Learning Resource Allocation and Pricing for Cloud Profit Maximization

Bingqian Du, Chuan Wu, Zhiyi Huang
The University of Hong Kong

Abstract

Cloud computing has been widely adopted to support various computation services. A fundamental problem faced by cloud providers is how to efficiently allocate resources upon user requests and price the resource usage, in order to maximize resource efficiency and hence provider profit. Existing studies establish detailed performance models of cloud resource usage, and propose offline or online algorithms to decide allocation and pricing. Differently, we adopt a black-box approach, and leverage model-free Deep Reinforcement Learning (DRL) to capture dynamics of cloud users and better characterize inherent connections beween an optimal allocation/pricing policy and the states of the dynamic cloud system. The goal is to learn a policy that maximizes net profit of the cloud provider through trial and error, which is better than decisions made on explicit performance models. We combine long short-term memory (LSTM) units with fully-connected neural networks in our DRL to deal with online user arrivals, and adjust the output and update methods of basic DRL algorithms to address both resource allocation and pricing. Evaluation based on real-world datasets shows that our DRL approach outperforms basic DRL algorithms and state-of-the-art white-box online cloud resource allocation/pricing algorithms significantly, in terms of both profit and the number of accepted users.

Introduction

Cloud computing has proliferated in recent years. Large public cloud platforms are available for various users to run computation jobs; enterprises are building on-premise clouds for internal usage, where users from different departments submit and run their jobs. A fundamental problem faced by the cloud provider/operator, no matter in public or private clouds, is how to efficiently allocate and price cloud resources, provisioned by physical servers in clusters, to the requesting users, such that resource utilization is most efficient and profit of the provider is maximized. [1]

The most common form of resource provisioning in today's cloud platforms is to run virtual machines (VMs) (or containers) with different resource configurations, and lease them to the users. For example, Microsoft Azure provides Burstable VMs (1 CPU core and 1GB RAM), Compute optimized VMs (2 CPU cores and 4GB RAM), General Purpose VMs (2 CPU cores and 8GB RAM), and a number of other types of VMs (Azu). Upon a user's VM request, the cloud provider should decide on which server to run the asked VM and how to price it.

Most cloud providers post prices which do not change in months, *e.g.*, Microsoft Azure (Azu) and Google Cloud (Goo) publish hourly pay-as-you-go prices for different types of VMs, leaving users to decide whether to use the cloud service, according to their budgets. Such static posted pricing does not adapt well to demand and supply changes in the cloud system, and may well jeopardize cloud provider's profit (Mazrekaj, Shabani, and Sejdiu 2016). We advocate dynamic resource pricing according to demand and supply in the cloud, which is closely related to VM placement in the server cluster, to maximize cloud provider's profit.

Dynamic resource allocation and pricing is challenging, as users may submit their VM requests at any time, asking for different types of VMs for different durations. The resource allocation problem is essentially an online, multi-dimensional packing problem, well known for its hardness, not to mention that an associated pricing scheme is also in need. There have been recent studies proposing online algorithms for dynamic resource pricing and allocation upon user arrivals (Zhang et al. 2015). These studies formulate resource allocation into detailed optimization problems, and compute allocation and pricing decisions based on online optimization techniques. Due to the complexity of a cloud system, simplification assumptions often have to be made, in order to convert the optimization problems to linear or convex problems, allowing efficient solutions.

Aiming to capture richer characteristics of the problem and derive better resource allocation and pricing decisions, we adopt a black-box approach and leverage deep reinforcement learning (DRL) to learn a better policy. DRL uses neural networks (NNs) as function approximators to describe connections between complex system input and optimal decisions. The success of DRL in playing games such as Go (Silver et al. 2016) and Atari (Mnih et al. 2015) shows that combining deep learning with reinforcement learning to directly learn Q-values for discrete actions from sampled data is a promising approach for solving real-world prob-

[1] In case of a private cloud, the prices may serve as gauges of job values to decide job admission, and the profit can be interpreted as total value of executed jobs.

lems in a model-free manner. The Deep Deterministic Policy Gradient (DDPG) method, which is based on the actor-critic model (Lillicrap et al. 2015), further enables DRL to handle problems with continuous action spaces. We employ DRL for obtaining a good cloud resource allocation and pricing policy, as DRL is potentially more suitable to model the complex dynamic cloud system than existing white-box approaches, as its NN may capture more inherent connections than a simplified linear/convex model.

The key challenges of designing a DRL approach for cloud resource allocation and pricing include: (i) our decisions include both discrete actions (VM placement) and continuous actions (pricing); (ii) the online user arrivals render non-Markovian time-variant dynamics. Basic DRL methods such as DQN (Mnih et al. 2015) or DDPG (Lillicrap et al. 2015) are not directly applicable, since their update methods are based on the Markovian Decision Process (MDP) assumption and they can handle either a discrete action space or a continuous one, instead of a mixed one. We adopt a number of techniques, in combination with a DDPG-based approach, to address the above challenges. Our main contributions are as follows:

1. We novelly use a model-free DRL approach to make cloud resource pricing and allocation decisions for profit maximization, rather than relying on white-box performance models of the cloud system.
2. We jointly consider VM pricing and placement of VMs in our DRL model, to maximize the profit of cloud provider to the largest extent.
3. We consider time-variant user dynamics, rather than simply assuming user arrivals are i.i.d.
4. We combine LSTM with DDPG to deal with time-variant user dynamics to resolve the non-MDP issue, and use a new update method to allow them to work together smoothly, to learn optimal decisions directly from input states.
5. We train our DRL model using real-world workload from Azure and Google clouds. Extensive validation experiments show that our DRL approach outperforms basic DRL algorithms and state-of-state-the-art white-box online cloud resource allocation/pricing algorithms significantly, in terms of both provider profit (by at least 30%) and the number of accepted users (by about 15% to 25% in majority of the cases).

Background and Related Work

Deep Reinforcement Learning. In reinforcement learning (RL), the learning agent observes a state s_t from the environment in each step t, and then chooses one action a_t according to its policy $\pi(s_t)$; after the action is taken, the state transits to the next state s_{t+1}, and the agent receives a reward r_t. The goal of the agent is to learn the best policy to maximize its accumulated discounted reward $R_t = \sum_{t=1}^{T} \gamma^{t-1} r_t$, where $\gamma \in [0, 1]$ is the discount factor. DRL uses deep neural networks in RL, enabling itself to handle large state and action spaces (Mnih et al. 2015; Lillicrap et al. 2015).

Handling a discrete action space, Deep Q-Network (DQN) (Mnih et al. 2015) outputs Q-values for all actions when receiving an input state, and chooses an action with the highest Q value; its Q-values are estimated using temporal difference learning. Double DQN (Van Hasselt, Guez, and Silver 2016) improves over DQN to alleviate its Q-value over-estimation issue.

For problems with continuous action spaces, the DQN type of approaches are not sufficient, due to the need of producing Q-values for all possible actions. The DDPG approach (Lillicrap et al. 2015) uses policy gradients for model training, to handle continuous control problems, which is built on the actor-critic architecture. In the actor-critic model, an actor network (the policy network) learns the policy π and produces actions, while a critic network (the value network) estimates the Q-value, $Q(s, a) = \mathbb{E}_{\pi}(R_t | s_t = s, a_t = a)$, for each state-action pair; Q-values produced by the critic network are used in the update of the actor network (to evaluate performance of the current policy), in order to stablize policy learning, especially in case of a large action space (*e.g.*, a continuous acton space).

For a RoboCup game with both discrete and continuous actions to take, Hausknecht *et al.* (Hausknecht and Stone 2015) use DRL to teach a robot how to kick the ball and score a goal, in order to win the game: each time a discrete action is chosen from an action set (Dash, Turn, Tackle, Kick), and then a real value is specified, representing direction or power, for the chosen action. They divide the output of the actor NN into two parts: one to produce a probability distribution over all discrete actions, and the other to output the real values corresponding to the discrete actions; inverting gradients are used to constrain the output continuous values within a specific range. We organize our actions in a similar fashion, while we do not constrain our action space for prices as such, and can still achieve good results with a much larger action space.

Reinforcement Learning for Resource Allocation. RL has been applied in resource allocation problems in recent years. Tesauro *et al.* (Tesauro et al. 2006) train a hybrid RL model for server allocation among multiple Web applications, to maximize the total payment. Galstyan *et al.* (Galstyan, Czajkowski, and Lerman 2004) study grid resource allocation using multi-agent RL, by treating users as heterogeneous RL agents, which learn by submitting requests for resources and using job completion time as rewards. Wang *et al.* (Wang et al. 2017) use tabular, dueling double deep Q-learning for cloud provisioning, aiming at balancing resource cost and performance. Bega *et al.* (Bega et al. 2017) use RL for network resource allocation in future 5G networks, for revenue maximization. These studies largely assume the environment is Markovian and apply existing DRL algorithms. Instead, we produce both VM provisioning and pricing decisions, and consider online user arrivals which may break the MDP assumption, if not properly handled.

Machine Learning for Pricing. In the context of a website selling online advertisement slots by auction, Shen *et al.* (Shen et al. 2017) use an LSTM network to predict advertiser bid distribution and a Monte Carlo tree search method to learn the best reserve prices for the website. Wu *et al.* (Wu, Joseph, and Russell 2016) use tabular Q-learning to set prices for Uber-type on-demand economies, by discretizing prices into ranges as actions. Kim *et al.* (Kim et al. 2014) use tabular Q-learning to decide the electricity prices in smart grids, for charging users' energy consumption. These studies mostly consider simpler models and use tabular RL to handle the relatively small state and action spaces. Our NN-based DRL model is more realistic and handles many more possible states and actions.

Problem Model

We consider a cloud data center with S physical servers, offering M types of virtual machines (VMs) to users. Each type of VM, $m = 1, 2, \ldots, M$, are composed of R types of resources (*e.g.*, CPU, memory, disk storage) at preconfigured amounts. Let V_{mr} be the demand of type-r resource in a type-m VM. C_{sr} is the capacity of type-r resource on server s, $\forall s = 1, \ldots, S, r = 1, \ldots, R$. Let $f_{sr}(\cdot)$ be the cost function of server s on the amount of type-r resource used on the server in time t, as indicated by U_{srt}. The cost is mainly due to power consumption of the server: in practice, most cloud data centers keep their servers on, which remain in the low-power idle mode if no VMs are running (to avoid boot-up delay from the off state); the power usage increases with the increase of resource consumption on the server. We define the cost function as follows (Zhang et al. 2015), where parameter ξ_{sr} indicates the relative weight of the cost due to consumption of type-r resource in the overall server cost.

$$f_{sr}(U_{srt}) = \begin{cases} \xi_{sr} U_{srt}, & U_{srt} \in [0, C_{sr}] \\ +\infty, & U_{srt} > C_{sr} \end{cases} \qquad (1)$$

In total N users submit VM requests to the cloud provider in a potentially large system timespan T. Without loss of generality, we assume the users arrive at different times in T. Upon arrival at t_i, user i requests one VM of type $m_i \in \{1, \ldots, M\}$, and specifies how long it will use the VM for, denoted by $\top_i$. The cloud provider releases to the user a unit-time-usage price P_i for running the VM; the user decides if he is to accept the price and run his VM in the cloud, or leave without taking the price, according to his budget b_i. The budget information is private to the user and not revealed to the cloud provider. If the user takes the price, the VM will be immediately launched on a server k_i (that the cloud provider decides) and run for $\top_i$ without interruption (*i.e.*, we practically do not consider VM preemption in the cloud). Upon completion of the VM usage at $t_i + \top_i$, the occupied resources will be returned to the cloud pool and can be used to create other VMs.

The cloud provider sets the prices for VMs dynamically according to demand and supply, *i.e.*, costs of the servers due to current resource consumption on the servers and estimated user budgets. The goal is to maximize the cloud provider's accumulated profit in T. Let binary variable x_i indicate whether user i accepts price P_i: $x_i = 1$ if yes and $x_i = 0$ otherwise. Overall profit of the cloud provider is computed by total payment minus overall cost, as follows:

$$Total\ Profit = \sum_{i=1}^{N} P_i \top_i x_i - \sum_{s=1}^{S} \sum_{r=1}^{R} \sum_{t=1}^{T} f_{sr}(U_{srt}) \qquad (2)$$

Table 1: Notation

V_{mr}	demand of type-r resource in a $type - m$ VM
C_{sr}	capacity of resource r on server s
m_i	type of VM requested by user i
k_i	server to deploy VM of user i
P_i	unit-time-usage price for user i's VM
$\top_i$	VM duration of user i
T	timespan of the cloud system
U_{srt}	amount of used type-r resource on server s at t
$f_{sr}(U_{srt})$	cost function of type-r resource on server s
ξ_{sr}	weight in cost function $f_{sr}(\cdot)$
x_i	indicator of user i's acceptance
$\omega^{\mu}, d^{\mu}, e^{\mu}$	parameters of actor network
ω^{ν}	parameters of critic network
L	# of sampled transitions from one episode

t_i	arrival time of user i	b_i	budget of user i
S	# of servers	R	# of resource types
N	total # of users	M	# of VM types
ν	critic network	μ	actor network
u_i	current observation upon user i	h_{i-1}	history before user i
r_i	reward due to user i	$\mathcal{D}$	experience memory
s_i	state for user i	a_i	action for user i

We seek to derive an online VM placement and pricing policy based on DRL, using which the cloud provider decides k_i and P_i upon the arrival of each user i, without knowing b_i. Table 1 summarizes key notation in the paper.

DRL Algorithm for Resource Allocation and Pricing

We first present our approaches to address the two DRL design challenges.

Joint Discrete and Continuous Actions. For each VM request, the cloud provider decides the server with available capacity to host the VM and the price for running the VM on the selected server, both are related to profit maximization. Hence, the action space of our DRL agent includes both discrete actions (server selection) and continuous actions (pricing). We base our DRL approach on the DDPG algorithm using the actor-critic model. The output of the actor neural network (NN) is divided into two parts: one gives the probability distribution for choosing among different servers and the other produces the corresponding unit-time-usage prices on the servers, if the asked VM is to be hosted on each. We choose the server according to the probability distribution and post the price of the corresponding server. We revise the actor-critic DDPG algorithm according to such a joint action space.

Non-Markovian Time-variant Dynamics. The online arrival of user requests may well be irrelevant with current resource availability in the cloud and previous user requests.

That is, the next state of the system is not completely decided by the previous state and action, violating the basic MDP assumption of RL algorithms. To render an MDP process, we add an LSTM module into our DRL NNs, and include the history into our state representation. The history encodes states and actions in the past. Existing studies have shown patterns in cloud workload (Han, Chan, and Leckie 2013)(Cortez et al. 2017); the LSTM module, well known for encoding time series (Hochreiter and Schmidhuber 1997), is useful for learning arrival patterns of the requests for different types of VMs, and user budget distributions.

DRL Model

In our problem, a *transition* occurs (*i.e.*, state transits to the next state) upon the arrival of a new user request.

State $s_i = [u_i, h_{i-1}]$: the first part $u_i = [\{U_{1rt_i}, \cdots, U_{Srt_i}\}_{\forall r=1,\ldots,R}, m_i, \top_i]$ includes the current resource availability on all servers and information of the new VM request i; the second part h_{i-1} is the history encoded by LSTM. h_0 is initialized to an all-zero vector.

Action $a_i = [v_{i1}, v_{i2}]$: the first part $v_{i1} = [o_1, \cdots, o_S]$ is a vector including probabilities of selecting the respective servers; the second part, vector $v_{i2} = [p_{i1}, \cdots, p_{iS}]$, contains the respective prices on the S servers, where p_{is} is the unit-time-usage price if VM i is to be run on server s. The DRL agent outputs the server k_i with the highest probability in v_{i1}, and the corresponding price p_{ik_i} from v_{i2}. The price is posted to user i, *i.e.*, $P_i = p_{ik_i}$.

Reward If user i accepts the posted price, the reward is the payment of user (*i.e.*, $P_i \top_i$) minus the increased cost of server k_i due to running user i's VM; otherwise, the reward is 0. Specifically, reward $r_i = P_i \top_i - (\sum_{r=1}^{R} \sum_{t=t_i}^{t_i+\top_i} [f_{k_i r}(U_{k_i rt} + V_{m_i r}) - f_{k_i r}(U_{k_i rt})])$.

Fig. 1 illustrates the architecture of our DRL model. The input to the actor network includes state u_i and history h_{i-1}; the output from the actor network is action a_i. The actor network contains two fully-connected layers followed by one LSTM layer, and then two output layers, for producing v_{i1} and v_{i2}, respectively. The input to the LSTM layer contains output from the second fully-connected layer (produced based on u_i), and the history vector h_{i-1} it produced in the previous step. h_{i-1} encodes information of $u_1, \ldots, u_{i-1}$. When producing v_{i1}, we further use masks to ensure zero probabilities for choosing servers without enough resources to host the VM; for producing v_{i2}, we use masks to make sure that each price can at least cover the increased cost of the respective server, if the VM is to be run on the server.

The input to the critic network includes state u_i, actions v_{i1} and v_{i2} produced by the actor network, and history h_{i-1}; the output is the corresponding Q-value. The input u_i, v_{i1} and v_{i2} are each connected to a fully-connected layer, and output from the three is concatenated and fed into an LSTM layer, followed by an output layer. The fully-connected layers are meant for extracting different features from the three types of input, which will then be combined to estimate the Q-value. The history h_{i-1} produced by the LSTM in the critic network encodes both states $u_1, \ldots, u_{i-1}$ and actions $a_1, \ldots, a_{i-1}$ from the actor network.

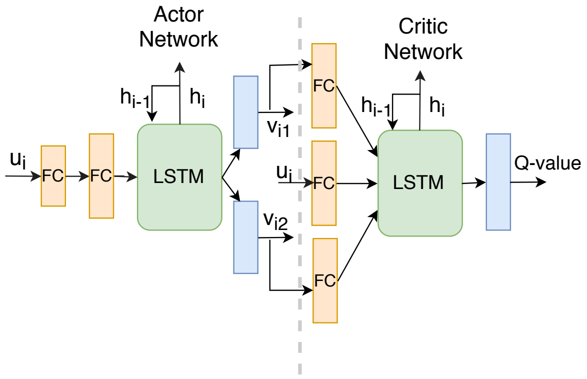


Figure 1: DRL Architecture

DRL Algorithm

To leverage the DDPG algorithm for DRL model training, we store all the transitions within an *episode* (containing all N transitions in T, *i.e.*, one epoch in repeated training) in an experience memory $\mathcal{D}$. However, if we update the actor and critic networks using all transitions in an episode, the strong correlation between transitions will make the training process unstable (Mnih et al. 2015). We adopt the experience replay technique, and randomly pick several transitions from an episode to use for updating the actor and critic networks. For each picked transition i, we cannot use its saved history h_{i-1} for updating the current actor (critic) NN, as it is produced using previous LSTM model parameters which have been updated all the time; to obtain the correct h_{i-1} produced by the current LSTM layer, we use all the historical states before i as input to the current LSTM model and obtain h_{i-1} accordingly (Song et al. 2017).

The update of the critic network, ν, is done by minimizing the mean square error between the output Q-value of the critic network and the target Q-value $y_i = r_i + Q(u_{i+1}, h_i, a_{i+1} | \omega^\nu, a_{i+1} \sim \mu)$, where ω^ν is the set of parameters in the critic NN and μ represents the actor network. The loss function for critic network update is as follows, where history h_i is used, making it different from the standard form (Mnih et al. 2015)(Lillicrap et al. 2015):

$$L(\omega^\nu) = \mathbb{E}_{u_i}[(Q(u_i, h_{i-1}, a_i | \omega^\nu) - y_i)^2] \tag{3}$$

Let ω^μ represent the set of parameters in the actor NN (*i.e.*, μ), excluding the parameters in its two output layers. Let d^μ and e^μ be the set of parameters in the output layer for producing v_{i1} and v_{i2}, respectively. Each step of actor network update includes two parts, updating parameters (ω^μ, d^μ) and (ω^μ, e^μ), respectively, by applying the chain rule to the expected Q-value from the critic network:

$$\begin{aligned}\Delta(\omega^\mu, d^\mu) &= \mathbb{E}_{u_i}[\partial Q(u_i, h_{i-1}, v_{i1}, v_{i2})|_{v_{i1}, v_{i2} \sim \mu(u_i, h_{i-1} | \omega^\mu, d^\mu, e^\mu)}] \\ &= \mathbb{E}_{u_i}[\frac{\partial Q(u_i, h_{i-1}, v_{i1}, v_{i2})}{\partial v_{i1}} \frac{\partial \mu(u_i, h_t - 1)}{\partial \omega^\mu \partial d^\mu}]\end{aligned} \tag{4}$$

$$\Delta(\omega^{\mu}, e^{\mu}) = \mathbb{E}_{u_i}[\partial Q(u_i, h_{i-1}, v_{i1}, v_{i2})|_{v_{i1}, v_{i2} \sim \mu(u_i, h_{i-1}|\omega^{\mu}, d^{\mu})}]$$
$$= \mathbb{E}_{u_i}[\frac{\partial Q(u_i, h_{h-1}, v_{i1}, v_{i2})}{\partial v_{i2}} \frac{\partial \mu(u_i, h_t - 1)}{\partial \omega^{\mu} \partial e^{\mu}}] \tag{5}$$

Further, we adopt the target network technique (Mnih et al. 2015), where a target network is a slightly older version of an NN, used to provide target values for loss computation in training the NN, for stabilizing the training process.

Our complete DRL algorithm, *LAPP*, is given in Alg. 1. $Max\_Episode$ indicates the maximum number of epochs we train the DRL model for. We sample $Batch\_Size$ episodes from the experience replay buffer (line 13), and further prepare a batch of L transitions sampled from each selected episode (lines 16-22); we update the actor and critic networks once using each batch (lines 23-25).

Performance Evaluation

Algorithm Implementation. We implement *LAPP* using TensorFlow on a server equipped with one Nvidia GTX 1080 GPU, Intel Xeon E5-1620 CPU with 4 cores, and 32GB memory. The actor NN we use has 300 and 400 neurons in the two fully-connected layers, respectively, and the output from the LSTM is a vector of 256 units (Ming et al. 2017); the activation function is softmax for outputting v_{i1} and rectifier for outputting v_{i2}. The critic NN has 400 neurons in each fully-connected layer and the output of the LSTM layer has a size of 256; the activation function is rectifier for its output layer. The learning rates in the actor network and the critic network are 10^{-4} and 10^{-4}, respectively. We set $Batch\_Size = 32$, $\gamma = 0.99$, and $L = 4$.

Datasets. We make use of two sets of public traces: (i) Microsoft Azure dataset (Cortez et al. 2017), which contains VM workload information (including VM request arrival time, VM lifetime, VM size in terms of CPU and memory configurations) for several subscriptions in one geographical region over a 30-day span, where a subscription includes logically related workload, as a unit for billing and access control (Cortez et al. 2017). (ii) Google cluster-usage dataset (Reiss, Wilkes, and Hellerstein 2011), which provides information about MapReduce type of jobs (start time, end time, maximum CPU and memory usage, etc.) in one datacenter in U.S. east over a one-month period; we map each job to one VM in our experiments.

DRL Training. We extract one week's workload of a subscription from the Azure dataset for training our DRL model. This subscription includes about 6000 VMs of 4 types: a type-1 VM uses 1 CPU core and 1.75GB RAM, a type-2 VM has 2 CPU cores and 3.5GB RAM, a type-3 VM has 4 CPU cores and 7GB RAM, and a type-4 VM occupies 8 CPU cores and 14GB RAM. For faster training convergence, we extract a workload of 320 VM requests from the subscription (*i.e.*, scale down the VM arrival rate by about 20), while retaining the same arrival pattern as in the complete workload.

Algorithm 1: DRL Algorithm for VM Placement and Pricing, *LAPP*

```
   Initialize: parameters $\omega^{\mu}, d^{\mu}, e^{\mu}, \omega^{\nu}$ in actor network $\mu$
               and critic network $\nu$
   Initialize: parameters $\omega^{\mu'} = \omega^{\mu}$, $d^{\mu'} = d^{\mu}, e^{\mu'} = e^{\mu}$,
               $\omega^{\nu'} = \omega^{\nu}$ for target actor network $\mu'$ and
               target critic network $\nu'$
   Initialize: Experience Replay Memory $\mathcal{D}$
1  for $episode = 1$ to $Max\_Episode$ do
       Initialize: history $h_0$
2      for user $i = 1$ to $N$ do
3          Observe current observation $u_i$, set $s_i = [u_i, h_{i-1}]$;
4          Inference on actor network to obtain $a_i = [v_{i1}, v_{i2}]$ and $h_i$;
5          Produce $k_i$ and $P_i$ from $a_i$;
6          if $P_i \top_i \leq b_i$ (user accepts the price) then
7              Allocate VM of user $i$ on server $k_i$;
8              Receive reward $r_i = P_i \top_i$ -
                 $(\sum_{r=1}^{R} \sum_{t=t_i}^{t_i+\top_i} [f_{k_i r}(U_{k_i rt} + V_{m_i r}) - f_{k_i r}(U_{k_i rt})])$;
9          else
10             Reward $r_i = 0$;
11         end
12         Store transition $(u_{i-1}, a_{i-1}, u_i, r_{i-1})$ into $\mathcal{D}[episode]$;
13         for $Batch$ = 1 to $Batch\_Size$ do
14             Sample an episode $\mathcal{D}[ep]$ from $\mathcal{D}$ randomly;
15             Sample $L$ transitions from $\mathcal{D}[ep]$ randomly;
16             for $l$ = 1 to $L$ do
17                 For transition $(u_i, a_i, x_{i+1}, r_i)$
                     corresponding to the $l$th sampled transition:
18                 Calculate $h_{i-1}$ using transitions before $i$;
19                 Calculate $h_i$ using $h_{i-1}$;
20                 Calculate target Q-value:
21                 $y_i = r_i + \gamma Q'(u_{i+1}, h_i, \mu'(u_{i+1}, h_i))$;
22             end
23             Update the critic network using SGD by
                 minimizing loss in (3) with the $Batch$ data;
24             Update the actor network using SGD with
                 gradients computed using (4) and (5) with the $Batch$ data;
25             Update target actor and critic networks
                 using standard method as DDPG (Lillicrap et al. 2015) with the $Batch$ data;
26         end
27     end
28 end
```

In training, we set the unit-time-usage budgets of users according to normal distributions $\mathcal{N}(\mu, \sigma^2)$ (Agmon Ben-Yehuda et al. 2013): μ and σ are set to (1.5, 0.1), (3.0, 0.2), (6.0, 0.3), and (12, 0.4) for the four types of VMs, respectively. The number of physical servers to host the VMs is 30. By default, each server is equipped with 8 CPU cores and 30GB RAM. ξ_{sr} in the server cost functions is set to 0.5 and 0.02 for CPU and RAM, respectively (Kansal et al. 2010).

The one-week workload is trained repeatedly for about 2000 times (*i.e.*, $Max\_Episode = 2000$) until model convergence, which takes about 15 hours.

Validation data. We evaluate our trained DRL model using a workload of more than 500 VM requests in another week from the same subscription in the Azure dataset, as well as a workload of about 200 VM requests from the Google dataset (we map each VM to the closet VM type in the Azure dataset, according to its resource composition). We use two types of distributions to produce user budgets in model validation: one is the normal distribution used in model training, and the other is a Pareto distribution, commonly used to describe wealth distribution in the real world (Souma 2002). Especially, we set the minimal value of the Pareto distribution to 0.5 and Pareto index to 2, and use this Pareto distribution to produce unit-time-usage budgets for users requesting type-1 VMs; we scale the budgets for type-1 VMs up, to produce user budgets for other types of VMs, according to resource configurations of those VMs as compared to those of type-1 VMs.

Baselines. We compare *LAPP* with 5 baselines:

(1) Basic DDPG algorithm ($DDPG$) (Lillicrap et al. 2015), with the same output as in our DRL algorithm.

(2) RPD_2 from (Zhang et al. 2015), which is a state-of-the-art online resource allocation and pricing algorithm based on the primal-dual optimization framework. It sets VM prices on each server based on an exponentially increasing function on the amount of consumed resources on the server, and places a VM on a server that leads to the largest profit gain. The rationale is to produce low unit prices when the system has abundant resources, in order to accept more user requests, and raise the prices to filter out low-value requests when the resources are rare, to save the resources for upcoming high-value requests.

(3) $Twice - the - Cost(TC)$ and (4) $Twice - the - Index(TI)$, which are similar to the online algorithm RPD_2, but set the unit price on a server to twice of the current marginal cost of the server, and the marginal cost of the server on twice of the current resource consumption (Blum et al. 2011)(Zhang et al. 2015), respectively.

(5) An algorithm (*Spot*) which uses prices similar to the Spot instance prices on Amazon EC2 (Agmon Ben-Yehuda et al. 2013), and places VMs in a round-robin fashion on servers with sufficient capacities. Especially, we use the $AR(1)$ $Reserve\ Price\ Algorithm$ devised in (Agmon Ben-Yehuda et al. 2013) to approximate EC2 Spot Instance pricing, which takes the smallest and the largest user budgets for a certain VM type as input to produce prices for VMs of the type. For each of our four types of VMs, we invoke the $AR(1)$ $Reserve\ Price\ Algorithm$ with the smallest/largest budget values according to our budget distribution in use.

Evaluation results. Fig. 2 and Fig. 3 show the overall profit (computed using Eqn. (2)) and the total number of accepted user requests achieved by *LAPP* and the first four baselines, using Azure workload trace, under different user budget distributions. *LAPP* achieves 25% more profit and accepts 5% more users requests, as compared to the basic DDPG algorithm, which does not include an LSTM module. This shows that including a history-based pattern prediction module in our DRL helps making better decisions. Compared with online algorithms RPD_2, TI, and TC, *LAPP* improves the profit by at least 40% (though accepting less user requests under the Pareto budget distribution than RPD_2). This reveals the effectiveness of a DRL-based approach, as compared to white-box online optimization approaches. When Google trace is used, Fig. 4 shows that *LAPP* still achieves the largest overall profit.

Since online algorithms are designed to work well in worst-case scenarios, we construct a sequence of VM requests that the online algorithms should be able to handle well, and investigate performance of *LAPP* in such a worst-case scenario. 40 VM requests arrive sequentially, one in a minute, each requesting to use a VM of 8 CPU cores and 14-GByte RAM (one server can only host one such VM) for 300 minutes; the unit-time-usage budget of the first 30 users is the same, 5, and that of the later 10 users is 13. If the first 30 low-value VM requests were accepted, with one VM on one server, then the next 10 high-value requests would be rejected due to lack of available resources. Fig 5 shows that *LAPP* still outperforms other algorithms by about 30%, implying that our DRL approach can handle worse-case scenarios better as well.

We next compare *LAPP* with the fifth baseline, *Spot*, with Azure and Google traces under two types of budget distributions, respectively. In Fig 6, N and P stand for Normal distribution and Pareto distribution, respectively; *LAPP* always obtains more overall profit.

For more detailed investigation, Fig. 8a shows the unit-time-usage prices produced by *LAPP* and *Spot*, respectively, using the Azure validation dataset and Pareto user budget distribution. Note that the *Spot* prices are produced given the largest and smallest budgets in the budget distribution, while *LAPP* is oblivious of the budget distribution. We observe that the majority of prices produced by *LAPP* are in similar ranges as the *Spot* prices, exhibiting *LAPP*'s ability in learning user budget distribution.

Fig. 8b further zooms into the sub sequence of the first 100 user requests, and illustrates the unit-time-usage prices produced by *LAPP* and the unit-time-usage budget of these users. Fig. 8c shows the number of available CPU cores on the server selected by *LAPP* to deploy the asked VM in each user request (no matter whether the corresponding prices are accepted or not), where the types of the VMs are also indicated. In Fig. 8c, for example, we can see that many user requests with indices near 50 ask for type-1 VMs (1 CPU core), followed by requests asking for type-3 or type-4 VMs (requesting 4 or 8 CPU cores each). Although the resources on the selected servers to host those type-1 VMs are sufficient (8 cores available as shown in Fig. 8c), the output prices on the chosen servers are largely higher than the respective user budgets (as shown in Fig. 8b). Hence, users submitting those type-1 VM requests do not accept

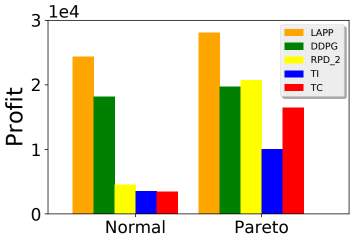


Figure 2: Profit: Azure trace

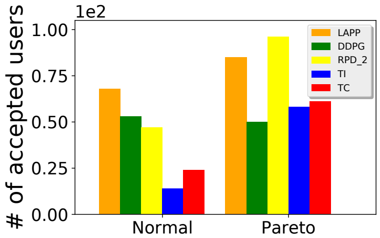


Figure 3: # of accepted user requests

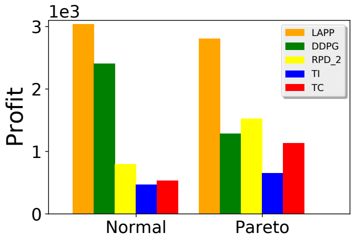


Figure 4: Profit: Google trace

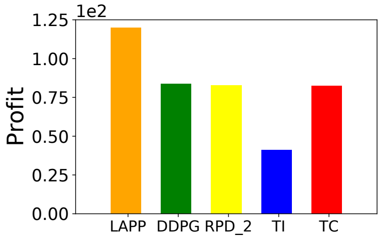


Figure 5: Profit: worst-case scenario

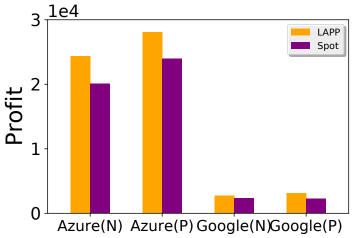


Figure 6: Profit: comparison with $SPOT$

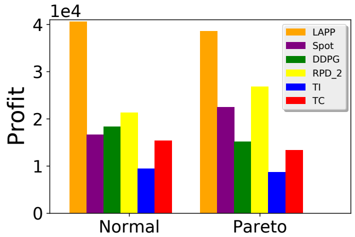


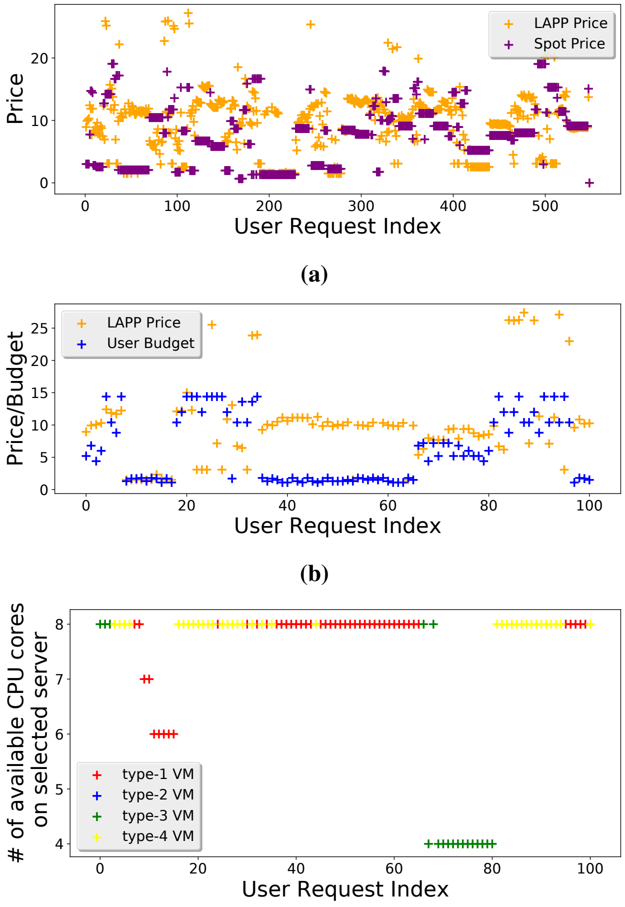
Figure 7: Heterogenous servers

the posted prices, and cloud resources are saved for serving upcoming VMs requiring more resources, from users with larger budgets. This shows that our DRL approach can learn/predict VM request patterns and make better online decisions.

Heterogenous Servers. We further evaluate our DRL approach in a more heterogenous setting with two types of servers: 15 servers each have 8 CPU cores and 30GB RAM, and the rest 15 each have 12 CPU cores and 25GB RAM. ξ_{sr} is set to 0.6 and 0.02 for CPU and RAM, respectively, for the first type of servers, and to 0.5 and 0.01 for the second type of servers. We train our DRL model under this heterogenous setting and validate the model using different Azure datasets. Fig. 7 shows that *LAPP* outperforms other algorithms by at least 25% in this more complex environment.

Conclusion

This paper studies a deep reinforcement learning approach for cloud resource allocation and pricing, targeting provider profit maximization. We combine LSTM with basic DRL to handle online user arrivals, ensuring the MDP assumption, and design new update methods for model training. We validate our DRL approach using real-world cloud workload, and compare it with state-of-the-art online cloud resource allocation and pricing algorithms. The results are quite promising: in majority of the cases, our DRL approach achieves significantly more profit and accommodate more user requests, under various user arrival patterns and budget distributions, even in worst-case scenarios which online optimization algorithms are designed to handle. We believe this an interesting attempt in comparing DRL approach with online optimization approaches, and plan to do more in-depth investigation of the pros and cons of both approaches in making online scheduling decisions in our future work.



(a)

(b)

(c)

Figure 8: Price, budget and available resources

Acknowledgements

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204715, 17225516, C7036-15G (CRF), grants NSFC 61628209 and HKU URC Matching Funding, and Huawei HIRP HO2016050002BE.

References

[Agmon Ben-Yehuda et al. 2013] Agmon Ben-Yehuda, O.; Ben-Yehuda, M.; Schuster, A.; and Tsafrir, D. 2013. Deconstructing amazon ec2 spot instance pricing. *ACM Transactions on Economics and Computation*.

[Azu] Microsoft azure linux virtual machines pricing. `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/`.

[Bega et al. 2017] Bega, D.; Gramaglia, M.; Banchs, A.; Sciancalepore, V.; Samdanis, K.; and Costa-Perez, X. 2017. Optimising 5g infrastructure markets: The business of network slicing. In *INFOCOM*.

[Blum et al. 2011] Blum, A.; Gupta, A.; Mansour, Y.; and Sharma, A. 2011. Welfare and profit maximization with production costs. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium*.

[Cortez et al. 2017] Cortez, E.; Bonde, A.; Muzio, A.; Russinovich, M.; Fontoura, M.; and Bianchini, R. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[Galstyan, Czajkowski, and Lerman 2004] Galstyan, A.; Czajkowski, K.; and Lerman, K. 2004. Resource allocation in the grid using reinforcement learning. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3*. IEEE.

[Goo] Google Compute Engine Pricing. `https://cloud.google.com/compute/pricing`.

[Han, Chan, and Leckie 2013] Han, Y.; Chan, J.; and Leckie, C. 2013. Analysing virtual machine usage in cloud computing. In *Services (SERVICES), 2013 IEEE Ninth World Congress*.

[Hausknecht and Stone 2015] Hausknecht, M., and Stone, P. 2015. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*.

[Hochreiter and Schmidhuber 1997] Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*.

[Kansal et al. 2010] Kansal, A.; Zhao, F.; Liu, J.; Kothari, N.; and Bhattacharya, A. A. 2010. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*.

[Kim et al. 2014] Kim, B.-G.; Zhang, Y.; Van Der Schaar, M.; and Lee, J.-W. 2014. Dynamic pricing for smart grid with reinforcement learning. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference*.

[Lillicrap et al. 2015] Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[Mazrekaj, Shabani, and Sejdiu 2016] Mazrekaj, A.; Shabani, I.; and Sejdiu, B. 2016. Pricing schemes in cloud computing: an overview. *International Journal of Advanced Computer Science and Applications*.

[Ming et al. 2017] Ming, Y.; Cao, S.; Zhang, R.; Li, Z.; Chen, Y.; Song, Y.; and Qu, H. 2017. Understanding hidden memories of recurrent neural networks. *arXiv preprint arXiv:1710.10777*.

[Mnih et al. 2015] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*.

[Reiss, Wilkes, and Hellerstein 2011] Reiss, C.; Wilkes, J.; and Hellerstein, J. L. 2011. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at `https://github.com/google/cluster-data`.

[Shen et al. 2017] Shen, W.; Peng, B.; Liu, H.; Zhang, M.; Qian, R.; Hong, Y.; Guo, Z.; Ding, Z.; Lu, P.; and Tang, P. 2017. Reinforcement mechanism design, with applications to dynamic pricing in sponsored search auctions. *arXiv preprint arXiv:1711.10279*.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature*.

[Song et al. 2017] Song, D. R.; Yang, C.; McGreavy, C.; and Li, Z. 2017. Recurrent network-based deterministic policy gradient for solving bipedal walking challenge on rugged terrains. *arXiv preprint arXiv:1710.02896*.

[Souma 2002] Souma, W. 2002. Physics of personal income. In *Empirical science of financial fluctuations*. Springer.

[Tesauro et al. 2006] Tesauro, G.; Jong, N. K.; Das, R.; and Bennani, M. N. 2006. A hybrid reinforcement learning approach to autonomic resource allocation. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*.

[Van Hasselt, Guez, and Silver 2016] Van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double q-learning. In *AAAI*.

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

[Wu, Joseph, and Russell 2016] Wu, T.; Joseph, A. D.; and Russell, S. J. 2016. Automated pricing agents in the on-demand economy.

[Zhang et al. 2015] Zhang, X.; Huang, Z.; Wu, C.; Li, Z.; and Lau, F. 2015. Online auctions in iaas clouds: Welfare and profit maximization with server costs. In *ACM SIGMETRICS*.