A Reinforcement Learning-Based Admission Control Strategy for Elastic Network Slices

Zhouxiang Wu, Genya Ishigaki, Riti Gour, and Jason P. Jue
Department of Computer Science, The University of Texas at Dallas, Richardson, Texas 75080, USA
Email: {Zhouxiang.Wu, gishigaki, rgour, jjue}@utdallas.edu

***Abstract*—This paper addresses the problem of admission control for elastic network slices that may dynamically adjust provisioned bandwidth levels over time. When admitting new slice requests, sufficient spare capacity must be reserved to allow existing elastic slices to dynamically increase their bandwidth allocation when needed. We demonstrate a lightweight deep Reinforcement Learning (RL) model to intelligently make admission control decisions for elastic slice requests and inelastic slice requests. This model achieves higher revenue and higher acceptance rates compared to traditional heuristic methods. Due to the lightness of this model, it can be deployed without GPUs. We can also use a relatively small amount of data to train the model and to achieve stable performance. Also, we introduce a Recurrent Neural Network to encode the variable-size environment and train the encoder with the RL model together.**

***Index Terms*—Network Slicing, Admission Control, Reinforcement Learning, Proximal Policy Optimization**

I. Introduction

5G communication networks are expected to provide service not only for traditional mobile communication but also for a wide range of other heterogeneous applications. Network slicing is an attractive technology for addressing this problem. Network slicing enables the creation and deployment of multiple virtualized network slices over a common physical infrastructure, with each slice consisting of a set of isolated virtual computing and networking resources that are capable of satisfying the unique requirements of specific applications. With the help of network function virtualization (NFV) and software-defined networking (SDN) [1] in 5G networks, swiftly instantiating network slice and allocating resources becomes possible.

Network slices may be characterized as either static or elastic. In a static slice, the amount of resources provisioned for the slice is fixed and cannot change during the service time of the slice. On the other hand, in an elastic slice, the amount of resources provisioned for the slice may change over the lifetime of the slice. With elastic slices, the network can accommodate a greater number of slices compared to the case in which each slice is allocated its maximum peak bandwidth; however, some spare capacity needs to be available to allow slices to dynamically transition to higher rates when needed. Thus, in addition to a dynamic resource allocation scheme for network slicing, a slice admission control policy must be deployed to ensure sufficient quality of service for elastic slices while maximizing revenue for the network operator.

Several slice admission policies [2] have been developed in recent years; however, they have various drawbacks. Also, existing works do not explicitly consider the elastic slice requests. We propose a novel reinforcement-learning-based admission control policy that utilizes foresight and attempts to optimize revenue in the long-term instead of only considering immediate rewards. Reinforcement learning is goal-directed learning and has been applied in many artificial intelligent controllers for games. This paper adopts a reinforcement learning technology named Proximal Policy Optimization (PPO) [3], which is the default method used in OpenAI Gym. We employ this technique to implement an admission controller that gains more revenue and a higher acceptance rate in the long term compared to existing heuristic methods. Our RL controller policy does not require prior knowledge of the request arrival process or the bandwidth distribution for elastic slice requests, and the policy can be deployed by the network operator independently. Despite lacking traffic load information, the controller can reject some requests to avoid congestion in the future based only on information about remaining bandwidth and the set of current slice requests in service. A critical prerequisite of deep learning is that the neural network input is a tensor with a fixed size; however, the number of slice requests in service changes from time to time. We use one variant of Recurrent Neural Networks (RNNs) [4] to transform the variable-size environment state description into a fixed-size vector. We test this model on a single link with various types of slice requests. Only about 100 pieces of data are needed to train our model. Thus, this model's overhead is limited, and in the test phase, the time required for each decision is around 3 msec.

The remainder of the paper is organized as follows. Section II defines the network slice requests and a pricing strategy. In Section III, we briefly introduce RL and RNN to facilitate understanding of the proposed model. Section IV describes the model in detail and includes discussion on training, validation, and test phases. In Section V, we show the procedure of producing data, and we evaluate the model's performance. Finally, we discuss related work in Section VI and conclude in Section VII.

II. Slice Request and Pricing Strategy

A. Slice Request

In general, a network operator receives two types of requests from slice operators: deployment requests and scale requests. A deployment request R^d defines a blueprint of a new slice

TABLE I
PREDEFINED CLASS

	Class I		Class II
b_0	0%	100%	50%
b_1	100%	0%	0%
b_2	0%	0%	50%

by specifying a slice topology, each source-destination pair (s, d) in the topology, a holding time h, the initial bandwidth requirement $b_{sd}(t_0)$, and the estimated distribution of future bandwidth $\hat{B}_{sd}$, where t_0 represents the time when the new slice will be deployed into the physical network. In this work, we assume that a slice operator chooses one of several predefined classes to specify the distribution of bandwidth $\hat{B}_{sd}$. Table I shows a simple example of predefined slice classes, where each slice specifies the percentage of its holding time during which it expects to use one of three possible bandwidth levels (b_0, b_1, and b_2). The type of slice (elastic or inelastic) is specified by this distribution of future bandwidth; for example, Class I in Table I corresponds to inelastic slice requests while Class II corresponds to elastic slice requests. Also associated with a set of slice classes is a pricing structure, which determines the price per unit bandwidth for each class at each bandwidth level. A scale request R^s, which is issued only by elastic slices, indicates a request for increase or decrease of resources assigned to the slice and specifies a new bandwidth allocation $b_{sd}(t_j)$ at next time step $t_j (t_j \in [0, h])$. In most cases, each slice operator has its own unique policy, based on its traffic pattern, that determines when it wants to send a scale request to the network operator. The definitions of the elastic slice classes will depend on the traffic being carried by the slices. A slice operator may estimate the intensity or distribution of traffic on a slice link as a function of time and map this to a discrete set of pre-determined bandwidth levels. The slice operator may either indicate the fraction of time at each bandwidth level or may provide more detailed distributions for the time spent at a given level. The network operator can then use this information to determine the parameters of the elastic slice classes that it will support.

B. Pricing Strategy

In order to model the profit of a network operator, the revenue from providing bandwidth to slices and the penalty of rejecting scale requests from the existing slices are defined as follows. When receiving a deployment request, the admission control module of a network operator decides if the request should be accepted or not. A rejection of a deployment request implies that the physical network does not have enough resources to serve the new slice. This scenario does not give the network operator an explicit penalty, since there is no commitment to the slice by the operator. However, the network operator loses opportunities to profit by rejecting too many new slices; i.e., the low utilization of physical infrastructure.

On the other hand, the admission module should not reject a scale request because the acceptance of a deployment request

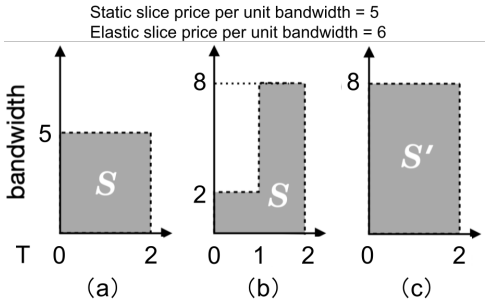

Fig. 1. Slice pricing: (a) Static slice, price = 50; (b) Elastic slice, price = 60; (c) Static slice with peak rate, price = 80.

could be seen as a contract that the operator promises to accommodate the future scaling of the slice. However, in reality, it is possible for a network operator to have insufficient resources to provision additional resources for a scale request if too many slices attempt to scale up their bandwidth at the same time. This unsuccessful scaling may incur a severe penalty, as it is a contract violation. An exception to the penalty for unsuccessful scale requests is the case in which the requested bandwidth $b_{sd}(t_j)$ in a scale request diverges from the reported distribution $\hat{B}_{sd}$. Note that the distribution of future traffic could be based on a rough estimation; however, inaccurate estimation of expected bandwidth levels by a slice operator may cause more rejections of the slice operator's scale requests. Therefore, there is an incentive for a slice operator to provide more accurate information to the network operator [5].

The revenue of a slice is calculated by the amount of bandwidth used by the slice over its holding time h. The price function p is a function of the predefined bandwidth classes. Suppose there is an elastic slice request and an inelastic slice request. We observe the following principles: (1) When the total bandwidth used by both slices is the same, an elastic slice should be charged slightly more per unit of bandwidth than a slice with static bandwidth (See Fig. 1-a, b), since the dynamics in resource allocation for an elastic request results in increased complexity for the network operator when pre-planning the provisioning. (2) When a slice operator opts for a dynamic slice with reduced overall bandwidth as opposed to a static slice provisioned for the slice's peak bandwidth requirement, the total amount charged should be reduced since the freed resources may be used by the network operator to accommodate additional slices and earn additional revenue (See Figure 1-b, c). In order for elastic slicing to be worthwhile to the network operator, the total revenue generated from elastic slices at a lower revenue per slice (but higher revenue per unit of reserved bandwidth) must be greater than the revenue generated by accommodating only fixed slices at a higher revenue per slice.

To discourage slice operators from issuing scale requests for amounts of bandwidth that significantly exceed its bandwidth distribution profile, we alleviate punishment by a factor $\alpha \in [0, 1]$ which is the similarity between history distribution and

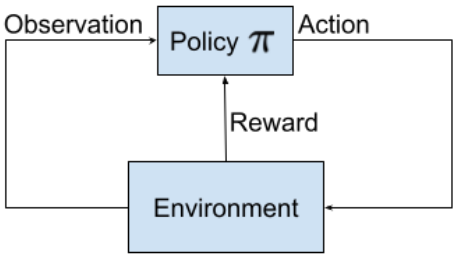

Fig. 2. RL Procedure

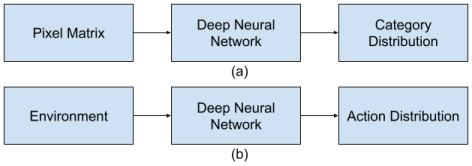

Fig. 3. RL compared with classification

reported distribution. Kullback–Leibler divergence $D_{KL}$is a measure of how one probability distribution is different from a second, reference probability distribution. Since the higher D_{KL} is, the larger difference between the two distributions. We use a variant of KL-divergence as the value of α, which should be between 0 and 1 and decrease with KL-divergence increasing. We choose $-exp^{-D_{KL}}$. We multiply this value with basic reward as the punishment for rejection of elastic scale requests.

III. Background on deep RL and RNN

In this paper, we use a policy-based RL method as shown in Fig. 2. The controller employs a policy π, which takes the environment state as the input and whose output is the selected action. Rewards are used to determine the best policy.

A. Policy Evaluation

Instead of following a traditional approach which utilizes RL with a Markov Decision Procedure, we choose to approach RL from the deep learning perspective. In traditional deep learning applications, such as image classification, the input may be a pixel matrix, and the output is a predicted label distribution, as shown in Fig. 3 (a). The procedure for deep reinforcement learning is similar to classification. The core part in both methods is a Deep Neural Network (DNN). As Fig. 3 (b) shows, the input of a DNN consists of environment state features, and the output is the action distribution. In classification, there is a ground truth label for each image. We can compute the difference between the output of the model and ground truth distribution precisely, then we optimize the DNN function based on the difference, which usually is called loss. However, in RL, we do not have ground truth for each environment state, so we cannot directly calculate the loss.

Instead, we use the following method to evaluate a policy π_θ, where θ is the parameter set for DNN. First, we use π to interact with the environment and record the process as a trajectory τ of the tuples. Each tuple contains state s_i, action a_i, and reward r_i. Suppose the controller ends the game within time T. Thus, the sequence is represented as $\{(s_1, a_1, r_1), (s_2, a_2, r_2), ..., (s_T, a_T, r_T)\}$. The total reward for the τ is given by:

$$R_\theta(\tau) = \sum_{t=1}^{T} r_t. \tag{1}$$

The cumulative reward is a reliable way to evaluate the policy. However, even with the same actor, R_θ could be different from time to time due to the environment's randomness. Thus, the expectation of the cumulative reward $\overline{R_\theta}$ is more suitable to evaluate the policy. Suppose we use the policy π_θ and play the game N times to obtain $\{\tau^1, \tau^2, ...\tau^N\}$. We calculate the expected cumulative reward as follows:

$$\overline{R_\theta} = \sum_\tau R_\theta(\tau) P(\tau|\theta) \approx \frac{1}{N} \sum_{n=1}^{N} R(\tau^n). \tag{2}$$

The gradient of the expected cumulative reward can be calculated by:

$$\begin{aligned} \nabla \overline{R_\theta} &= \sum_\tau R(\tau) \nabla P(\tau \mid \theta) \\ &= \sum_\tau R(\tau) P(\tau \mid \theta) \nabla \log P(\tau \mid \theta) \\ &= E_{\tau \sim P_\theta(\tau)}[R(\tau) \nabla \log P_\theta(\tau)]. \end{aligned} \tag{3}$$

Algorithm 1 Policy Gradient

1: Initialize the policy parameter θ at random and learning rate μ
2: **while** Not Converge **do**
3: Generate trajectories $\{\tau^1, \tau^2, ...\tau^N\}$ on policy π_θ
4: Calculate expectation of cumulative reward $\overline{R_\theta}$
5: Update policy parameters by $\theta \leftarrow \theta + \mu \nabla \overline{R_\theta}$
6: **end while**

B. Proximal Policy Optimization

Each batch of trajectories in the policy gradient updates the parameters θ only once. We call this type of policy-based method the on-policy method. The on-policy feature is a problem when trajectories are challenging to produce, which is the case for most realistic situations. The goal of Proximal Policy Optimization (PPO) is to use the trajectories from policy $\pi_{\theta'}$ to train θ with fixed θ'. Thus, we can reuse the trajectories to save resources. In the policy gradient, the gradient of expected cumulative rewards is defined as follows:

$$\nabla \overline{R_\theta} = E_{\tau \sim P_\theta(\tau)}[R(\tau) \nabla \log P_\theta(\tau)]. \tag{4}$$

With the help of importance sampling [6], the new gradient function could be represented as follows:

$$\nabla \overline{R_\theta} = E_{\tau \sim p_{\theta'}(\tau)} \left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)} R(\tau) \nabla \log p_\theta(\tau) \right]. \tag{5}$$

However, P_θ cannot diverge too far from $P_{\theta'}$ since the variance of expectation of cumulative reward increases with the difference between P_θ and $P_{\theta'}$. PPO employs KL divergence between P_θ and $P_{\theta'}$ to constrain the difference. We introduce the main idea of PPO and omit the implementation details due to paper length restrictions.

C. RNN

Recurrent neural networks (RNNs), are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. RNNs have the following advantages.

1) Possibility of processing inputs of any length
2) Model size not increasing with the size of input
3) Computation taking into account historical information
4) Weights are sharing across time

There are two well-known variants of RNNs, which are long short-term memory (LSTM) and gated recurrent unit (GRU) [7]. This paper takes the advantage of RNNs and employs GRU as an encoder of variable-size features.

IV. Proposed Models

This section discusses how to extract features from the environment, the RNN encoder, the DNN structure, and the objective function.

A. Environment Description

Feature engineering is a critical cornerstone for successful machine learning. Thus, the accurate description of environment features is crucial for reinforcement learning. We divide the environment features into two parts. The first part includes the requests in service $\{R_1, R_2, ..., R_N\}$, where N is the number of requests in service. Each request is a tuple $R_i = (B_i, T_i, R_i)$ where B_i is the request bandwidth, T_i is the remaining service time, and R_i is the request type in the one-hot encoding format. $R_i \in \{(1,0,0),(0,1,0),(0,0,1)\}$, where (1,0,0), (0,1,0) and (0,0,1) corresponds to the static slice deployment request type, the elastic slice deployment request type, and the elastic slice scale request type, respectively. Thus, each R_i can be represented by a 1×5 vector. For example, there are three requests in service. The first request R_1 is a static deployment type whose request bandwidth is two units, and the remaining service time is 1 second. The second request R_2 is an elastic deployment type whose request bandwidth is 1 unit and remaining service time is 3 seconds. The third request R_3 is elastic scale type whose bandwidth B_3 is 8 units and whose service time T_3 is 0.05 second. These three requests can be represented as the following sequence: $\{(2,1,1,0,0),(1,3,0,1,0),(8,0.05,0,0,1)\}$, and the RNN module can take one row at a time. The second part includes the value of the remaining bandwidth, slice request, which is represented in a 1×5 vector as above, except replacing the remaining service time with the total service time. As a result, we encode the second part as a 1×6 vector.

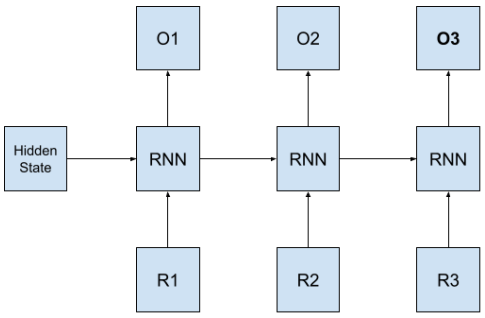

Fig. 4. Encoding Procedure

B. RNN Encoder

Since the number of slice requests in the system changes from time to time, we utilize RNN to encode requests in service as a fixed-length vector. To manifest the essential principle, we choose a single-level RNN module to explain the procedure of transforming from requests to a fixed-size vector.

In Fig. 4, we have three service requests $\{R_1, R_2, R_3\}$, and three corresponding outputs: $\{O_1, O_2, O_3\}$. Hidden State (HS) is initialized to the zero vector. We concatenate a request R_i and a hidden state from the last step HS_{i-1} and put the result into the RNN model to get output O_i. We use O_3 as input to another fully connected layer. The output of this layer is the code of these three requests. In this way, we transform variable-size codes of requests into a fixed-size vector, and the vector contains information of all requests. With the help of PyTorch, we could effortlessly replace RNN with other variants, such as LSTM, GRU, and multi-level variants. The output length is a hyper-parameter, which we set to 10 in our experiment. Then, we concatenate the final layer's output and the second part of the environment features into a 1×16 vector. We utilize this vector as input of the DNN model.

C. DNN Model

We use the Proximal Policy Optimization (PPO) algorithm to train an intelligent controller. As mentioned in Section III, the heart of PPO is a deep neural network model. The input to this model consists of features extracted from the environment, as described in the subsection B, and the output is an action distribution. We introduce two more functions. One of them is ReLu, and another is Softmax. ReLu is the most popular activation function in deep learning and is shown as follows:

$$h(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

Softmax could convert any list of real numbers to a distribution. For example, if the input of Softmax is a list $[a_1, a_2, .., a_N]$, then the output is a list $[\sigma_1, \sigma_2, .., \sigma_N]$ where $\sigma_i = \frac{e^{a_i}}{\sum_{n=1}^{N} e^{a_n}}$. The output of Softmax is a discrete distribution. We employ two fully connected layers with ReLu and apply the Softmax function for the output layer. The entire procedure is shown in Fig. 5.

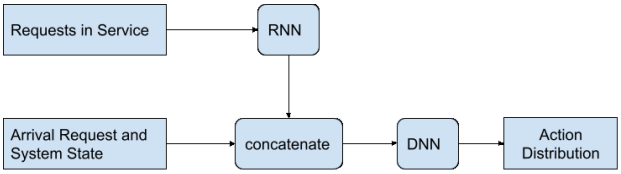

Fig. 5. Complete Procedure

D. Objective Function

We use the policy π_{θ^k} to interact with the environment and obtain N trajectories, which contains T_n tuples for each trajectory n. We show the first trajectory τ^1 as follows:

$$\tau^1 : \{(s_1^1, a_1^1, R(\tau^1)), (s_2^2, a_2^2, R(\tau^2))...(s_2^{T_1}, a_2^{T_1}, R(\tau^{T_1}))\}.$$

The remaining trajectories $\{\tau^2, \tau^3...\tau^N\}$ have the exact same format as τ^1. To increase accuracy, we replace $R(\tau^n)$ with $A_\theta(s_t^n, a_t^n)$, which is defined as follows with the discount factor $\gamma \in [0, 1]$:

$$A^{\theta^k}(s_t^n, a_t^n) = \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n. \tag{7}$$

We define two more values, $J^{\theta^k}(\theta)$ and $J_{PPO}^{\theta^k}(\theta)$.

$$J^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \frac{p_\theta(a_t \mid s_t)}{p_{\theta^k}(a_t \mid s_t)} A^{\theta^k}(s_t, a_t) \tag{8}$$

$$J_{PPO}^{\theta^k}(\theta) = J^{\theta^k}(\theta) - \beta KL\left(\theta, \theta^k\right) \tag{9}$$

We use the Algorithm 2 to update parameter θ and generate policy π_θ.

Algorithm 2 PPO

1: Initialize the policy parameter θ^0, β, KL_{max} and KL_{min}
2: **for** each iteration **do**
3: Using θ^k to interact with the environment to collect $\{s_t, a_t\}$and compute advantage $A^{\theta^k}(s_t^n, a_t^n)$
4: Find θ optimizing $J_{PPO}^{\theta^k}$
5: **if** $KL\left(\theta, \theta^k\right) > KL_{max}$ **then**
6: increase β
7: **end if**
8: **if** $KL\left(\theta, \theta^k\right) < KL_{min}$ **then**
9: decrease β
10: **end if**
11: **end for**

While the policy π_{θ^k} interacts with the environment, we employ the exploit strategy, which samples actions from the output action distribution. While this model actually makes decisions, we apply a greedy strategy, which takes the action with the highest possibility.

V. Performance Evaluation

In this section, we introduce our experimental setup and compare the performance of the proposed scheme with First-In-First-Out (FIFO) and two random methods.

TABLE II
Predefined Class

Scenario No.	ID	bandwidth	λ	μ	distribution
Scenario 1	0	[2]	1	1	[1]
	1	[5]	1	1	[1]
	2	[2,8]	1	1	[0.5,0.5]
Scenario 2	0	[2]	0.33	1	[1]
	1	[5]	0.5	0.5	[1]
	2	[2,8]	1	0.33	[0.5,0.5]
	3	[2,4,6]	0.25	0.25	[0.5,0.2,0.3]
	4	[1,9]	0.33	0.33	[0.5,0.5]

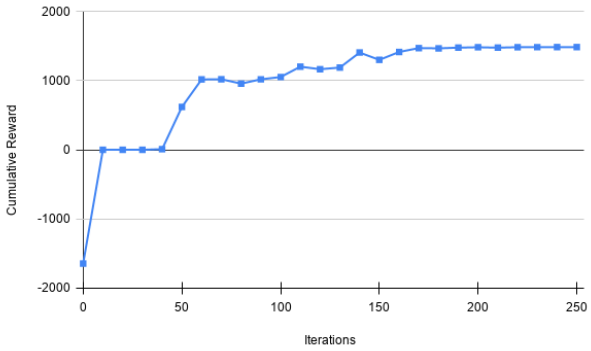

Fig. 6. Cumulative Reward in Scenario 1 Traning

A. Simulated Data

We consider two scenarios, with each consisting of a set of possible elastic and inelastic request types. Each type contains four elements: ID, request bandwidth, arrival rate λ, service rate μ, and reported distribution. Each request arrival event follows a Poisson process with parameter λ, and service time follows an exponential distribution with parameter μ. We define a specific numerical value for each element in Table II.

For the elastic type, the amount of time it stays in a given state is sampled from an exponential distribution with parameter 50. Thus, the average time an elastic request stays in one state is 0.02 units of time, which means that the switches between states occur frequently during one elastic request.

B. Training Phase

In Scenario 1, on average, requests require 12 units of bandwidth, and we set the system to contain ten units of bandwidth to stress the controller to learn an effective strategy rather than accept all requests. We generate ten trajectories and generate 20 requests for each type and each trajectory to train the model. In the validation phase, each type of request has 300 requests, and we produce five trajectories. We employ the validation data to locate the best model during training. To be more accurate, we produce 300 of each type request and ten trajectories in the test phase. Fig 6. shows average validation cumulative rewards during training. After 200 iterations, the performance of the model converged.

TABLE III
RESULT SUMMARY

No.	Criteria	RL	FIFO	Rand1	Rand2
1	Revenue	1442.431	1408.226	599.603	1335.188
	AR	0.630	0.432	0.135	0.411
	AR for SD	0.566	0.588	0.350	0.558
	AR for ED	0.313	0.728	0.497	0.682
	AR for ES	0.684	0.400	0.071	0.379
2	Revenue	9472.215	9317.903	4081.671	8392.998
	AR	0.683	0.653	0.405	0.621
	AR for SD	0.758	0.749	0.493	0.712
	AR for ED	0.851	0.869	0.593	0.810
	AR for ES	0.666	0.632	0.378	0.601

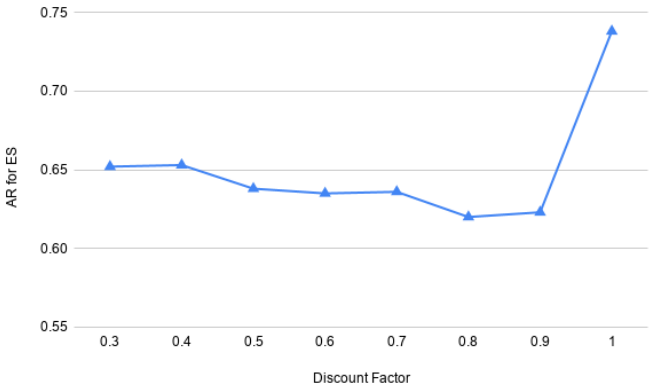

Fig. 7. AR for ES change with γ in Scenario 2

In Scenario 2, the basic setup is similar to Group 1. We change the total bandwidth to 20 units to test the model's performance in a relatively rich environment. There is a potential risk that the agent may accept requests when there are insufficient remaining resource to fulfill the request. In this situation, we reject the request and give a harsh punishment to the agent to indicate that the behavior is illegal.

C. Result

We compare our model with three other simple methods that we have developed. The first method is FIFO which rejects requests only when the system cannot provide the requested resource. The second method is to randomly reject requests according to the ratio of residual bandwidth in the system to total bandwidth. The third method has a ten percent chance of rejecting the request. Table III shows the results, where Rand1 and Rand2 correspond to the second and third methods. AR is the request acceptance ratio and AR for SD, AR for ED, and AR for ES corresponds to the acceptance ratio of static deployment request type, the acceptance ratio of elastic deployment request type, and the acceptance ratio of elastic scale request type, respectively.

During the experiment, we find that the discount factor γ is a critical hyper-parameter. We need to set γ to 0.9 to maximize the model's performance in a relatively lower total bandwidth setup. On the other hand, when the system has higher bandwidth, we could set the discount factor to 0.4 or 0.5 to obtain higher cumulative revenue. A higher value of γ indicates that the model places a greater emphasis on future benefits. When we set γ equal to 1, the model would focus on accepting elastic scale requests to obtain the highest value for AR for ES, as shown in Fig 7. Our environment is different from the typical RL environments because, unlike a game, our environment does not contain a beginning and an end; thus, γ selection is different from the traditional RL.

Generally, based on the experimental results, we conclude that our model is able to utilize foresight in order to avoid punishment by rejecting deployment requests before the bandwidth is fully occupied; however, the overall revenue and acceptance ratios increase despite these rejections.

VI. Related Work

In [2] the authors describe the four main objectives of slice admission control strategies, including revenue optimization, quality of service (QoS) control, inter-slice congestion control, and slice fairness assurance. The authors also divide the strategies into five groups: FIFO, priority-based, random, greedy, and optimal. We compare our model with FIFO and two random methods. The priority-based methods always require prior knowledge of ranking for the slice types. The greedy policy makes decisions greedily based on history. The optimal policy is not realistic in our situation due to its time-consuming nature.

VII. Conclusions

We propose a lightweight deep RL model for making admission control decisions for elastic slices. The model obtains higher revenue compared with some heuristic methods. Our model training process requires only a relatively small amount of data, and it could reuse data which avoids wasting time and resources on collect training data. We have examined other RL technologies such as Q-learning and Pathwise Derivative Policy Gradient. However, these methods' performance were found to be unstable. We plan to fine-tune more advanced RL methods and apply the models in the future.

References

[1] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2014.

[2] M. O. Ojijo and O. E. Falowo, "A survey on slice admission control strategies and optimization schemes in 5g network," *IEEE Access*, vol. 8, pp. 14977–14990, 2020.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[4] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[5] V. Sciancalepore, K. Samdanis, X. Costa-Perez, D. Bega, M. Gramaglia, and A. Banchs, "Mobile traffic forecasting for maximizing 5g network slicing resource utilization," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pp. 1–9, IEEE, 2017.

[6] R. M. Neal, "Annealed importance sampling," *Statistics and computing*, vol. 11, no. 2, pp. 125–139, 2001.

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