, \underset{\tilde{\boldsymbol{a}}}{\text{argmax}}\ Q\left(\boldsymbol{s}_{i+1}, \tilde{\boldsymbol{a}}; \acute{\theta}, \acute{\alpha}, \acute{\beta}\right)\right)$$
$$\tilde{\boldsymbol{a}} = \underset{\acute{\boldsymbol{a}}}{\text{argmax}}\ Q\left(\boldsymbol{s}_{i+1}, \acute{\boldsymbol{a}}; \acute{\theta}, \acute{\alpha}, \acute{\beta}\right). \tag{5}$$

where $Q(\boldsymbol{s}_{i+1}, \acute{\boldsymbol{a}}; \acute{\theta}, \acute{\alpha}, \acute{\beta})$ is the target Q function, which is introduced in the Double Deep Q-Network architecture to avoid overestimation of Q function [6]. $\acute{\theta}$, $\acute{\alpha}$, and $\acute{\beta}$ are the parameters in the previous iteration, and replaced by θ, α, and β at each given iteration.

B. Integrating the state prediction into D3QN

In the scaling problem, although D3QN decides of an action based on the current state, there is delay longer than 60 seconds between the execution of the scaling command and activating NFs, which leads to decreasing the performance of D3QN. Especially, in our platform, we need a little more than 120 seconds for scaling in or out. Action 0 also has a duration, 20 seconds in our setup. Since the data is collected every 15 seconds, it takes 15 seconds to get the information on the next state to which we added a 5-second buffer.

Figure 2 is an example of what the delay is and how the delay degrades the performance. The vertical axis represents the number of UEs connected to NFs (i.e., the loads of the NFs). In this figure, the number of connected UEs remains constant for several iteration steps (e.g., between iteration steps 710 and 720 or between iteration steps 740 and 750). This is because NF has crashed under a heavy load. Then,

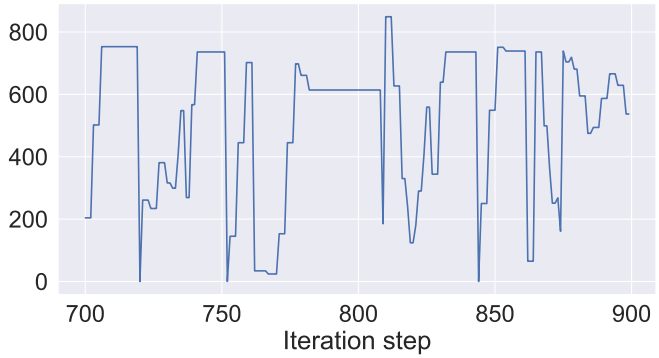


Fig. 2: Example of number of connected UEs

after approximately 20 steps, the number of UEs connected decreases suddenly because D3QN restarts the NF. Thus, there is approximately 300 seconds from the time the crash occurs and when the NF becomes available again.

The Q function might be trained to avoid the crashes by considering the future reward. However, even though the crash occurs, if D3QN takes an action 0, D3QN can get rewards approximately 7 times. On the other hand, if D3QN takes scale in or out action, D3QN gets a penalty due to a crash since it takes more than 120 seconds. As a result, D3QN might be optimized to take an action 0 even in the crash. Additionally, D3QN should take a scaling out action 120 seconds before the crash occurs

Thus, in the proposed method, SGD Regressor is introduced for predicting the next state $\hat{\boldsymbol{s}}_{i+1}$. Integrating next state prediction into the Q function enables the D3QN to decide the action based not only on the current state but also the future state which reduces the effect of the delay. In the proposed method, SGD Regressor is trained independently from D3QN as follows.

$$\hat{\boldsymbol{s}}_{i+1} = SGD(\boldsymbol{s}_{i-k}, \ldots, \boldsymbol{s}_i), \tag{6}$$

where we set $k = 3$ in our experiments, and SGD is trained using the buffered previous states.

As displayed in Figure 1, the predicted next state $\hat{\boldsymbol{s}}_{t+1}$ is input to the shared full connect neural networks. Then, the proposed method redefines the Q function as follows.

$$Q\left(\boldsymbol{s}, \hat{\boldsymbol{s}}, \boldsymbol{a}; \theta, \alpha, \beta\right) = V(\boldsymbol{s}, \hat{\boldsymbol{s}}; \theta, \beta) + (A(\boldsymbol{s}, \hat{\boldsymbol{s}}, \boldsymbol{a}; \theta, \alpha) - \frac{1}{|A|} \sum_{\acute{\boldsymbol{a}}} (A(\boldsymbol{s}, \hat{\boldsymbol{s}}, \acute{\boldsymbol{a}}; \theta, \alpha). \tag{7}$$

Here, the training of the SGD is executed independently from the training of D3QN.

Finally, the Q function in the proposed method is trained by minimizing the following equation.

$$\mathcal{L} = Y - Q\left(\boldsymbol{s}_i, \hat{\boldsymbol{s}}_{i+1}, \boldsymbol{a}_i; \theta, \alpha, \beta\right), \tag{8}$$

TABLE I: Average values of metrics for both methods

Method	Reward	# of UEs	CPU usage	Memory usage
Basic D3QN	-2.08	392.95	35.1%	17.7%
Proposed	-1.39	407.17	40.0%	20.4%

with

$$Y = R_i + \gamma Q\left(\boldsymbol{s}_{i+1}, \hat{\boldsymbol{s}}_{i+2}, \underset{\tilde{\boldsymbol{a}}}{\operatorname{argmax}}\ Q\left(\boldsymbol{s}_{i+1}, \hat{\boldsymbol{s}}_{i+2}, \tilde{\boldsymbol{a}}; \acute{\theta}_i\right)\right)$$

$$\tilde{\boldsymbol{a}} = \underset{\acute{\boldsymbol{a}}}{\operatorname{argmax}}\ Q\left(\boldsymbol{s}_{i+1}, \hat{\boldsymbol{s}}_{i+2}, \acute{\boldsymbol{a}}; \acute{\theta}_i\right).$$

Note that although we have selected D3QN and SGD as a DRL and prediction method as a first step. However, the architecture of the proposed method can be used with other DRL and prediction methods.

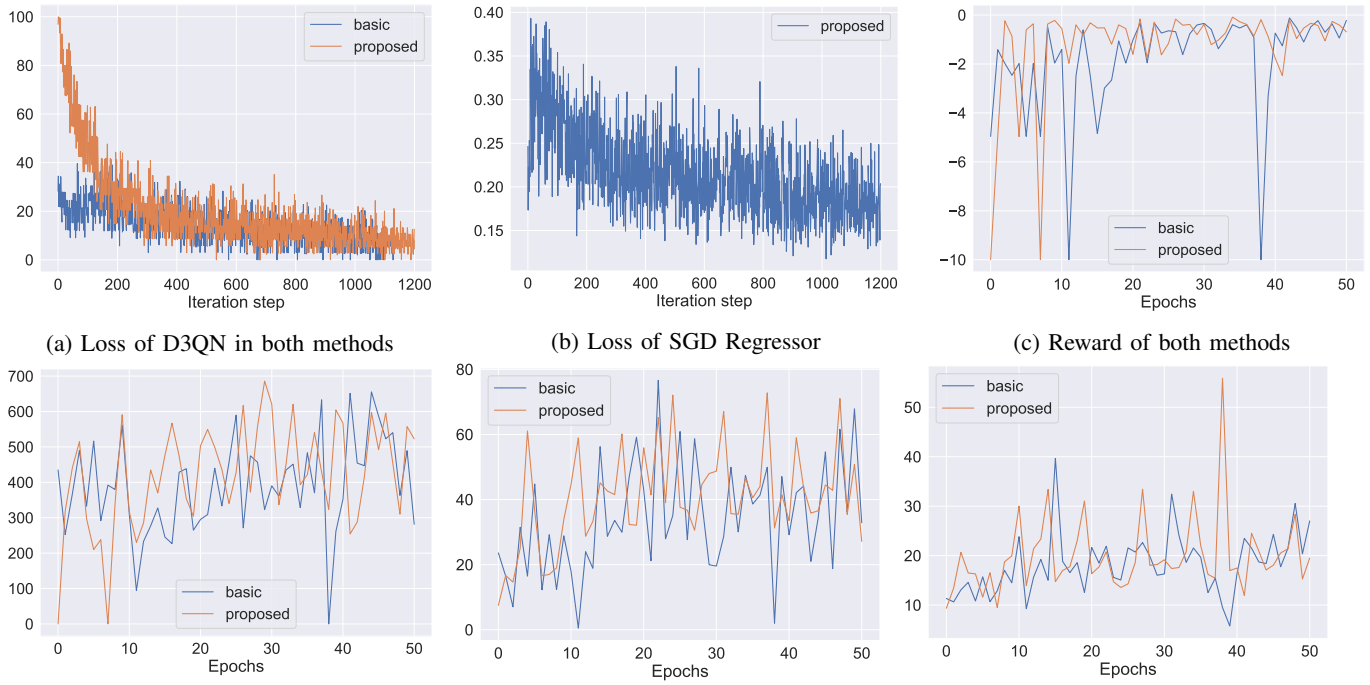
III. Evaluation

The method described in Section II is compared here with the basic D3QN on a testbed that allows to deploy multiple instances of NFs and generates load on them.

A. Experiment platform and setting

For the experiments, we prepared MME, SGW, and PGW as NFs using the Magma Open Source software [7]. In our experiments, these NFs are deployed as a single core network package known as Access Gateway (AGW) on Flexible Engine [8], which is a Cloud Infrastructure as a Service (IaaS), and we set a limit on the total resources by setting the maximum number of AGWs to five. Specific Python scripts integrate the RL pipeline with Magma Network Management System (NMS) Application Programming Interface (API) as well as Flexible Engine API which facilitates the collection of metrics from the core network and automating the actions on the platform.

Also, we generated the load to those NFs based on the sine wave, which represents the timestamps of user device attach requests to MME. According to the load, the devices are attached (resp. detached) to (resp. from) the running instances of the MME. Especially, since the load is heavy at the peak, a crash might occur on the MME if an appropriate number of instances aren't running. Thus, both the baseline D3QN and proposed state-prediction D3QN method have to learn the timing of scaling out those NFs to avoid the crash. Regarding architecture of D3QN, as illustrated in Figure 1, the two layered full connected neural networks is used for the shared part of $V(\boldsymbol{s}; \theta, \beta)$ and $A(\boldsymbol{s}, \boldsymbol{a}_i; \theta, \alpha)$. Rectified linear unit (ReLU) function is used as the activation function between layers. The number of dimensions of the shared layers is set to 64 and 128, respectively. The dimension of the last layers in $A(\boldsymbol{s}_i, \boldsymbol{a}_i; \theta, \alpha)$ and $V(\boldsymbol{s}_i; \theta, \beta)$ are three (i.e. the number of actions) and one (i.e. the scalar value for the current state) respectively. The number of epochs is set to 50 and each epoch has 200 steps. However, if a crash occurs, we enforce to proceed the next epoch even before reaching the 200 steps by resetting the environment, i.e., restarting NFs.



(a) Loss of D3QN in both methods

(b) Loss of SGD Regressor

(c) Reward of both methods

(d) # of connected UEs of both methods

(e) CPU usage of both methods

(f) Memory usage of both methods

Fig. 3: Metrics of both methods

B. Experimental results

We evaluated the proposed method i.e., the state-predicted D3QN method, by comparing it to the basic D3QN as a baseline. Table I and Figures 3 show the results of both methods. In the table, the average values of each metric during the epochs are described. In the figures, horizontal axis on Figures 3a and 3b show the number of steps since the start of the test, while other figures use the number of epochs.

First of all, the state-predicted D3QN method improves the average reward by approximately 30%, which means the the state-predicted D3QN method selects the better action compared with the basic D3QN. Note that, the rewards are negative in both methods because the positive rewards are always less than 1 due to calculating using the normalized number of UEs, and the penalty is -10, which is larger than the positive rewards. Figure 3c shows the sequence of the rewards during the experiments. The reward of the state-predicted D3QN method exceeds that of the basic D3QN in almost all periods after 10 epochs.

On the other hand, the state-predicted D3QN method gets the penalty in some cases during the first 10 epochs. This is because not only D3QN but also SGD Regressor are not well trained in the first several epochs. Thus, D3QN cannot select an appropriate action because the predicted state might be different from the actual next state. Figure 3a shows the loss of D3QN in the both methods. Since the loss of D3QN in the proposed method is much larger than that of basic D3QN, it also shows D3QN in the proposed method was not trained well comparing basic D3QN during the first several epochs. However, as in the figure, D3QN and SGD Regressor are trained faster and the loss of D3QN in the proposed method reaches the same level as that of basic D3QN.

Table I shows that the number of connected UEs, which represents the amount of handled traffic load by NFs, is increased by 15, Resource utilization including CPU usage and memory usage which are related to the cost, are also increased by 13.9% points and 15.25%, respectively. This indicates that the proposed method is able to control the NFs scaling more efficiently. Figures 3d, 3e, and 3f are sequence during the experiment. Similarly to the reward, the number of connected UEs and resource utilization increased after 10 epochs.

IV. Conclusion

In this paper, we pointed out the scaling process delay that occurs during the process of scaling in or out NFs, which has to be considered in the real system. Then, D3QN-based scaling method with state prediction is introduced to handle the delay. As a first step, the SGD Regressor is tested as a prediction method and the results show an improvement of the performance of D3QN.

Future works include evaluating the proposed method with different workloads derived from commercial traffic and under a variety of conditions. Since most of the prediction methods are adaptable to the proposed method, comparing the performance of different prediction methods and predicting time steps are also future works.

References

[1] J. V. Bibal Benifa and D. Dejey, “Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment,”

Mob. Netw. Appl., vol. 24, no. 4, p. 1348–1363, aug 2019. [Online]. Available: https://doi.org/10.1007/s11036-018-0996-0

[2] Z. Wang, C. Gwon, T. Oates, and A. Iezzi, "Automated cloud provisioning on aws using deep reinforcement learning," *arXiv preprint arXiv:1709.04305*, 2017.

[3] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. García Garino, "Reinforcement learning-based application autoscaling in the cloud: A survey," *Engineering Applications of Artificial Intelligence*, vol. 102, p. 104288, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197621001354

[4] Y. Jin, M. Bouzid, D. Kostadinov, and A. Aghasaryan, "Model-free resource management of cloud-based applications using reinforcement learning," in *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2018, pp. 1–6.

[5] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, 2016, pp. 1995–2003. [Online]. Available: https://proceedings.mlr.press/v48/wangf16.html

[6] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 2094–2100.

[7] MAGMA, 2021, accessed on 16th, August, 2022. [Online]. Available: https://magmacore.org/

[8] Flexible Engine. Cloud - Orange Business Services. [Online]. Available: https://cloud.orange-business.com/en/offers/infrastructure-iaas/public-cloud/