M Ű E G Y E T E M 1 7 8 2

Scaling resources with reinforcement learning

Ph.D. dissertation summary
by **Nguyen Tuan Hai**

Department of Networked Systems and Services
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics

Research supervisor:
Prof. Do Van Tien
Budapest University of Technology and Economics

Budapest, Hungary
2022

1 Introduction

Imagine you walk into your university's local grocery store to grab an iced coffee before your morning lecture. However, inside you see that only one out of the three cashier desks is open and the line to it seems never-ending. The large crowd deters you from joining the queue and you decide to skip your morning coffee. Alternatively, imagine your university announces that students may register to courses for the following semester through its online education system. The registration process starts at 6pm on Monday. You spend your whole afternoon assembling the perfect timetable, but so do thousands of other students. When the clock hits 6pm every student tries to login at the same time causing the university servers to overload.

The problems in the above two cases stem from a common root: there is not enough resources to serve a given size of demand. This is a ubiquitous issue that we may encounter daily. How many flights do we need to operate on an airplane route? How many mobile network cells do we need in a densely populated area? How many server nodes do we need for the launch of a much-anticipated online game? How many researchers do we need to change a lightbulb?[1] Though these questions look very similar from far, each of them poses challenges unique in their own context. Therefore, it is not practical to try to find a general method that can solve each case study. During my studies I focused on resource provisioning in information and communication technology (ICT) systems, more particularly, in cloud-based systems in 5G networks, and in actor-based systems for highly concurrent tasks.

Determining the correct amount of resources is not a trivial task. Having too much of it leads to overprovisioning. This means that we may end up having excess resources that unnecessarily increase our operational costs. For example, hiring too many cashiers or renting too many servers. On the other hand, not having enough resources may lead to underprovisioning. This may cause degradation in the quality of service (QoS), resulting in unsatisfied customers and loss of revenue. Furthermore, we also need to take the varying nature of the demand into account. During peak hours we may need more cashiers in the store or more computer servers in the cloud. This calls for a dynamic solution that could adjust the resources based on the demand. I used machine learning techniques, more particularly reinforcement learning (RL) and deep reinforcement learning (DRL), to dynamically adjust resources.

[1]The answer is two. One implements an AI, the other changes the lightbulb 10 thousand times to generate the training data for the AI.

2 Motivations and goals

In network function virtualization (NFV), network services (NS) are composed of physical (PNF) or virtual (VNF) network functions. This concept was introduced by the European Telecommunications Standard Institute (ETSI) [4, 5]. NFV allows the rapid and flexible deployment of NSs while also providing reliability and scalability. In practice a network function may be made up of multiple VNF component (VNFC) instances and the operator may start multiple VNFCs to serve the incoming demand. For example, imagine a network function that processes video frames. Depending on the number of users and the frame rate of the video streams, the operator may need more VNFCs, otherwise the processing would be delayed.

In the actor model, the unit of computation is the actor. These actors can send messages to each other, and process incoming messages. These actions may be carried out asynchronously, allowing a highly concurrent operation, suitable for distributed systems, like IoT systems [3, 6]. This architecture hides low-level elements, such as threads or locks, making development of applications more convenient. However, the size of the underlying pool of threads may greatly impact the operation, as idle threads in a large thread pool may need to reserve a large amount of memory and context switching between threads may slow down the system.

The User Plane Function (UPF) is one of many network functions in the 5G core (5GC). It is responsible for forwarding the packets of a user equipment (UE) to the data network (DN). This means that the service of UEs greatly depends on the number of UPF instances running [12]. In a cloud, UPFs would run in containers called Pods and a cloud orchestrator like Kubernetes [2] would control the number of Pods in the system. DRL could be a good alternative to Kubernetes's built-in Horizontal Pod Autoscaler (HPA). However, the DRL's stochastic behavior may occasionally cause the execution of incorrect actions which would degrade the performance. This could be remedied with classification methods that assign scaling actions to system states deterministically.

During my studies my objectives were the following:

- Develop a method with deep reinforcement learning (DRL) to scale VNF instances under constant and varying traffic. The resulting method would be flexible, meaning that the QoS level could be adjusted by the operator.
- Develop and compare DRL methods for scaling the thread pool size of an actor-based system considering varying traffic and thread start time. Also, compare the results with built-in methods, such as the time out rule used by the `ThreadPoolExecutor` in Java.
- Show that DRL can perform better at scaling UPF Pods than the HPA and then combine DRL with a classification method such as support vector machines (SVM) to remedy the DRL's stochastic action selection.

3 Methodology

In general, autoscaling methods can be classified into five categories: queuing theory, threshold-based rules, control theory, time series analysis [1], and reinforcement learning [8]. The advantage of RL is that it does not require expert knowledge on the field of the problem. An RL agent learns the quality of its decisions by interacting with the environment and collecting experience. This is also often called the model-free property of RL.

In order to apply RL to a problem, first we would need to define a corresponding Markov decision problem (MDP). An MDP is a 5-tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$, where $\mathcal{S}$ is the set of states in the system; $\mathcal{A}$ is the set of actions; $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ describes the transition probabilities between states; $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ describes the immediate reward between state transitions; and $\gamma \in [0, 1]$ is the so-called discount factor [14].

The learning loop of an RL agent is the following: the agent observes a state $s_i \in \mathcal{S}$; the agent uses its policy $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$ to determine the action a_i, where $\mathcal{P}(\cdot)$ denotes the probability distribution over a set; action a_i is executed on the environment; the agent observes the reward r_i and the next state s_{i+1}; learning from the tuple (s_i, a_i, r_i, s_{i+1}) the agent improves its policy π.

Correctly identifying $\mathcal{S}$ and $\mathcal{A}$ is crucial. If their dimension is too high, it would lead to a so-called explosion of the state- or action-space. This means that they would be so large, it would be infeasible for an RL-agent to explore them fully in reasonable time. On the other hand, if they are too small, it may be impossible to differentiate between two otherwise distinct states of the system or it would limit the possible actions of the agent in a state. The definition of the reward function r is also very important as it describes the objective the RL agent tries to optimize. If not well defined, the RL agent may get stuck in local minima during learning. The transition function p describes the model, however the RL agent does not need to know this beforehand, as it only needs to interact with the environment and it can learn only from the state transitions and the reward, model-free.

There are various RL methods in the literature. One of the most well studied is Q-learning [15] which tries to maximize the state-action value function $Q(s, a)$ (also called q-factors) for each $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Due to the size of $\mathcal{S}$ usually it is impractical to store the q-factors in memory. Instead, neural network approximation is used to create deep Q-learning or deep Q-networks (DQN) [10]. Another popular class of methods is policy gradient methods like the actor-critic (AC) [9] or proximal policy optimization (PPO) [13, 11].

4 Theses

4.1 Scaling VNFC instances with DRL

Let us assume a NS that is made of VNFCs. For example, a VNF that processes video streams frame by frame for mobile users. Further assume that requests arrive according to a Poisson process with constant rate λ and that service times are distributed exponentially. With ΔT time interval the operator may decide to scale in, scale out, or do nothing. Upon arrival, a request may find no idle VNFC instance to serve it. I considered two cases: requests are enqueued in an infinite buffer and are served in a first-in first-out (FIFO) manner; there is no buffer and requests are blocked. The goal of the operator is to minimize the number of VNF instances while meeting a threshold for delay (T_{th}) or blocking ($p_{b,th}$) in average.

Let the state description be $\{I(t), J'(t), K'(t)\}$, where $I(t)$ is the number of VNFC instances, $J'(t)$ indicates if there is a non-busy instance, and $K'(t)$ indicates whether there is a request in a queue. Let the action space be $\mathcal{A} = \{\texttt{ScaleIn}, \texttt{ScaleOut}, \texttt{None}\}$. The reward functions that describe the objective to optimize are:

$$r_{m,b} = \begin{cases} \frac{I(t_m^+)-n}{n} & \hat{p}_{b,m} > p_{b,th} \\ \frac{-I(t_m^+)}{n} & \hat{p}_{b,m} \leq p_{b,th} \end{cases}$$

and

$$r_{m,T} = \begin{cases} \frac{I(t_m^+)-n}{n} & \hat{T}_m > T_{th} \\ \frac{-I(t_m^+)}{n} & \hat{T}_m \leq T_{th} \end{cases}$$

for blocking and for delay, where $p_{b,th}$ and T_{th} are parameters set by the QoS.

I propose the use of DRL to autoscale VNFC instances. During my studies I found that the actor-critic method can quickly and reliably find the optimal policy.

Thesis group 1 *[P1] A DRL agent trained with the actor-critic method can minimize the number of VNFC instances in a NS.*

I formulated the MDP for this particular VNF problem and through simulations showed that DRL is an adequate tool to control.

Thesis 1.1 *With the above described MDP, the actor-critic algorithm is capable of finding the optimal number of VNFCs in a VNF scaling problem without expert knowledge, given a traffic with Poisson arrivals and fixed arrival rate.*

For Thesis 1.1 I carried out the experiments considering a constant λ arrival rate. For the case of varying arrival rates, I trained multiple DRL agents for various λ values. I propose the multi-agent scheme where an agent is selected from this set depending on the current arrival rate.

Thesis 1.2 *In case of varying traffic, a multi-agent scheme can scale VNFCs to minimize their number while keeping QoS level.*

4.2 Scaling thread pool in actor-based systems

Assume the MDP formulation of an actor-based IoT system, where at time t sessions arrive according to a Poisson process with arrival rate $\lambda(t)$ and the length of the sessions is exponentially distributed with rate μ. During each session, fixed sized packets are sent to the system periodically with a constant inter-arrival time. These packets are processed by threads in a thread pool. If all threads are busy, packets are enqueued in buffer with a size limit. If the buffer is also full, packets are blocked.

Let $\{X(t), Y(t), B(t), \hat{\lambda}(t)\}$ describe the state space, where $X(t)$ is the number of active sessions, $Y(t)$ is the size of the thread pool, $B(t)$ is the number of busy threads, and $\hat{\lambda}(t)$ is the approximated arrival rate. I considered two action spaces. In the on/off action case let $\mathcal{A}_1 = \{\texttt{START\_THREAD}, \texttt{STOP\_THREAD}, \texttt{NoOp}\}$, which means that the agent may start or stop a thread or do nothing. In the multi action case let $\mathcal{A}_2(t) = \{B(t), B(t)+1, \ldots, N\}$, where N is the upper limit for the size of the thread pool. Finally let the reward function be

$$r_i = \begin{cases} -\kappa\hat{\delta}_i & \text{if } \delta_{th} < \hat{\delta}_i \text{ or } p_{b,th} < \hat{p}_{b,i} \\ -Y(T_i) & \text{if } \delta_{th} \geq \hat{\delta}_i \text{ and } p_{b,th} \geq \hat{p}_{b,i} \end{cases},$$

where δ_{th} and $p_{b,th}$ are the threshold delays and blocking rates set by the QoS, $\hat{\delta}_i$ and $p_{b,i}$ are the measured delay and blocking rate in the i-th decision epoch, and κ is an adjustable hyperparameter.

I propose the use of DRL to control the size of a thread pool in an actor-based system.

Thesis group 2 *[P2] DRL can optimize the size of a thread pool in an actor-based system.*

I formulated the MDP for a thread pool and used PPO to train a DRL agent. I also identified the hyperparameters the solution was sensitive for and devised a method that combines population-based training with grid search to optimize these hyperparameters. Through simulations I showed that DRL outperforms the timeout method often used by thread pools.

Thesis 2.1 *With the above described MDP, DRL can minimize the number of threads in the thread pool of an actor-based system while keeping the mean blocking rate below a given blocking threshold. The hyperparameters κ and ξ can be searched with the mix of population-based training [7] and grid search.*

Thesis 2.2 *When the starting time of the threads is small, a DRL agent trained with PPO performs comparably to the timeout method. With larger starting times the DRL agent outperforms the timeout method.*

Thesis 2.3 *Starting and stopping multiple threads in one action does not improve the performance.*

4.3 Scaling of UPF Pods in the 5G/6G core

Assume the MDP formulation of a 5G core network running on a cloud, where groups of servers (called pods) can be initialized or stopped. At time t, PDU sessions arrive according to a Poisson process with rate $\lambda(t)$ and their service time is distributed exponentially with rate μ.

Let $\{d_{\text{on}}(t), d_{\text{boot}}(t), l_{\text{sess}}(t), l_{\text{free}}(t), \hat{\lambda}(t)\}$ describe the state space, where $d_{\text{on}}(t)$ denotes the number of running Pods; $d_{\text{boot}}(t)$ denotes the number of booting Pods; $l_{\text{sess}}(t)$ is the number of PDU sessions; $l_{\text{free}}(t)$ is the available capacity of additional sessions the system could handle; and $\hat{\lambda}(t)$ is the approximated arrival rate. Let the action space be $\mathcal{A} = \{\texttt{start}, \texttt{stop}, \texttt{noaction}\}$, with each action denoting the start or stop of a Pod, or no action respectively. The reward is

$$r_i = \begin{cases} -\kappa \hat{p}_{b,i} & \text{if } \hat{p}_{b,i} > p_{b,th} \\ -\hat{d}_{\text{on}}(T_i) & \text{if } \hat{p}_{b,i} \leq p_{b,th} \end{cases}$$

, where $p_{b,th}$ is the threshold for the blocking rate, $\hat{p}_{b,i}$ is the measured blocking rate in the i-th decision epoch, and κ is a hyperparameter.

In [P3] I formulate the MDP for the control of UPF Pods in the 5G Core running on the cloud. I further proposed the use of DRL to scale UPF Pods in the 5G Core. I showed that DRL outperforms the HPA in Kubernetes.

Thesis group 3 *[P3] Agents trained with DRL or SVM can optimize the number of UPF Pods in the 5G Core.*

Thesis 3.1 *With the above described MDP formulation (state description, action space, reward function), a RL agent minimizes the UPF Pod count while keeping the mean blocking rate below a certain blocking threshold $p_{b,th}$.*

Thesis 3.2 *A RL agent trained with PPO under the average reward scheme outperforms the HPA.*

PPO uses a stochastic action selection method, which means that it outputs a probability distribution over the possible actions in a given state. This is very useful during training, because it allows the RL agent to pick any action even if it has a low probability, and allows the agent to explore the state space and discover potentially better policies. However, this also means that during operation, the RL agent may pick the wrong action with very low probability. This problem commonly arises when there is a big and sudden change in traffic. Instead of using the stochastic action selection method, it is possible to generate a dataset of state-action pairs using the RL agent and train a classifier on them that assigns actions to states.

Thesis 3.3 *A linear SVM classifier that assigns actions to states trained on the dataset generated by the RL agent performs better when traffic changes suddenly. In average, the trained agent does not outperform the RL agent, but it still works better than the HPA.*

One drawback of the SVM-based agent that it cannot learn online. Therefore, I propose its use for cases where sudden and large changes in traffic is common and one time training of the agent is sufficient. Otherwise, I suggest the use of the DRL agent trained with PPO.

5 Application of results

Results of Thesis group 1 can be applied in the context of VNFs for scaling VNFC instances. The proposed DRL agent could be implemented as a separate component that communicates with the NFV Management and Orchestration (NFV-MANO). In this setup the NFV-MANO periodically feeds information to the DRL agent and asks it for scaling advice. The agent then suggests a scaling operation based on the information it received. Scaling operations are then triggered by either the VNF Manager (VNFM) or the network function virtual orchestrator (NFVO).

Results of Thesis group 2 can be applied to thread pools in Akka, more particularly, thread pools in the JVM. In this case a DRL agent could be integrated into a Java `ExecutorService`. The `ExecutorService` would monitor the thread pool and run a DRL agent. The service would then start new threads or terminate idle threads in the thread pool based on the DRL agent's suggestions.

Results of Thesis group 3 could be applied in a cloud orchestrator like Kubernetes to scale UPF Pods. In this scenario UPF instances would be packed into Kubernetes Pods with each Pod running a single UPF instance. A custom metrics server would monitor the state of 5G Core by keeping track of the UPF Pod count and the number of active PDU sessions in the system. This state information would be sent to the DRL-based autoscaler to process. This autoscaler would then suggest scaling actions through the Kubernetes Scale interface. A DRL-based agent would be able to learn online during operation. An SVM-based agent would need a trained DRL agent first to generate a dataset and after training it could be integrated into the autoscaler.

Publications

[P1] Hai T. Nguyen, Tien Van Do, Attila Hegyi, and Csaba Rotter. An approach to apply reinforcement learning for a vnf scaling problem. In *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 94–99, 2019.

[P2] Hai T. Nguyen, Tien V. Do, and Csaba Rotter. Optimizing the resource usage of actor-based systems. *Journal of Network and Computer Applications*, 190:103143, 2021.

[P3] Hai T. Nguyen, Tien Van Do, and Csaba Rotter. Scaling upf instances in 5g/6g core with deep reinforcement learning. *IEEE Access*, 9:165892–165906, 2021.

Other papers

[B1] Yang Yuan Li, Tien Van Do, and Hai T. Nguyen. A comparison of forecasting models for the resource usage of mapreduce applications. *Neurocomputing*, 418:36–55, 2020.

[B2] Tien Van Do, N.H. Do, H.T. Nguyen, Csaba Rotter, Attila Hegyi, and Peter Hegyi. Comparison of scheduling algorithms for multiple mobile computing edge clouds. *Simulation Modelling Practice and Theory*, 93:104–118, 2019. Modeling and Simulation of Cloud Computing and Big Data.

[B3] Hai T. Nguyen and T. V. Do. A model for a computing cluster with two asynchronous servers. In Nguyen-Thinh Le, Tien van Do, Ngoc Thanh Nguyen, and Hoai An Le Thi, editors, *Advanced Computational Methods for Knowledge Engineering*, pages 197–211, Cham, 2018. Springer International Publishing.

References

[1] M. Alazab, S. Khan, S. S. R. Krishnan, Q. Pham, M. P. K. Reddy, and T. R. Gadekallu. A multidirectional lstm model for predicting the stability of a smart grid. *IEEE Access*, 8:85454–85463, 2020.

[2] Cloud Native Computing Foundation. Kubernetes. `https://kubernetes.io/`, 2021. Accessed: 2021-06-10.

[3] D. Diaz Snchez, R. Simon Sherratt, P. Arias, F. Almenarez, and A. Marn. Enabling actor model for crowd sensing and iot. In *2015 International Symposium on Consumer Electronics (ISCE)*, pages 1–2, June 2015.

[4] ETSI Group Specification. ETSI GS NFV 001 V1.1.1. Network Functions Virtualisation (NFV): Use Case, 2013, 2013.

[5] ETSI Group Specification. ETSI GS NFV 003 V1.2.1. Network Functions Virtualisation (NFV): Terminology for Main Concepts in NFV. 2014., 2014.

[6] Andreas Moregrd Haubenwaller and Konstantinos Vandikas. Computations on the edge in the internet of things. *Procedia Computer Science*, 52:29 – 34, 2015. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

[7] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks, 2017.

[8] Tania Lorido-Botran, José Miguel-Alonso, and José Antonio Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12:559–592, 2014.

[9] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529 EP –, Feb 2015.

[11] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysaw Dbiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Jzefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[12] Csaba Rotter and Tien Van Do. A queueing model for threshold-based scaling of UPF instances in 5G core. *IEEE Access*, 9:81443–81453, 2021.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[15] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.