Machine learning based metro edge computing network resource optimization algorithm for 5G and beyond applications

Luong Quoc Dat
Dept. of Electrical Engineering
Technical University of Eindhoven

***Abstract*—Nowadays, there is an increasing demand for computing resources in metro edge computing (MEC) networks to serve multiple services. However, deploying large computing resources at every node in the MEC network is inefficient and expensive. One solution is to deploy the micro-services in multiple nodes and routing the traffic load in the network. In this work, we propose a solution to distribute the user traffic load properly as well as strategically deploying services in the MEC network using the reinforcement learning (RL) method. The results show that the RL agent can provide the location of the resources to serve multiple services without prior expert knowledge in different network topologies with real user traces. Moreover, the RL agent has been tested in both simulation and real-life MEC network setup.**

I. Introduction

In recent years, it is undeniable that there is a significant increasing trend in the number of users as well as devices connect to the Internet [1]. This trend leads to a huge demand in MEC networks infrastructure in terms of service expectations. Therefore, MEC networks should efficiently support a variety of access applications with different dynamic user traffic patterns from various sources such as enterprises, home applications, media streaming, and IoT applications. These applications not only require high resource demands but also require low latency response. For those latency-sensitive applications, cloud computing in data centers is not efficient enough to support. Because of the long distance between the data centers and end-users, long delay from signal propagation leads to high latency. To address this issue, the computing servers had been brought to near the end-users [2]. This is edge computing. Edge computing refers to the enabling technologies that allow computation to be performed at the metro network so that computing happens near data sources.

However, the cost of deploying such large computing resources in the metro network is expensive. In real life, computing resources are limited. Therefore, the computing resources in a MEC network node might not be able to serve all the aggregated requests from the access network. This leads to a high drop request rate and affects the quality of services. One solution to solve the problem is to strategically schedule and routing the requests to be served at other MEC network nodes in the network to reduce the burden in the particular node. By scheduling where the requests should be served, we also decided where the resources should be allocated accordingly. With a good arrangement, users can experience a good quality of service with low latency and high accepted requests ratio. Therefore, it is important to find a way to utilize the resources in the metro network in a way that satisfies the quality of services for clients. Normally, this work has been done with the knowledge of experts in network traffic engineering [3]. They relied on the node capacities, traffic users to assign bandwidth for each node to process. For the specific case with known resource computing distribution in the metro network, traffic user patterns, network topology, service requirements, and service priority, this solution works very well. However, this type of solution is working only for a particular case, if there is a change in the user traffic pattern or a new service introduces in the network, the solution is not working properly anymore. Moreover, all the information is not always available in real life for example the user patterns or exact computing resource capacity of each node. Therefore, the solution needs to be online, automated, adapted to different user's demands, network topology, and multiple services deployed in the same network.

In recent years, Machine learning (ML) has been explored as a bright candidate to deal with the problem in the optical network including resource allocation to optimize the network by using Reinforcement Learning (RL) [4]. According to Danish Rafique *et. al.* in [5], the main motivations to apply the ML method in the metro network are heterogeneity, reliability, capacity, and complexity, which are not suitable with the static traditional design principle anymore. The author also argued that network self-configuration, including resource allocation and service (re)configurations—both for physical and virtual infrastructure are among the core application of RL in the metro network. Applying Machine Learning, especially Reinforcement Learning in the metro network is not new. In [6], Pham Tran Anh Quang *et. al.,* also using reinforcement learning to deal with the allocation of virtual network functions (VNF) for realizing network services. The author proposed reinforcement learning with a neural network and "memory play" which allows having the advantages of supervised learning to allocate VNFs in the network. In [7], Stefan Schneider *et. al.*, showed that with the use of reinforcement learning in optimization of resource utilization, they can achieve a better result in terms of successful request rate served compared to other methods such as heuristic method [8], baselines shortest path and load balance [9]. All the of above works show the potential of applying DRL method in maximizing the resource utilization in metro network. Indeed, based on the literature, the author took the idea of using DDPG method for solving continuous action space and scheduling table for routing the traffic in the network. However, in previous work, the RL is only designed to solve one service scenario without real pattern traffic for each service. Moreover, all the results are only in simulation and prototype without testing in a real network.

In this work, we design the RL method in a real-life scenario with multiple services with real traffic user patterns to maximize the utilization of resources in the MEC network to serve 5G and beyond services. It is a model-free RL algorithm, therefore it does not require network knowledge during designing and training. Because it is based on experiences and feedback with real-life traces during training, it can adapt to different user's demands as well as different network capacity distributions. Based on the work of [7], we further develop the multiple services feature in their simulation tool. Therefore, we can train and test the RL agent in a multiple-service environment, which makes the scenario more realistic instead of evaluating with only one service. We also further set the priority for each service to investigate how the RL model can coordinate among services. Moreover, we also aim to apply RL not only in simulation but also in a real-life setup environment. For the demonstration purpose, in the TU/e ECO lab, there is an optical metro ring network that has already been set up. The ring optical metro network consists of three nodes connected by optical links. Overall, the results show that the RL agent can provide the location of the resources to serve several services without prior expert knowledge in different network topologies with real user traces. The RL agent can work independently to the locations of client aggregated traffic as well as the location of computing resources in the network. Moreover, the RL agent also learns the priority concept to maximize the successful request rate for higher-priority services.

The rest of the report is constructed as the following. First of all, the user traffic generation and network services are presented in section II. The idea of how we approach RL and implement RL in the metro network in the Markov decision process is discussed in section III. The implementation in metro network are presented in section IV. The results of RL agent in both simulation and real metro network are discussed in section V. Finally, at the end in section VI is conclusion.

II. User traffic generation and network services

A. Network services

There are many user traffic traces of different services in real-life represented in various papers [1], [2], [10]. Among the services that could be deployed in the optical metro network, web searching, web serving, and online shopping are chosen. According to [1], they are among the most used WWW services on the Internet nowadays. To design these services, we used the most popular platform on the Internet today: Nginx server, which has been taken from Apache since April 2019 [11]. To deploy all these application servers to the MEC network, we have 2 options: container and virtual machine (VM). In this work, all the applications are deployed using docker container technology because containers are more portable and efficient [12]. As in Figure 1, containers take up less space than VMs (container images are typically tens of MBs in size), they also can handle more applications, and require fewer VMs and Operating systems [13].

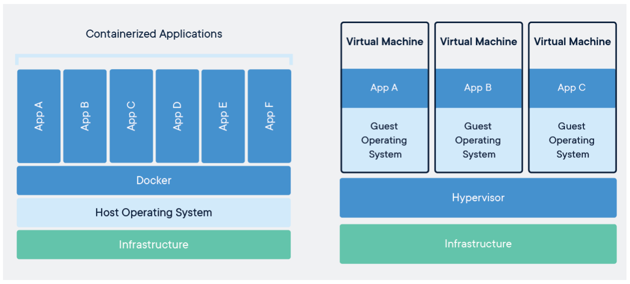


Fig. 1. Container and Virtual Machine

The web service, the search service, and the online shopping service are designed as simple static website services. Each service contains two network functions: a reverse proxy server and a static HTML web server. The reverse proxy increased the security by keeping the actual address of the backend server remains unknown to users. In this setup, the reverse proxy server just listens to the request from clients and forward it further to the server address. The web servers contain static HTML files which are different among the services. For example, the search service server contains a static HTML Google web page, the shopping service server contains a static HTML Amazon website, and the web service server contains a static HTML Wikipedia website. We can have several servers in the same node by assign each server to each different port as we can see in Figure 2. In this case, we only deployed the proxy server in node 4 but noted that the proxy server also can be deployed at node 3 and node 2. All of these servers are built based on Nginx images in DockerHub.

B. User generator

Because the access network is not available in the lab, simulated traffic is generated from nodes to the metro network as client containers. To the most knowledge of the author, most of the user behavior can be assumed daily periodic based on research papers [14]–[17]. Based on the patterns in these researches, a software program python-based has been constructed to simulate the user. To simulate multiple concurrent users at the same time we used multi-threading in python. Each thread represents a user. By changing the number of threads in the program, we can change the number of user requests to services. Because the service servers contain static HTML files, the amount of traffic will be proportional to the number of user requests. Therefore, the pattern of traffic captured over time in the research papers can be translated to the pattern of user number over time.

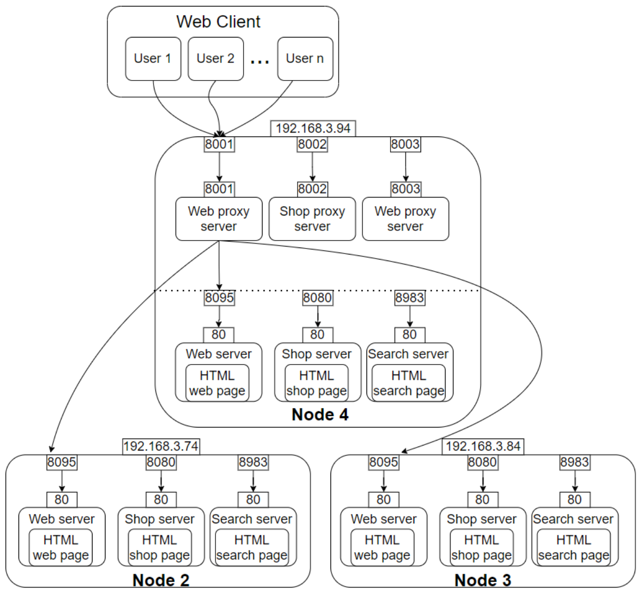


Fig. 2. Services and client deployment in MEC network

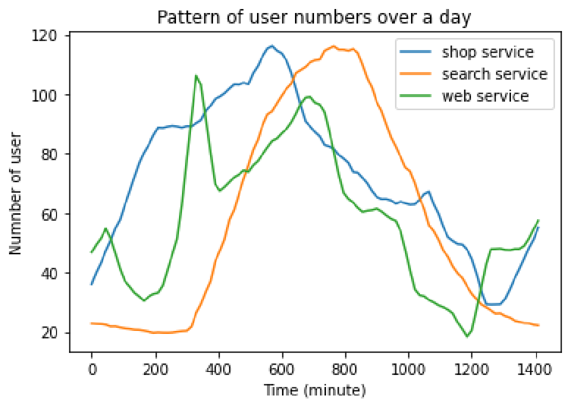


Fig. 3. Number of user over a day according to traffic pattern in research papers

The client container takes two main parameters as input: the number of users and the service port number. By updating the number of users over time according to the traces from research papers, we can simulate the overall aggregated traffic. By adjusting the service port number, we can choose which service the client container subscribes to. User requests can come from any node and they can request any service in the network. Therefore, there can be multiple client containers in any node. Each client container can subscribe to any network service. For example, in the Figure 2, we consider a client container for web service. This client container contains n threads, each thread represented a user. This number of threads can be updated as a global environment in the client container. Each thread requests the web service with the fixed request rate of 2 requests/sec via requests package in python library.

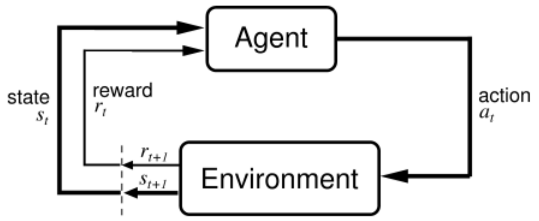


Fig. 4. Agent-environment interaction

III. Reinforcement learning approach

Briefly speaking, reinforcement learning deals with learning via interaction and feedback, or in other words learning to solve a task by trial and error. As we can see from Figure 4, the agent is the learner and the decision-maker from the environment. An action belongs to a set of actions that the agent can perform and the state is the state of the agent in the environment. For each action selected by the agent, the environment provides a reward which usually is a scalar value. By observing and receiving reward from environment, the agent learn how to react to solve the problem in environment.

In this section, the mathematical network model is presented in III-A. To optimize the goal in the network model using the RL method, a Markov Decision Process (MDP) is proposed in III-B. Finally, an RL agent to solve the MDP is designed in section III-C.

A. Network modelling

The metro network is model as a graph $G =< N, L >$ with N is the number of nodes and L is the link between them. Each node $n \in N$ is modeled with its own amount of computing resource capacity n_{cap} and each link L is model with its own link delay depends on the length of the link L_{delay}. Each service $s \in S$ is model as a service chain function (SFC) which is combined from several virtual network functions (VNF) $vnf \in VNF$. For example, web service can be model as a service chain with proxy server and web server combine: $s_{search} =< vnf_{proxyserver}, vnf_{server} >$. Moreover, service s has its own service priority in the network. Each VNF has its own processing time for requests from clients. To model the clients for different services, the traces in section II are used. The traffic flow from client f is modelled by client request rate f_{rate} and which service it is requested f_s. The metric data are captured after every Δ time. The metrics that we captured are the statistic of latency, the number of dropped requests and successful requests of each SFC over the Δ time, and the run time of the program.

For traffic routing and VNF placement, the request routing table is introduced. The idea is the same with the scheduling table in [7]. Each node in the MEC network has a traffic routing table we can see in Table I. Let us say the network has three nodes: node 2, node 3, and node 4. In node 2, we have incoming traffics from clients to several services such as search service and web service. According to Table I, when a search request coming to node 2 from the client and demand

TABLE I
REQUEST ROUTING TABLE IN NODE 2

Service	VNF	Node 2	Node 3	Node 4
Search	Search proxy server	0.5	0.5	0
	Search server	0.33	0.33	0.33
Web	Web proxy server	1	0	0
	Web server	0.5	0.5	0
...				

to be processed with $vnf_{proxyserver}$, 50% chance it will be processed at the proxy server of node 2 and 50% chance it will be routed to node 3 processed at the proxy server of node 3. By doing this, it also does the provisioning where should the VNF be. For example, according to Table I, the web proxy server is only placed at node 2 in the entire network. To present the table in mathematical way, let us denote decision variable $x(n, s, vnf, n)|n \in N, s \in S, vnf \in VNF$. This variable decided how likely where a traffic flow should be served for each node, each service, and each VNF in that service. There are 2 constraints to this variable. To avoid fragmentation, it should have a minimum value x_{min}. The total percentage of $x(n, s, vnf, n)$ for a VNF should be 1.

We assume that the network function does not occupy any fixed computing resources. Each flow from client will occupy one unit in the computing resource capacity of the node during the vnf processing time. For example, if there is a flow coming to node 2 and requested to be served at node 2. It will occupy one computing resource of node 2 during processing time. We assume the processing time is fixed for each VNF. Therefore, the computing resource usage at each node $n_{used} \in [0, n_{cap}]|n \in N$ depends on the traffic load coming to the node and asked to be processed. In details, it depends on the request routing table (or $x(n, s, vnf, n)$) and user request data rate f_{rate}. A client flow is dropped if there is not enough capacity in the node.

The objective is to increase the quality of service for users, which is to satisfy the low latency and maximize the successful request rate. The latency is already be solved by edge computing. By correct rearrangement of the traffic routing table, we can maximize the successful request rate from users. To formulate the objective, we have the following equation:

$$O = \frac{\sum_{s \in S} succ_s \times priority_s - \sum_{s \in S} drop_s \times priority_s}{\sum_{s \in S} succ_s \times priority_s + \sum_{s \in S} drop_s \times priority_s} \tag{1}$$

The objective O calculates the successful request rate of all services. Equation 1 also shows that the higher the priority of service, the more that service contributes to the total success rate. By maximizing this term, it encourages the network to serve more the service requests from users, especially the high priority one.

B. Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used for modeling decision-making problems. It contains:

Observation space: $O = f_{rate}(n, s)|n \in N, s \in S, n_{used}/n_{cap}|n \in N$. It observes the traffic flow from client at each node for each service and the utilization percentage of each node during the last Δ time.

Action space: $A = x(n, s, vnf, n)|n \in N, s \in S, vnf \in VNF$. The action space described the request routing table as in Table I. For each node, we have a particular traffic routing table and the action is to update the table after every Δ time. As we can see, the action space is continuous. This is a challenge for traditional reinforcement learning but it can be solved using DDPG which is presented in the following section.

Reward: $R = O_{\Delta}$. We use the objective function but we calculate it only based on the last Δ time.

C. RL design

The main idea here is that the RL agent observes new observations from metrics collected for the last Δ interval of time. Based on the monitored information and also the reward from the last action, it decides how to route the requests from each node to another. The RL agent is trained offline. During this time, it interacts with the environment and learns to give the best action given the observation. After training, it will be able to schedule where the requests should be served, allocate the IT resources accordingly. Because the action space in our MDP is a continuous space, a deep deterministic policy gradient (DDPG) algorithm is chosen [18]. DDPG is the algorithm that learns a deterministic policy and a Q-function by using each to improve the other.

In our problem, we want to maximize the successful request rate of all services. We want to maintain this successful rate over a long time. When we train a RL model, we want to know which action, start at which state and follow which rule to maximize the successful request rate. Therefore, it is good to know the total expected reward if the agent starts at a specific state s, takes an action a and then takes the action according to the same policy π until finish the trajectory τ. This is the concept of action-value function $Q(s, a)$:

$$Q(s, a) = E_{\tau \ \pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2}$$

To calculate the optimum action-value function $Q^*(s, a)$ we need to find the optimum policy which maximizes the action value-function:

$$Q^*(s, a) = \max_{\pi} E_{\tau \ \pi}[R(\tau)|s_0 = s, a_0 = a] \tag{3}$$

If we know the optimum action-value function $Q^*(s, a)$, then in any given state, we can find the optimum action by:

$$a^*(s) = \arg\max_{a} Q^*(s, a) \tag{4}$$

Using eqn. 3 and 4, the agent interleaves learn both how to maximize the Q-function and action policy.

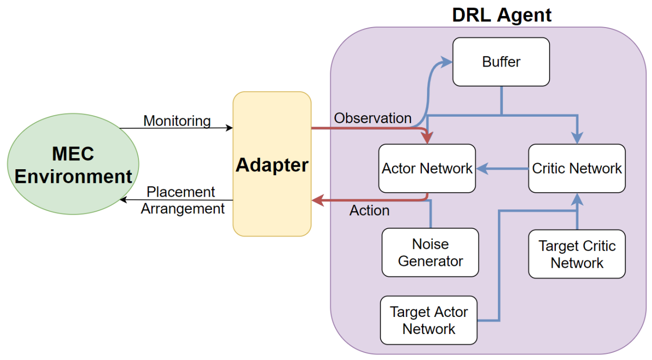


Fig. 5. The design of DDPG agent, the blue line represented the training process, the red line represented the testing process after training.

This works perfectly with the discrete action space because calculating $a^*(s)$ from Eqn 4 is very straightforward. We just check and calculate with each action a and chose the one that maximizes the term $Q^*(s,a)$. However, in our problem, the action is continuous, which means if we calculate the maximum normally, it required huge computation resources as well as calculation time. This is unacceptable since the agent acting in the environment and needs to respond to the environment in a fast manner.

To solve this problem, DDPG comes up with an idea of learning a policy $\mu(s)$ to optimize the action to satisfy the maximum of $Q^*(s,a)$. Therefore, in steads of calculating $\max_a Q(s,a)$, we can just approximate it with $Q(s,\mu(s))$. In DDPG, this gradient-based learning rule for policy $\mu(s)$ is presented as actor network and the $Q(s,a)$ is presented as critic network. Both of them are neural networks.

Design

The actor network $\mu(s)$ takes the observations as the input and gives the action at the output. In DDPG, the action is deterministic: $a = \mu(s)$. This means that with the same observation state, the output action is always the same.

The critic network $Q(s,a)$ takes the observation and the action and gives out the Q-value as calculated in eqn. 2. The Q-value tells us how good the action is to maximize the long-term expected reward. The better the action, the higher the Q-value. The long-term reward is calculated as followed: $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$. Substitute this term into eqn. 2, we have the so-called Bellman equation for action-value function:

$$Q(s,a) = E_{s' \ P}[r(s,a) + \gamma \times E_{\tau \ \pi}[Q_\pi(s',a')]] \tag{5}$$

and if the policy is optimum then we have:

$$Q*(s,a) = E_{s' \ P}[r(s,a) + \gamma \times max_{a'}[Q*(s',a')]] \tag{6}$$

It said that the value of starting point is the reward from being there, plus the value of wherever the agent lands next.

Training

To train the critic network and actor network, we use replay buffer D. It contains the experiences that the agent interacts with the environment. For each interaction, it stores a tuple which contains: observation, action, reward and next observation $(s,a,r,s') \in D$. These data then can be fed into the critic and the actor network for training. The size of the buffer is limited. Therefore, after it reaches its capacity it will discard the oldest values to store the new values.

Let us say the set of parameters in actor network is θ. We need to update these parameters to get the perfect set which should give us the best action given the observation. Training this actor network is very straight forward. Let say if we have already a well-trained critic network, we can just update the parameters to have the maximum Q-value at the critic network:

$$\theta = \max_\theta E_{s \ D}[[Q^\phi(s,\mu(s))]] \tag{7}$$

Let us denote the set of parameters in the critic network is ϕ. We also would like to train this network by finding the best set of ϕ also. Let say we have an optimal actor network and it gives $\mu_\theta(s)$. We already know from the Bellman equation that the optimal Q-value is the reward plus the value comes from the optimal actor network in the next observation. Therefore, we will try to minimize the difference between the actual Q-value we have from the critic network (with the parameters ϕ) and the optimal Q-value:

$$L(\phi,D) = E_{(s,a,r,s')\in D}[(Q_\phi(s,a) - (r + \gamma Q_\phi(s',\mu_\theta(s'))))^2] \tag{8}$$

By doing this, we are trying to make the Q-function be more like its target. However, in this case, the set of parameters ϕ in both the actual Q-value and the target are the same. When we update the parameters, they move together which makes the loss minimization unstable and the training process is unlikely to converge. To solve this, DDPG and most other Q-learning methods come up with the idea of the target network. The target network will not share the same set of ϕ with the network but it will lag the first with some time delay. In DDPG, they use the soft copy to slowly update the parameters in the target network from the main network by some constant averaging:

$$\phi_{targ} = \rho\phi_{targ} + (1-\rho)\phi \tag{9}$$

The same idea applied to the critic network $\mu(s)$:

$$\theta_{targ} = \rho\theta_{targ} + (1-\rho)\theta \tag{10}$$

To summarize, to train the critic network, we need to use data from buffer D, with the Bellman equation 6, assume we have the optimal target actor network and we also have a target critic network, we update the parameter ϕ in the way to minimize the difference between main critic network and target critic network:

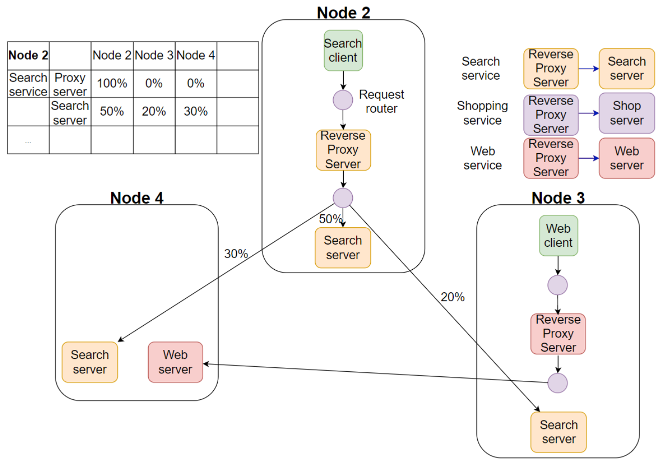
$$L(\phi,D) = E_{(s,a,r,s')\in D}[(Q_\phi(s,a) - (r + \gamma Q_{\phi_{targ}}(s',\mu_{\theta_{targ}}(s'))))^2] \tag{11}$$

The agent interleaves training the actor and critic network together. It updated the θ and ρ together. When we update the actor network, we freeze the critic network and when we update the critic network, we freeze the target critic network and the target actor network.

Because of deterministic policy, we need to come up with a way for exploration during training. In DDPG, we add noise in action. In the original paper of DDPG [18], the noise generator is Ornstein–Uhlenbeck. However, according to recent papers, normal Gaussian white noise work perfectly fine and it is less complex than Ornstein–Uhlenbeck noise.

Testing

After training, we remove the noise generator to have the best action for testing. New observations are passed directly to the actor network $\mu(s)$ to get the best action as we can see in the red line in 5.



Node 2		Node 2	Node 3	Node 4	
Search service	Proxy server	100%	0%	0%	
	Search server	50%	20%	30%	
...					

Fig. 6. The MEC network environment

IV. Implementation

We design the setup to demonstrate how RL works in a real MEC network. But first, we need to design a complete MEC network environment for the RL agent to learn. We also need to design the adapter between the environment and the RL agent to do translating the action to the network and calculating reward, pre-process the monitoring data for the RL agent. At last, we will do the implementation of the RL agent.

A. Simulation

In this work, we develop the network simulator from the simulator in [7]. It can simulate the network nodes, links, service chain with network functions and user traffic flows as we discussed in section III. However, we need to modify it to fit our problem. First, we design the network graph topology based on our real setup in the lab. Second, we use the traces that we got from section II to simulate the user traffic pattern. Third, we designed multiple services feature in the simulator so it can deploy multiple services in the simulator's network.

To implement the multiple-service feature, we need two data. We add up all user patterns for all services together to create a total arrival rate data-trace. Meanwhile, we create another data-trace that specifies how many percentages each service occupies at a specific time in the total arrival rate data-trace. After that, we put both traces as input to the network simulator. In the simulator, we designed a trace processor: for each incoming request from users, we based on the percentage data that we had to classify which service the request should subscribe to.

B. Metro edge computing network

In this work, we assume there are optical links between the nodes and the bandwidth is unlimited in these links. In real life, we need to make sure there are links among these network nodes. However, in this work, the main target is to optimize the computing resources capacity in each node such as CPU. In future work, the RL agent can be expanded to optimize both computing resources in each node and bandwidth resources in each link. The bandwidth can be assigned according to the request routing table at each node by using ONOS.

For the client generators and network services part, we already presented in section II. To implement the idea of the request routing table, we use the Nginx server load balancers with weights that we called request routers. Based on the weights assigned to the request router, we can split the incoming requests accordingly. Therefore, when updating the request routing table in each node from the action of the RL agent, we update the weights in these request routers. The request router is also designed as a container and deployed in all nodes in the network.

To monitor the drop requests and successful requests, round-trip time latency as well as node utilization, we used Prometheus. Prometheus will place the node exporter at each client container to scrape the metrics data and send them to the Prometheus server for storing and processing. This is where our adapter takes data and processes to give out the observations and reward to the RL agent.

To demonstrate the capacity of the node, we set a limit request rate threshold zone. All servers in the same node use the same limit request zone. If the number of requests coming is higher than the threshold, it will be discarded.

C. Adapter

a) Adapter for simulator: We mainly use the simulator for training, therefore, we need to design an adapter for the simulator to work with the RL agent. As we can see, the adapter containing a wrapper. This basically will process all the metrics we collect from the simulation environment and prepare the observation according to the observation space we defined in Section III-B. We also design the reward calculator based on the reward function. It calculates the successful request rate with service priority weights. This information will be fed into the RL agent. The action from the RL agent will be processed to form the request routing table and apply to the environment.

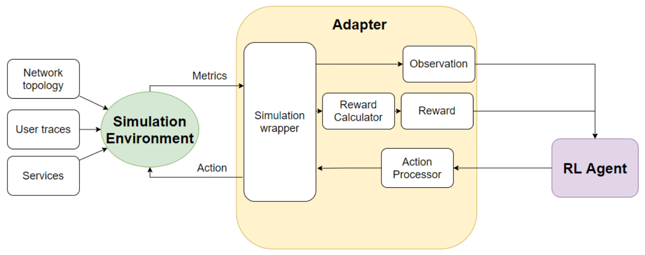


Fig. 7. The adapter design for simulator

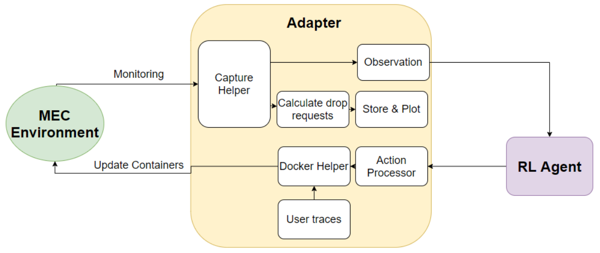


Fig. 8. The adapter design for running in MEC network

b) Adapter for MEC network: To run the RL agent work in the MEC network, we need to design an adapter to translate action into a real setup as well as process the information from the MEC network into the observation space. As we can see in Figure 8, the Adapter captured the monitor data using the "Capture helper" class. This Capture Helper read data such as client request rate and server usage in Prometheus server, calculate the average of them over Δ time, and translate them into observation space format for RL agent. On the other hand, it also passes the information about the dropped and successful request rate to calculate the drop rate and store them for evaluation. The Capture Helper read the information from the Prometheus server continuously for an interval of Δ time. Then it sends the observation after each interval time.

When the Adapter received the action from the RL agent, we need to translate it into the request routing table. First of all, the action is pre-processed in the Action processor. There are two jobs here: to convert all action values which are lower than x_{min} to 0 and to normalize the action according to each row in the request routing table to ensure the probability sum up to 1. After that, we update the weights in each request router at all nodes according to the action we got from the RL agent using docker. Moreover, we also need to update the user number in client containers according to the realistic trace.

D. RL Agent

To develop our DRL agent, the Stable baselines 3 library with DDPG Agent for reinforcement learning framework is used [19]. In this design, we set the specify parameters: the actor network and critic network is designed with 1 hidden fully connected layer with 64 nodes, activation function for all node is ReLU function, discount factor for reward function is $\gamma = 0.99$, soft update for target network $\rho = 0.0001$, learning rate $\alpha = 0.01$, buffer size $D = 10000$, noise variation for Gaussian white noise $\sigma = 0.2$, threshold $x_{min} = 0.1$ and learning start after 1 episode.

According to [20], the reinforcement learning training outcome depends a lot on the random seed. Therefore, with different seeds in the beginning as initial, we can have very different results in reinforcement learning. We should run multiple times with different seeds to find the best solution with a particular seed. In this implementation, we run training in each case 10 times and chose the best for testing.

V. Evaluation

A. Evaluation scenarios

We evaluate the RL agent with simulation and our real MEC network. First, we evaluate to check the compatibility between the simulation environment and the MEC network environment. After that, we evaluate the real MEC network topology with 3 nodes, each node with its computing capacity. We evaluate in 2 cases: the capacity is evenly distributed among 3 nodes and the capacity is unevenly distributed among 3 nodes. We also evaluate the case of multiple services (search, web, shopping) in our network with different priorities among them. Each client has its traffic pattern to its service. For simulation, we set the $\Delta = 100$ time steps and we set an episode that has 9600-time steps. For the MEC network, we set the $\Delta = 15$min and an episode is equal to a day with 96 Δ time intervals. For a particular training case, the number and location of the client are fixed and do not change over time. Noted that the Δ is the time the DRL agent observes the environment before giving action. After that, we further investigate the scalability of the RL agent when we increase the number of nodes in a ring topology. We also further increase the number of clients in the metro network to evaluate the ability to handle multiple clients of the RL agent.

B. Experiment procedure

In all scenarios, we always train in the simulation to get the well-trained model with different random seeds. Then we pick up the best model among them. After training, we save the model and test it (remove noise from action) in our MEC network to verify its ability in a real-life environment. This is way much faster than training in a real-life scenario.

C. Baseline algorithms

For benchmark, we use simple baseline algorithms: load balancing and shortest path algorithm. These baseline algorithms are also used as benchmark algorithms in [9] and [7].

a) Load balancing: Distribute the user requests to all node in the network which has capacity equally. This applies to all request routers in our network. This algorithm tries to balance the load among all nodes evenly to increase the throughput.

b) Shortest path: SP tries to minimize the latency as well as the number of network function instances. It works like a simplified version of first-fit algorithm. It places only 1 network function at a single node which closest to the traffic source. For example, it placed the first network function (reverse proxy server) at the node where the client requests come to the network and the next network function (server) at the closest node to it which has the capacity to deploy. It also favors the node which has a less number of network functions.

D. Simulation and MEC network evaluation

Before investigating the RL agent in the MEC network, we evaluate how accurate to transfer an RL agent trained in the simulator to a different environment: the MEC network. We evaluate the trained RL agent in both environments and check how it performs in both cases. We chose the scenario of the same capacity in the network. We set up the simulation with the computing resource capacity of $n_{cap} = 8$ for each node in the network meanwhile in the real MEC network we set the limit request zone for each node as 200 requests/sec. The pattern is the multiple-service pattern (which is the combination of all services). Figure 9 shows the reward return during the testing phase for an episode in both cases. With the same observation and action values, the rewards are quite comparable in both cases with a relatively small mean square error $MSE = 0.01615$. This shows that the simulation can present the real MEC network environment for evaluation. However, there is still some mismatch between the simulation and the real MEC network environment. This is due to our assumptions about the MEC network when we modeling the network. To be detailed, in our network model, we assume the server processing time is fixed meanwhile the processing time in the real MEC network varies. In real life, the server might discard the request based on other issues not only because of the limited request rate. Therefore, we need to tune the simulation parameters (for example, n_{cap}) to minimize the differences between the two environments. To sum up, given the same trace, configuration, network graph, and follow the same policy from trained RL agent, and the good choice of simulation parameters, the reward function behavior for both cases are similar to each other.

E. MEC network evaluation

In the MEC network, we have a 3-node topology as we have seen in section IV. Firstly, we evaluate our model in two scenarios: distribute capacity equally and distribute capacity unequally. Secondly, we evaluate our model with multiple services with different priorities. In both cases, we extensively train the model in simulation before applying the model in the MEC network.

a) Different capacity distribution: In the capacity distribute equally scenario, we set the capacity in all nodes with the same capacity of 200 requests/sec. We investigate with multiple-service trace with one service. We deployed the client containers in two nodes in the network as we can see in Figure 10.

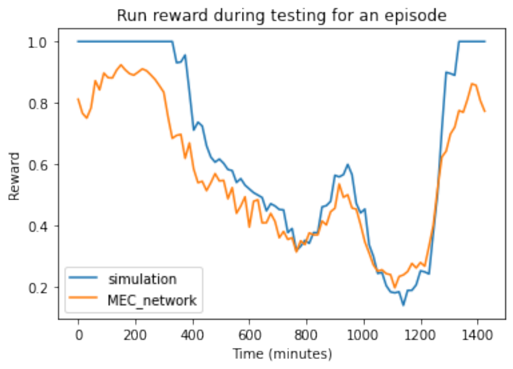


Fig. 9. Reward function behavior for an episode in simulation and MEC network

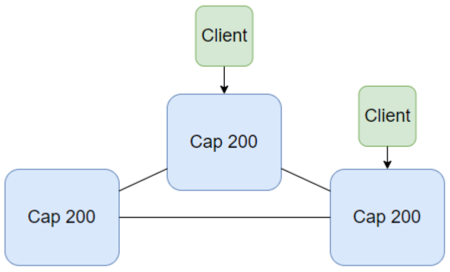


Fig. 10. The balance capacity scenario

We peak at the solution of the RL agent at the time where the traffic is the highest in Figure 11. As we can see, it learns to put the traffic equally to all servers to fully utilize the resources in all nodes. This is a good sign that our RL agent learns and can work in the MEC network environment.

In Figure 12, we can see that within a 1-day test, our RL agent performs better than other benchmark algorithms. The

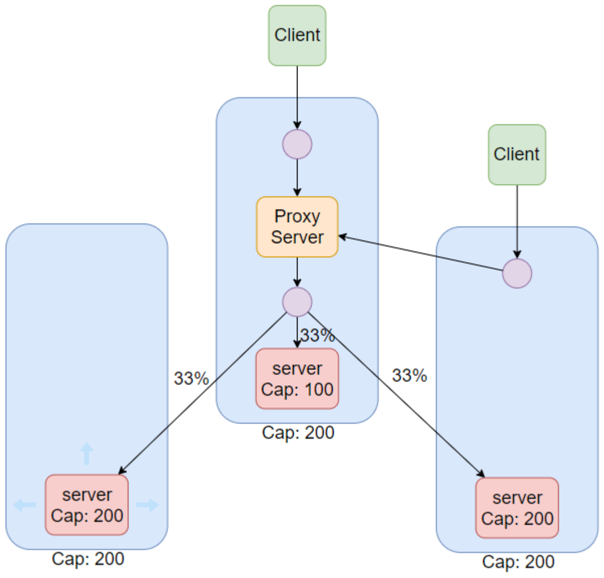


Fig. 11. The RL solution for balance capacity scenario

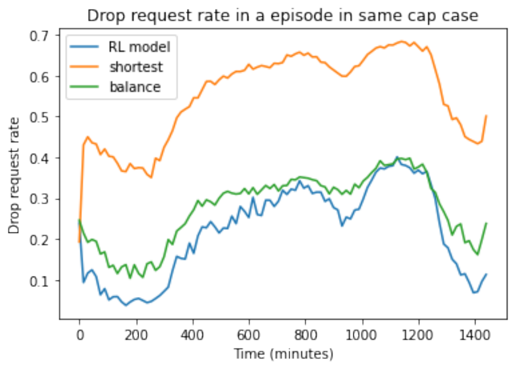


Fig. 12. The drop request rate in a day tested in real MEC network environment with balance capacity scenario

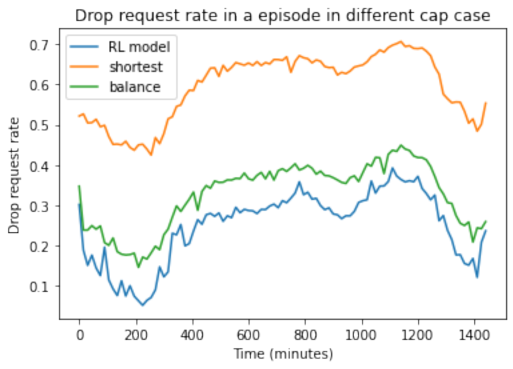


Fig. 14. The drop request rate in a day tested in real MEC network environment with unbalance capacity scenario

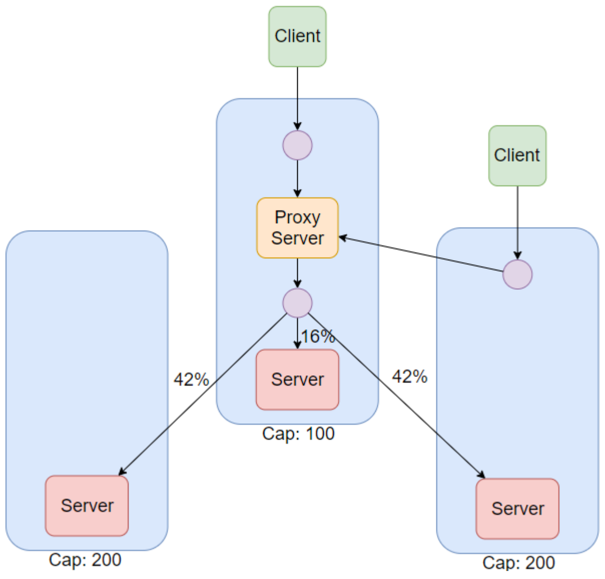


Fig. 13. The RL solution for unbalance capacity scenario

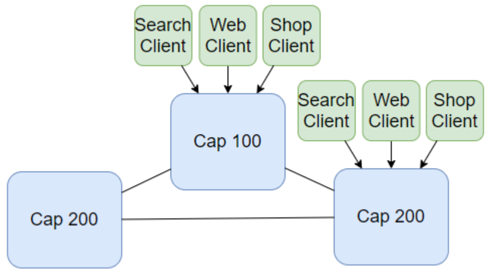


Fig. 15. The multiple service scenario

shape of the drop request rate is quite similar to the trace pattern. It is understandable since the higher the request rate from the user, the faster the node capacity gets full and the higher the drop request rate.

In the capacity distribution unequally scenario, we decrease the capacity in node 3 with only 100 requests per second as we can see in Figure 13. We also look at the solution of our model when the traffic rate is the highest. It shows that the RL agent distributed the traffic accordingly to the node capacities ratio.

In Figure 14, it also proves that our RL model outperforms other baseline algorithms. Because the capacity distribution is unevenly among the nodes, the load balance algorithm is not as good as the one in Figure 12. The results in both cases proves that our RL model can adapt to different types of capacity distribution scenarios.

We also notice that the average drop request rate (in our RL algorithm) increases when we reduce the total capacities in the network from 21.9% in the balance case to 24.7% in the unbalanced case. This happened because we reduce the capacity in the node. Therefore, requests get discarded.

b) Multiple services with priority: In this experiment, we use multiple services in our training then applied them in the MEC network. We use the scenario of unbalanced capacity distribution. As we can see in Figure 15, the clients which request three services (search service, web service, and shop service) are deployed in node 3 and node 2. The priority we set for search service, web service, and shop service are 7, 2, and 1, respectively.

Figure 16 shows the drop request rate for different services in our MEC network. It shows that the RL agent learns how to prioritize the higher priority service. In this case, even though the overall drop request rate of our RL model is not as good as the load balance algorithm, the highest priority service (search service) does perform significantly better with the lowest drop rate request. Noted that the other services suffer higher dropped rate. In the case of shop service, it even performed worse than the load balance algorithm. To sum up, the RL agent learns to prioritize to serve the important service.

F. Scalability evaluation

To test the scalability of the RL agent, we increase the network size in our simulation and test if our RL agent can provide a good solution in such a case. We also evaluate the case when the number of clients increases in the network given that the network also increases capacity to provide for those clients.

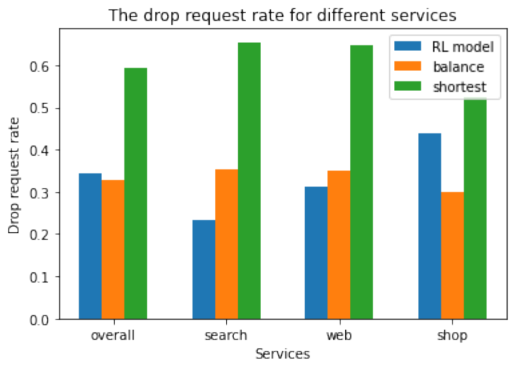


Fig. 16. The drop request rate in multiple services case

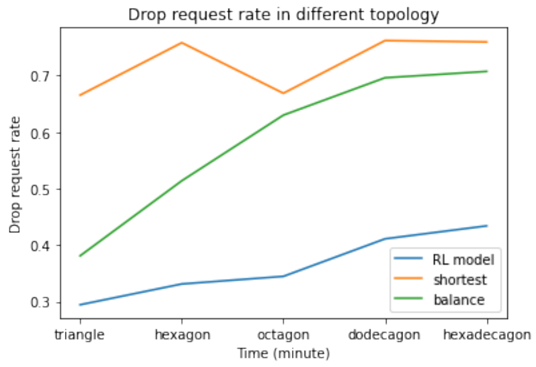


Fig. 17. The drop request rate when increase the network size

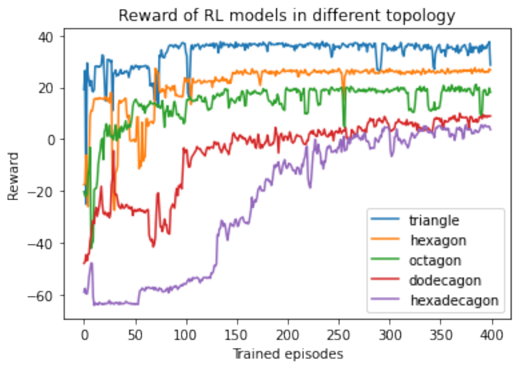


Fig. 18. The RL reward convergence during training for different network size

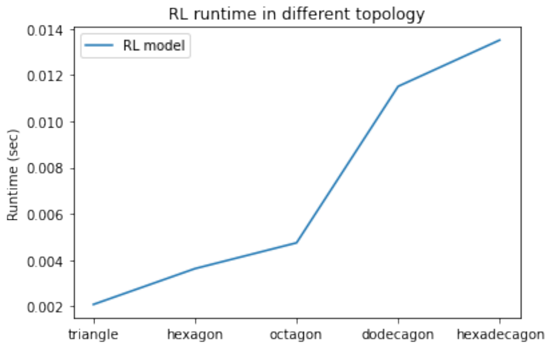


Fig. 19. The average run time of RL model in different network size

a) Increase network size: We increase the network size from 3 nodes to 6, 8, 12, and 16 nodes. When we scaling, each node in the network has a random capacity but the total capacity of all nodes is the same. The client number is keeping the same in all cases. As we can see in Figure 17, the RL model still outperforms other baseline algorithms, the drop rate increases when we increase the ring size. This is understandable since the more node, the more action space the RL model has. Therefore it will take more time for the RL agent to learn to find the best result. It might get stuck to the local minima point and take more training time to explore a better solution. As we can see in Figure 18, the more complex topology takes more time and receive less reward in training to converge than the simple one. This explained why the complex ring size got a higher dropped request rate. We also noted that the running time for the RL agent increases as the network ring size increases as we can see in Figure 19. This is because when we increase the network size, the increase action space, we increase the parameters in actor neural network and critic neural network in RL model. Therefore, it takes a longer time to make a decision. But still, the decision is in a fast manner (milliseconds) compare to the monitoring interval $\Delta = 15$ minutes.

b) Increase clients: In the last case, we would like to investigate the case when the number of aggregated traffic in the network increases from 2 to 8 in the 8-node network. The client placement in the network is random. The total capacity in the network increase as the number of client increase to make sure we have enough capacity for such clients. However, the capacity distribution in the metro network is random. For each case, we have to retrain the model and test them in the simulation. As we can see in Figure 20, the most important thing is that the RL model is to be able to keep the drop request rate low despite the number and location of clients as long as we provide enough total capacity in the metro network. On the other hand, other algorithms depend on the location of clients and how the total capacity is distributed in the network. For example, in the 4 clients case, the shortest path algorithm performs well because it has the closest neighbor node which has a lot of capacity. Therefore it can significantly reduce the drop request rate. But in most other cases, it poorly perform. Noted that the RL agent needed to be trained for each number client cases because with different number of client and different location of clients, the observation is different. The RL agent needed to learn again based on this information to optimize the request routing table for each cases. In conclusion, RL agent learns how to route the traffic for all the clients to be served and get the lowest drop request rate despite where the client requests come from and how many request sources.

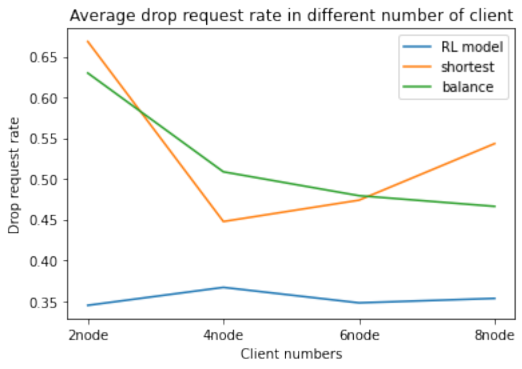


Fig. 20. The drop request rate when increase number of clients

VI. Conclusion

We have demonstrated that the reinforcement learning-based algorithm can be used to solve the resource utilization problem. It proves that with the good traffic arrangement solution from RL agent, the dropped request rate can be minimized. This achievement together with low latency from edge computing technologies can give a good quality of service to end-users in MEC network for services in 5G and beyonds.

VII. Acknowledgements

I would like to thank my supervisor, professor Nicola Calabretta for the guidance during my thesis. I also would like to thank my daily supervisors, Bitao Pan and Xiaotao Guo, and Dr. Fulong Yan for fruitful discussions and advice.

References

[1] C. International, "Cisco Visual Networking Index: Forecast and Methodology Cisco Visual Networking Index: Cisco Visual Networking Index: Forecast and Methodology," *Forecast and Methodology*, pp. 6–17, 2017.
[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
[3] J. Gil Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.
[4] R. Gu, Z. Yang, and Y. Ji, "Machine learning for intelligent optical networks: A comprehensive survey," *Journal of Network and Computer Applications*, vol. 157, no. December 2019, p. 102576, 2020. [Online]. Available: https://doi.org/10.1016/j.jnca.2020.102576
[5] D. Rafique and L. Velasco, "Machine learning for network automation: Overview, architecture, and applications," *J. Opt. Commun. Netw.*, vol. 10, no. 10, pp. D126–D143, Oct 2018. [Online]. Available: http://jocn.osa.org/abstract.cfm?URI=jocn-10-10-D126
[6] P. T. A. Quang, Y. Hadjadj-Aoul, and A. Outtagarts, "A deep reinforcement learning approach for vnf forwarding graph embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1318–1331, 2019.
[7] S. Schneider, A. Manzoor, H. Qarawlus, R. Schellenberg, H. Karl, R. Khalili, and A. Hecker, "Self-driving network and service coordination using deep reinforcement learning," in *2020 16th International Conference on Network and Service Management (CNSM)*, 2020, pp. 1–9.
[8] S. Dräxler, S. Schneider, and H. Karl, "Scaling and placing bidirectional services with stateful virtual and physical network functions," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 123–131.
[9] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 1871–1879.
[10] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.
[11] "January 2021 web server survey," Jan 2021. [Online]. Available: https://news.netcraft.com/archives/2021/01/28/january-2021-web-server-survey.html
[12] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
[13] "What is a container?" [Online]. Available: https://www.docker.com/resources/what-container
[14] F. Kooti, K. Lerman, L. M. Aiello, M. Grbovic, N. Djuric, and V. Radosavljevic, "Portrait of an online shopper: Understanding and predicting consumer behavior," *WSDM 2016 - Proceedings of the 9th ACM International Conference on Web Search and Data Mining*, pp. 205–214, 2016.
[15] I. Ullah, G. Doyen, G. Bonnet, and D. Gaïti, "A survey and synthesis of user behavior measurements in P2P streaming systems," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 3, pp. 734–749, 2012.
[16] M. Taghavi, A. Patel, N. Schmidt, C. Wills, and Y. Tew, "An analysis of web proxy logs with query distribution pattern approach for search engines," *Computer Standards and Interfaces*, vol. 34, no. 1, pp. 162–170, 2012.
[17] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng, "Understanding user behavior in large-scale video-on-demand systems," *Proceedings of the 2006 EuroSys Conference*, pp. 333–344, 2006.
[18] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.
[19] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.
[20] A. Irpan, "Deep reinforcement learning doesn't work yet," https://www.alexirpan.com/2018/02/14/rl-hard.html, 2018.