Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Ákos Jakub

APPLICATION OF MULTI-AGENT DEEP REINFORCEMENT LEARNING TO MICROSERVICE RESOURCE MANAGEMENT

Scientific Students' Association Report 2022

Consultants:

Dr. Balázs Sonkoly

Dr. Gábor Szűcs

Balázs Fodor

Contents

Abstract

Microservice applications are software systems whose components exist separately and communicate with each other through well-defined interfaces. One of the biggest challenges when running software following a microservice architecture in an environment is the scaling of the application components, i.e. the dynamic change of resources depending on the incoming traffic. On the one hand, we don't want to assign too many resources to the application, because in a cloud environment you have to pay for the resource used, and on the other hand, we do not want low resource utilization, because this leads to the deterioration of the application's performance, which can cause the loss of users. A big challenge is how to coordinate the management of microservices, where the components also affect each other, but the resources are specified per component.

In my thesis, I offer a potential solution to this problem with multi-agent deep reinforcement learning-based algorithms, which can be used to cost-effectively manage the resource demand of large-scale microservice applications in real time. To solve the task, I use the actor-critic, off-policy, model-free artificial intelligence system developed by Open AI, the Multiagent Deep Deterministic Policy Gradient (MADDPG) algorithm. One of the central elements of the MADDPG is that the model determines the quality of the actors' decisions taking into account the current state of the entire system, which requires a complex structure. As a result, depending on the field of application, even in the case of 10-12 agents, we run into serious scaling limitations, which makes it impossible to effectively train large-scale microservice applications - sometimes consisting of a few dozens of microservices. In order to eliminate this problem, in my thesis I also present an improved algorithm in the form of the State Encoded Multi Agent Deep Deterministic Policy Gradient (SEMADDPG) algorithm, which, compared to its predecessor, is supplemented with a new architectural element in the form of an autoencoder. The new structure allows critics to have a more compact and smaller representation of their inputs, thereby ensuring that the model learns efficiently even with a large number of microservices. The task of the agents is to scale cooperatively in a simulated environment - taking into account the load from the microservices directly connected to them and managed by other agents - so that all services can function without problems, with minimal resource use. When evaluating the performance of the models, I take into account the learning time of the models, their resource management, and their robustness, for which I use the Kubernetes Horizontal Pod Autoscaling procedure as a benchmark. Both reinforcement learning algorithms manage to outperform the horizontal pod autoscaling, while the SEMADDPG approach could provide a faster converging multiagent system compared to simple MADDPG.

1. Introduction

Software applications were traditionally built as monolithic pieces of software, and adding new features requires reconfiguring and updating everything from processes and communications to security within the application. Traditional monolithic applications have long life cycles, are updated infrequently and the changes in the code usually affect the entire application. This costly and cumbersome process delays advancements and updates in enterprise application development. The microservice cluster architecture was designed to solve this problem. This design was built in order to create distributed applications which consist of smaller independent components that communicate with each other through well-defined interfaces. Each function of the application operates as an independent service. This architecture allows each service to scale or update without disrupting other services in the application. With the help of microservices, we can build a massively scalable and distributed system, which avoids the bottlenecks of a central database and improves business capabilities, such as enabling continuous delivery/deployment applications and modernizing the technology stack.

Microservice architecture has the following main attributes:

- Application is broken into modular, loosely coupled components
- Application can be distributed across clouds and data centers
- Adding new features only requires those individual microservices to be updated
- Network services must be software-defined and run as a fabric for each microservice to connect to

Despite their robustness there are various challenges when it comes to implementing and maintaining microservice systems. Drifting away from the monolithic perspective the problems of architecture planning arise in various forms, such as the management of a complex system and their mostly codependent microservices. Application performance monitoring tools that collect various measures are used to collect performance-relevant runtime data of the entire system, starting from low-level measures such as CPU and memory usage, virtualization layers and middlewares. Most of these tools collect historical data from a microservice and try to model the normal behavior. The goal of these models are to detect anomalies, such as exceptionally high response times or resource usage. Performance

relevant data in microservice architectures can be collected from the microservice inside a container, from the container, and from interrelated microservices. However, this poses several additional challenges.

- **Problem of autoscaling**: In order for most model to up- or downscale a microservice, fist have to collect data from the environment
- **Problem of normal behavior**: It is often hard to determine what behavior of the microservice can be considered normal. Most of these applications face frequent changes, since there are various updates, scaling events, virtualizations and code changes in the lifecycle of a microservice.
- **Classic auto scaling solutions lack robustness**: Existing techniques for performance anomaly detection may therefore raise many false alarms. Most of the classical algorithms struggle to deal with sudden resource spikes.
- **Single focus only**: Many autoscalers can scale only one service and therefore do not take into account the dependencies between services. This can lead to a locally optimal, but globally suboptimal scaling solution for the whole microservice.

Despite these problems, there are applications for design-time performance modeling emerging in new areas, such as reliability and resilience engineering and the design of runtime adaptation strategies. Some new applications require changes to the scaling algorithms themselves. This means that new ways of representing the data are needed to capture the recent advances in technology. For example, in settings where there are lots of machines that are constantly being added or removed (like in Amazon Lambda), traditional models based on the notion of distinct (virtual) machines are inadequate. New modeling strategies must be found to accurately represent such structures and the behavior of the automated infrastructure components. Another question arises from the size of the models, as large microservice installations may consist of tens of thousands of service instances. We expect that it will be necessary to reduce the level of detail to be able to cope with such models in a timely fashion. To tackle these problems a variety of machine learning approaches can be useful to experiment on. During my thesis I researched various reinforcement learning based scaling solutions. I have developed a C++ based framework that enables us to model simple and complex microservice environments down to request handling mechanics, and also implemented two multiagent deep reinforcement learning algorithms to manage microservice resources, namely the MADDPG algorithm originally developed by OpenAI and a slightly extended variation of the algorithm, the SEMADDPG.

In my thesis, after a review of reinforcement learning and multiagent solutions in Section 2, I analyze several such machine learning approaches for resource management and study their strengths and weaknesses in Section 3. I present the environment I have developed, its properties and constraints in Section 4. In Section 5, I detail the implemented models, as well as their advantages and disadvantages. In Section 5.2, I present and analyze the SEMADDPG algorithm, which is my structurally expanded version of the previous multi-agent deep learning approach. Finally in Section 6 I compare the approaches above.

2. Reinforcement learning

2.1. Basics

Situated in between supervised learning and unsupervised learning, the paradigm of reinforcement learning deals with learning in sequential decision making problems in which there is limited feedback. Reinforcement learning differs from supervised learning in not needing labeled input/output pairs to be presented, and in not needing sub-optimal actions to be explicitly corrected. For an RL algorithm the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). Reinforcement learning is the closest thing to real life learning in terms of continuous exploration and exploitation of an unknown environment, since most RL algorithms do not assume knowledge of an exact mathematical model of the Markov Decision Process (MDP) relating to the environment. Also there are problems, where the target MDPs are so complex that exact methods with basic dynamic programming become infeasible.

The Markov Decision Process provides a mathematical framework for solving the RL problem, since almost all RL problems can be modeled as an MDP. All states in an MDP that come in between an initial-state and a terminal-state. The agent's goal is to maximize the total reward it receives during an episode which starts at the initial state and ends at the terminal state. There are two main concepts relating to Markov decision processes.

2.1.1. Markov property

The Markov process consists of a sequence of states that strictly obey the Markov property, which states that the future depends only on the present and not on the past. When an RL problem satisfies the Markov property, i.e., the future depends only on the current state and an action, but not on the past, it is formulated as a Markov Decision Process (MDP).

The goal of a reinforcement learning model (from now on: agent) is to find a sequence of actions that will maximize the sum of future rewards. We have to put emphasis on the fact that the further the future reward is for the agent, the less it has to take into account, since it seems an unsure outcome from the current state. For this reason we use discounting during an episode or the entire life of the agent, depending on the task.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{j=0}^{T} \gamma^j r_{t+j+1}$$

Equation 2.1: Sum of all future rewards denoted as G

In Equation 2.1 r_{t+1} represents the reward at time t+1 while γ r_{t+2} represents the discounted reward at the next timestep. Even though the sum of the future rewards can be infinite (if we are talking about a continuous problem) according to the equation above, if γ<1, then G will have a finite value. If γ=0, the agent is only interested in the immediate reward and discards the long-term return. Conversely, if γ=1, the agent will consider all future rewards equal to the immediate reward. In short, the Agent must be able to exploit this information that we have been able to express with this return G to make their decisions. We refer to the above expression as a discounted return.

2.1.2. V function

Generally speaking, the V-function answers the question of "What reward can the agent expect from here?". More formally, the V-function, also referred to as the state-value function, measures the goodness of each state. In other words, how good or bad it is to be in a particular state according to the return G when following a policy π. That is, we can define the V-function as an expected total reward (discounted or undiscounted — depending on the value of gamma) that is obtainable from the state. In a formal way, the value of $V_\pi(s)$ is:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s = s_t] = \mathbb{E}_\pi\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} \,|s = s_t\right]$$

Equation 2.2: Value function of a policy given a state expressed with discounted rewards

Equation 2.2 describes the expected value of the total return G, at time step t starting from the state s at time t and then following policy π. It is used expectation $\mathbb{E}[.]$ in this definition because the environment transition function might act in a stochastic way.

2.1.3. Q value

Q-function or simply Q defines a value for each state-action pair, which is called the action-value function. It represents the value of taking action a in state s under a policy π,

denoted by $Q_{\pi}(s, a)$, as the expected Return G starting from s, taking the action a, and thereafter following policy π

$$Q_{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|S_t = s, A_t = a] = \mathbb{E}_{\pi}\left[\sum_{j=0}^{T} \gamma^j r_{t+j+1} | S_t = s, A_t = a\right]$$

Equation 2.3: Q-value function of a policy given a state, and an action expressed with discounted rewards

In Equation 2.3 expected value $\mathbb{E}[.]$ is used again because the environment transition function might act in a stochastic way. There is also a strict relationship between the Q value and the V value. We denote with $\pi(a|s)$ the probability that a policy, π, selects an action, a, given a current state, s. Note that the sum of probabilities of all outbound actions from s is equal to 1 as it can be seen in Equation 3.4:

$$\sum_{a} \pi(a|s) = 1$$

Equation 2.4: Sum of action probabilities over a state

We can assert that the state-value function is equivalent to the sum of the action-value functions of all outgoing (from s) actions a, multiplied by the policy probability of selecting each action:

$$V_{\pi}(s) = \sum_{a} \pi(a|s) \cdot Q_{\pi}(s,a)$$

Equation 2.5: Value function expressed with Q value

What Equation 2.5 essentially represents is that we can write the value function of a policy in state s if we weight all possible actions in state s (according to the policy) with the Q values of that policy in state s with respect to the action.

2.1.4. Bellman equation

Almost all Reinforcement Learning algorithms executed by the Agents involve estimating value functions of states or of state-action pairs. These are the so-called Value-based Agents. The Bellman equation simplifies the computation of the value function, such that rather than summing over multiple time steps, we can find the optimal solution of a complex problem by breaking it down into simpler, recursive subproblems and finding their optimal solutions.

2.1.5. Bellman equation of optimality

Bellman proved that the optimal state value function in a state s (denoted by $V_*(s)$) is equal to the action a, which gives us the maximum possible expected immediate reward, plus the discounted long-term reward for the next state s', as it is denoted in Equation 2.6

$$V_*(s) = \max_a \sum_{s'} P^a_{ss'}(r(s,a) + \gamma V_*(s'))$$

Equation 2.6: Bellman optimality equation for V value

Bellman also proved that the optimal state-action value function in state s and taking action a is:

$$Q_*(s,a) = \sum_{s'} P^a_{ss'}(r(s,a) + \gamma \max_{a'} Q_*(s',a'))$$

Equation 2.7: Bellman optimality equation for Q value

The Bellman equation is a keystone to find the optimal values of the value functions to obtain an optimal policy for any type of agent.

2.2. Q learning

Based on the Bellman equation, Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the Q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. Q-learning seeks to learn a policy that maximizes the total reward.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy.

"Q" in this setting refers to the function that the algorithm computes – the expected rewards for an action taken in a given state. There are various solutions regarding this type of reinforcement learning, like tabular methods, such as Q-table or function estimation solutions, such as Deep Q learning. The main task of these types of algorithms is to estimate a proper Q value for each state action pair, and it works quite well for finite action spaces

with finite state spaces, and sometimes continuous action spaces can as well, especially with Deep Q learning.

The main drawback of Q-learning is that the learning process is expensive for the agent, mainly in the beginning steps. Because, every state-action pair should be visited frequently in order to converge to the optimal policy. If this part of the training process is missing, we might arrive at a suboptimal/partially optimal solution to the problem. The other main drawback is that it is very hard to apply classic Q learning to continuous action spaces.

2.3. Policy Gradient Methods

Policy gradient methods are a type of reinforcement learning techniques that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. They do not suffer from many of the problems that have been present in traditional reinforcement learning approaches such as the lack of guarantees of a value function, the intractability problem resulting from uncertain state information and the complexity arising from continuous states & actions.

Most traditional reinforcement learning methods have no convergence guarantees and there exist even divergence examples. Continuous states and actions in high dimensional spaces cannot be treated by most RL approaches.

Policy gradient methods differ significantly as they do not suffer from these problems. For example, uncertainty in the state might degrade the performance of the policy (if no additional state estimator is being used) but the optimization techniques for the policy do not need to be changed. Continuous states and actions can be dealt with in exactly the same way as discrete ones while, in addition, the learning performance is often increased. Convergence at least to a local optimum is guaranteed.

The advantages of policy gradient methods for real world applications are numerous. Among the most important ones are that the policy representations can be chosen so that it is meaningful for the task and can incorporate domain knowledge, that often fewer parameters are needed in the learning process than in value-function based approaches and that there is a variety of different algorithms for policy gradient estimation in the literature which have a rather strong theoretical underpinning. Additionally, policy gradient methods can be used either model-free or model-based as they are a generic formulation of solving the Bellman equation. Despite their usefulness and elegance, policy gradient methods also have significant

problems. They are by definition on-policy and need to forget data very fast in order to avoid the introduction of a bias to the gradient estimator. Hence, the use of sampled data is not very efficient. In tabular methods such as Q-table, value function methods are guaranteed to converge to a global maximum while policy gradients only converge to a local maximum and there may be many maxima in discrete problems. Policy gradient methods are often quite demanding to apply, mainly because one has to have considerable knowledge about the system one wants to control to make reasonable policy definitions. Finally, policy gradient methods always have an open parameter, the learning rate, which may decide over the order of magnitude of the speed of convergence; these facts have led to new approaches inspired by expectation-maximization approaches which later became known as actor critic solutions.

2.4. Actor critic methods

There are various algorithms that can tackle the challenges of resource management, but reinforcement learning approaches prove to be pioneers in the field of controlling dynamic systems. In the last few years, the development achieved in the field of neural networks has helped the field to flourish, as many algorithms based on deep reinforcement learning have been developed, the structure and complexity of which is close to human real-time decision-making [11], but in some cases it exceeds it by a lot. Considering the problems of resource management, an algorithm is needed that can handle a large complex state space and also a large and continuous action space. It is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces. Timothy P. Lillicrap et al. [6] presented an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. This model-free approach is called Deep Deterministic Policy Gradient (DDPG). A key feature of their approach is its simplicity: it requires only a straightforward actor-critic architecture and learning algorithm with very few “moving parts”, making it easy to implement and scale to more difficult problems and larger networks. Interestingly, DDPG can sometimes find policies that exceed the performance of the planner, in some cases even when learning from low dimensional state representations.

The DDPG algorithm maintains a parameterized actor function $\mu(s|\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action. The critic Q(s, a) is

learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to the expected return from the start distribution J with respect to the actor parameters as it can be seen in the Equation 2.8.

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t} \right]\end{aligned}$$

Equation 2.8: Gradient update of the actors

Silver et al. [6] proved that this is the policy gradient, the gradient of the policy's performance. As with Q learning, introducing non-linear function approximators means that convergence is no longer guaranteed. However, such approximators appear essential in order to learn and generalize on large state spaces. NFQCA [7] which uses the same update rules as deterministic policy gradients (DPG) but with neural network function approximators, uses batch learning for stability, which is intractable for large networks. A minibatch version of NFQCA which does not reset the policy at each update would be required to scale to large networks. Lillicrap's contribution here is to provide modifications to the vanilla DPG, inspired by the success of DQN, which allow it to use neural network function approximators to learn in large state and action spaces online. One challenge when using neural networks for reinforcement learning is that most optimization algorithms assume that the samples are independently and identically distributed. Obviously, when the samples are generated from exploring sequentially in an environment this assumption no longer holds. Additionally, to make efficient use of hardware optimizations, it is essential to learn in mini batches, rather than online. As in DQN, they used a replay buffer to address these issues. The replay buffer is a finite sized cache. Transitions are sampled from the environment according to the exploration policy and the current state, current action, next state, and reward were stored in the replay buffer. When the replay buffer is full the oldest samples are discarded. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. Because DDPG is an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.

Another large problem present in Q learning was the divergence of efficient actions. Directly implementing Q learning with neural networks proved to be unstable in many environments. Since the network $Q(s, a|\theta^Q)$ being updated is also used in calculating the target value, the Q

update is prone to divergence. Their solution was to use ”soft” target updates, rather than directly copying the weights. They created a copy of the actor and critic networks that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks, as denoted by Equation 2.9.

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

Equation 2.9: Target network update for critic and actor networks

This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist. This may slow learning, since the target network delays the propagation of value estimations. However, in practice their findings suggested that this was greatly outweighed by the stability of learning [6].

2.5. Multiagent systems

Multiagent Reinforcement Learning (MARL) algorithms are designed to deal with systems consisting of multiple agents who are interacting with each other within a shared environment. Each agent makes a decision in each timestep, working towards an individual predetermined goal, while also trying to help the other agents achieve their goals. The goal of MARL algorithms is to learn a policy for each agent such that all of the agents working together can achieve the goal of the system. The agents are learnable units that aim to learn an optimal policy on the fly in order to maximize the long-term cumulative discounted reward from interacting with the environment. However, due to the complexities of the environments or the combinatorial nature of the problem, training the agents is typically a challenging task. MARL deals with several problems that are categorized as NP-Hard, such as manufacturing scheduling, the vehicle routing problem, and some multi-agent games [12].
A multiagent system is often exceedingly complicated, making preprogramming the system unfeasible for practical reasons. Additionally, learning and adaptation are necessary since the dynamics of the agents and their surroundings are subject to change over time. It was acknowledged in the early work on MARL for stochastic games that no agent operates in a vacuum. The learning agent may come across completely competitive, zero-sum games,

among other scenarios and game types. There are also general-sum cooperative games, in which players work together to maximize their rewards. In a multiagent system, the agents typically need to monitor the actions of the other agents so that a coherent behavior emerges. We also need to think about the scalability issue. The agents have a huge number of potential states and coordinated activities to keep track of. It is possible to formalize learning in stochastic games as a MARL issue. At the present state, agents simultaneously choose actions, and at the following state, they are rewarded. The goal of a multiagent reinforcement learning algorithm is to learn equilibrium strategies through interaction with the environment and also other agents. Generally, in a MARL problem, agents may not know the transition function or the reward function from the environment. Instead, agents are required to select actions and observe the received reward and the next state in order to gain information of the transition function or the reward function. Rationality and convergence are two desirable properties for multiagent learning algorithms in stochastic games. When we say that an agent is rational, we mean that, if the other agent's policies converge to stationary policies, then the learning algorithm will converge to a policy that is the best response to the other agent's policies. In the stochastic learning algorithms, we also have the idea of convergence: if all the agents use rational learning algorithms and their policies converge, then they must have converged to an equilibrium. Each agent will make the best response to all other agents [8].

3. Review of possible solutions

3.1. Related work

Quality of Service violation is a very important topic when it comes to developing and deploying microservice applications. Many algorithms exist for the sole purpose of managing microservice instances and/or to suggest resource allocation for the services. Autoscaler solutions are mostly categorized by their underlying theoretical models [15], [16]. Such solutions can be:

- **Threshold-based**: these solutions use a set of predefined rules to scale microservices. Such an algorithm was proposed by Khazaei [17], where he implemented auto-scalability and monitoring-as-a-service for any type of cloud software system, making scaling decisions based on an adjustable linear combination of CPU, memory, and network utilization.
- **Queuing model-based**: these scalers use queueing models, which can determine the number of servers based on the length of the queue. A good example for such an approach was created by Kaboudan [18]. He presented a discrete-time queuing model with a dynamic number of servers using a threshold-based scaling policy.
- **Control theory-based**: the goal of the controller is to align resources to the services in a way that makes the output approximate a reference value. Ali-Eldin implemented such an algorithm based on control theory [19], which was used to design a reactive-adaptive proportional controller that acted based on the load dynamics, and could respond to sudden changes in it.
- **Reinforcement learning-based**: using machine learning algorithms to control dynamic systems in real time. A good example for such a solution was developed by Haoran Qiu and Subho S. Banerjee [3], where they used an ensemble technique in order to categorize and eliminate service level objective violations.
- **Performance prediction-based**: various regression oriented models can be used to predict the response time or the latency of the application. Wajahat et al. [24] proposed such a neural network (NN)-, and regression-based application agnostic auto scaling solution [20].
- **Demand forecast-based**: first predict the possible demand for specific services, and after that scale the resources accordingly. A good example would be Chen's dynamic

server provisioning technique [21] which was able to minimize the energy consumption of data centers

Due to the complexity of the issue most of the researches use machine learning approaches. Such was the work of Yanqi Zhang and Weizhe Hua [1]. Their system the Sinan was one of the first complex resource allocation frameworks to introduce ensemble learning approaches. The system simultaneously used convolutional neural networks to extract latent variables and to predict the end-to-end latency of each microservice, while an XGBoost algorithm used the latent variables and the resource allocation and tried to predict the possibility of QoS violation. While this approach is way more robust than classic algorithms such as Horizontal Pod Autoscaling (HPA), being a static model it requires retraining over time, and some background knowledge about the nature of the microservice cluster to scale, which are common challenges when implementing scaling algorithms. The current methods for performance modeling or heuristic-based approaches are not effective because they do not take into account the dynamically changing status of the system. The interactions between microservices can also be complex and cause cascading effects when provisioning resources for interrelated microservices. This approach would potentially consume a tremendous amount of resources. Although one workflow has a fixed microservice set and microservice composition structure, the processing time of each microservice is not fixed, due to variant sizes of input data. Some simple policies, e.g., Earliest Deadline First, might mitigate this unpredictability, but such solutions are not able to adapt quickly to vast condition changes, and thus perform poorly on variant environment conditions which is often the real-world scenario. They also require a lot of knowledge and effort to implement and validate. However, reinforcement learning (RL) is a good option for learning how to provide resources because it does not rely on inaccurate assumptions and can directly learn from the workload and operating conditions online. The resource adaptation algorithm needs to consider operation cost and not use more consumers than what is allocated to them. It also requires training data obtained from giving the microservice workflow system control inputs and receiving feedback in order to profile the microservice system, but such an algorithm has access to a limited number of such interactions.

Due to the sequential nature of the decision-making process, RL is especially well-suited for resource management problems and it has been shown that deep neural networks can express the complex dynamics and decision-making policies in such a system-application

environment. The FIRM framework developed by Haoran Qiu and Subho S. Banerjee [3] was a much advanced approach to efficiently tackle most of the problems presented above with an advanced RL approach. They also used ensemble technique in order to categorize and eliminate service level objective violations. The first part of the structure was a support vector machine that identifies critical paths in the microservice dependency graphs, while an actor-critic reinforcement learning approach, named Deep Deterministic Policy Gradient (DDPG) is responsible for the reprovisioning decisions regarding resource utilization and scaling the number of instance replicas to optimize the workload. While this approach is more robust than a pure supervised learning solution, it still heavily relies on the fact that the SVM has to have some form of prior knowledge on the microservice system itself, in the form of a service dependency graph. Creating such a graph might be sustainable for a smaller microservice system, but for a full scale industry level application, structuring such a graph might be infeasible - also we would have to train a supervised model to identify the Service Level Objective (SLO) violation points in the whole system.

Despite the question of scalability, the FIRM system efficiently established that the use of reinforcement learning to solve dynamic problems might be the best course of action for such a task. Using an actor-critic model free model such as DDPG, FIRM can estimate and control a fine-grained set of resources, including CPU time, memory bandwidth and network bandwidth. Since it is a high complexity deep reinforcement solution, it can deal with large, and complex state spaces, and equally large and complex continuous action spaces, therefore it has the capacity to utilize resources on a granular level, further elevating the possibilities of creating an optimal management policy.

- Model-free RL does not need the ergodic distribution of states or the environment dynamics (i.e., transitions between states), which are difficult to model precisely. When microservices are updated, the simulations of state transitions used in model-based RL are no longer valid.
- The Actor-critic framework combines policy-based and value-based methods, and that is suitable for continuous stochastic environments, converges faster, and has lower variance.

Despite that, model free approaches also have some shortcomings when it comes to the real life applicability. DDPG as a temporal difference method requires large amounts of episodes

to train efficiently, and therefore their use is heavily bound to the available training data for a microservice. In a simulated environment this is not a challenge, but if we do not possess the necessary historical request loads about our microservice, the pure online training on incoming data might take a long time.

A possible good approach for the problem might be to incorporate model based techniques, as did Zhe Yang, Phuong Nguyen and Klara Nahrstedt [4] when they proposed a model based variant of the DDPG, realizing the limitations of model free approaches. Their framework, the MIRAS, combined two main concepts: environment model learning and policy optimization. In model learning a network tries to capture the behavior of the microservice infrastructure using sample collected from real interaction between RL agent and the environment, and in policy optimization another network tries to learn on both real world samples and samples provided by the environment modeling network. Throughout the learning the algorithm alternates between these two approaches.

Such a technique provides a good solution for the lack of available information regarding our microservice system, yet a single RL agent still lacks the robustness that is required for a system with few dozens of microservices. The more parts we have in the system, the higher the complexity of the action space becomes. While neural networks can deal with such problems, the increase in the number of possible actions makes the training process slow, and/or it would require a large neural network model. A good heuristic might be to train separate models on each microservice, but it brings even deeper challenges. A serverless FaaS platform is multi-tenant where heterogeneous functions from all customers compete for shared resources in a cluster. Multi-tenancy makes the environment non-stationary from each agent's own perspective, as it is also affected by the actions of other agents, which breaks the standard assumption that underpins single agent reinforcement learning algorithms. Since the transitions and rewards depend on the joint actions of all agents, whose decision policies keep changing in the learning process, each agent can enter an endless cycle of adapting to other agents in the shared environment.

To tackle this problem Haoran Qiu, Weichao Mao and Chen Wang proposed a multiagent proximal policy optimization method [5]. Compared to classic single agent Proximal Policy Optimization (PPO), in the system each model treats the other models like they are also part

of the environment. This implementation is agnostic to the order or the size of the agent group, and therefore provides a feasible solution for microservice resource management.

3.2. My approach

Based on the approaches found during the literature research, I came to the conclusion that I must solve the problem of resource management with deep reinforcement learning, and I must choose a model that tries to estimate the resource needs of individual microservices as efficiently as possible in a model-free approach. Given that we want to find an optimal solution for the entire microservice system on a global level, I found the multiagent approach inevitable. Most of the previous approaches in the literature used information collected from other existing microservices in some form, but this type of training has both advantages and disadvantages. On the one hand, it is beneficial to be able to model real systems by using real measurement data, but if we only rely on the use of external data, it limits the measurement of the real capabilities of the implemented models, so in my thesis I present a framework that simultaneously supports existing microservices and integration of new stochastic heuristic microservice models into a reinforcement learning environment and conversion into a Markov decision process. Despite the fact that the environment can receive external data, during my thesis I only trained models on simulated microservice systems, because I did not have such data at my disposal.

As a reinforcement learning algorithm, I chose the MADDPG model developed by OpenAI [14], because it is a robust model in which the training is centralized, while the production level execution is decentralized. This can be practical from a production point of view, because only the relatively smaller actors need to be deployed at the end of the training for the purpose of microservice management. Unfortunately, the model has a strong shortcoming, which can be seen in scaling limitations. Given that all critics belonging to each actor must simultaneously see and take into account the entire environment as all states belonging to all actors, the introduction of new actors not only increases the size of the entire model drastically, but also the time for individual models to converge increases. In order to eliminate this problem, I implemented a structural modification in the MADDPG algorithm, with the help of which I can increase the speed of the convergence of the model and reduce the complexity of critic networks.

4. Modeling the microservice execution environment

4.1. The environment

Given the complexity of both the task and the approaches to be implemented, I have developed a modeling environment that can also be used to model:

- the properties of the microservices in a cluster
- the relationships between the services
- the resource usage of individual microservices
- the load of the microservices on each other
- external requests

The MicroserviceGym library allows us to model any clusters of microservices. Structurally, the environment can be represented as a directed acyclic graph. The nodes in the graph represent each microservice (later in the documentation, nodes and microservices are used interchangeably, denoting the same concept), while the edges between them are intended to show the relationship between said microservices. A microservice has the ability to send requests to another microservice, providing an extra load for that node. From a hierarchical point of view, this is how we distinguish three types of nodes in the system:

- Chief node: In the case of a chief node, we are talking about a service that has only workers, since it is not used by another service during its operation; such services are only and exclusively get requests from outside the system.
- Worker node: Worker nodes are located at the bottom of the system usage hierarchy graph, they do not use any other services, but they also serve one or more other services.
- Basic microservice: every other microservice that is not a chief or a worker node, can be considered a simple microservice that can send and receive requests from other microservices.

4.1.1. Modeling details

Connections may occur between each node if one microservice uses the resources of another. The weights of each edge are represented with linear regression: the value of the stress currently on the chief node will be passed to the workers with a weighting of as it can be seen in Equation 4.1.

$$m \cdot x + b : m \sim U(0.75, 1.2), b \sim N(loc, scale)$$

Equation 4.1: Simplified notation of load computing on an edge

While x is the number of requests sent from one node to another, m is a random number sample from a uniform distribution between 0.75 and 1.2 and b is a random number that is sampled from a normal distribution which parameters can be set. This implementation is advantageous from the point of view that it is possible for a system to model a certain level of stochasticity. From now on the load from microservice x to microservice y in iteration i will be denoted with $\lambda^{i}_{x,y}$ such as

$$\lambda^{i}_{x,y} = m \cdot \lambda^{i}_{y} + b^{i} : m \sim U(0.75, 1.2), b^{i} \sim N(loc, scale)$$

Equation 4.2: Load from microservice x to microservice y in the i'th sequence

b^{i} represents a random value sampled from a normal distribution in the i'th iteration, while λ^{i}_{y} denotes the full load on node y in the i'th iteration as it can be seen in Equation 4.2. To get its value let's denote the chief nodes of microservice y with Γ. That means that all nodes in gamma propagate request loads toward node y, as presented in Equation 4.3.

$$\lambda^{i}_{y} = \sum_{\forall \gamma \in \Gamma} \lambda^{i}_{\gamma,y}$$

Equation 4.3: Full load on node y in the i'th sequence

To further elevate the environment close to life in terms of complexity the instance handling also has a stochastic behavior. Each service node has an instance container for horizontal scalability. These instances represent at a low level the resources behind a given microservice: each instance has an instance capacity that shows how many requests that instance can handle. Considering this, and the number of instances, we can determine how many requests we can handle at a given time for a given microservice. Assuming that we cannot increase the number of these instances indefinitely in a real environment, each node has a single maximum instance number. Of course, with the horizontal increase in the number of individual instances, the overall request handling capabilities of said microservices scale non-linearly in real life, so I introduce a stochastic polynomial term to artificially modify the performance of the microservice. This kind of randomness between microservices elevates the environment closer to a life situation in terms of complexity, however this solution does not perfectly match with a real life architecture. The request handling capacity of instances,

denoted by ω_y^i also varies stochastically for each different microservice and it can be computed as in Equation 4.4.

$$\omega_y^i = (util^i \cdot \mu_y \cdot capacity_y)^{U(0.8,1)_y}$$

Equation 4.4: Request handling capacity of microservice y in the i'th sequence

The request handling capacity of microservice y in the i'th sequence will be calculated by the actual utilization percentage in the i'th sequence, multiplied by the maximum available microservice number, and the instance capacity on service y. This whole term is taken to the power of a random number sampled from a uniform distribution between 0.8 and 1 specific for the microservice y.

The resource usage of microservice y at i'th iteration can be described as the fraction of the incoming loads and the request handling capacity, as presented in Equation 4.5.

$$usage_y^i = \frac{\lambda_y^i}{\omega_y^i}$$

Equation 4.5: Usage percentage of microservice y in i'th sequence

The usage of the nodes will be the key metric that determines how stable the system will be. Between 0 and 1, the microservice has more resources than it needs to serve incoming requests, in which case usage shows how much of the resources the microservice uses. If it is greater than 1, the application is overloaded and QoS violation might happen.

The task of the agents was to estimate the number of instances belonging to each microservice. Reflecting on the fact that even in real life we are not able to scale resources to infinity, a maximum instance number is defined for each microservice, which is automatically calculated by the environment before the agents are taught: with the predefined maximum load, the system runs 100 test episode, with 300 steps per episode and examines the maximum loads that can come to each microservice. Taking into account the request handling capacity of each node, the system retrospectively calculates the amount of instances that would be needed to fully serve all incoming requests under maximum load. It is important to emphasize that with this approach we also assume that the load of our system has/may have a maximum upper limit, above which no more requests are received. Although resource scaling in real life is a never-ending dynamic system control problem, on the contrary, during

teaching, the number of steps during one episode had to be maximized. An episode consists of a fixed number of steps and ends when the maximum number of steps is reached or if a QoS violation occurs in the case of any microservice within the system. A step represents a time resolution with arbitrary granularity, but it reflects a state where all requests received in the microservice system have been propagated through all the codependent microservices. During training, each agent receives a reward depending on how well they were able to guess a usage of the microservice in the i'th iteration.

4.1.2. Implementation

The logical part of the library's code is written entirely in C++ in order to train the agents managing microservices as quickly as possible, and during the process the vast majority of the time is spent on backpropagation of the neural network. In order to be able to interact smoothly with the developed system, and so that Python-specific libraries can also use the system, I used a library called Cython which is an optimizing static compiler for both the Python programming language and the extended Cython programming language. With this tool I created a library that has the speed of a pure C++ program with capability to interact with it through Python.

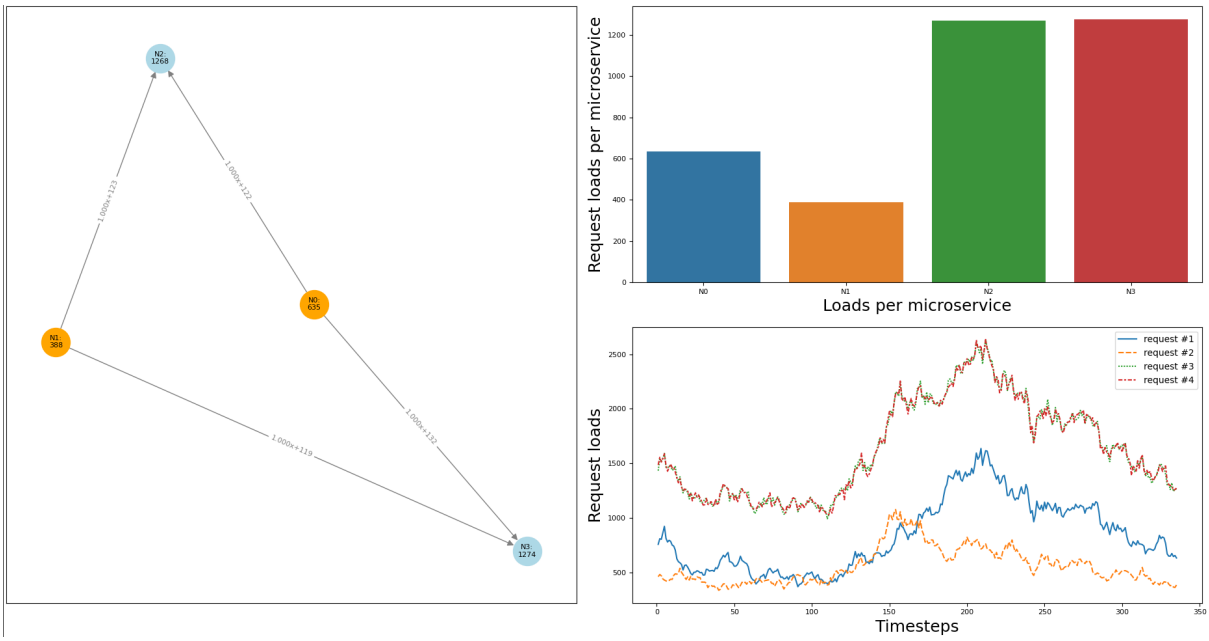

Figure 4.1: Usage percentage plot of microservice y in i'th sequence

Additional feature of this solution is that other Python packages can be used in Cython. Leveraging this I also created a monitoring dashboard for evaluation purposes as it can be seen in Figure 4.1 using Seaborn and Matplotlib packages.

4.2. Training

Throughout the training process I tried to present various levels of environment complexity for the models to learn on. The difference between each environment complexity level was presented both in the number of microservices within the environment and in the values of the noise factors and multipliers found on the edges for the microservices. During the running simulations 3 levels were separated, in the form of small, medium and complex microservice systems, which had the following properties:

- **Small environment**: 4 microservices, strictly linear connections with small noise between the nodes, linear request handling on each node, the states are the previous incoming global request values
- **Medium environment**: 6 microservices, simple linear or 0.5 multiplicated edges between nodes with medium noise, polinomial request handling, the states are the previous incoming request values from the chief node of each microservice
- **Complex environmen**t: 15 microservices, random linear multiplication on each edge between nodes, polinomial request handling, the states are the previous incoming request values from the chief node of each microservice

As I highlighted earlier, both models were evaluated at these 3 levels of complexity. At all three levels, during each episode, the models could take a maximum of 2000 steps and during these steps the models must collect as much reward as possible. In each case the training lasted for 10,000 episodes. At the beginning of each episode, the loads of chief nodes changed in each case following a random Brownian motion. At the end of each training episode, an evaluation episode was also run, which serves the purpose of determining the change in the performance in the set of actors.

The goal of the artificial intelligence approaches are to manage the resources of the microservices by adding or removing instances so that it approaches a predefined utilization level, similar to the operation of the HPA algorithm [13]. HPA helps provide seamless service by dynamically scaling up and down the number of resource units, called pods, without having to restart the whole system. It supports high availability by adjusting the number of execution and resource units, known as pods (the terms pod and instance are interchangeable throughout the thesis), based on various requirements. When triggered, HPA

creates new pods to share the workloads without affecting the existing ones currently running inside the cluster, however this process takes time and while it happens performance degradation and QoS violations may occur.

An important difference compared to previous solutions is that, since there is only one learning approach for both models, which does not contain a separate QoS violation detection structure, during the training, efforts must be made to prevent such a case from occurring. I was able to force this behavior out of the models by simply disallowing them to violate QoS, meaning that if any microservice overloaded in any system, the episode would end. The idea would give itself, that ideally we would like our resources assigned to microservice to always be at 100% utilization, because this is not the situation where a QoS violation occurs, but resource management is the most economically efficient at this point. It is important in what direction we are willing to be permissive during the training: if avoiding overutilization is a higher priority, we should be more permissive with QoS violations, if the loss of our customers is a bigger threat to us than the potential additional costs resulting from overutilization, then more emphasis should be placed for the stability of the system. During the training of the models, more emphasis is placed on the latter aspect, because for providers operating larger microservice systems such as Amazon Prime or Netflix the primary value is the user. There are many solutions for mitigating the additional technical costs that such a company can live with, and hard churn of users is a problem that can directly and drastically affect financial security in the long term. Due to this consideration, I preferred maintaining the stability of the entire system during the creation of the reward function, where the reward is based on the usage percentage of specific microservices regarding the utilized instances, as can be seen in equation 4.6, where x is the usage percentage of the microservice.

$$g(x) = \begin{cases} \log_2(x + 1.1) & : x < 0.9 \\ \dfrac{1}{5\,x - 4} - 1 & : \text{otherwise} \end{cases}$$

Equation 4.6: Reward function

Given that we are talking about a highly stochastic environment, where not only the number of incoming requests but also the relationships between microservices can change, a safe approach for the model is to choose either the suboptimal solution or end the episode quickly, thus eliminating any penalties that may arise later. I designed the function above, such that

the custom reward function gives the biggest possible reward at 90% utilization which happens to be 1. As we get closer to a possible quality drop, the reward becomes miniscule in order to prevent the end of the episode, as presented in figure 4.2. I choose 90% utilization since it is relatively close to perfect instance utilization, however it leave some wiggle room for the RL algorithms to not automatically end the episode if there would be a smaller spike in the incoming load.

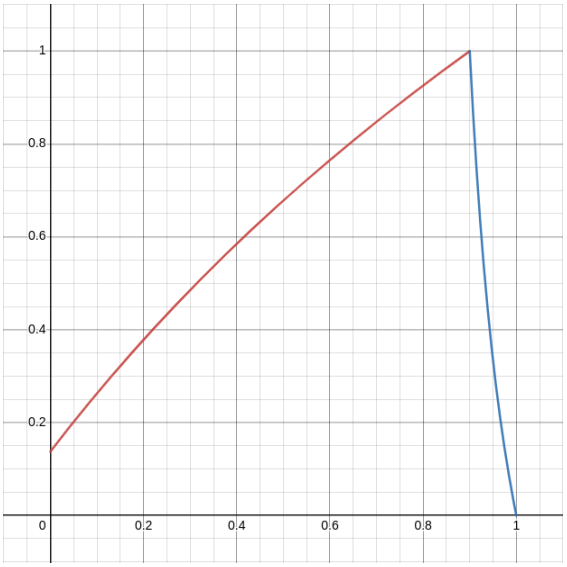

Figure 4.2: Reward function visualization

5. Proposed RL models

5.1. MADDPG

The simplest approach to learning in multi-agent settings is to use independently learning agents. This was attempted with Q-learning, but it does not perform well in practice. Independently-learning policy gradient methods also perform poorly. One issue is that each agent's policy changes during training, resulting in a non-stationary environment and preventing the naive application of experience replay. Previous work has attempted to address this by inputting other agent's policy parameters to the Q function in the work of Tesauro [9], explicitly adding the iteration index to the replay buffer, or using importance sampling. The nature of interaction between agents can either be cooperative, competitive, or both and many algorithms are designed only for a particular nature of interaction. The goal of the OpenAI-s team [14] was to build a model that works in all of these cases. However, they provided additional constraints: the learned policies can only use local information at execution time, no differentiable model of the environment dynamics, no particular structure on the communication method between agents (we don't assume differentiable communication).

Algorithm 1 Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode = 1 to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for t = 1 to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + N_t$ w.r.t the current policy and exploration
    Execute actions $a = (a_1, ..., a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, ..., a'_N)|_{a'_k = \mu'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum (y^i - Q_i^{\mu}(\mathbf{x}^j, a_1^j, ..., a_N^j))^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, ..., a_i, ..., a_N^j)_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \rightarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Algorithm 5.1: MADDPG algorithm

Fulfilling the above constraints they created a general-purpose multi-agent learning algorithm that could be applied not just to cooperative games with explicit communication channels, but competitive games. The canonical form of the original algorithm can be read as in Algorithm 5.1. A new approach was also presented in terms of centralized training with decentralized execution. For MADDPG the policies use extra information to ease training, so long as this information is not used at test time. It is unnatural to do this with Q-learning, as the Q function generally cannot contain different information at training and test time. Here the critic is augmented with extra information about the policies of other agents.

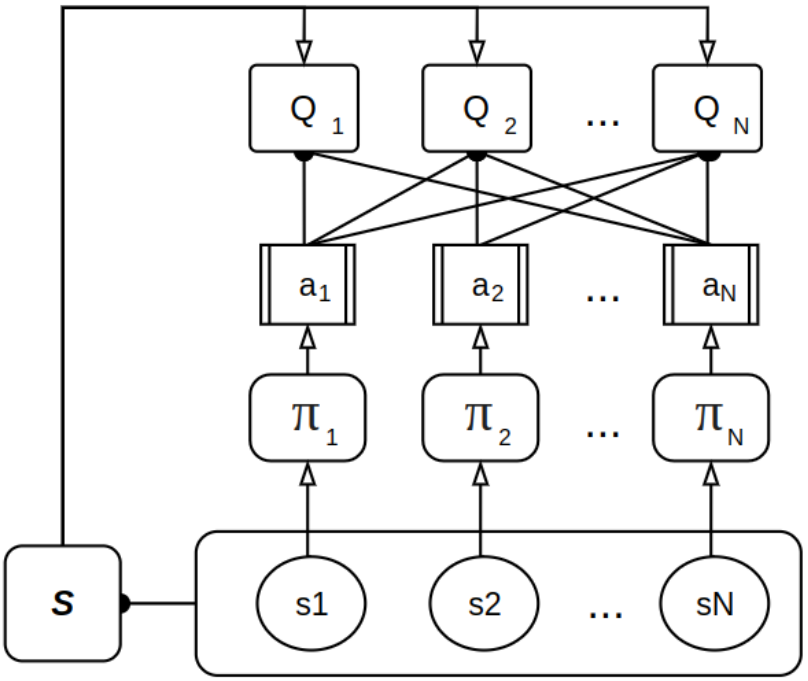

Figure 5.1: MADDPG model structure

Here, as it can be seen on figure 4.1 $Q(\pi)_i(x, a_1, ..., a_N)$ is a centralized action-value function that takes as input the actions of all agents, $a_1, ..., a_N$, in addition to some state information x, and outputs the Q-value for agent i. In the simplest case, x could consist of the observations of all agents, $x = (o_1, ..., o_N)$, but we could also include additional state information if available - the former approach was used for my MADDPG implementation. Since each $Q(\pi)_i$ is learned separately, agents can have arbitrary reward structures, including conflicting rewards in a competitive setting. Each actor takes into account its specific state and tries to make an action based on that, however every critic takes into account the actions of all actors, and all the states, and tries to evaluate one actor based on the whole system. The

separation of critics per actor is necessary, since if we had used only one critic, we would trade the size of the whole network to complexity of Q value approximation for all of the actors. The model as can be seen in Figure 5.1 can be interpreted as an ensemble of reinforcement learning algorithms.

Similar to the simple DDPG, MADDPG is an actor-critic approach, where the actor's task is to determine the ideal action based on the state, while the critic's task is to determine the Q value based on the state and the action given by the actor, and all both the critic network and the actor network have a target network to avoid divergence in learning and more stable learning. Thus, in total, an agent is composed of 4 neural network-based approximators, in a multiagent approach this is multiplied by the number of agents. As with centralized training in the past, it is important to emphasize that in a multi-agent approach, each critic network must have access to the state seen by all other agents and the actions of the agents. This is necessary so that the critic network can evaluate the action given by the actor for each agent based on the corresponding reward function, so that in the long run all the agents can cooperate with each other in synchronism. I grouped the multi-agent model into a SuperAgent class to make the code clearer and to unify the MADDPG algorithm. A primary motivation behind MADDPG is that, if we know the actions taken by all agents, the environment is stationary even as the policies change, since:

$$P(s'|s, a_1, \ldots, a_N, \pi_1, \ldots, \pi_N) = P(s'|s, a_1, \ldots, a_N) = P(s'|s, a_1, \ldots, a_N, \pi'_1, \ldots, \pi'_N)$$

Equation 5.1: Environment stationarity

for any $\pi_i \neq \pi'_i$. This is not the case if we do not explicitly condition on the actions of other agents, as done for most traditional RL methods [10]. I chose this algorithm, because it is based on a robust actor critic method which can efficiently map large state spaces to continuous actions. The model is expected to determine as accurately as possible how much resources each microservice needs. Although the task could have been approached with discrete action spaces, in the long run I found it more advantageous to use an algorithm that can handle more refined actions. It is the job of the actors of each agent to estimate the utilization of microservices as a percentage. Given that each microservice has a maximum instance size, agents need to estimate what percentage of that instance size should be used. Every actor has to output a value between 0 and 1, which represents the percentage of used instances compared to the maximum available instances in the microservice. For example if a microservice has a maximum instance size of 100 and the actor predicts utilization

percentage 0.31, then 31 instances will be started for the service (rounded up to the nearest integer).

Both the actor structure, and the critic network were simple fully connected neural networks, with five layers and ReLu activations, however the critic network was slightly larger in terms of neuron number. The learning rate of the critic was set to 0.0075, while the learning rate of the actor was 0.0003. The buffer had a batch size of 1200 and 6000 as maximal size.

5.2. SEMADDPG

The classic MADDPG algorithm is a robust model that can cope with training in both cooperative and competitive environments, but it has minor shortcomings. One such problem is the lack of scalability. The curse of dimensionality that occurs in the case of critical networks not only increases the training time, but also slows down the convergence of the model towards the optimal policy. Inspired by model-based approaches, I argue that the training time could be reduced here as well, because each critic sees the same values in their input, which includes the states seen by all the actors and the actions taken by the actors. As we saw earlier, the purpose of this model element is for each critic to have access to the current state of the entire environment, and in light of this, the individual actors are evaluated. Each critic provides the evaluation of the Q value only to the corresponding actor, which means that although the entire system can converge faster than in the case of a single critic, this implementation increases the model size - which is made worse by the fact that, similarly to the plain DDPG algorithm, the targets are also present here networks to stabilize training.

To eliminate this problem, I introduced a special state encoding procedure into the algorithm, the purpose of which is that with the help of an autoencoder, each critic receives only a compressed vector of the current states of the actors, which gives a smaller and more compact picture of the current state of the system. Recently, a variety of (unsupervised) representation learning algorithms have been proposed based on the idea of autoencoding where the goal is to learn a mapping from high-dimensional observations to a lower-dimensional representation space such that the original observations can be reconstructed (approximately) from the lower-dimensional representation. During teaching, the task of an encoder decoder structure will be to develop a latent representation for each state that is identical to the original state vector in terms of information content, but much smaller in terms of size. The degree of compression is indicated by a new parameter, which is a fractional number between 0 and 1

and denoted by χ. If a compact state vector is with 10 elements and $\chi = 0.7$, only 7 elements will be included in the latent representation.

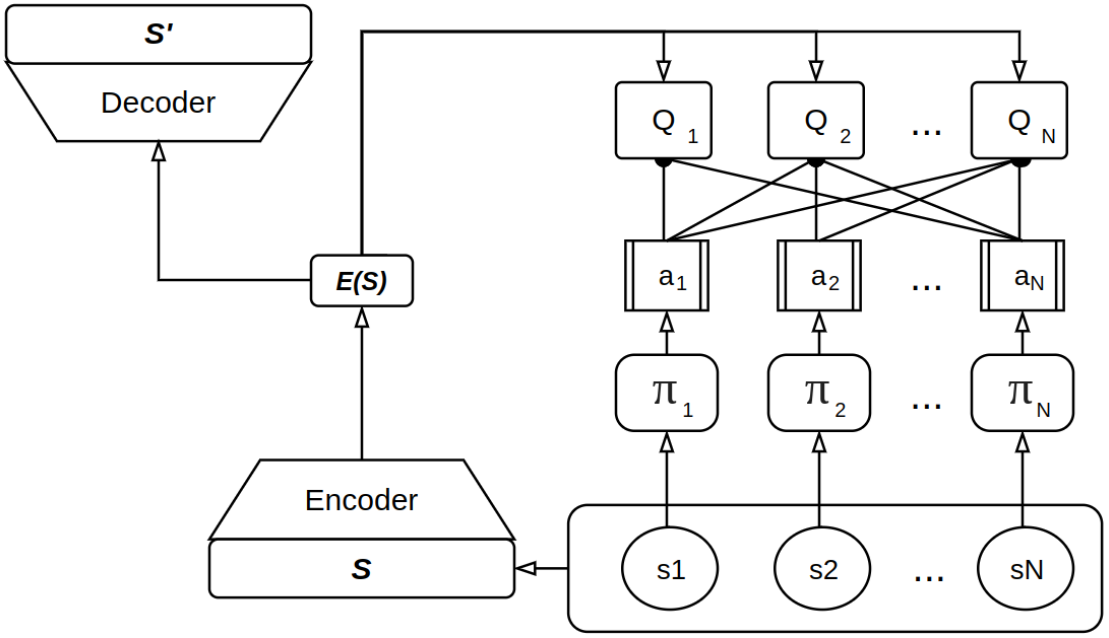

Figure 5.2: SEMADDPG model structure

I had several model design attempts, where I tried to compress not only the states, but also the corresponding actions, but at the same time, these trainings were not successful, the system diverged strongly, and it was not possible to create or maintain optimal policies. A similar result was shown when, during the training of the encoder, I not only took into account the reconstruction loss, but also back propagated the loss coming from each critic. The intuition behind my idea was that regardless of the fact that the autoencoder strives for the most efficient compression of the states, it is not a guarantee that it will retain the features of the latent representation that mostly determined the learning ability of each critic. When I also propagated this error vector in addition to reconstruction loss, the performance of the autoencoder started to deteriorate drastically, and with it the entire reinforcement learning performance dropped, so I used a simple reconstruction loss for the final model, as can be seen in the Algorithm 5.2. Compared to the base MADDPG, I not only used encoder decoder helper structure, but also used Prioritized Experience Replay (PER) in order to enhance the performance of the model. This addition was not applied in the MADDPG algorithm since the original implementation had a simple replay buffer implementation only. The final structure of the model can be seen in Figure 5.2.

Algorithm 2 State Encoded Multi-Agent Deep Deterministic Policy Gradient

for episode = 1 to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for t = 1 to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + N_t$ w.r.t the current policy and exploration
    Execute actions $a = (a_1, ..., a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in prioritized replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of K samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Let $\mathbf{S}$ be the union of $\mathbf{x}^j$ and $\mathbf{x}'^j$ states and $\mathbf{s}^j \in \mathbf{S}$
      Update encoder $\mathcal{E}$ and decoder $\mathcal{P}$ by minimizing the loss:

$$\mathcal{L}(\mathbf{s}^j) = (\mathbf{s}^j - \mathcal{P}(\tilde{\mathbf{s}}^j))^2 \text{ where } \tilde{\mathbf{s}}^j = \mathcal{E}(\mathbf{s}^j)$$

      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\tilde{\mathbf{s}}^j, a'_1, ..., a'_N)|_{a'_k = \mu'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum (y^i - Q_i^{\mu}(\tilde{\mathbf{s}}^j, a_1^j, ..., a_N^j))^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, ..., a_i, ..., a_N^j)|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \rightarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Algorithm 5.2: SEMADDPG algorithm

5.3. Horizontal Pod Autoscaling as a baseline algorithm

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely used by big tech companies, and engineers. Kubernetes implements horizontal pod autoscaling as a control loop that runs intermittently. Horizontal Pod Autoscaling is a scaler algorithm which automatically updates a workload resource, with the aim of automatically scaling the workload to match demand, which periodically adjusts the desired scale of its target to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric we specify. The dynamic of the algorithm is straightforward. Based on metrics collected in the microservice environment, the algorithm finds the accurate number of instances to utilize for a microservice, based on predefined preferences, such as utilization percentage, and the intermission time. According to the literature review most big companies

use more advanced approaches for the scaling of microservice resource handling, however this built-in algorithm is still commonly used, despite its shortcomings:

- **Lacks robustness**: This algorithm may be useful for small microservice systems but it is not designed to handle a huge number of microservices
- **Takes time to make changes**: There is a lag between detection and scaling in the environment. For the auto scaling solution to be effective, a key requirement is to have the auto scaling lag be less than the maximum time available to autoscale
- **Fail to detect workload spikes**: There might be huge intraday spikes for the usage of microservice systems which the HPA cannot detect in time which leads to QoS violations

In order to be able to effectively and realistically measure the performance of the horizontal pod autoscaling algorithm, we must make various compromises to it from the point of view of the environment, and the operation of the algorithm must be adjusted to the conditions of the environment. Given that the developed MDP system represents a time resolution of arbitrary granularity, for benchmarking purposes the HPA takes into account the loads belonging to the previous time window and predicts the utilization required for the next time slot. With this, we achieve that the trailing effect of the HPA is represented in the modeled system at the same time, and that we get a realistic basis of comparison compared to the multiagent algorithms implemented later. It is important to point out that we are more permissive in a couple of aspects compared to the HPA algorithm: while reinforcement learning approaches are not allowed to overload the system, in the case of HPA it must be enabled for individual microservices to be overloaded.

6. Evaluation

In the course of my work, I trained both algorithms under almost identical conditions, so that the differences between the State Encoded MADDPG and the plain MADDPG models could be validated as much as possible. In the figures below, I present the training that took place in environments of different complexity based on this evaluation score. In order to be able to interpret the values shown in the figures in a comprehensive way, I divided the value of the scores achieved in each episode uniformly by the number of nodes in the microservice system, so that the maximum number of points available during each episode will be the same as the maximum number of steps - since the maximum available reward for each microservice is 1 in each episode.

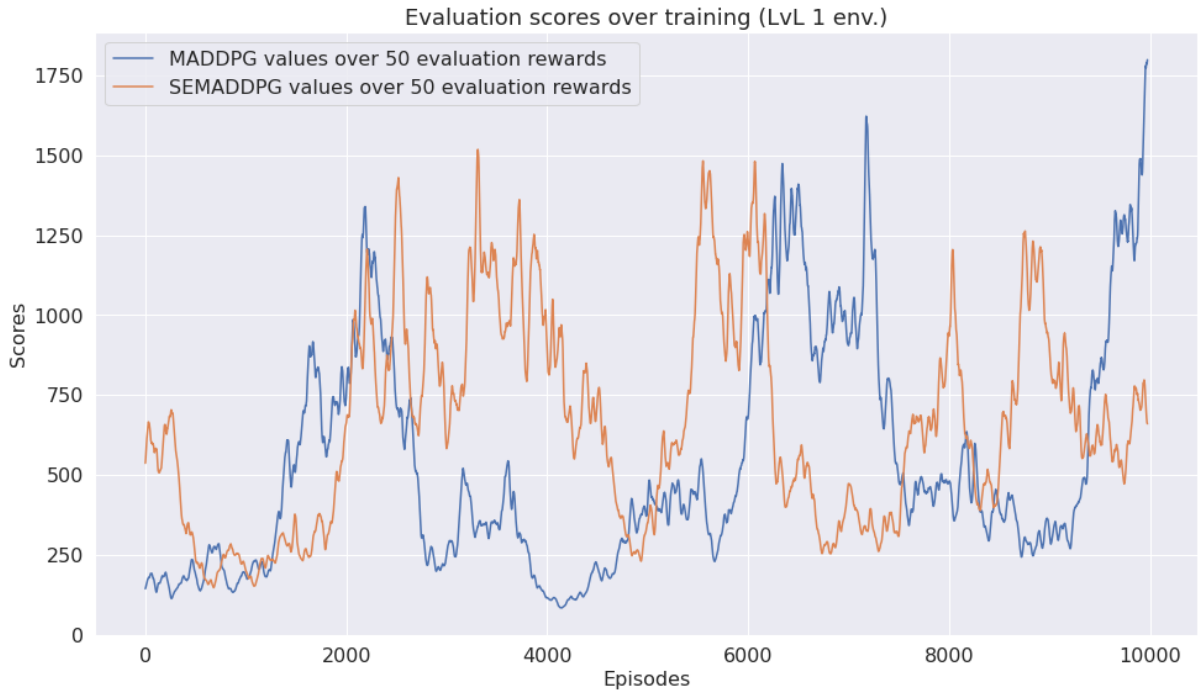

Figure 6.1: Level 1 environment validation scores

I started each level of training with the MADDPG algorithm as an RL baseline and then proceeded with the SEMADDPG training. The first evaluation scores of the 1st level training can be seen in Figure 6.1. It became clear already during the first training that both approaches were able to create a suboptimal policy quite quickly - roughly around the 2000th episode - with the help of which it achieved around 1250-1350 points per episode. Although here MADDPG started to converge somewhat faster, later on there was a drop in reward. The initial high score of the SEMADDPG could be attributed to lucky random initialization, as this phenomenon did not occur in any other training. In contrast, SEMADDPG was able to

maintain this suboptimal policy. Given the highly stochastic nature of the underlying environment, it is not surprising that the multi-agent models show a highly fluctuating behavior instead of a clear and consistent reward increase, as can be seen in Figure 6.1.

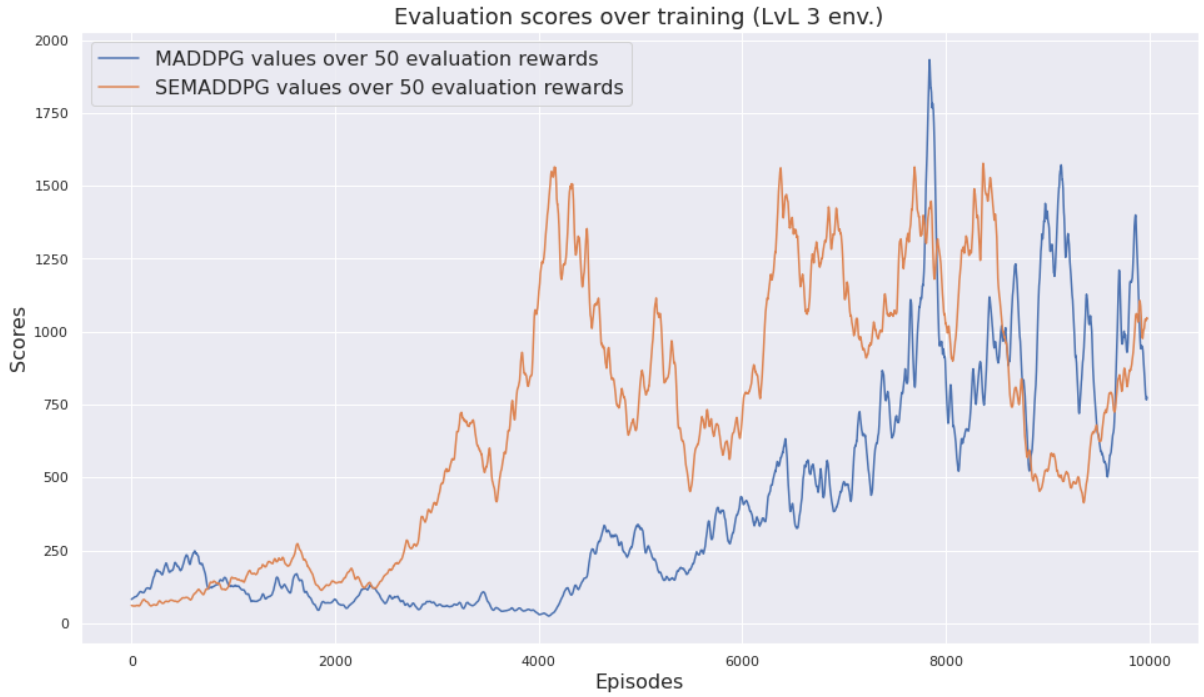

Figure 6.2: Level 2 environment validation scores

Although the training on the second complexity level did not contain many more microservices (only six in total), the number of codependent relationships in the system was higher, because more microservices burden other microservices with loads. On Figure 6.2 one of the strengths of SEMADDPG can be observed much more significantly: although in the initial episodes the individual models managed to collect approximately the same amount of rewards, as the encoder belonging to the critics starts to converge, the critics are able to guide the actors towards an optimal policy more and more quickly and efficiently. This phenomenon also appeared during two further re-trainings, also in the case of a more complex state. Similarly to the environment of individual complexity, SEMADDPG is able to approach a strategy with a relatively good score several times, and is able to maintain this strategy with a good score. In contrast, MADDPG converges somewhat more slowly, but it is important to point out that when it converges, it outperforms the SEMADDPG model in the vast majority of cases.

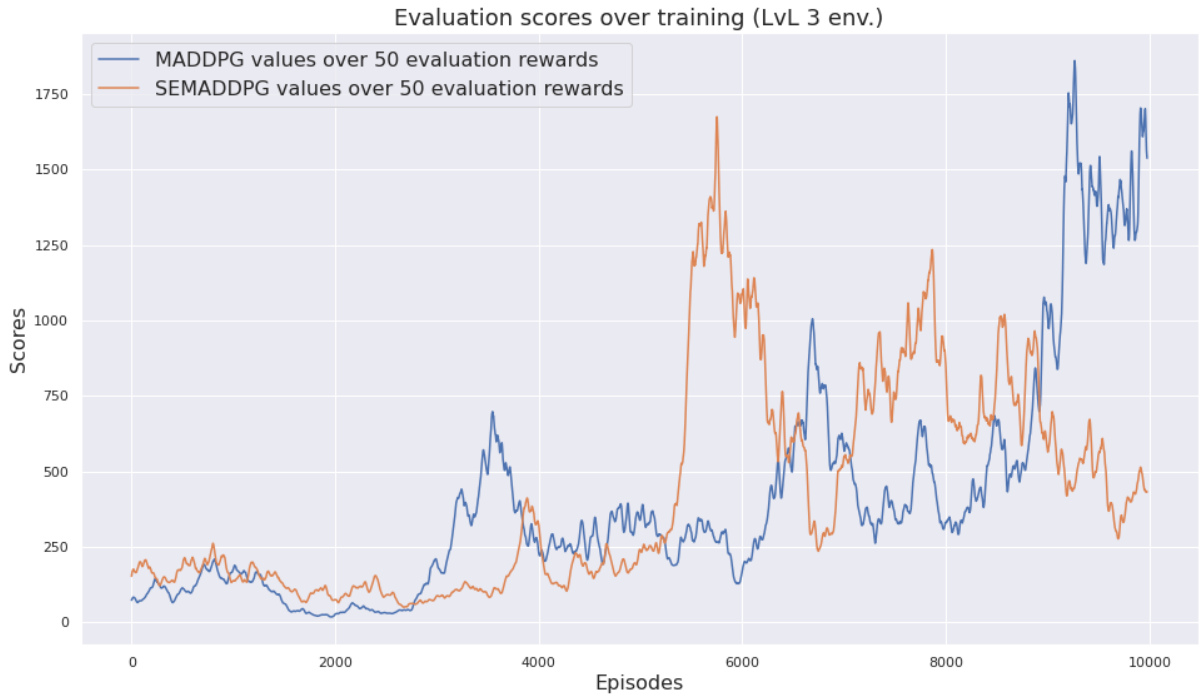

Figure 6.3: Level 3 environment validation scores

Even in the case of teaching in the most complex environment, it can be observed that SEMADDPG is able to develop a policy that can be considered optimal, but presumably due to the strong stochastic nature of the system, it cannot maintain the score associated with this policy later on as it can be seen on Figure 6.3. Regardless, MADDPG produces a much slower but more stable convergence, which ultimately results in a higher aggregate score.

Model metrics			
Model name and complexity	Training time	Best val. score(scaled)	Best val. score
HPA @ Level 1	-	1021.53	4086.12
MADDPG @ Level 1	3.21 hours	**1799.02**	**7196.08**
SEMADDPG @ Level 1	**2.74 hours**	1518.35	6073.40
HPA @ Level 2	-	986.53	5919.18
MADDPG @ Level 2	7.39 hours	**1932.92**	**11597.50**
SEMADDPG @ Level 2	**5.91 hours**	1577.34	9464.09
HPA @ Level 3	-	872.41	13086.15
MADDPG @ Level 3	19.76 hours	**1860.22**	**27903.34**
SEMADDPG @ Level 3	**15.41 hours**	1674.23	25113.47

Table 6.1: Comparison of models

Table 6.1 shows a comparison of each model and training. It is clear that the reinforcement learning-based approaches were able to outperform the HPA algorithm in all cases without exception. On average, HPA's scaled score was 903.9 points lower than that of the best performing MADDPG algorithm during all training, while it was 629.8 points behind the SEMADDPG algorithm on average. Although 90% load was set as the ideal value for HPA, in many cases the algorithm still struggled with this task. It was a general observation that

HPA scaled relatively well in less complex microservice systems and in the case of nodes with fewer chief nodes - in some cases outperforming RL approaches. Nevertheless, in the case of microservices of greater complexity or a stronger stochastic nature, the algorithm struggled to create an allocation that meets the predefined requirements. Considering that during the development of the reward function, I primarily sought to maintain the stability of microservice systems; based on the results it can be said that in critical QoS violation intolerant systems, Reinforcement Learning-based approaches can dominate classical scaling solutions.

Based on table 6.1, the figures and the measurement results, it can be concluded that the SEMADDPG algorithm can be an effective solution for managing a more complex microservice system, as it is able to create a suboptimal policy in a shorter time. On average, it took 18.86% less time to teach SEMADDPG, but in my opinion, this value can be improved with further implementation optimization. However, it is important to take into account that the policy created in this way cannot achieve perfect results, and due to the compression of the states, this globally optimal policy cannot necessarily be approached. At the same time, we can move forward on this problem thanks to the robustness of MADDPG: since the model enables centralized training, at the same time, the execution is decentralized, we will only use the actors in our final deployed system. This also means that our actors with suboptimal policies trained with state encoded critics can be further trained with critics processing the entire state space used in the classic MADDPG. Presumably, not only the time required for convergence to be improved, but also the performance of the final actors could approach the accuracy of MADDPG. This combined approach is an aspect that is definitely worth investigating.

7. Summary

Resource management in microservices is still a highly complex issue to this day. During my thesis, I addressed this topic through multiagent deep reinforcement learning. I have developed an artificial modeling environment that allows us to model microservice clusters at a low level together with the connections found between services. The library-like environment lets us set the number of external requests as well as the number of request sources. For the developed interface, I implemented the MADDPG algorithm detailed in the OpenAI publication, and tried to improve it. The new SEMADDPG algorithms turned out to be a lighter structure in terms of dimensions regarding the critic networks, and converged faster to a suboptimal global policy, however it still has limitations when it comes to real world applicability. As a result, during the continuation of the research, great emphasis must be placed on the design and development of small, fast-learning and efficient models. Despite this both models were able to outperform Horizontal Pod Autoscaling, indicating that machine learning algorithms - especially multiagent reinforcement learning algorithms can be a gamechaning solution when it comes to scaling robust microservice systems.

8. Bibliography

[1] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance Engineering for Microservices: Research Challenges and Directions. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion). Association for Computing Machinery, New York, NY, USA, 223–226. https://doi.org/10.1145/3053600.3053653

[2] Zhang, Yanqi, Hua, Weizhe, Zhou, Zhuangzhuang, Suh, Ed, and Delimitrou, Christina. Sinan: Data-Driven Resource Management for Interactive Multi-tier Microservices. Retrieved from https://par.nsf.gov/biblio/10188080. Workshop on ML for Computer Architecture and Systems (MLArchSys)

[3] QIU, Haoran, et al. {FIRM}: An Intelligent Fine-grained Resource Management Framework for {SLO-Oriented} Microservices. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020. p. 805-825.

[4] Z. Yang, P. Nguyen, H. Jin and K. Nahrstedt, "MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows," 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 122-132, doi: 10.1109/ICDCS.2019.00021.

[5] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2022. Reinforcement learning for resource management in multi-tenant serverless platforms. In Proceedings of the 2nd European Workshop on Machine Learning and Systems (EuroMLSys '22). Association for Computing Machinery, New York, NY, USA, 20–28. https://doi.org/10.1145/3517207.3526971

[6] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

[7] Hafner, R., Riedmiller, M. (2011) Reinforcement learning in feedback control. Mach Learn 84, 137–169

[8] H. M. Schwartz (2020) “Multi-Agent Machine Learning: A Reinforcement Approach”

[9] Gerald Tesauro. 2003. Extending Q-Learning to general adaptive multi-agent systems. In Proceedings of the 16th International Conference on Neural Information Processing Systems (NIPS'03). MIT Press, Cambridge, MA, USA, 871–878.

[10] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17). Curran Associates Inc., Red Hook, NY, USA, 6382–6393.

[11] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015)

[12] Oroojlooy Jadid, Afshin & Hajinezhad, Davood. (2019). A Review of Cooperative Multi-Agent Deep Reinforcement Learning.

[13] Nguyen, Thanh Tung et al. “Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration.” Sensors (Basel, Switzerland) 20 (2020): n. pag.

[14] Lowe, Ryan, et al. "Multi-agent actor-critic for mixed cooperative-competitive environments." Advances in neural information processing systems 30 (2017).

[15] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” J. Grid Comput., vol. 12, no. 4, pp. 559–592, 2014.

[16] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Elasticity in cloud computing: State of the art and research challenges,” IEEE Trans. Services Comput., vol. 11, no. 2, pp. 430–447, Mar./Apr. 2018.

[17] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, “Elascale: Autoscaling and monitoring as a service,” in Proc. 27th Annu. Int. Conf. Comput. Sci. Softw. Eng. (CASCON), 2017, pp. 234–240.

[18] M. A. Kaboudan, “A dynamic-server queuing simulation,” Comput. Oper. Res., vol. 25, no. 6, pp. 431–439, 1998.

[19] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth, “Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control,” in Proc. ACM 3rd Workshop Sci. Cloud Comput. (ScienceCloud), 2012, pp. 31–40.

[20] M. Wajahat, A. Gandhi, A. A. Karve, and A. Kochut, “Using machine learning for black-box autoscaling,” in Proc. IEEE 7th Int. Green Sustain. Comput. Conf. (IGSC), 2016, pp. 1–8.

[21] G. Chen, “Energy-aware server provisioning and load dispatching for connection-intensive Internet services,” in Proc. 5th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2008, pp. 337–350.