Adversarial Deep Learning for Online Resource Allocation

BINGQIAN DU, ZHIYI HUANG, and CHUAN WU, The University of Hong Kong

Online algorithms are an important branch in algorithm design. Designing online algorithms with a bounded competitive ratio (in terms of worst-case performance) can be hard and usually relies on problem-specific assumptions. Inspired by adversarial training from Generative Adversarial Net and the fact that the competitive ratio of an online algorithm is based on worst-case input, we adopt deep neural networks (NNs) to learn an online algorithm for a resource allocation and pricing problem from scratch, with the goal that the performance gap between offline optimum and the learned online algorithm can be minimized for worst-case input.

Specifically, we leverage two NNs as the algorithm and the adversary, respectively, and let them play a zero sum game, with the adversary being responsible for generating worst-case input while the algorithm learns the best strategy based on the input provided by the adversary. To ensure better convergence of the algorithm network (to the desired online algorithm), we propose a novel per-round update method to handle sequential decision making to break complex dependency among different rounds so that update can be done for every possible action instead of only sampled actions. To the best of our knowledge, our work is the first using deep NNs to design an online algorithm from the perspective of worst-case performance guarantee. Empirical studies show that our updating methods ensure convergence to Nash equilibrium and the learned algorithm outperforms state-of-the-art online algorithms under various settings.

CCS Concepts: • **Computing methodologies → Planning under uncertainty**; **Artificial intelligence;**

Additional Key Words and Phrases: Neural networks, adversarial learning, online algorithm

ACM Reference format:
Bingqian Du, Zhiyi Huang, and Chuan Wu. 2022. Adversarial Deep Learning for Online Resource Allocation. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 6, 4, Article 13 (February 2022), 25 pages.
https://doi.org/10.1145/3494526

1 INTRODUCTION

Traditional algorithm design assumes that input will be revealed to the algorithm all at once. However, a great number of problems arising from reality do not fall into this category. Consider the resource allocation problem in a cloud computing platform. Users' requests for renting resources can arrive at any time; the platform needs to decide whether to rent resources out to the current user without the knowledge of future requests. Such a problem depicts an online setting, where decisions are made for partial inputs that have been revealed so far. Decisions are hard to make in

This work was supported in part by grants from Hong Kong RGC under contracts HKU 17204619, 17208920, and C5026-18G (CRF).

Authors' address: B. Du, Z. Huang, and C. Wu, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong, China; emails: bqdu@hku.hk, {zhiyi, cwu}@cs.hku.hk.

2376-3639/2022/02-ART13 $15.00
https://doi.org/10.1145/3494526

such online settings because of future uncertainty. Still considering the cloud resource allocation problem, renting resource to the current user can be beneficial but may also take up resources that could be allocated to later users with much higher budgets. To deal with online problems, online algorithms have been studied for decades. One common metric to evaluate an online algorithm is the competitive ratio, which measures the gap between the performance of the offline optimum and the online algorithm in the worst case (i.e., under the adversary input) [36].

Inspecting an online algorithm from the competitive ratio perspective, we can view the relationship between the online algorithm and the adversary input as two players at the **Nash equilibrium (NE)** of a zero sum game. For the "algorithm" player, it aims to minimize the gap between offline optimum and its own performance to achieve a good competitive ratio; for the "adversary" player, it targets maximizing this gap by generating hard cases for the algorithm to handle. When both of them arrive at the NE, any strategy change of the adversary will not cause worse performance of the algorithm than its performance at the NE (which corresponds exactly to the worst case in online algorithm analysis). The policy of the algorithm network at the NE is thus the online algorithm with worst-case performance guarantee.

Generative Adversarial Net (GAN) [14] has been a remarkable attempt to combine game theory and deep learning. In GAN architecture, there are two **neural networks (NNs)**: a *generative* NN G and a *discriminative* NN D. G learns to map latent variables to data distribution F while D tries to distinguish real data from F and data generated from G. G and D have opposite goals; the training process of G and D is equivalent to having G and D play a two-player zero-sum game. GAN has been proved effective in generating data distributions similar to real data in a number of successful applications to computer vision [6, 31] and natural language processing [43] [42].

Inspired by competitive ratio analysis of the online algorithm and the GAN model, we investigate designing online algorithms using a deep learning method instead of standard theoretical frameworks such as primal-dual [7, 9]. In this article, we focus on online resource allocation and pricing for social welfare maximization, a classic category of online problems. Representing the online algorithm and the adversary input generation as two NNs, we formulate their interaction as dynamics of two players in a zero sum game: the adversary generates worst-case input while the algorithm learns the best strategy based on the input provided by the adversary. The goal of the algorithm is to minimize the difference between offline optimal social welfare and the social welfare obtained by the online algorithm; the adversary maximizes this difference so that the worst case is ensured. Traditional online algorithm design typically considers the performance ratio between offline optimal solution and online solution; we use performance difference as the objective to ease problem formulation and update design, which also reflects the performance gap between an offline optimal solution and an online solution, and has been used in online algorithm literature [15]. Since an online algorithm produces sequential decisions, the original NN architectures and update methods of GAN cannot be applied to our problem. The standard method for dealing with sequential decision making in deep learning literature is **Reinforcement Learning (RL)**. We do not utilize RL as most of the existing works do since RL is known to heavily rely on the exploration-exploitation trade-off and can easily be trapped in sub-optimal solutions. To deal with sequences and achieve better convergence (to a good online algorithm), we carefully design a per-round update method for both algorithm and adversary NNs from their respective optimization formulations, to break the strong correlation between different timesteps in a sequence so that no more exploration-exploitation heuristics are needed and update can be done for every possible action instead of just sampled actions during the training process. To the best of our knowledge, our work is the first to design worst-case-based online algorithms using a deep learning approach.

We carefully analyze the NE achievable by our approach. We also carry out careful empirical studies under different numbers of arriving users and resource units, and different user budget distributions. The results show the convergence to NE with our update methods and the superior performance of our learned algorithm compared with existing state-of-the-art online resource allocation and pricing algorithms. In this article, we explore the possibility of designing a worst-case-based algorithm using a deep learning framework. We are aware that existing deep learning based methods cannot provide formal theoretical guarantee and worst cases are rare in practice. However, the learned online algorithm and worst cases can provide insights to algorithm designer for better understanding the problem when worst cases are not obvious. We verify the effectiveness of our learning-based approach through empirical studies under both worst cases and random (common) cases.

2 RELATED WORK

2.1 Deep Learning for Game Playing

Game playing is a classic area that has been investigated by researchers for decades. With the prevalence of deep learning, it is natural to ask whether deep learning can be applied to solving complicated games. There have been several attempts in this regard. AlphaGo [33] masters the game of Go without relying on any human knowledge; it combines Monte Carlo Tree Search and RL to improve its policy quality and policy evaluation accuracy with continuous self-play. Green Security Game has been studied by Wang et al. [40], by training a Deep Q-Network (a kind of RL model) to learn an approximate best response. Yu et al. [42] apply a GAN model for text generation; policy gradient and the REINFORCE algorithm are used for the training of their generator, whereas the same training method of the discriminator as in the original GAN work [14] is used in their discriminator training. Multi-agent deep RL is utilized by Celli et al. [10] to solve a sequential zero-sum game. Solving the zero-sum game in a linear discrete-time system is investigated by Luo et al. [24]. They develop a data-based policy iteration Q-learning algorithm to learn the optimal Q-function from data collected in real systems. A regret-based RL algorithm is proposed by Steinberger et al. [37] for an imperfect information multi-agent model-free setting to find the equilibrium. We can see that existing deep learning methods for game playing are mostly based on RL, but as pointed out in the work of Arulkumaran et al. [2], the main challenge faced by RL is that long-range time dependencies make the consequences of a single action unclear after many transitions of the environment, and its observations are limited by its sampled actions.

We use a GAN-like framework; however, unlike GAN's one-shot output, our model deals with sequential decision making, which makes the training even more challenging. Based on the drawbacks of RL described earlier, we do not utilize the RL method as most of the existing studies do; instead, we design a novel per-round update method to tackle challenges in training of both algorithm and adversary NNs, to ensure better convergence.

2.2 Online Resource Allocation Problems

Online allocation and pricing for single-type, non-recycled resources is a fundamental online problem. It is equivalent to the classic online knapsack problem when pricing is not considered [17]. The online knapsack problem was first studied by Marchetti-Spaccamela and Vercellis [27], and they considered the average case analysis. Following that, Buchbinder and Naor [8] proposed a general framework for design and analysis of online algorithms for packing problems (the knapsack problem falls into this category) based on two assumptions: (1) the budget per unit of resource of all users is lower bounded by L and upper bounded by U, and (2) the resource demand of a single user is infinitesimal compared to total resource number. Their algorithm gives an $\mathcal{O}(\log(U/L))$ competitive ratio with a fractional solution; however, how to round the solution to integers is not clear. Based on the same assumptions, Zhou et al. [45] cast the online single ad slot auction problem as an online knapsack problem and propose a KP-Threshold algorithm that achieves a competitive ratio of $\ln(U/L) + 1$.

There have been recent studies on online resource allocation and pricing algorithm design. Online posted price and resource allocation under the same setting as ours was studied by Zhang et al. [44]. Their deterministic online posted pricing and resource allocation algorithm is proved to be optimal under the same two assumptions as earlier. Unlike Zhang et al. [44], we do not make assumptions with respect to the relationship between resource demand and total resource supply and approach the online problem using a deep learning method. In addition, we provide a randomized algorithm, which can potentially capture more complicated features of the problem for better decision making than deterministic ones.

Some deep learning methods have been applied to solving online resource allocation and pricing problems. Tesauro [38] uses a decompositional RL method to learn a strategy for online server allocation. They empirically show that an RL-based method is comparable to a performance model framework based on queuing theory. Resource management in an online scenario is investigated by Mao et al. [26]. They translate the packing problem with resource constraints to a learning problem and propose to use RL for the learning process. Wang et al. [41] aim at achieving automated balance of performance and cost for cloud provisioning. By analyzing the performance of RL under tabular, deep, and dueling double deep Q-learning with the CloudSim simulator, they show the effectiveness of RL. Du et al. [11] use a Long Short-Term Memory (LSTM) NN and a deep deterministic RL algorithm to learn resource allocation and pricing strategy for cloud computing platform, based on the assumption that user request sequences follow a fixed time series distribution. Existing learning-based methods typically make assumptions on the distribution of user requests, which contradicts worst-case input in competitive analysis of online algorithms. We approach online resource allocation and pricing using a deep learning method without making any assumption regarding user request sequence distribution, and worst-case performance is considered in this article.

2.3 Deep Learning for Classic Theoretical Problems

Deep learning has been a new trend for solving classic optimization problems and has shown superior performance in computing time and solution quality. Khalil et al. [18] propose a combination of RL and the graph embedding method to design algorithms for NP-hard combinatorial optimization problems. Empirical results show the effectiveness of their learning-based method. The Traveling Salesman Problem is investigated via the neural approach by Bello et al. [4]. They utilize the recurrent NN and policy gradient method to tackle such a learning problem and achieve close-to-optimal results on Euclidean graphs with up to 100 nodes. Kool et al. [19] propose new models and training methods for routing problems, such as the Traveling Salesman Problem and the Vehicle Routing Problem. They design model-based attention layers to better capture the underlying problem structure and a greedy rollout-based REINFORCE algorithm to find optimal solutions. Evaluation shows that their method outperforms a large number of baselines and obtains results close to highly optimized and specialized algorithms. Graph-based combinatorial optimization problems are further studied by Li et al. [22] via a graph convolutional network. The trained graph convolutional network is used to guide a tree search to generate a large number of good candidate solutions. Evaluation shows that their method performs on par with highly optimized state-of-the-art heuristic solvers. Unsupervised learning methods are considered by Karalias and Loukas [16]. By carefully designing the loss, they bridge the gap between the discrete feasible solution for the combinatorial optimization problem and the continuous output of the NN, which provides a performance guarantee for the solution found by the NN. Empirical results show that their method outperforms heuristics and solvers in both solution quality and computation time.

Existing literature has shown the ability of neural approaches in solving classic optimization problems, mainly due to the powerful computation and the generalization ability of the NN. By carefully designing the training method and the model, the NN is a promising way to find a high-quality solution for hard optimization problems. Based on this observation, we believe the neural approach can be a novel tool for online algorithm design, with the goal of finding better solutions. To the best of our knowledge, our work is the first to tackle online optimization through an NN-based method.

2.4 Online Algorithm with Advice

Online algorithms usually assume no knowledge about future input, but with accumulated historical traces, it is likely that some information about the input can be incorporated to improve the performance of online algorithms. Medina and Vassilvitskii [28] utilize a predictor for reserve price optimization, which is the first work to relate the revenue gain with the quality of a machine learning predictor. Following this work, Lykouris and Vassilvtiskii [25] study the caching problem with the augmentation of a machine learning oracle and prove an improved bound when the oracle has a low error rate, as compared with an unconditional worst case. They propose *robustness* and *consistency* to evaluate the model performance with respect to the worst case and best prediction, respectively. Kumar et al. [20] improve the ski-rental algorithm and non-clairvoyant job scheduling using machine learning predictions. Their proposed algorithm achieves a good trade-off between robustness and consistency, improves the performance with better predictions, and does not degrade much when predictions are worse. Gollapudi and Panigrahi [12] also study the ski-rental problem. The main difference is that they consider multiple machine learning experts. They prove the improvement of their algorithm with the aid of prediction.

Another line of works consider incorporating historical data traces directly to improve the performance of online algorithms. Prodan and Nae [32] propose a prediction-based method for resource provisioning, with predictions produced by a neural net trained using historical data traces. On top of the NN, they design generic analytical game load models for resource allocation. Vera et al. [39] propose a framework based on Bellman inequalities for designing online allocation and pricing algorithms. Following this work, Banerjee et al. [3] utilize approximate dynamic programming based on Bellman inequalities, which resolve offline relaxations to make the controller reduce its sensitivity to estimation error.

Our work differs from these studies. In our work, we do not consider incorporating an online algorithm with machine learning prediction of input based on historical traces. Instead, we directly use the machine learning approach to learn the online algorithm and worst cases, still respecting the assumption that no information about the input is known to the online algorithm beforehand.

2.5 Convergence Rate Analysis for Differentiable Games

There have been some recent efforts to study the convergence behavior of GAN and other learning-based games. Singh et al. [35] analyze the two-player two-action iterative general-sum game, where each agent updates its strategy through gradient descent. They prove that the strategies may not always converge, but the average payoffs always converge to the expected payoffs of NE. Nagarajan and Kolter [30] study the convergence dynamics of GAN by showing that the equilibrium point is locally asymptotically stable for GAN formulation. Based on this result, they further propose a regularization term to speed up convergence. Letcher et al. [21] argue that gradient descent does not always converge to the local optimum of objective in GANs, when there are multiple interacting losses. By decomposing the game Jacobian, they propose Symplectic Gradient Adjustment to find stable fixed point. The non-asymptotic local convergence of a two-player

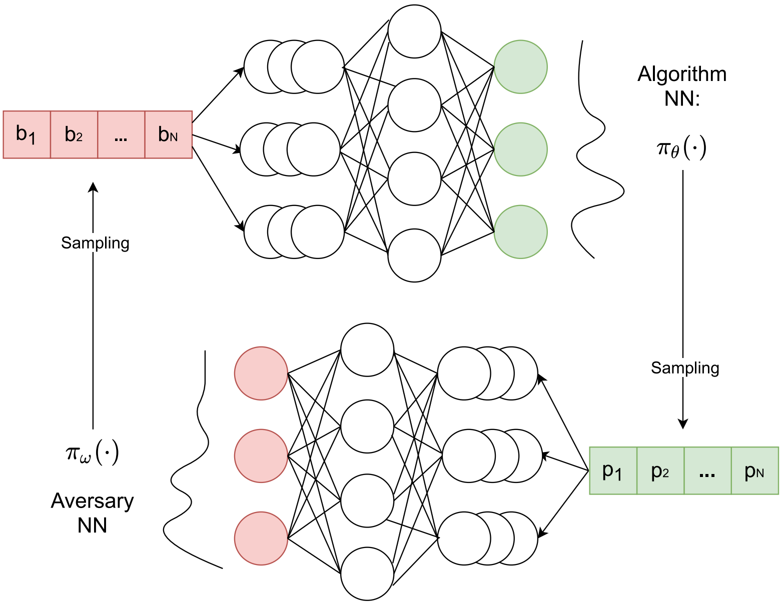

Fig. 1. System overview.

smooth game was studied by Liang and Stokes [23]. They prove that the iteration needed for the Simultaneous Gradient Ascent to converge is dependent on the off-diagonal interaction term.

3 MODEL AND TRAINING METHODS

An overview of our method is given in Figure 1. We adopt two NNs, namely the *algorithm* NN and the *adversary* NN. The output (colored circle) of the algorithm NN is a probability distribution over the price set for each user, whereas the output from the adversary NN is probability distributions over the budget set for all users. The input (colored square) to the algorithm NN or the adversary NN is decided by the learned strategy of the other NN. To be more specific, in each training iteration, the price sequence as the input to the adversary NN and the budget sequence fed into the algorithm NN are sampled according to the output of the algorithm NN and the adversary NN, respectively. The price set A and the budget set B are assumed to be known and finite in our formulation, which can be estimated based on historical traces if not given.

We now formulate our problem, model the algorithm and adversary NNs, and derive the NN update methods for learning the online algorithm. Importance notations are summarized in Table 1.

3.1 Online Resource Allocation and Pricing Problem

We consider social welfare maximization for single-type non-recycled resource allocation and pricing. There are R units of resource supply in total. N users arrive over time, each requesting one unit of the resource. The budget of user $i \in N$ is b_i, denoting how much the user is willing to pay for buying one unit of the resource. When user i arrives, the online algorithm posts its price p_i for one unit of the resource. We assume the algorithm is not aware of the current user's budget b_i but knows posted prices and budgets of past users. If p_i is no larger than b_i, user i accepts the price and receives one unit of the resource. The portion of social welfare due to accepting user i is $(b_i - p_i) + p_i = b_i$.

Table 1. Notation Table

i	User index	N	No. of users
R	No. of resource units	b_i	Budget of user i
B	Budget set	p_i	Posted price for user i
A	Price set	Ω	Budget sequence set
$\pi_\theta(\cdot)$	Algorithm NN	$\pi_\omega(\cdot)$	Adversary NN
τ_o	Joint mixed strategy	C	Payoff matrix of the game
τ_p	Mixed strategy of algorithm	τ_b	Mixed strategy of the adversary
y_i	No. of available resources when user i arrives		
X_i	Random variable denoting user i is accepted or not		
x_i	Realized acceptance decision for user i		
l_i	Indicator on whether budget i will be accepted in offline optimum		
Gap	Difference between offline optimum and online solution		
$u_p(\cdot)$	Utility of algorithm: additive inverse of the expected gap		
$u_b(\cdot)$	Utility of adversary: the expected gap		

3.2 Algorithm

In the posted price scenario, the accept or reject decision would normally be made by a user by comparing its budget to the current price. However, when formulating the optimization problem for the algorithm NN, to allow the NN to adjust its pricing strategy, we instead consider the probability of a user being accepted or rejected by the NN as the decision variable. For the online algorithm, upon arrival of a user, the decision variable is a probability distribution over all possible prices. We use X_i as the random variable denoting whether user i is accepted or not. x_i is the realized acceptance decision of random variable X_i: $x_i = 1$ if $b_i \geq p_i$ and there exists available resource, and $x_i = 0$ otherwise. $x_i = 1$ implies that one unit of resource is allocated to user i, whereas $x_i = 0$ indicates that user i does not consume any resource. Let $P(X_i = 1|X_1 = x_1, \ldots, X_j = x_j, 1 \leq j < i)$ denote the probability of user i being accepted conditioned on the realized acceptance of previous users, which is also the probability for choosing prices no larger than b_i. Similarly, $P(X_i = 0|X_1 = x_1, \ldots, X_j = x_j, 1 \leq j < i)$ is the probability of not allocating resource to user i conditioned on the realized decisions of previous users—that is, the probability for choosing prices larger than b_i. For simplicity of notation, we use $P(X_i = 1|x_1 \ldots x_j)$ and $P(X_i = 0|x_1 \ldots x_j)$ as a shorthand for $P(X_i = 1|X_1 = x_1, \ldots, X_j = x_j, 1 \leq j < i)$ and $P(X_i = 0|X_1 = x_1, \ldots, X_j = x_j, 1 \leq j < i)$, respectively.

The goal of the online algorithm is to minimize the gap between offline optimal social welfare and the social welfare achieved by the algorithm, which is equivalent to maximizing the social welfare of the algorithm solely because the offline optimum is a constant for a given user budget sequence. The social welfare maximization problem to solve upon arrival of user i can be formulated as follows (it is the offline optimization problem if user i is the first user). Note that since rejecting one user will bring 0 social welfare increment and will not affect resource consumption, the terms related to $P(X_i = 0|x_1 \ldots x_j)$ are not shown in the following formulation and their gradient will be 0:

$$\max_{\substack{\mathrm{P(X_j=1|x_1\ldots x_{i-1})}, \\ \forall j \in [i, N]}} f = \sum_{j=i}^{N} b_j P(X_j = 1|x_1 \ldots x_{i-1}), \tag{1}$$

subject to

$$x_1 + x_2 + \cdots + x_{i-1} + \sum_{j=i}^{N} P(X_j = 1|x_1 \dots x_{i-1}) \leq R, \tag{1a}$$

$$P(X_j = 1|x_1 \dots x_{i-1}) \in [0,1], \forall j \in [i,N]. \tag{1b}$$

The objective function (1) is the expected social welfare achieved by the algorithm conditioned on the decisions before user i. (1a) is the resource constraint, which bounds the expected resource consumption by the number of available resources, due to the randomization nature of our algorithm NN. The conditional probability based on previous realized decisions would ensure the optimality of the complete sequence, that users with the largest R budgets would be accepted. (1b) presents the decision variables in the problem.

The preceding problem is a **linear program (LP)**, where strong duality holds. We can relax constraint (1a) by introducing Lagrangian multiplier λ and obtain the following Lagrangian function:

$$\begin{aligned}\mathcal{L}(P(X_j = 1|x_1 \dots x_{i-1}), \lambda) = &\sum_{j=i}^{N} b_j P(X_j = 1|x_1 \dots x_{i-1}) + \lambda(R - x_1 - \cdots - x_{i-1} - \sum_{j=i}^{N} P(X_j = 1|x_1 \dots x_{i-1}))\\ = &\sum_{j=i}^{N} (b_j - \lambda) P(X_j = 1|x_1 \dots x_{i-1}) + \lambda(R - x_1 - \cdots - x_{i-1}).\end{aligned} \tag{2}$$

The dual function is then

$$\mathcal{G}(\lambda) = \max_{\substack{\mathbf{P(X_j=1|x_1\dots x_{i-1})},\\ \forall j \in [i,N]}} \mathcal{L}(P(X_j = 1|x_1 \dots x_{i-1}), \lambda).$$

Let OPT be the optimal objective value of the primal problem, and let λ^* represent $(\lambda^*|x_1 \dots x_{i-1})$. Since strong duality holds, we have

$$\begin{aligned}OPT = \mathcal{G}(\lambda^*) &= \max_{\substack{\mathbf{P(X_j=1|x_1\dots x_{i-1})},\\ \forall j \in [i,N]}} \mathcal{L}(P(X_j = 1|x_1 \dots x_{i-1}), \lambda^*)\\ &= \max_{\substack{\mathbf{P(X_j=1|x_1\dots x_{i-1})},\\ \forall j \in [i,N]}} \sum_{j=i}^{N} (b_i - \lambda^*) P(X_j = 1|x_1 \dots x_{i-1}) + \lambda^*(R - x_1 - \cdots - x_{i-1}).\end{aligned}$$

Given λ^*, $\lambda^*(R - x_1 - \cdots - x_{i-1})$ is a constant, and solving the primal problem is equivalent to solving

$$\max_{\substack{\mathbf{P(X_j=1|x_1\dots x_{i-1})},\\ \forall j \in [i,N]}} \sum_{j=i}^{N} (b_i - \lambda^*) P(X_j = 1|x_1 \dots x_{i-1}), \tag{3}$$

subject to

$$P(X_j = 1|x_1 \dots x_{i-1}) \in [0,1], \forall j \in [i,N]. \tag{3a}$$

The gradient of function (3) on $P(X_j = 1|x_1, \dots x_{i-1})$ is $b_i - (\lambda^*|x_1 \dots x_{i-1})$. According to Simon and Blume [34], $(\lambda^*|x_1 \dots x_{i-1}) = \frac{\partial f(P^*(X_j=1|x_1 \dots x_{i-1}))}{\partial R}$ measures the rate of the change of the optimal value of problem (1) with respect to resource capacity (in constraint (1a))—that is, how much the objective function value will increase if another unit of resource is available. We will use this gradient in the stochastic gradient descent (SGD) method in solving the optimization problem, via training an algorithm NN. For algorithm NN training, we know complete budget sequences, as provided by the adversary NN. Given an acceptance decision sequence $x_i, \dots, x_N$ and the observation that optimal Lagrangian multiplier represents the infinitesimal change in the optimal social welfare with one more unit of resource, $(\lambda^*|x_1 \dots x_{i-1})$ can take any value between the yth budget and the $(y+1)$th budget in the ordered budget sequence in decreasing budget order, with $y = y_i = R - \sum_{l=1}^{i-1} x_l$ being the current amount of available resources upon arrival of user i. We set $(\lambda^*|x_1 \dots x_{i-1})$ to be the average of the yth budget and the $(y+1)$th budget.

We provide more discussions to better justify our update method for the algorithm NN:

- *Conditional probability*: The problem formulation (1) for the algorithm considers the probability of accepting a user, conditioned on the acceptance realization of previous users. The conditional probability is to avoid the situation that Rth budget and the $(R+1)$th budget are the same so that the update signal for both of them would be 0, which would result in sub-optimal solution. Specifically, if the decision variable is $P(X_i)$, denoting the probability of accepting user i, the problem can be formulated as

$$\max_{P(X_i), \forall\ in[1,N]} \sum_{i=1}^{N} b_j P(X_i) \text{ subject to: } \sum_{i=1}^{N} P(X_i) \leq R, P_i \in [0,1], \forall i \in [1,N].$$

  Recall that given available resource number R, λ^* can take the average of the Rth budget and the $(R+1)$th budget. If the Rth budget and the $(R+1)$th budget are the same, the gradient $(b_i - \lambda^*)$ for both of them would be 0 while the optimal solution should accept one of them. To avoid this case, we realize the acceptance decision x_i for the current user i and consider conditional probability so that optimal solution of the algorithm formulation can be ensured for the current budget sequence.
- *Greedily optimizing w.r.t. sampled budget sequence*: The problem the algorithm tries to solve is $\min_{alg} \max_{adv}$ *Gap*. In the framework proposed, the adversary is adjusting its strategy to provide a worst sequence with the largest gap. To solve this problem, the algorithm should adjust its strategy to minimize the gap/maximize social welfare for the worst case sampled by the adversary. We note that the NE strategy may not be the optimal solution for each single sequence, so the optimizing step taken for each sequence could make the strategy of the NN deviate from the NE strategy; but on average, the value achieved by algorithm NN would be equivalent to the value of NE when the two NNs are playing the zero sum game (which is also the reason we save the last 1k training results to obtain the average performance in the evaluation section).

3.3 Adversary

We consider an oblivious adversary, which generates the complete worst-case input sequence before the sequence is handled by the online algorithm [5]. Oblivious adversary is the most studied type of adversary, which is popular in practical settings since most of the time, the input sequence is independent of the algorithm. For example, in the ski rental problem, the weather is independent of the choice of buying or renting skis.

Given the pricing strategy of the online algorithm, the goal of our adversary is to generate the user arrival sequence (a.k.a. budget sequence) to maximize the gap between the offline optimal social welfare and the social welfare achieved by the online algorithm. The complete budget sequence has a length of N; for the ith slot in the sequence, the adversary produces a probability distribution $P_i(\cdot)$ over budget set $B = \{b_1, b_2, \dots b_m\}$, which contains all possible budget choices, where m is the number of possible budget values. For each user, the budget is independently chosen according to the probability distribution $P_i(\cdot)$ over all budgets.

Given the price sequence produced by the algorithm, $p_1, p_2, \ldots, p_N$, the optimization problem of the adversary can be formulated as follows:

$$\max_{\substack{\mathbf{P_i(b_i^{(j)})} \\ i\in[1,N],j\in[1,|\Omega|]}} \sum_{j\in[1,|\Omega|]} \prod_{i=1}^{N} P_i\left(b_i^{(j)}\right) Gap^{(j)}, \tag{4}$$

subject to

$$Gap^{(j)} = \sum_{i=1}^{N} \left(b_i^{(j)} l_i^{(j)} - b_i^{(j)} \mathbb{1}\left(b_i^{(j)} \geq p_i\right) \mathbb{1}\left(y_i^{(j)} > 0\right)\right), \forall j \in [1, |\Omega|] \quad (4a)$$

$$l_i^{(j)} = \begin{cases} 1, & if\ b_i^{(j)} \in benchmark \\ 0, & \text{otherwise} \end{cases}, \forall i \in [1,N], j \in [1,|\Omega|] \quad (4b)$$

$$y_i^{(j)} = R - \sum_{l=1}^{i-1} \mathbb{1}(b_l^{(j)} \geq p_l),\ \forall i \in [1,N], j \in [1,|\Omega|] \quad (4c).$$

Here, $(b_1^{(j)} \ldots b_N^{(j)})$ is a combination of N elements from set B. We use Ω to denote the set of all such combinations, which is also the set of all possible complete budget sequences of the adversary. $j \in [1, |\Omega|]$ is used to denote one specific combination/budget sequence, where $|\Omega|$ is the number of all possible combinations. For the complete budget sequence with index j, $b_i^{(j)}$ is the budget choice for user i. $P_i(b_i^{(j)})$ denotes the probability of choosing budget value $b_i^{(j)}$ for user i. The probability of choosing the complete budget sequence j equals the product of the probabilities of choosing each budget $b_i^{(j)}$ in the sequence. The objective function in (4) gives the expected gap over all possible budget sequences of the adversary. Note that since different budget sequences are generated using the same adversary NN, we have $P_i(b_i^{(j)}) = P_i(b_i^{(j')})$ as long as $b_i^j = b_i^{(j')}$.

$Gap^{(j)}$ is defined in (4a), with $l_i^{(j)}$ and $y_i^{(j)}$ defined in (4b) and (4c), respectively. $l_i^{(j)} = 1$ indicates that budget $b_i^{(j)}$ is counted into the offline optimal social welfare (referred to as the *benchmark*), which implies that $b_i^{(j)}$ is among the top-R largest budgets in sequence j—that is, user i with this budget is allocated with one unit of resource in the offline optimal solution, and $l_i^{(j)} = 0$ otherwise. $y_i^{(j)}$ is the number of available resource units upon arrival of user i when budget sequence j is discussed. A user will be allocated one unit of resource by the algorithm if and only if (1) his budget is no smaller than the posted price by the algorithm (i.e., $b_i^{(j)} \geq p_i$) and (2) there are available resources to allocate (i.e., $y_i^{(j)} > 0$). The gap between the offline optimal social welfare and the social welfare of the online algorithm can be computed by summing up the difference between the chosen budget for each user (if it is in the benchmark) and the budget if it is accepted by the algorithm in the respective slot, as in (4a).

Unlike the optimization problem on the algorithm side, it is hard to relax constraints to derive the gradient of the objective function on $P_i(b_i^{(j)})$ in the preceding adversary's optimization problem. This is because the numbers of accepted users in the benchmark and by the algorithm are both constrained by the resource number, and directly enumerating all possible combinations in Ω to obtain the gradient on $P_i(b_i^{(j)})$ can soon be intractable when the user sequence becomes long and the budget set becomes large. Nonetheless, the algorithm-side gradients can be used as a reference for the derivation of gradients on the adversary side. Since $(\lambda^* | x_1 \ldots x_{i-1})$ measures the change of the optimal objective value of the algorithm when one more unit of resource is available, it is a tight lower bound of all accepted user budgets from user i to N in the optimal solution of the algorithm, derived upon user i's arrival. So the gradient on the probability $P(X_j = 1|x_i \ldots x_{i-1})$ of accepting user i, $b_i - \lambda^*$, measures the difference between the objective value computed when the decision of accepting user i is made by the algorithm and the objective value calculated when the optimal choice for user i is made, assuming optimal choices were adopted in all later slots ($j > i$) (so we only need to compare b_i with λ^*). We provide a simple example for better illustration. Consider the following user budget sequences: $\{1, 3, 3, 3\}$. Suppose the available resource number is 2, and then $\lambda^* = 3$. The gradient for accepting the first user is $1 - 3$, which equals the difference between $(1 + 3) - (3 + 3)$—that is (the objective value when accepting the first user and taking optimal solution for latter slots) − (the objective value of the optimal solution).

Since the gradient of the algorithm's objective function on the probability of choosing a price at slot i measures the effect of the price choice at slot i to the objective value when all later slots are fixed to optimal choices, following this principle, on the adversary side, the gradient on the probability of choosing budget b_l for user i ($P_i(b_l)$) should reflect the objective value of the adversary when budgets of all later slots after i are fixed to optimal budgets and the budget of user i is set to b_l. We hence adopt such a heuristic approach for more efficiently computing gradients of the adversary's optimization problem. To derive such gradients, the next question is how to compute the optimal budgets for unrealized slots, since solving the adversary's optimization problem for those slots is not as obvious as solving the algorithm's optimization problem.

Given a partially realized budget sequence and complete price sequence, we can calculate the optimal budget choices for unrealized slots in polynomial time (i.e., solving the optimization in (4) with budgets in the first $i - 1$ slots realized). We only need to consider two cases for gap maximization:

(a) For each user $j \in [i, N]$, set the budget to be the largest budget in B that is smaller than price p_j in the algorithm's price sequence, or post the smallest budget in B if no budget in B is smaller than price p_j.

(b) Suppose the number of unrealized slots is $U = N - i + 1$ and k is the current number of available resource units. Consider each slot index $j \in [k + i, N]$: for slots between $[i, j - 1]$, find k slots with the smallest prices according to the algorithm's price sequence and set the smallest budget from B that is no less than the respective price for each of these slots (such that these users will be accepted but with smallest social welfare increment)[1]; supposing the index of the last slot among these k accepted slots is j', for slots after j', set the largest budget from B, and for slots before j' other than the k slots picked earlier, set the respective budget the same way as described in case (a) (such that corresponding users are maximally rejected to increase benchmark value). In this way, we obtain $(U - k)$ budget sequences, and the one with the largest gap will be the budget sequence output in this case.

Case (a) represents the case that almost no resource is allocated starting from user i onward; case (b) corresponds to the case that all remaining resources are allocated in the following. We do not need to consider other scenarios where part of the remaining resources are allocated in the unrealized slots, which will always lead to a smaller gap than that in case (a). Consider the following situation when one budget value is different from that in the respective slot in case (a) and the corresponding user is accepted by the algorithm, resulting in one more unit of resource usage (or the same amount of resource usage if case (a) has already used up all resources) as compared to (a): (i) if the slots whose budgets are counted into the benchmark are not affected,

[1] If all budget values are smaller than respective prices at these k slots, we will not consider the current j anymore and will continue to the next j.

since social welfare achieved by the algorithm is increased by the accepted budget value, then the gap between the benchmark and the algorithm will be smaller; (ii) if the new budget value allows the corresponding slot to be counted into the benchmark, replacing another slot with a smaller budget, no matter whether the user of this replaced budget is accepted or rejected by the algorithm, the gap will always not be larger after replacement. Suppose a user's budget b' is replaced by another user's budget b^* in the benchmark; $b^* > b'$ due to the replacement in the benchmark. If the user with budget b' is accepted by the algorithm after replacement (it definitely was accepted by the algorithm before replacement due to the same price and budget for this slot before replacement), the change of the gap is $(b^* - b' - b^*) - (b' - b') = -b'$; if b' is rejected by the algorithm after replacement, there are two possible scenarios: (1) b' is also rejected before replacement, and then the change of the gap in this scenario is $(b^* - b^*) - b' = -b'$; (2) b' is accepted before replacement but is rejected due to resource exhaustion by b^* after replacement, and then the corresponding gap is $(b^* - b^*) - (b' - b') = 0$.

The complete algorithm for computing optimal budget sequence of adversary given the price sequence of algorithm is Algorithm 1. The input contains the complete price sequence from the algorithm side, the partially realized budget sequence, the total number of resource units, and the budget set. Lines 4 and 5 compute the number of available resources and social welfare of the algorithm for the partially realized budget sequence part. Case (a) is considered in lines 7 through 12, and case (b) is implemented from line 13 to line 26: every possible resource running out situation is considered (line 13), where the available resource number of slots with the smallest prices will be accepted with the smallest possible budgets (lines 14–18) while the highest possible budgets causing algorithm rejection will be set for other slots (line 14); the highest budgets will be set for slots after using up resources (lines 21 and 22). Among all situations, the one with the largest gap will be the final solution in case (b) (lines 25 and 26).

3.4 Training NNs

Let $\pi_\theta(h_{i-1}, x_i)$ represent the NN model of the algorithm, where θ is the set of parameters in this algorithm NN, and h_{i-1} is the history information before user i and x_i have received information of new user i. h_{i-1} and x_i are input to the algorithm NN $\pi_\theta(\cdot)$. In our implementation, x_i consists of the current index i, the current available resource number y_i, the last user's budget b_{i-1}, and the last realized price p_{i-1}; h_{i-1} is the encoded history before user i using x_1 to x_{i-1}, computed by $h_{i-1} = vec[\delta([x_1, \ldots x_{i-1}] * W + \boldsymbol{b})]$, where W is a matrix to give different weights to different slots and information, $\boldsymbol{b}$ is a bias matrix, $\delta(\cdot)$ is an activation function, and $vec[\cdot]$ is to flat the result to a vector so that h_{i-1} and x_i can be concatenated together as input to the algorithm NN. The output of the algorithm NN P_i^θ is a probability distribution over all possible price choices for user i.

Let $\pi_\omega(v)$ represent the NN model of the adversary, where ω is the set of parameters in this adversary NN and v is the input to the adversary NN. v is a vector of latent variables sampled from some prior distribution $p(v)$; we use the Gaussian distribution in our implementation, the same as in GAN [29]. The output P^ω is N probability distributions over budget set B, to produce the budget sequence.

The complete algorithm for training the two NNs is given in Algorithm 2. Suppose the size of the price set is n and the size of the budget set is m. At each iteration t, we first sample a batch of s latent variables, budget sequences, and price sequences (lines 3–5). When updating the algorithm NN (line 17), we calculate the gradient of f in (1) on parameter θ, to optimize f. According to the chain rule, $\frac{\partial f}{\partial \theta} = \frac{\partial f}{\partial P^\theta(p)} \times \frac{\partial P^\theta(p)}{\partial \theta}$, where $P^\theta(p)$ is the probability output of the algorithm NN. The gradient of f on the probability of accepting user i is $b_i^{(j)} - (\lambda^* | p_1^{(j)} \ldots p_{i-1}^{(j)})$ if $p_l \leq b_i^{(j)}$ and 0 if $p_l > b_i^{(j)}$, as discussed in Section 3.2.

ALGORITHM 1: OPT_BUDGET

Input: Complete price sequence P: $p_1, \dots, p_N$; Partially realized budget sequence: $b'_1, \dots b'_{l-1}$; Total resource number R; Budget set $B = \{b_1, b_2, \dots b_m\}$
Output: Optimal budgets for unrealized slots $b^*_l, \dots b^*_N$ so that the complete budget sequence $b'_1, \dots b'_{l-1}, b^*_l, \dots b^*_N$ with the largest gap can be produced

```
1:  opt_gap = 0, res = R, alg_perf = 0, rej_b_seq = {$b'_1, \dots, b'_{l-1}, 0, \dots, 0$} (with $N$ slots)
2:  $Gap_{rej}$ = 0 {variables for case (a)}
3:  acc_b_seq = {}, $Gap_{acc}$ = 0 {variables for case (b)}
4:  for $i = 1$ to $l - 1$ do
5:      If $b'_i \geq p_i$ and res > 0: alg_perf += $b'_i$, res -= 1
6:  end for
7:  for $i = l$ to $N$ do
8:      temp_b = min($B$)
9:      for $j = 1$ to $m$ do
10:         If $b_j < p_i$ and temp_b < $b_j$: temp_b = $b_j$ {make users rejected}
11:     end for
12:     rej_b_seq[i] = temp_b
13:     if $i - l + 1 \geq res$ then
14:         temp_b_seq = rej_b_seq, find $res\#$ of smallest price from $p_l, \dots, p_i$, mark their index as
            $a1$ to $ar$ {use up resources}
15:         for $k = a1$ to $ar$ do
16:             temp_b_seq[k] = $\infty$
17:             for $j = 1$ to $m$ do
18:                 If $b_j \geq p_k$ and $b_j$ < temp_b_seq[k]: temp_b_seq[k] = $b_j$
19:             end for
20:         end for
21:         for $o = ar$ to $N$ do
22:             temp_b_seq [o] = max($B$)
23:         end for
24:     end if
25:     $Gap$ = Benchmark (temp_b_seq) - Alg (temp_b_seq, $P$)
26:     If $Gap > Gap_{acc}$: acc_b_seq = temp_b_seq, $Gap_{acc} = Gap$
27: end for
28: $Gap_{rej}$ = Benchmark (rej_b_seq) - Alg (rej_b_seq, $P$)
29: If $Gap_{acc} > Gap_{rej}$:
30: return $Gap_{acc}$, acc_b_seq
31: Else:
32: return $Gap_{rej}$, rej_b_seq
```

Similarly, fix a price sequence j, and the gradient of the adversary objective on the probability of choosing budget b_l at user i equals the gap value (line 10) when all slots after i are fixed to optimal budget choices (line 9). Adversary NN update is performed based on the cumulative gradient computed from the entire batch (line 12).

Note that in each training round, we can calculate the gradient of the objective on the probability of any possible output price/budget choice so that all prices'/budgets' probabilities can be updated, instead of only sampled outputs as in standard RL.

ALGORITHM 2: Stochastic gradient descent training of algorithm and adversary NNs

```
for t = 1, 2, ... do
    for $\xi$ steps do
        Sample $s$ latent variables $\{v^{(1)}, \dots, v^{(s)}\}$ from prior $p(v)$
        Sample $s$ budget sequences $\{b^{(1)}, \dots, b^{(s)}\}$ by feeding sampled latent variables to adversary NN
        Sample $s$ price sequences $\{p^{(1)}, \dots, p^{(s)}\}$ by feeding sampled budget sequences to algorithm NN
        Update adversary NN: $gradient = 0$
        for $j = 1$ to $s$, $i = 1$ to $N$, $l = 1$ to $m$ do
            $b_i^{(j)} = b_l$
            $gap^{(j)}, seq^{(j)}$ = OPT_BUDGET $(p^{(j)}, \{b_1^{(j)}, \dots, b_i^{(j)}\}, R, B)$
            $gradient \mathrel{+}= gap^{(j)} \nabla_{\omega^{t-1}} P_i^{\omega^{t-1}}(b_l)$
        end for
        $\omega^{t-1} = \omega^{t-1} + gradient$
    end for
    $\omega^t = \omega^{t-1}$
    Sample $s$ latent variables $\{v^{(1)}, \dots, v^{(s)}\}$ from prior $p(v)$
    Sample $s$ budget sequences $\{b^{(1)}, \dots, b^{(s)}\}$ by feeding sampled latent variables to adversary NN
    Update algorithm NN: $\theta^t = \theta^{t-1} + \sum_{j=1}^{s} \sum_{i=1}^{N} \sum_{l=1}^{n} (b_i^{(j)} - (\lambda^* | p_1^{(j)} \dots p_{i-1}^{(j)})) \mathbb{1}(p_l \leq b_i^{(j)}) \nabla_{\theta^{t-1}} P_i^{\theta^{t-1}}(p_l)$
end for
```

4 THEORETICAL ANALYSIS

4.1 Existence of NE

We next discuss the existence of NE of the game played by the algorithm and the adversary, and the convergence to the NE by our algorithm and adversary NN training.

Suppose the price set of algorithm is $A = \{p_1, p_2, \dots, p_n\}$ and the budget set of adversary is $B = \{b_1, b_2, \dots b_m\}$. There are in total N users. We use $\mathbb{A} = \{\alpha_1, \alpha_2, \dots \alpha_{n^N}\}$ and $\mathbb{B} = \{\beta_1, \beta_2, \dots, \beta_{m^N}\}$ to denote the pure strategy set of the algorithm and the adversary, respectively, where $\alpha_l, \forall l \in [1, |\mathbb{A}|]$, contains N prices chosen from set A and $\beta_l, \forall l \in [1, |\mathbb{B}|]$, contains N budgets from set B. A mixed strategy of a player is a random distribution over its pure strategies. The set of such mixed strategies is denoted by τ_p and τ_b for the algorithm and the adversary, respectively. The joint mixed strategy set is $\tau_o = \tau_p \otimes \tau_b$.

For ease of presentation, we use $u_p(\cdot)$ to represent the additive inverse of the expected gap as the algorithm's utility and $u_b(\cdot)$ to represent the expected gap as the adversary's utility so that the goal for both algorithm and adversary is to maximize their own utility (recall that the algorithm's objective is to maximize the expected social welfare it achieves, which is equivalent to maximizing the additive inverse of the expected gap).

A joint mixed strategy $\tau_o^* \in \tau_o$ is the NE, if the following holds, where τ_{po}^* (τ_{bo}^*) represents the mixed strategy of the algorithm (adversary) in the joint mixed strategy τ_o^*: $\forall \tau_p^l \in \tau_p, u_p(\tau_o^*) \geq u_p(\tau_{bo}^*, \tau_p^l)$, and $\forall \tau_b^l \in \tau_b, u_b(\tau_o^*) \geq u_b(\tau_b^l, \tau_{po}^*)$, or equivalently, $\forall$ pure strategy $\alpha^l \in \mathbb{A}, u_p(\tau_o^*) \geq u_p(\tau_{bo}^*, \alpha^l)$, and $\forall$ pure strategy $\beta^l \in \mathbb{B}, u_b(\tau_o^*) \geq u_b(\beta^l, \tau_{po}^*)$. It indicates that at NE, neither the algorithm player nor the adversary player can have better utility by unilaterally changing its strategy.

Proposition 1. *There exists a mixed NE in the game played by the algorithm and the adversary.*

Proof. The existence of the mixed NE is based on Brouwer's lemma: a continuous function $F(\cdot)$, which maps a non-empty, compact, convex set to the set itself, always has a fixed point x^* such that $x^* = F(x^*)$. Since the probability simplex is non-empty, compact, and convex, all we need to show is to find such a $F(\cdot)$.

We first define a function $g_p^l(\tau_o) = max(u_p(\tau_{bo}, \mathbb{A}_l) - u_p(\tau_o), 0)$, where τ_{bo} means the adversary's strategy in τ_o. Given joint mixed strategy τ_o, this function evaluates by fixing the mixed strategy of adversary τ_{bo}, whether the pure strategy $\mathbb{A}_l \in \mathbb{A}$ would be better than the mixed strategy of algorithm in τ_o for the algorithm.

Next, we define a continuous function $F_p : \tau_o \rightarrow \tau_o$, with $F_{pl}(\tau_o) = \frac{\tau_{po}^l + g_p^l(\tau_o)}{1+\sum_{j=1}^{|\mathbb{A}|} g_p^j(\tau_o)}, \forall l \in [1, |\mathbb{A}|]$, where τ_{po}^l is the algorithm's probability of choosing pure strategy $\mathbb{A}_l$ in its mixed strategy in τ_o. The denominator is to normalize $F_{pl}(\tau_o)$ so that $\sum_{l=1}^{|\mathbb{A}|} F_{pl}(\tau_o) = 1$—that is, the values of $F_{pl}(\tau_o)$ for all l form a probability distribution. Hence, $F(\tau_o)$ for all l maps the non-empty, compact, convex set to the set itself, with only the algorithm's mixed strategy updated. Based on Brouwer's lemma, there exists a fixed point, which is a joint mixed strategy, τ_o^*, satisfying

$$\tau_{po}^{*l} = \frac{\tau_{po}^{*l} + g_p^l(\tau_o^*)}{1 + \sum_{j=1}^{|\mathbb{A}|} g_p^j(\tau_o^*)}, \forall l. \tag{5}$$

There are the following two possible cases regarding Equation (5). Case (i): if $\sum_{l=1}^{|\mathbb{A}|} g_p^l(\tau_o^*) = 0$ (i.e., fixing the adversary's mixed strategy according to the definition of $g_p^l(\cdot)$), varying the algorithm's strategy cannot obtain larger utility for the algorithm: no pure strategy of the algorithm is better than this mixed strategy; deviating this mixed strategy toward any pure strategy cannot lead to a larger utility. By the definition of *Nash equilibrium*, the mixed strategy τ_o^* in this case is already an NE. Case (ii): if $\sum_{l=1}^{|\mathbb{A}|} g_p^l(\tau_o^*) > 0$, then for $\forall l, g_p^l(\tau_o^*) > 0$, as otherwise τ_{po}^{*l} cannot be the same. $g_p^l(\tau_o^*) > 0$ is equivalent to $u_p(\tau_{bo}^*, \mathbb{A}_l) > u_p(\tau_o^*)$. Taking expectation of both sides of this inequality over τ_{po}^*, we can derive $\sum_{l=1}^{|\mathbb{A}|} \tau_{po}^{*l} u_p(\tau_{bo}^*, \mathbb{A}_l) > \sum_{l=1}^{|\mathbb{A}|} \tau_{po}^{*l} u_p(\tau_o^*)$. For the right-hand side of this new inequality, we have $\sum_{l=1}^{|\mathbb{A}|} \tau_{po}^{*l} u_p(\tau_o^*) = u_p(\tau_o^*) \sum_{l=1}^{|\mathbb{A}|} \tau_{po}^{*l} = u_p(\tau_o^*)$, whereas for the left-hand side, we can get $\sum_{l=1}^{|\mathbb{A}|} \tau_{po}^{*l} u_p(\tau_{bo}^*, \mathbb{A}_l) = u_p(\tau_o^*)$ by definition. Combining both sides together, we reach a contradiction $u_p(\tau_o^*) > u_p(\tau_o^*)$; so this case is impossible. Therefore, we are only left with case (i), which indicates NE. Similar discussion can be applied to the adversary side. □

4.2 Convergence

4.2.1 Convergence Proof. We next show that the strategies of the algorithm and the adversary converge to NE. We follow an idea similar to Goodfellow et al. [14].

Proposition 2. *If in each round t of training in Algorithm 2 the adversary is allowed to reach its optimal strategy given the current strategy of the algorithm, the algorithm NN converges to the NE strategy. At the same time, the optimal strategy of the adversary given the NE strategy of the algorithm is also the adversary's NE strategy.*

Proof. Given the strategy of the adversary, the algorithm's problem (1) is equivalent to problem (3), which is a convex function of the algorithm strategy. According to Goodfellow et al. [14], we know that for a function $g(x) = sup_{\alpha \in \mathcal{A}} g_\alpha(x)$ with $g_\alpha(x)$ convex in x for every α, we have

$\partial g_{\alpha^*}(x) \in \partial g(x)$ if $\alpha^* = argsup_{\alpha \in \mathcal{A}} g_\alpha(x)$. In other words, the gradient of $g_\alpha(x)$ on x when α is optimally calculated based on the value of x is equivalent to the gradient of function $sup_{\alpha \in \mathcal{A}} g_\alpha(x)$. In our case, we can solve for the NE by calculating the gradient on the algorithm strategy when the adversary strategy is optimally calculated based on the algorithm strategy. Due to the convexity of objective (3) for computing the algorithm strategy, sufficiently small gradient descent updates of the algorithm strategy can lead to convergence to the optimal algorithm strategy.

The training process will not end until both the algorithm and the adversary converge to NE strategy, as otherwise one of them would keep changing its strategy to get better utility and the updating process would continue. □

To allow the adversary to reach its optimal strategy given the current strategy of the algorithm, in each training iteration, ξ would be very large in theory and the gradient of adversary's objective function on $P_i(b_i^{(j)})$ should be computed exactly by enumerating all possible budget sequences in optimization (4). In addition, when an NN is used, we are optimizing θ instead of the strategy itself, as in the proof of convergence, and we may not ensure that the adversary is allowed to reach its optimal strategy. However, we will show in our empirical studies even when using $\xi = 1$ and our heuristic approach in computing the gradient (see Section 3.3), the algorithm and adversary NNs converge close to NE.

4.3 Calculation of NE Strategies

We first describe how to compute a NE strategy in our zero-sum game, given mixed strategy τ_p for the algorithm, τ_b for the adversary, and the payoff matrix C with dimension $\mathbb{B} \times \mathbb{A}$, where each entry (i, j) corresponds to the gap achieved by the budgets-prices pair $(\mathbb{B}_i, \mathbb{A}_j)$. If τ_p is known, the expected gap with pure strategy $\mathbb{B}_i$ of the adversary is the ith element in $C \times \tau_p$ (the vector containing expected gaps corresponding to all possible pure strategies of the adversary). The adversary wants to maximize the gap, so it will play strategies corresponding to the largest value in $C\tau_p$. To minimize the gap, the best strategy of the algorithm is to minimize the maximum value in $C\tau_p$: $v_p = min_{\tau_p} max\{(C\tau_p)_1, \ldots, (C\tau_p)_{|\mathbb{B}|}\}$. In contrast, the best response of the adversary is to maximize the minimum value in $\tau_b C$: $v_b = max_{\tau_b} min((\tau_b C)_1, \ldots, (\tau_b C)_{|\mathbb{A}|})$.

The computation of τ_p, NE strategy of the algorithm, can be formulated into the following LP, where c_i is the ith row of matrix C:

$$Min_{\tau_p} v_p, \tag{6}$$

subject to

$$\begin{aligned} \tau_p^j &\geq 0, \forall j \in [1, |\mathbb{A}|] \\ \tau_p^1 + \tau_p^2 + \cdots + \tau_p^{|\mathbb{A}|} &= 1, \\ c_i \tau_p &\leq v_p, \forall i \in \{1, |\mathbb{B}|\}. \end{aligned}$$

The computation of τ_b, NE strategy of the adversary, can be formulated into the following LP, where c_j represents the jth column of C:

$$Max_{\tau_b} v_b, \tag{7}$$

subject to

$$\begin{aligned} \tau_b^i &\geq 0, \forall i \in [1, |\mathbb{B}|] \\ \tau_b^1 + \tau_b^2 + \cdots + \tau_b^{|\mathbb{B}|} &= 1, \\ \tau_b c_j &\geq v_b, \forall j \in \{1, |\mathbb{A}|\}. \end{aligned}$$

LPs (6) and (7) are duals of each other. This can easily be verified by the Lagrangian function: $Max_{\tau_b, v_b} L = v_b + \sum_{j=1}^{|\mathbb{A}|} \tau_p^j (\tau_b c_j - v_b) + v_p(1 - \tau_b^1 - \cdots - \tau_b^{|\mathbb{B}|}) + \sum_{i=1}^{|\mathbb{B}|} z_i \tau_b^i$. The gap at the NE is $v^* = v_p = v_b$ given strong duality of LPs. We can see that solving the NE strategies of our zero-sum game is equivalent to solving two LPs.

Next, we discuss the gap and strategies at NE of our system in two cases.

4.3.1 Resource Number No Smaller Than Job Number ($R \geq N$). In this case, pricing/budgeting strategies for different users are the same because the resource is always abundant upon arrival of each user. So we only need to focus on one user. For any user, the pure strategies for the algorithm and the adversary are sets A and B, respectively; τ_p and τ_b denote the mixed strategy over set A and B, respectively, and p_l and b_l are the probability of choosing a respective pure strategy. The payoff matrix C of a single user is as follows (since the resource is enough to accept the user):

$$C_{m \times n} = \left[c_{ij} = b_i \mathbb{1}(b_i < p_j) \right]_{\forall i \in \{1, m\}, j \in \{1, n\}}.$$

We can derive the NE value using (6) (or (7)).

If the smallest price from set A is no larger than any budget in set B, one of the columns of C is a zero vector (because $\mathbb{1}(b_i < p_j)$ will always be zero if p_j is the smallest price), and we can get $z^* = 0$—that is, the optimal strategy of the algorithm is to accept all users with the smallest price.

4.3.2 Resource Number Smaller Than Job Number ($R < N$). The principle of calculating NE in this case is the same as in the previous case, by replacing the pure strategy set from A to $\mathbb{A}$ and B to $\mathbb{B}$ because we cannot calculate NE by a single slot now. τ_p and τ_b denote the mixed strategy over set $\mathbb{A}$ and $\mathbb{B}$; α_l and β_l are pure strategy l in each set, respectively. The payoff matrix C is

$$C_{|\mathbb{B}| \times |\mathbb{A}|} = \left[c_{ij} = Benchmark(\beta_i) - Alg(\beta_i, \alpha_j), \right]_{\forall i \in \{1, |\mathbb{B}|\}, j \in \{1, |\mathbb{A}|\}}.$$

We note that the number of pure strategies of both the algorithm and the adversary is increasing exponentially with the length of user sequence, causing computation complexity in solving the corresponding LP due to enormous numbers of decision variables and constraints.

5 EMPIRICAL STUDIES

Through empirical studies, we seek to further answer two questions. First, are our algorithm and adversary NN update methods effective in converging to NE? Second, how is the performance of our learned online algorithm compared to state-of-the-art online algorithms on the same problem?

The algorithm NN consists of three parts: the history encoding layer, the fully connected layers to process input from the encoding layer, and the output layer. The activation function of the first two parts is Leaky ReLU, and Softmax is used as the output layer function. The adversary NN is composed of fully connected layers and an output layer. Activation function for the fully connected layers is Leaky ReLU, and Softmax is the output layer function.

In our experiments, the algorithm NN has three fully connected layers following history encoding and input concatenation, and then the output layer follows; the adversary NN has four fully connected layers before the output layer. The number of neurons in each layer is adjusted accordingly under different experimental settings, within the range of [30, 80]. ξ in Algorithm 2 is set to 1 (same as in the work of Goodfellow [13]). One episode indicates training with one complete price/budget sequence.

5.1 NN Training Effectiveness

We evaluate our training of algorithm NN and adversary NN separately by replacing the other NN by a **multiplicative weight (MW)** updated player and compare the strategy learned by the respective NN and the NE strategy calculated with standard game theory methods. MW is a commonly used algorithm in playing zero-sum games to approximately converge to the NE [1], which maintains weights over all pure strategies and the probability of choosing each pure strategy is proportional to the corresponding weight. In the training process, weights are updated as follows: $w^{t+1}(a) = w^t(a) * (1 + \eta r^t(a))$, where η is the learning rate (set to 0.01 in our experiments) and $r^t(a)$ is the utility of pure strategy a in updating iteration t (normalized between 0 and 1 to limit the change of weights). We set $s = 10$ (Algorithm 2) in this set of experiments.

5.1.1 Algorithm NN. Consider a scenario where the total number of units of resources is 5; both the price set and the budget set are $\{1, 2, 3, 4, 5\}$. The adversary has 25 pure strategies, each including the first i slots in the sequence $seq = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5]$, $i \in [1, 25]$. Such pure adversary strategies represent worst-case user request sequences in standard online algorithm analysis [44]: different sequence lengths lead to different final resource utilization levels, and increasing budgets make online algorithm perform worst as compared to the offline optimum, as the online algorithm may well exhaust resources by allocating them to earlier users with small budgets, whereas the offline optimum allocates resources to latecomers. In deciding the mixed strategy of the adversary, $r^t(a)$ represents the expected gap of budget sequence a.

Given the adversary strategies, we can derive the NE of the game by formulating the zero-sum game into an LP (8) and solve it using standard method (e.g., an interior point method), and obtain that the expected gap and the probability of accepting each user at NE is 7.834 and [0, 0, 0, 0, 0, 0, 0, 0.083, 0.5, 0.5, 0.333, 0.333, 0.333, 0.333, 0.333, 0.25, 0.25, 0.25, 0.25, 0.25, 0.2, 0.2, 0.2, 0.2, 0.2], respectively. Note that this strategy is not the only NE strategy of the algorithm: for example, a probability of 0.75 for accepting the 9th user and a probability of 0.25 for accepting the 10th user also constitute a NE strategy, as long as the gap between the algorithm strategy and each pure strategy of the adversary is no larger than 7.834. The formulation of effectiveness test of algorithm NN is as follows. The problem formulation is similar to (6). Instead of computing the probabilities of choosing different prices, we let decision variables in the LP be the probabilities of accepting different users so that the number of decision variables is only 25 (we assume the algorithm knows the maximum length of user request sequence only when computing this NE). P_i denotes the probability of accepting user i. The LP is as follows:

$$\min_{P_i,\, i\in[1,25]} z, \tag{8}$$

subject to

$$P_1 + P_2 + \cdots + P_{25} \leq 5, \ \text{(8a)}$$
$$0 \leq P_i \leq 1, \forall i \in [1, 25] \ \text{(8b)}$$
$$\sum_{i=1}^{j} seq[i] P_i \leq z, \forall j \in [1, 25] \ \text{(8c)}.$$

(8a) is the resource constraint. (8c) corresponds to all pure strategies of the adversary, where $seq[i]$ denotes the ith budget in the budget sequence. We can then solve this LP to get NE. The training curve of the algorithm NN is shown in Figure 2. We can see that the time-averaged gap (averaged over the last 500 episodes) converges to around 8 after 40k episodes of training, very close to the computed gap at NE, showing effectiveness of our update method of training the algorithm NN in ensuring convergence to NE. We note that in this experiment, even though the size of the price set is only 5, when there are 25 users, the number of pure strategies of algorithm is as many as 5^{25}.

We can then compute the probability of accepting each user from the NN output (averaged over the last 9k episodes): $[0, 0, 0, 0, 0, 0.226, 0.222, 0.213, 0.173, 0.115, 0.523, 0.36, 0.324, 0.212, 0.153,$

$0.534, 0.352, 0.226, 0.203, 0.113, 0.345, 0.221, 0.209, 0.208, 0.091]$. The largest gap resulted from the preceding strategy under all pure strategies of the adversary is within 0.5 from the gap value at NE (i.e., 7.835), showing closeness of our strategy with the computed NE strategy of the algorithm. The running time for completing 100k episodes of algorithm NN training is less than 15 minutes.

5.1.2 *Adversary NN.*

When testing the effectiveness of adversary NN, the mixed strategy of the algorithm is updated by MW. There is an exponential number of decision variables in the LP ((7) in Section 4.3) to solve for the adversary's NE strategy using a standard game theory method (as its probability distribution is over all combinations of N element from the budget set). To make this problem computable (without causing out-of-memory error when solving the LP using Python linprog on a server with 64 GB of memory), we set the sequence length to 7, price/budget set as $A = \{1, 2, 3\}$, and the total number of resource units to 3. The pure strategies of the algorithm are the following three price sequences: $seqs = [1, 1, 2, 2, 3, 3, 3], [1, 1, 1, 2, 2, 2, 3], [1, 2, 2, 2, 3, 3, 3]$, which are chosen following pricing function design in existing online algorithms [44], which posts lower prices at the start when resources are abundant so that more users can be accepted while raising the prices with the consumption of resources. In computing mixed strategy of the algorithm using MW, $r^t(a)$ is the additive inverse of the expected gap for choosing price sequence a.

By solving LP (7) in our concrete setting, we obtain that the expected gap is 4.333 and the mixed strategy of the adversary is to choose among budget sequences $[1, 1, 1, 1, 2, 2, 2], [1, 1, 2, 2, 3, 3, 3], [1, 2, 1, 2, 3, 3, 3]$ with a $\frac{1}{3}$ probability each, at NE. Note that this NE strategy of the adversary is not unique—any strategy leading to a gap between this strategy and any pure strategy of the algorithm no smaller than 4.333 is an NE strategy.

Figure 3 shows the training curve of the adversary NN, which took about 8 minutes until convergence. We can see that the time-averaged gap is around 4.33 after 60k episodes of training, close to the NE gap value, exhibiting the NE learning ability of our update method for the adversary NN. The strategy learned by our adversary NN is to choose budget 1 with probability of 1 for the first four users and budget 2 with probability of 1 for the last three users, which corresponds to the first budget sequence. Alhough different from the NE strategy computed earlier, only choosing the first sequence is also an NE strategy since its expected gap is no less than the NE gap value, given unilateral strategy change of the algorithm.

5.1.3 *Training Algorithm and Adversary NNs Together.*

Next, we show the effectiveness of our training methods when algorithm and adversary NNs are trained together, under the following setting: the budget set is $B = \{2, 4, 6\}$, and the price set is $A = \{1, 3, 5, 7\}$; the maximal sequence length is 7, and the total resource number is 3; and the pure strategy set of the adversary contains all combinations of seven budgets from set B, and the pure strategy set of the algorithm contains all combinations of seven prices from set A.

By solving the corresponding LPs in (6) and (7), we obtain that the gap at NE is 3.279. The expected gap values throughout training of the algorithm and the adversary NNs are shown in Figure 4, which are calculated using both algorithm and adversary strategies learned. We observe that after about 80k episodes of training, the expected gap approaches the computed gap at NE, further exhibiting the effectiveness of our training methods. The overall running time for 100k-episode training of two NNs is about 20 minutes.

5.1.4 *Training Efficiency.*

To better illustrate the convergence behavior of our method, we further test the algorithm NN and the adversary NN with larger price/budget set and longer sequences. Figure 5 and Figure 6 present the convergence results of the algorithm NN when the adversary is updated by MWU. The price and budget can be any integers between 1 and 20. The budget sequence for Figure 5 is constructed by repeating each budget choice from 1 to 20 twice (sequence

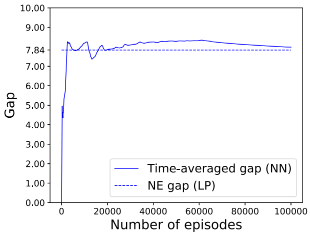

Fig. 2. Algorithm NN training process.

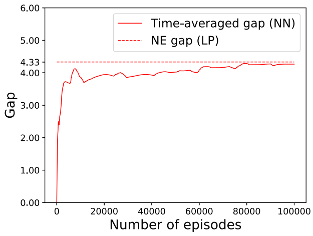

Fig. 3. Adversary NN training process.

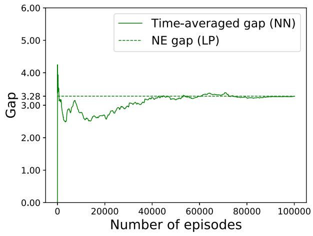

Fig. 4. Two NNs training process.

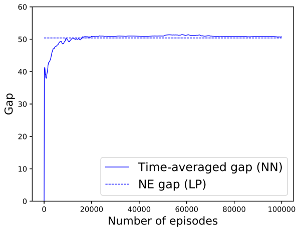

Fig. 5. Algorithm NN training process.

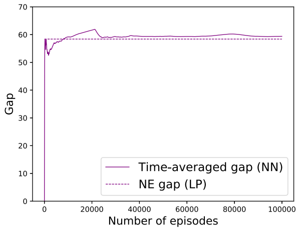

Fig. 6. Algorithm NN training process.

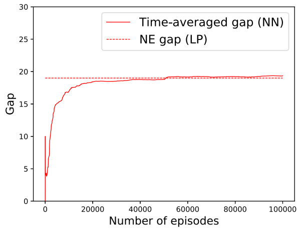

Fig. 7. Adversary NN training process.

length = 40), whereas the budget sequence for Figure 6 is constructed in a similar manner by repeating each value in the budget set from low to high for three times (sequence length = 60). The total resource number is set to 10 for both budget sequences. The NE values computed by solving similar LPs as (8) for these two budget sequences are 50.39 and 58.39, respectively. From the training results, we can see that given larger budget/price sets and longer sequences, the update method for the algorithm NN can still ensure the convergence to NE. The running times for 100,000 episodes for Figure 5 and Figure 6 are about 20 minutes and 30 minutes, respectively. As for the adversary NN, we extend its price and budget sets to integers between 1 and 10 (LP with larger sets would be unsolvable due to memory error). The sequence length is 7, and the resource number is 3. Pure strategies/price sequences of the algorithm (updated by MWU) are as follows: [[1, 2, 3, 4, 5, 6, 7], [2, 4, 4, 6, 6, 9, 10], [1, 2, 2, 2, 2, 2, 3], [6, 6, 7, 7, 8, 9, 10]]. The LP is solved on a server with 256 GB of RDIMM memory, and the NE value computed is 19. Figure 7 shows the training convergence for the game described previously between the adversary NN and the MWU-based algorithm, where the convergence took about 16 minutes.

Next, we evaluate the time needed for conducting training on NVIDIA GeForce 1080Ti and compare the time consumed by our method with other online algorithms during the inference stage. We report the time needed for running one episode in the training process, which starts with sequence sampling according to the opponent's strategy and ends with one update step based on the gradient derived from Section 3. We also report the time needed to make decisions for the same budget sequence with our algorithm NN and with other online algorithms (a.k.a. the inference time) in Table 2. The sequence length is fixed to 100. We observe that the training process would well take hours to complete if the sequence length is long. However, training is done offline. However, the average inference time is less than 0.001 second with all methods. Even though decision making with our NN-based method is slower than the other methods, the inference time is still small enough.

Table 2. Average Processing Time

Training (in seconds)		Inference (in seconds)				
Alg NN	**Adv NN**	**NN**	**OPT-Online**	**KP-Threshold**	**Randomized**	**Greedy**
0.067	0.14	0.0008	0.00007	0.00007	0.00006	0.00004

5.2 Comparison with Existing Online Algorithms

To evaluate the pricing strategy learned by our algorithm NN, we compare its performance with state-of-the-art online algorithms. Let U and L be the upper bound and lower bound of the user budget per unit of resource, and let z denote the fraction of used resource. The baseline algorithms we compare with are as follows. The first is *OPT-Online* [44], which is a deterministic online algorithm for single-type, non-recycled resource pricing and allocation, obtaining optimal competitive ratio based on two assumptions (see Section 2). Its pricing strategy is based on the relationship between total resource demand and resource supply and uses U, L, z; so does its competitive ratio. The optimality proofs of OPT-Online rely on continuous price/budget, whereas the price/budget set in our setting is discrete. When budgets are discrete, some important properties, such as the largest accepted budget by the offline optimum equals the largest accepted budget by the online algorithm minus ϵ, do not hold any more, which indicates that the gap between the offline optimal and online algorithm would be smaller when the price/budget set is discrete. We investigate a discrete price set to facilitate the learning algorithm design and leave the continuous price function case as our future work. The second is *KP-Threshold* [45], which uses price function $\psi(z) = (Ue/L)^z(L/e)$ (e is the base of natural logarithm) to decide which job to accept. The third is the *Randomized* algorithm (mentioned in the work of Zhou et al. [45]), with a competitive ratio of $O(\log_2(U/L))$: for each item, this algorithm samples an integer i uniformly randomly from $[0, \log_2(U/L)]$; if the ratio of user budget per unit resource over L is no smaller than 2^i, the user is accepted. The fourth is the *Greedy* algorithm, which accepts every arrived user until all resources have been allocated, with a competitive ratio of U/L.

In our following evaluation, all results are averaged over 1,000 sequences. NN parameters of the last 1k training episodes are saved and uniformly randomly chosen for evaluation. The price/budget set is $[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. The s in Algorithm 2 is set between 32 and 64.

We first compare the competitive ratio achieved by our learned algorithm with the benchmark algorithms. Competitive ratio is computed as the ratio of the offline optimum and performance of the online algorithm under the worst-case user request (a.k.a. budget) sequence, for each algorithm. For our algorithm, the worst-case budget sequence is provided by the adversary NN. For OPT-Online and KP-Threshold, given the final resource utilization level, we can calculate the largest posted price p^* of the respective algorithm; we set the budget values for accepted user as close as possible to the price that online algorithm posted upon its arrival so that the objective value of online algorithm can be minimized, and budgets of other users to $p^* - \epsilon$ so that the objective value of offline optimum can be maximized [44]. For the Randomized online algorithm, the worst-case sequence contains gradually increased budgets. For the Greedy algorithm, the worst-case sequence starts with the lowest budget value, whereas after resources have been used up, the highest budget value will follow.

Figure 8 shows the competitive ratio when the user sequence length is 100. The total resource supply varies, representing different relationships between resource demand and supply [44]. We can see that our algorithm always achieves the smallest competitive ratio. Especially when the resource is more scarce, our competitive ratio is smaller—almost half of those of other algorithms.

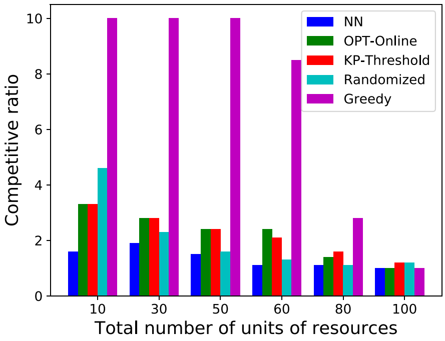

Fig. 8. CR: same sequence length.

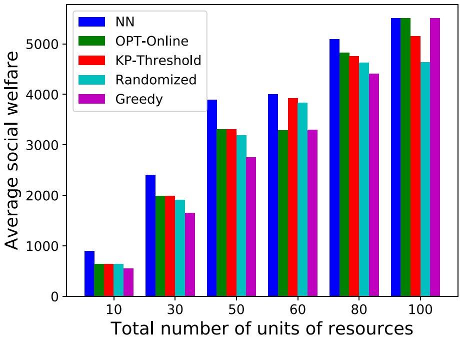

Fig. 9. Social welfare: same sequence length.

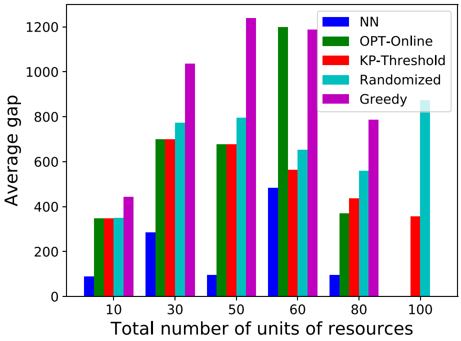

Fig. 10. Gap: same sequence length.

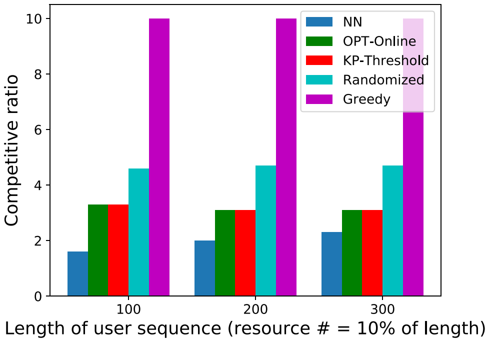

Fig. 11. CR: same supply/demand ratio.

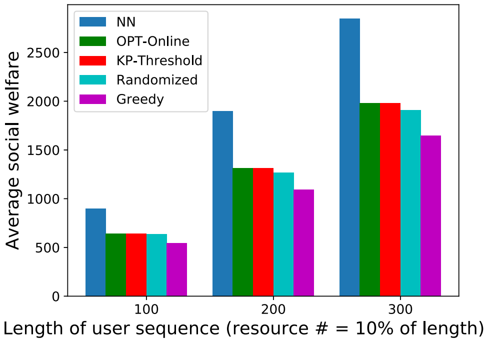

Fig. 12. Social welfare: same supply/demand ratio.

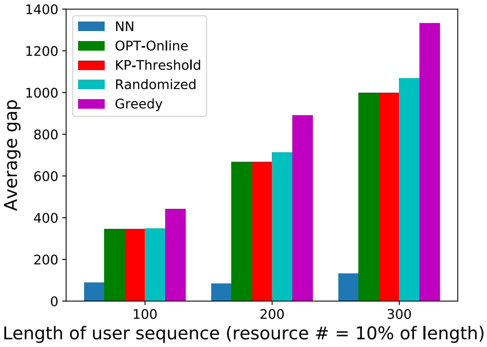

Fig. 13. Gap: same supply/demand ratio.

Figures 9 and 10 compare the social welfare and gap achieved by the algorithms under uniformly randomly generated budget sequences of length 100. Our algorithm achieves the best social welfare and smallest gap to offline optimum. In the case that resource number is no smaller than the user number, we see the gap achieved by our algorithm is zero, consistent with our analysis in Section 7.

Figure 11 compares the competitive ratios when the user sequence length varies while the ratio between total resource supply and total resource demand is fixed to 10%. Our algorithm outperforms baselines with lower competitive ratios.

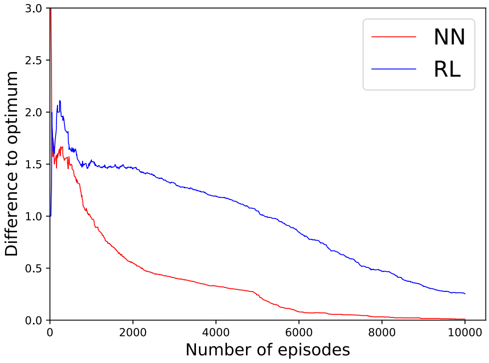

Fig. 14. ϵ-greedy exploration.

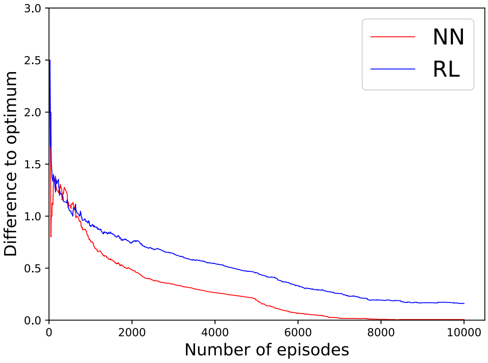

Fig. 15. Guided exploration with sequence length = 5.

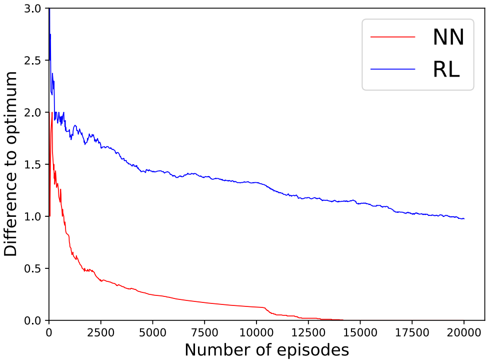

Fig. 16. Guided exploration with sequence length = 10.

Figures 12 and 13 further compare the performance when the user sequence length varies and the ratio between total resource supply and total resource demand is fixed (10%). Similar observations can be made.

5.3 Comparison with RL

We next show the superior convergence performance of our method over the RL algorithm. The budget and price sets are both [1, 2, 3] in this experiment, and the resource number is set to 5. For a given budget sequence (each budget is chosen uniformly at random from the budget set), we train two NNs with the same architecture by our update method and the REINFORCE update method, separately. REINFORCE is a commonly used RL algorithm for sequential decision making in the deep learning literature [19, 42], which updates sampled price values by gradient descent according to the average performance gap of complete price sequences, which are formed by sampled price values and following RL-learned strategy for unsampled slots. In Figure 14, the sequence length is 5; the exploration strategy for RL is a simple ϵ-greedy strategy ($\epsilon = 0.3$)—that is, with probability ϵ, the sampled price according to the RL strategy will be replaced by a random value from the price set to encourage exploration. We can see that our method outperforms RL in both solution quality and convergence speed. Next, we equip RL with an improved exploration strategy and keep the sequence length at 5. Instead of replacing a sampled price with a random price from the price set, we replace it with the optimal price with probability 0.3 in the training process, to guide the sample process to better decisions. Figure 15 plots the training process in this scenario. We can see that RL heavily relies on a good exploration strategy and performs better when the guided exploration strategy is applied, but our method is still better than RL. Last, we compare RL plus the same guided exploration with our method with a longer sequence (sequence length = 10). Figure 16 shows the training process. We can see that when the sequence becomes longer, training convergence becomes harder for RL due to its larger exploration space and the difficulty to break the correlation between steps.

6 CONCLUSION

To the best of our knowledge, our work presents the first attempt to design an online algorithm addressing the worst-case input through NNs, with the case of a classic online problem. The randomized strategy, our novel per-round update method, and the powerful learning ability of NNs enable better online algorithms, as shown by our evaluation results. We also provide empirical evidence showing that our methods ensure convergence to NE. As future work, we are working on extending our method to algorithm design of other online problems.

REFERENCES

[1] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing* 8, 1 (2012), 121–164.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

[3] Siddhartha Banerjee, Itai Gurvich, and Alberto Vera. 2020. *Constant Regret in Online Allocation: On the Sufficiency of a Single Historical Trace*. https://people.orie.cornell.edu/ig264/Online_Optimization_with_Samples.pdf.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

[5] Shai Ben-David, Allan Borodin, Richard Karp, Gabor Tardos, and Avi Wigderson. 1994. On the power of randomization in on-line algorithms. *Algorithmica* 11, 1 (1994), 2–14.

[6] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2018. Large scale GAN training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*.

[7] Niv Buchbinder, Kamal Jain, and Joseph Seffi Naor. 2007. Online primal-dual algorithms for maximizing ad-auctions revenue. In *Proceedings of the European Symposium on Algorithms*. 253–264.

[8] Niv Buchbinder and Joseph Naor. 2009. Online primal-dual algorithms for covering and packing. *Mathematics of Operations Research* 34, 2 (2009), 270–286.

[9] Niv Buchbinder. 2009. The design of competitive online algorithms via a primal–dual approach. *Foundations and Trends® in Theoretical Computer Science* 3, 2–3 (2009), 93–263.

[10] Andrea Celli, Marco Ciccone, Raffaele Bongo, and Nicola Gatti. 2019. Coordination in adversarial sequential team games via multi-agent deep reinforcement learning. *arXiv preprint arXiv:1912.07712*.

[11] Bingqian Du, Chuan Wu, and Zhiyi Huang. 2019. Learning resource allocation and pricing for cloud profit maximization. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*.

[12] Sreenivas Gollapudi and Debmalya Panigrahi. 2019. Online algorithms for rent-or-buy with expert advice. In *Proceedings of the International Conference on Machine Learning*. 2319–2327.

[13] Ian Goodfellow. 2016. NIPS 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems*. 2672–2680.

[15] Klaus Jansen and Roberto Solis-Oba. 2011. *Approximation and Online Algorithms: 8th International Workshop, WAOA 2010, Liverpool, UK, September 9–10, 2010, Revised Papers*. Vol. 6534. Springer.

[16] Nikolaos Karalias and Andreas Loukas. 2020. Erdos goes neural: An unsupervised learning framework for combinatorial optimization on graphs. *arXiv preprint arXiv:2006.10643*.

[17] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. Multidimensional knapsack problems. In *Knapsack Problems*. Springer, 235–283.

[18] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*. 6348–6358.

[19] Wouter Kool, Herke Van Hoof, and Max Welling. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

[20] Ravi Kumar, Manish Purohit, and Zoya Svitkina. 2018. Improving online algorithms via ML predictions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 9684–9693.

[21] Alistair Letcher, David Balduzzi, Sébastien Racaniere, James Martens, Jakob Foerster, Karl Tuyls, and Thore Graepel. 2019. Differentiable game mechanics. *Journal of Machine Learning Research* 20, 1 (2019), 3032–3071.

[22] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*. 539–548.

[23] Tengyuan Liang and James Stokes. 2019. Interaction matters: A note on non-asymptotic local convergence of generative adversarial networks. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*. 907–915.

[24] Biao Luo, Yin Yang, and Derong Liu. 2020. Policy iteration Q-learning for data-based two-player zero-sum game of linear discrete-time systems. *IEEE Transactions on Cybernetics* 51, 7 (2020), 3630–3640.

[25] Thodoris Lykouris and Sergei Vassilvtiskii. 2018. Competitive caching with machine learned advice. In *Proceedings of the International Conference on Machine Learning*. 3296–3305.

[26] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. 50–56.

[27] A. Marchetti-Spaccamela and C. Vercellis. 1995. Stochastic on-line knapsack problems. *Mathematical Programming* 68, 1 (Jan. 1995), 73–104. https://doi.org/10.1007/BF01585758

[28] Andrés Munoz Medina and Sergei Vassilvitskii. 2017. Revenue optimization with approximate bid predictions. *arXiv preprint arXiv:1706.04732* (2017).

[29] Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

[30] Vaishnavh Nagarajan and J. Zico Kolter. 2017. Gradient descent GAN optimization is locally stable. *arXiv preprint arXiv:1706.04156* (2017).

[31] Dong Nie, Roger Trullo, Jun Lian, Caroline Petitjean, Su Ruan, Qian Wang, and Dinggang Shen. 2017. Medical image synthesis with context-aware generative adversarial networks. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention*. 417–425.

[32] Radu Prodan and Vlad Nae. 2009. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems* 25, 7 (2009), 785–793.

[33] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354–359.

[34] Carl P. Simon and Lawrence Blume. 1994. *Mathematics for Economists*. Vol. 7. Norton, New York, NY.

[35] Satinder Singh, Michael Kearns, and Yishay Mansour. 2013. Nash convergence of gradient dynamics in iterated general-sum games. *arXiv preprint arXiv:1301.3892*.

[36] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM* 28, 2 (1985), 202–208.

[37] Eric Steinberger, Adam Lerer, and Noam Brown. 2020. DREAM: Deep regret minimization with advantage baselines and model-free learning. *arXiv preprint arXiv:2006.10410*.

[38] Gerald Tesauro. 2005. Online resource allocation using decompositional reinforcement learning. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*. 886–891.

[39] Alberto Vera, Siddhartha Banerjee, and Itai Gurvich. 2021. Online allocation and pricing: Constant regret via Bellman inequalities. *Operations Research* 69, 3 (2021), 821–840.

[40] Yufei Wang, Zheyuan Ryan Shi, Lantao Yu, Yi Wu, Rohit Singh, Lucas Joppa, and Fei Fang. 2019. Deep reinforcement learning for green security games with real-time information. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1401–1408.

[41] Zhiguang Wang, Chul Gwon, Tim Oates, and Adam Iezzi. 2017. Automated cloud provisioning on AWS using deep reinforcement learning. *arXiv preprint arXiv:1709.04305*.

[42] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. SeqGAN: Sequence generative adversarial nets with policy gradient. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*.

[43] Yizhe Zhang, Zhe Gan, and Lawrence Carin. 2016. Generating text via adversarial training. In *Proceedings of the NIPS Workshop on Adversarial Training*, Vol. 21.

[44] Zijun Zhang, Zongpeng Li, and Chuan Wu. 2017. Optimal posted prices for online cloud resource allocation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (2017), 23.

[45] Yunhong Zhou, Deeparnab Chakrabarty, and Rajan Lukose. 2008. Budget constrained bidding in keyword auctions and online knapsack problems. In *Proceedings of the International Workshop on Internet and Network Economics*. 566–576.

Received January 2021; revised August 2021; accepted October 2021