2020 IEEE 36th International Conference on Data Engineering (ICDE)

An End-to-End Deep RL Framework for Task Arrangement in Crowdsourcing Platforms

Caihua Shan † Nikos Mamoulis § Reynold Cheng † Guoliang Li # Xiang Li † Yuqiu Qian ‡
† The University of Hong Kong, § University of Ioannina, # Tsinghua University, ‡ Tencent Inc.
{chshan, ckcheng, xli2}@cs.hku.hk, nikos@cs.uoi.gr, liguoliang@tsinghua.edu.cn, yuqiuqian@tencent.com

***Abstract*—In this paper, we propose a Deep Reinforcement Learning (RL) framework for task arrangement, which is a critical problem for the success of crowdsourcing platforms. Previous works conduct the personalized recommendation of tasks to workers via supervised learning methods. However, the majority of them only consider the benefit of either workers or requesters independently. In addition, they do not consider the real dynamic environments (e.g., dynamic tasks, dynamic workers), so they may produce sub-optimal results. To address these issues, we utilize Deep Q-Network (DQN), an RL-based method combined with a neural network to estimate the expected long-term return of recommending a task. DQN inherently considers the immediate and the future rewards and can be updated quickly to deal with evolving data and dynamic changes. Furthermore, we design two DQNs that capture the benefit of both workers and requesters and maximize the profit of the platform. To learn value functions in DQN effectively, we also propose novel state representations, carefully design the computation of Q values, and predict transition probabilities and future states. Experiments on synthetic and real datasets demonstrate the superior performance of our framework.**

***Index Terms*—crowdsourcing platform, task arrangement, reinforcement learning, deep Q-Network**

I. Introduction

Crowdsourcing is an effective way to address computer-hard tasks by utilizing numerous ordinary humans (called *workers* or *the crowd*). In commercial crowdsourcing platforms (i.e., Amazon MTurk [1] or CrowdSpring [2]), requesters first publish tasks with requirements (e.g., collect labels for an image) and awards (e.g., pay 0.01 per labeling). When a worker arrives, the platform shows him/her a list of available tasks (posted by possibly different requesters), which are ordered by a certain criterion, e.g., award value or creation time. The worker can select any of the tasks in the list based on summary information for each task, such as the title, the description and the award. Finally, s/he clicks on a task, views more detailed information and decides whether to complete it or not.

As shown in Fig. 1, the current platforms only provide a simple sorting or filtering function for tasks, i.e., sorting by creation time, filtering by category, etc. Due to the large number of available tasks, previous work [22], [34] pointed out that manually selecting a preferred task is tedious and could weaken workers' enthusiasm in crowdsourcing. Some supervised learning methods (e.g., kNN classification or probabilistic matrix factorization) are proposed to conduct personalized recommendation of tasks to workers. However, these approaches have several shortcomings.

First, previous works only consider the recommendation and assignment of tasks, aiming at optimizing the individual benefit of either the workers or requesters. If we only consider the workers' preferences or skills, we may not find a sufficient number of workers for tasks in domains of rare interest. On the other hand, if we only consider the benefit of the requesters, i.e., collecting high-quality results by a given deadline, the assignment of tasks might be *unfair* to workers, lowering their motivation to participate. The goal of a commercial platform is to maximize the number of *completed* tasks, as they make a profit by receiving a commission for each such task. To achieve this, crowdsourcing platforms should attract as many tasks as possible by requesters and as many as possible workers to complete these tasks. Hence, it is necessary to *balance* the benefit of both workers and requesters by satisfying the objectives of both parties to the highest possible degree.

Second, previous works are not designed for handling real dynamic environments. New tasks are created and old tasks expire all the time. The *quality* of a given task (e.g., accuracy of labeling) also keeps changing as it gets completed by workers. Besides, we do not know which worker will come at the next moment, and the workers' preferences are evolving based on the currently available tasks. The models based on supervised learning cannot update the preferences of workers quickly. We show by experimentation that, even if we update supervised learning-based models every day, their performance is still not satisfactory.

Further, the majority of existing works are designed for maximizing the immediate (short-term) *reward*, i.e., select the task with the maximum predicted completion rate for the coming worker, or choose the task that yields the maximum quality gain. They disregard whether the recommended tasks will lead to the most profitable (long-term) reward in the future; hence, they may generate suboptimal suggestions.

To address the above issues, we propose a *Deep Reinforcement Learning* framework for task arrangement in this paper. We model the interactions between the environment (workers and requesters) and the agent (the platform) as a Markov Decision Process (MDP). We apply *Deep Q-Network* (DQN), a widely used reinforcement learning method, training a neural network to estimate the reward for recommending each task. DQN naturally considers the immediate and future

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00012

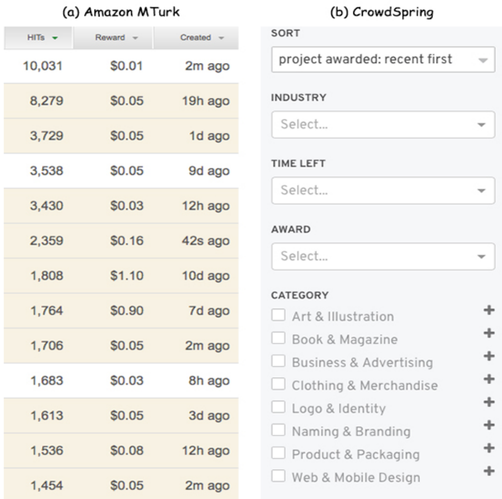


Figure 1: Sorting or Filtering Functions

reward simultaneously in the online environment (i.e., continuously coming workers and changing available tasks). Besides, DQN can be updated quickly after each worker's feedback, seamlessly handling dynamic and evolving workers and tasks.

Despite the advantages of DQN in crowdsourcing platforms, it cannot be directly applied into our task arrangement problem. A typical DQN for recommendation systems only models the relationship between users and items, i.e., workers and tasks in our context. Here, we should also take into consideration the relationships among all available tasks. To capture all the information of the environment, we design a novel state representation that concatenates the features of workers and currently available tasks, as well as a particular Q-Network to handle the set of available tasks with uncertain size and permutation-invariant characteristics.

Besides, workers and requesters have different benefits, and we choose to use two MDPs to model them. If we only consider to recommend tasks of interest for workers, the actions decided by the MDP for a worker are independent to those for other workers. However, the assigned tasks and the corresponding feedback of previous workers do affect the action assigned to the next worker and the quality of tasks (i.e., the benefit of requesters). Thus, we design two separate DQNs to represent these two benefits and then combine them.

Furthermore, DQN is a model-free method which computes the transition probability of (future) states implicitly. Since each (future) state is composed of the (next) coming workers and the available tasks, these workers and tasks could generate a large number of state representations and thus very sparse transitions between states. This further leads to possibly inaccurate estimation of the transition probability and slow convergence. To address such problem, we revise the equation of computing Q values and predict transition probabilities and future states explicitly, after obtaining worker feedback. Specifically, we utilize the worker arrival distribution (which will be discussed in Sec. III-D and Sec. IV-D) to predict when the next timestamp will be, who the next worker is, and how many tasks will be available.

Our contributions can be summarized as follows:

1) To the best of our knowledge, we are the first to propose a Deep Reinforcement Learning framework for task arrangement in crowdsourcing platforms.

2) We apply a Deep Q-Network (DQN) to handle both immediate and future rewards, aiming at optimizing a *holistic objective* from the perspectives of both workers and requesters in the long term.

3) We design a novel and efficient state representation, revise equations for computing Q values and predict transition probabilities and future states explicitly.

4) We use both synthetic and the real datasets to demonstrate the effectiveness and efficiency of our framework.

The rest of the paper is organized as follows. We present our system in Sec. II and describe the main modules in our system in Sec. III and IV. Experiments on synthetic and real data are conducted in Sec. V. We discuss related work in Sec. VI and conclude in Sec. VII.

II. Deep Reinforcement Learning Framework

A. Problem Definition

The goal of the proposed task arrangement system is to assign a task or recommend a sorted list of tasks to a coming worker. Because the profit model of the platform is to charge a commission fee of each completed task, the system should satisfy workers and requesters simultaneously – (1) each worker can find more relevant tasks to complete and (2) requesters publish as many tasks as possible while making each task obtain high-quality results. Moreover, since the tasks and workers are dynamically changing, the system should cope with these dynamic changes and assign tasks in real-time.

B. Problem Formulation as MDPs

We model the task arrangement problem as a reinforcement learning problem. While the crowdsourcing platform (the *agent*) interacts with requesters and workers (the *environment*), requesters influence the pool of available tasks in the agent by setting the start date and a deadline of tasks and obtaining the result of each task after its deadline. The agent recommends tasks to coming workers, and workers influence the agent by task completions.

Since workers and requesters have different optimization goals, we first propose two Markov Decision Processes (MDPs) to optimize them separately, and then combine them together to optimize simultaneously.

MDP(w) (for the benefit of workers): Following the MDP setup of a typical item recommendation system [37], [38], our MDP considers the benefit of workers as follows. At a timestamp i, a worker w_i comes and there is a set of available tasks $\{T_i\}$ posted by requesters.

- State s_i is defined as the recent completion history of w_i, i.e., the representation of the state is the *feature* of the worker w_i, i.e., $f_{s_i} = f_{w_i}$.
- An action a_i is to recommend some of the available tasks to w_i. There are two kinds of actions based on

the problem setup. If the problem is to recommend one task, the possible actions are all available tasks, i.e., $a_i = t_j, \forall t_j \in \{T_i\}$. If the problem is to recommend a sorted list of tasks, possible actions are all possible permutations of available tasks, where $a_i = \sigma(T_i) = \{t_{j_1}, t_{j_2}...\}$ and σ is a ranking function.

- Reward r_i is decided by the feedback of w_i given (s_i, a_i). $r_i = 1$ if w_i completes a task. Otherwise $r_i = 0$.
- Future State s_{i+1} happens when the same worker w_i comes again. The worker feature f_{w_i} is changed if $r_i > 0$. Thus $f_{s_{i+1}}$ is the updated worker feature f_{w_i} by r_i, i.e., the feature of worker w_i when w_i comes again.
- Transition $Pr(s_{i+1}|s_i, a_i, r_i)$ is the probability of state transition from s_i to s_{i+1}, which depends on the success (r_i) of completing a certain task of a_i by w_i.
- The discount factor $\gamma \in [0, 1]$ determines the importance of future rewards compared to the immediate reward in reinforcement learning.

According to the MDP(w) definition, the global objective is to maximize the cumulative completion rate of workers in the long run. We explore and exploit the relationships between workers and tasks, in order to learn the optimal strategy for each worker, even when the interest of the worker is evolving.

MDP(r) (for the benefit of requesters): Again, each timestamp i is triggered by the coming worker w_i and there exists a set of available tasks $\{T_i\}$. However, as we now consider the sum of the qualities of tasks posted by requesters, some elements of the MDP are different:

- State s_i is defined as the previous completion history of w_i and currently available tasks $\{T_i\}$. The worker quality q_{w_i} and the task quality $q_{t_j}, \forall t_j \in \{T_i\}$ are also considered. f_{s_i} is the combination of all these features, i.e., $f_{s_i} = [f_{w_i}, f_{T_i}, q_{w_i}, q_{T_i}]$.
- Action a_i is the same as in MDP(w).
- Reward r_i is decided by the feedback of w_i given (s_i, a_i). r_i is the quality gain of the completed task by w_i. If w_i skips all the recommended tasks, $r_i = 0$.
- Future State s_{i+1} happens when the next worker w_{i+1} comes, no matter whether $w_{i+1} \neq w_i$. The worker feature f_{w_i} and the quality $q_{t_{j'}}$ of a completed task may change if $r_i > 0$.
- Transition $Pr(s_{i+1}|s_i, a_i, r_i)$ depends on the success and quality gain (r_i) of completing a certain task of a_i by w_i. Moreover, it is related to the next worker w_{i+1}.
- The discount factor γ is the same as in MDP(w).

Based on the definition of MDP(r), the global objective is to maximize the cumulative quality gains of tasks in the long run. This is similar to a matching problem. We consider the worker-task relationships for all available tasks to obtain the overall maximum sum of task qualities.

Unifying States. The main differences between MDP(w) and MDP(r) are the definitions of states and rewards, and the happening times of future states. Since they have different global objectives, rewards should be defined and future states should be triggered correspondingly. State(w) and state(r) have

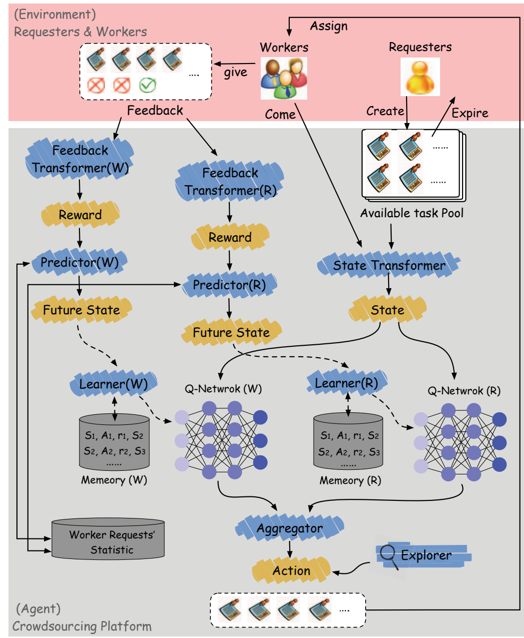


Figure 2: DRL Framework

an inclusion relation, i.e., state(r) considers the relationship among the currently available tasks, but state(w) does not. In Sec. V, we empirically find that the performance of MDP(w) is improved if we also consider the task relationships. Therefore, we unify the state definition in two MDPs, where state(w) is also composed by w_i and T_i and its representation becomes $f_{s_i} = [f_{w_i}, f_{T_i}]$.

Integration of MDP(w) and MDP(r): We combine the two MDPs to meet the two optimization goals. We adopt *Deep Q-Networks* (DQN) to model the MDPs. In each MDP, DQN is used to estimate the value $Q(s_i, a_i)$ of taking the action a_i in the state s_i. Thus, there are two values $Q_w(s_i, a_i)$ and $Q_r(s_i, a_i)$ for MDP(w) and MDP(r) respectively. We use weighted sum to balance them to a single value $Q(s_i, a_i) = wQ_w(s_i, a_i) + (1 - w)Q_r(s_i, a_i)$. The agent selects the action based on combined $Q(s_i, a_i)$ while updating two DQNs separately. We discuss the details of DQN in Section II-D.

C. System Overview

Fig. 2 illustrates the system framework. A worker w_i comes and sees a set of available tasks $\{T_i\}$ posted by requesters at timestamp i. The representation of a state includes the feature of worker w_i and the available tasks T_i through the *State Transformer*, i.e., $f_{s_i} = \textit{StateTransformer}[f_{w_i}, f_{T_i}]$.

Taking Actions. Then, we input f_{s_i} into two Deep Q-networks, *Q-network(w)* and *Q-network(r)*, to predict Q values for each possible action a_i at s_i, considering the benefit of workers $Q_w(s_i, a_i)$ and requesters $Q_r(s_i, a_i)$ separately. We use the aggregator to combine two benefits and generate the final action assigned to w_i. The agent could select the action $a_i = t_j$ with the maximum combined $Q(s_i, t_j)$. If the agent recommends a task list, the action is $\sigma(T_i) = \{t_{j_1}, t_{j_2}, ...\}$

where t_{j*}'s are ranked in descending order of $Q(s_i, t_{j*})$. An explorer is also used to perform the trial-and-error actions, i.e., select a task or generate a task list randomly.

Obtaining Rewards & Future States. When w_i is assigned one task, s/he can decide to complete or skip it. If w_i sees a sorted list of tasks, we assume that workers follow a *cascade model* [6] to look through the task list and complete the first interesting task. The feedback is the completed task and the uncompleted tasks suggested to w_i. Since the reward definitions are different in MDP(w) and MDP(r), we use two *feedback transformers* to quantify the workers' feedback. As we said before, we explicitly predict transition probabilities and future states to ensure stable convergence and real-time behavior. Two *future state predictors* are utilized for *Q-Network(w)* and *Q-Network(r)* separately to predict future states, based on the historical statistics.

Learning models. If the action is to assign a task, we can store one transition (s_i, a_i, r_i, s_{i+1}) (a_i is the assigned task) into the memory pool, which is used to store the training data. When the action is to recommend a list of tasks, the feedback includes the completed task and the uncompleted (suggested) tasks. Thus, we store the successful transition (s_i, a_i, r_i, s_{i+1}) where a_i is the completed task, and the failed transitions $(s_i, a_i, 0, s_{i+1})$ where a_i is an uncompleted task. Each time we store one more transition into the memory pool, we use *learner*s to update the parameters of two Q-networks, obtain a good estimation of $Q_w(s_i, a_i)$ and $Q_r(s_i, a_i)$ and derive the optimal policy π. In the following sections, we will introduce these parts of the system in detail.

D. Deep Q-Network

1) Q-Learning: Q-learning could learn an optimal policy in MDP, i.e., $\pi : \mathcal{S} \rightarrow \mathcal{A}$, which tells the agent what action in $\mathcal{A}$ to take under what state in $\mathcal{S}$ and maximizes the expected cumulative reward in a long run. It defines a state-action value function $Q^\pi(s, a)$ as the expected return following the policy π given state s and action a, where $Q^\pi(s, a) = \mathbb{E}[\sum_{i=0}^{\inf} \gamma^i r_i | s_0 = s, a_0 = a, \pi]$. Based on Bellman's equation [27], the optimal policy is related to the optimal Q value function satisfying:

$$Q(s_i, a_i) = \mathbb{E}_{s_{i+1}}[r_i + \gamma \max_{a'} Q(s_{i+1}, a') | s_i, a_i].$$

Thus, Q-learning learns $Q(s_i, a_i)$ iteratively by choosing the action a_i with the maximum $Q(s_i, a_i)$ at each state s_i. Then it updates $Q(s_i, a_i) \leftarrow (1 - \alpha)Q(s_i, a_i) + \alpha(r_i + \gamma \max_{a'} Q(s_{i+1}, a'))$ where $\alpha \in [0, 1]$ is the learning rate.

2) Deep Q-Network: In practice, we may have huge state and action spaces, making it impossible to estimate $Q(s, a)$ for each s and a. Besides, it is hard to store and update a huge number state-action pairs. To alleviate this, we can use a highly nonlinear and complex function to approximate, i.e., $Q(s, a) \approx Q(s, a; \theta)$. Based on this, *Deep Q-Network* [28] is proposed, which uses a neural network with parameters θ as the Q-network. The network is learned by minimizing the mean-squared loss function as follows:

$$\begin{aligned} L(\theta) &= \mathbb{E}_{\{(s_i, a_i, r_i, s_{i+1})\}}[(y_i - Q(s_i, a_i; \theta))^2] \\ y_i &= r_i + \gamma \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta) \end{aligned} \tag{1}$$

where $\{(s_i, a_i, r_i, s_{i+1})\}$ is the historical data, stored in a large memory buffer sorted by occurrence time. By differentiating the loss function with respect to θ, the gradient update can be written as:

$$\begin{aligned} \nabla_\theta L(\theta) =& \mathbb{E}_{\{(s_i, a_i, r_i, s_{i+1})\}}[(r_i + \gamma \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta) \\ & - Q(s_i, a_i | \theta)) \nabla_\theta Q(s_i, a_i | \theta)] \end{aligned} \tag{2}$$

In practice, *stochastic gradient descent* can be used to efficiently optimize the loss function.

III. Modules for MDP(w)

We introduce the modules related to MDP(w) in the section. We first describe the feature of tasks and workers. Then we design *State Transformer* to construct the state in MDP(w). Next we discuss how to transform the reward from the worker feedback, and how to predict future state from the statistic data. Finally, we revise the equations in DQN to update the model based on predicted future states.

A. Feature Construction

1) **Feature of a task** *t_j:* According to previous studies [13], the top-3 motivations of workers in crowdsourcing are the payment, the task autonomy and the skill variety. Task autonomy is the degree of freedom given to the worker for completing this task, e.g., open tasks (e.g., collecting some entities) and close tasks (e.g., labeling some entities). Skill variety is the diversity of skills that are needed for solving and fit with the skill set of the worker. Thus, we construct the task features using *award*, *task category* and *skill domain*, which correspond to the top-3 three motives. Award is a continuous attribute which needs to be discretized. Category and domain are categorical attributes, and we use one-hot encoding to encode them. Then, we can concatenate them together to obtain the feature vector of task t_j.

2) **Feature of a worker** *w_i:* The features of a worker are determined by the distribution of recently completed tasks by him/her (e.g., in the last week or month). This information can be used to model the probability of a worker to complete a task in the near future. Hence, the feature of a worker is the average feature vector of completed tasks recently.

B. State Transformer and Q Network

1) **Challenges:** We define the state s_i to be composed of the set of available tasks $\{T_i\}$ and the worker w_i at timestamp i. However, it is hard to represent the set of available tasks. First of all, tasks are dynamic and their number is not fixed. We need to design a model can process input of any size. Secondly, the model should be *permutation invariant* (i.e, it should not be affected by the order of tasks). Simple forward neural networks violate both requirements. Methods like LSTM [11] or GRU

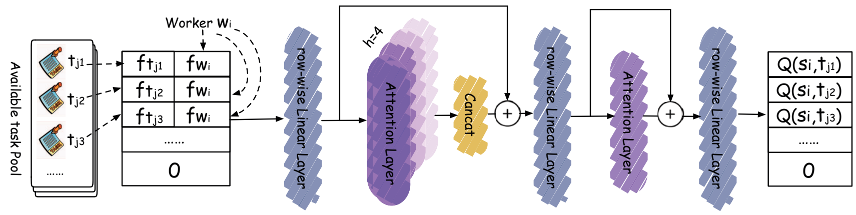


Figure 3: Q Network

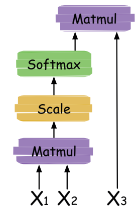


Figure 4: One Attention Layer

[5] that process a variable-length sequence of data, are relative sensitive to the order.

Some approaches in recommender systems based on DQN [37], [38] input the features of each task and worker into a forward neural network independently to estimate the value of each task. However, they ignore the relationship among all available tasks. The value of a task is the same no matter which other tasks are available. This is not true in our setup because tasks are 'competitive' and influence the value of other tasks. For the above reasons, we need to design a novel representation for a set of available tasks.

2) **Design:** Inspired by [35] and [14], we design our *State Transformer* and *Q-Network* to obtain the state s_i and values of each available task $Q(s_i, t_j)$, as shown in Fig. 3. Firstly, we concatenate the features of each task $f_{t_{j*}}$ in the pool of available tasks with the features of the worker f_{w_i}. To fix the length, we set the maximum value of an available task $\max_T$ and use zero padding, i.e., add zeros to the end of f_{s_i} and set its dimension to $[\max_T, |f_{t_{j*}}| + |f_{w_i}|]$.

Then we use row-wise Linear Layers and (multi-head) Attention Layers to project f_{s_i} into Q values, which keeps permutation-invariance. Row-wise Linear Layer is a row-wise feedforward layer which processes each row independently and identically. It calculates function

$$\text{rFF}(X) = \text{relu}(XW + b)$$

where X is the input, W and b are the learnable parameters and relu is an activation function.

The structure of the Attention Layer is shown in Fig. 4. Its input are three matrices X_1, X_2, X_3 and it calculates

$$\text{Att}(X_1, X_2, X_3) = \text{softmax}(\frac{X_1 X_2^T}{\sqrt{d}})X_3.$$

The pairwise dot product $X_1 X_2^T$ measures how similar each row in X_1 and X_2 is, with a scaling factor of $\frac{1}{\sqrt{d}}$ and softmax function. The output is a weighed sum of X_3. Multi-head Attention Layer is proposed in [29]. It projects X_1, X_2, X_3 into h different matrices. The attention function Att is applied to each of the h projections. The output is a linear transformation of the concatenation of all attention outputs.

$$\text{MultiHead}(X_1, X_2, X_3) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Att}(X_1 W_i^{X_1}, X_2 W_i^{X_2}, X_3 W_i^{X_3})$$

We have to learn the parameters $\{W_i^{X_1}, W_i^{X_2}, W_i^{X_3}\}_{i=1}^h$ and W^O. Here we use multi-head Self-Attention layers, where $X_1 = X_2 = X_3 = X$. When $X \in \mathbb{R}^{n \times d}$, a typical choice for the dimension of W_i^X (resp. W^O) is $n \times \frac{d}{h}$ (resp. $n \times d$).

We can prove that row-wise Linear Layer and multi-head Self-Attention Layers are both permutation-invariant. The stack of these layers are also permutation-invariant. Please see the Appendix for details.

We now summarize the design of our Q-network. Each row in the input f_{s_i} is the pair of features of t_j and w_i. The first two rFF layers are used to transform the task-worker features into high-dimensional features. Next, we use the multi-head self-attention layer to compute the pairwise interaction of different task-worker features in the set. Adding to the original features a rFF layer helps keeping the network stable. Thirdly, we use a self-attention layer again, which gives the Q-network the ability to compute pairwise as well as higher-order interactions among the elements in the set. The final rFF layer reduces the feature of each element into one value, representing $Q(s_i, t_j)$. Because of permutation-invariance, no matter the order of t_j, $Q(s_i, t_j)$ is the same. Besides, $Q(s_i, t_j)$ is decided by not only the pair of w_i and t_j, also the other available tasks $t_{j'} \in T_i$.

C. Action $\mathcal{A}$, Feedback and Reward $\mathcal{R}$

The workers of a crowdsourcing platform aim at achieving a good experience. Payment-driven workers aim at finding high award per unit of time tasks while interest-driven workers hope to answer tasks that match their interest. Mixed-interest workers decide by balancing these factors. Our goal is to help them in finding tasks interesting to them as soon as possible, i.e., at maximizing the completion rate of recommended tasks.

If the agent is to assign one task, it selects the action $a_i = t_j$ with the maximum $Q(s_i, t_j)$. We assume workers follow a *cascade model* to look through the list of tasks, so if the agent recommends a task list, the action is $\sigma(T_i) = \{t_{j_1}, t_{j_2}, ...\}$ where t_{j*}'s are ranked in descending order of $Q(s_i, t_{j*})$.

As for the feedback and reward, the feedback is completed or skipped when the action is one task. Thus, the immediate reward is 1 if the worker completes the task or 0 if the worker rejects it. When the action is a list of k tasks, the immediate reward is 1 if the worker finishes one of the tasks or 0 if the worker never finishes any of them.

D. Future State, Memory Storer, and Learner

1) **Challenges:** The future state s_{i+1} is the time when the same worker w_i comes again. Thus, the times of r_i and the future state s_{i+1} are different. Besides, it may take a long time for the same worker to come again (the median value of the time gap is one day in our data) and for the transition (s_i, a_i, r_i, s_{i+1}) to be stored and used to update the *Q-network(w)*.

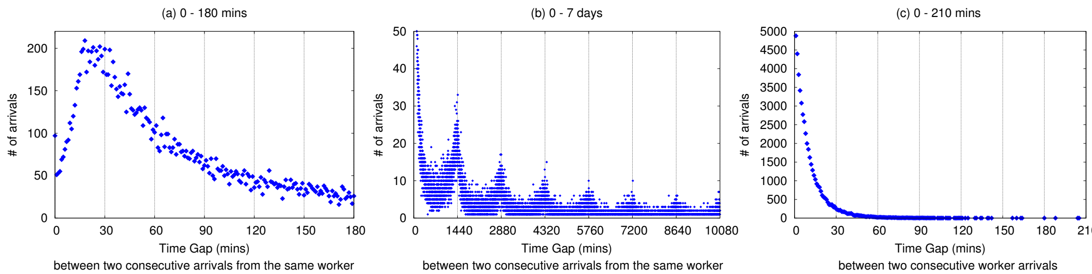


Figure 5: Time Gap between Two Consecutive Arrivals of Workers

However, parameters in *Q-network(w)* are shared by all workers. (s_i, a_i, r_i) includes information on whether the worker with feature f_{w_i} likes the task with feature f_{a_i}. Even when the task is new and few workers complete it, the information in (s_i, a_i, r_i) is very important. Therefore, we predict s_{i+1} in order to update *Q-network(w)* using (s_i, a_i, r_i, s_{i+1}) immediately. *Q-network(w)* will have a better estimation for a_i and decide whether assigning a_i to the workers who are similar to w_i.

Therefore, we design a *predictor(w)* to predict the transition probability $Pr(s_{i+1}|s_i, a_i, r_i)$ and the feature of the future state $f_{s_{i+1}}$ after we obtain the feedback and reward r_i for (s_i, a_i). This helps our framework to satisfy the requirement of handling online changes and achieving real-time interaction.

2) **Design:** First, the worker feature f_{w_i}, i.e., the distribution of recently completed tasks, needs to be updated by r_i. Based on the MDP(w) definition, $w_{i+1} = w_i$ and the worker feature $f_{w_{i+1}}$ at s_{i+1} is the updated feature f_{w_i}. Second, we consider T_{i+1} and its feature $f_{T_{i+1}}$ at s_{i+1}. The change between T_i and T_{i+1} comes mainly from the expired tasks. We need to check whether $t_j \in T_i$ has expired at Time_{i+1} (i.e., the happening time of s_{i+1}) and remove expired tasks from T_{i+1}. Time_{i+1} is stochastic and we need to learn its distribution from the environment. From the history, we find that there is a pattern of the same worker arrivals, i.e., a worker comes again within a short time, or comes again after 1 day, 2 days, etc. up to one week later (see the distribution of the time gap between two consecutive arrivals from the same worker in Fig. 5(a) and 5(b)). To capture the pattern, we maintain a function $\phi(g)$, where g is the time gap, and $\phi(g = \text{CurrentTime} - \text{TimeOfLastArrival}_w)$ is the probability whether the worker w comes again currently. We set $g \in [1, 10080]$ minutes since the probability of $\phi(g) > 0, g > 10080$ is small and can be ignored. Note that $\phi(g)$ is initialized by the history and iterative updated when we have a new sample.

Finally the distribution of Time_{i+1} is $\text{Time}_i + \phi(g)$, $g \in [1, 10080]$. Given a possible Time_{i+1}, *predictor(w)* checks whether tasks are expired and generates s_{i+1} and $f_{s_{i+1}}$.

For *learner(w)*, we use the method introduced in Sec. II-D to update the parameters of *Q-Network(w)* by transitions stored in the memory. Our loss function can be written as

$$
\begin{aligned}
&L(\theta) = \mathbb{E}_{\{(s_i,a_i,r_i)\}}[(y_i - Q(s_i, a_i; \theta))^2] \\
&y_i = r_i + \gamma \sum_g Pr(s_{i+1}|g) \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta)
\end{aligned} \tag{3}
$$

where $Pr(s_{i+1}|g) = \phi(g)$ and $g \in [1, 10080]$. Actually, we do not calculate $\max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta)$ for all possible g. The value $\max_{a_{i+1}} Q$ may change when a task $t_{j'} \in T_i$ expires. Thus, the maximum times we compute $\max_{a_{i+1}} Q$ is max_T.

Here, we also use the double Q-learning algorithm [28] to avoid overestimating Q values. The algorithm uses another neural network $\widetilde{Q}$ with parameters $\widetilde{\theta}$, which has the same structure as the Q-Network Q, to select actions. The original Q-Network Q with parameters θ is used to evaluate actions:

$$y_i = r_i + \gamma \sum_g Pr(s_{i+1}|g) \widetilde{Q}(s_{i+1}, \arg\max_{a_{i+1}} Q(s_{i+1}, a_{i+1}|\theta)|\widetilde{\theta}).$$

Parameters $\widetilde{\theta}$ are slowly copied from parameters θ during learning. Accordingly, the gradient update is

$$
\begin{aligned}
&\nabla_\theta L(\theta) = \mathbb{E}_{\{(s_i,a_i,r_i)\}}[r_i + \gamma \sum_g Pr(s_{i+1}|g) \\
&\widetilde{Q}(s_{i+1}, \arg\max_{a_{i+1}} Q(s_{i+1}, a_{i+1}|\theta)|\widetilde{\theta}) - Q(s_i, a_i)] \nabla_\theta Q(s_i, a_i).
\end{aligned} \tag{4}
$$

Prioritized experience replay [24] is used to learn efficiently.

IV. Modules for MDP(r)

Same as the previous section, we describe the modules related to MDP(r). We first supplement the quality of tasks in the task feature. Then, we introduce the modules, *state transformer*, *feedback transformer(r)* and *learner(r)* one by one. Besides, we also detail the module *explorer* to perform trial-and-error actions.

A. Feature Construction

In addition to the features of tasks and workers introduced in Sec. III-A, we also use the quality of workers $q_{w_i} \in [0, 1]$ and the quality of tasks $q_{t_j} \in \mathbb{R}$ to predict the benefit of requesters. We assume that we already know the the quality of workers from their worker answer history or the qualification tests with the ground truth. The quality of tasks is decided by all the workers who completed it. We assume that workers who come at timestamps $i \in I_{t_j}$, complete the task t_j. We use the Dixit-Stiglitz preference model [8] to calculate task quality q_{t_j} based on the law of *diminishing marginal utility*:

$$q_{t_j} = (\sum_{i \in I_{t_j}} (q_{w_i})^p)^{1/p}, p \geq 1. \tag{5}$$

where p controls how much marginal utility we can get with multiple workers.

Let us explain the above equation using two typical examples. The first is AMT, where each task has multiple independent micro-tasks and each micro-task is only allowed to be answered by one worker. The quality of mirco-tasks is equal to the quality of the answering worker. Since the micro-tasks are independent, the quality of the task is the sum of the qualities of the micro-tasks which comprise it, where $q_{t_j} = \sum_{i \in I_{t_j}} q_{w_i}$, $p = 1$. The second example is competition-based crowdsourcing platforms, where tasks can be answered by many workers, but only one worker is selected to be awarded after the deadline. The quality should be defined as $q_{t_j} = \max_{i \in I_{t_j}} q_{w_i}$, i.e., p is set to infinity.

B. State Transformer and Q Network

The *State Transformer* and the *Q-Network* are as defined in Sec. III-B; we only need to add the two dimensions (q_{w_i} and q_{t_j}) to the input.

C. Action $\mathcal{A}$, Feedback and Reward $\mathcal{R}$

Same as before, the action $a_i = t_j$ with the maximum $Q_r(s_i, t_j)$ is recommended, if the agent assigns one task to w_i. To recommend a list, the action is $a_i = \sigma(T_i) = \{t_{j_1}, t_{j_2}, ...\}$, where t_{j*}'s are ranked in descending order of $Q_r(s_i, t_{j*})$.

From the requester's perspective, the goal is to obtain the greatest possible quality of results before the deadline of tasks. Thus the immediate reward is $q_{t_j}^{\text{new}} - q_{t_j}^{\text{old}}$ if the worker is assigned to the task t_j and finishes it. The reward is 0 if the worker skips the task. When the action is to recommend a list of k tasks, the immediate reward is $q_{t_{j*}}^{\text{new}} - q_{t_{j*}}^{\text{old}}$ if the worker selects the task $q_{t_{j*}}$ and completes it. The reward is 0 if the worker does not finish any task.

D. Future State, Memory Storer and Learner

1) **Challenges:** Different from MDP(w), the next worker in MDP(r) arrives fast as the platforms have many active workers. However, we find that when we use the real worker w_{i+1} to combine s_{i+1}, it is hard for Deep Q-network to converge. The reason is that there are too many possibilities between w_i in s_i and w_{i+1} in s_{i+1}, leading to very sparse transitions between s_i and s_{i+1}. It is hard for *Q-network(r)* to estimate the accurate transition probability $P(s_{i+1}|s_i, a_i, r_i)$. Hence, we use the expectation of the next worker instead of the real next worker, to construct future state s_{i+1} and update *Q-network(r)*.

2) **Design:** After we obtain the feedback and reward r_i for (s_i, a_i), the first thing is to update the worker feature f_{w_i} when $r_i > 0$. Besides, we also need to update the quality in the task feature f_{t_j} which is completed.

From the benefit of requesters, the qualities of tasks are influenced by all workers. Thus the future state s_{i+1} happens when the next worker w_{i+1} (no matter whether $w_{i+1} = w_i$) comes. Here the *future state predictor(r)* not only needs to estimate the next timestamp and check for expired tasks, but also has to predict the next worker.

We first explain how we predict Time_{i+1}. Fig. 5(c) shows the distribution of the time gap between two consecutive arrivals, no matter whether these two arrivals are from the same or different workers. It is a long-tail distribution, which means that workers come to the platform and complete tasks frequently. We also maintain a function $\varphi(g)$, where g is the time gap, and $\varphi(g = \text{Time}_{i+1} - \text{Time}_i)$ is the probability that a worker comes at Time_{i+1} if the last worker comes at Time_i. We set $g \in [0, 60]$ minutes because 99% of time gaps in the history are smaller than 60 minutes. Same as $\phi(g)$, $\varphi(g)$ is also built from the history and iteratively updated at each new sample. Then the distribution of Time_{i+1} is $\text{Time}_i + \varphi(g)$.

After we know Time_{i+1}, we compute the distribution of the coming workers. For each worker $w \in W^{\text{old}}$ who already came before, we know the feature of worker f_w and the time gap between his/her last arrival time and Time_{i+1} (i.e., $g_w = \text{Time}_{i+1} - \text{TimeOfLastArrival}_w$). From function $\phi(g)$ defined in Sec. III-D, we obtain probability $\phi(g_w)$. Besides, we also consider the probability that a new worker comes. From the history, we also maintain the rate of new workers p_{new}, and we use the average feature of old workers $\bar{f}_w$ to represent the feature of new workers. Finally, we normalize, integrate and obtain the probability for a coming worker w:

$$Pr(w_{i+1} = w) = \begin{cases} (1 - p_{\text{new}}) \frac{\phi(g_w)}{\sum_{w' \in W^{\text{old}}} \phi(g_{w'})} & \text{when } w \in W^{\text{old}} \\ p_{\text{new}} & \text{when } w \text{ is new} \end{cases}$$

Given g and w_{i+1}, we use the method described in Sec.III-D to calculate T_{i+1} and s_{i+1}.

For *learner(r)*, our loss function is

$$\begin{aligned} L(\theta) &= \mathbb{E}_{\{(s_i, a_i, r_i)\}}[(y_i - Q(s_i, a_i; \theta))^2] \\ y_i &= r_i + \gamma \sum_g \sum_{w_{i+1}} Pr(s_{i+1}|g, w_{i+1}) \\ &\qquad \widetilde{Q}(s_{i+1}, \arg\max_{a_{i+1}} Q(s_{i+1}, a_{i+1}|\theta)|\widetilde{\theta}) \end{aligned} \tag{6}$$

where $Pr(s_{i+1}|g, w_{i+1}) = \varphi(g) Pr(w_{i+1}|g)$ and $g \in [0, 60]$ while $w_{i+1} \in W^{\text{old}}$ or w_{i+1} is new.

Accordingly, the gradient update is

$$\begin{aligned} \nabla_\theta L(\theta) = \mathbb{E}_{\{(s_i, a_i, r_i)\}}[r_i + \gamma \sum_g \sum_{w_{i+1}} Pr(s_{i+1}|g, w_{i+1}) \\ \widetilde{Q}(s_{i+1}, \arg\max_{a_{i+1}} Q(s_{i+1}, a_{i+1}|\theta)|\widetilde{\theta}) - Q(s_i, a_i)]\nabla_\theta Q(s_i, a_i). \end{aligned} \tag{7}$$

However, computing $\widetilde{Q}(s_{i+1}, \arg\max_{a_{i+1}} Q(s_{i+1}, a_{i+1}))$ for all possible g and w_{i+1} may take a long time. Here are two possible methods to speed this up. One method is to limit the number of possible workers. We can set a threshold to disregard workers with low coming probability. Another method is to use the expectation of the feature of all possible w_{i+1} instead of computing them. The expectation of the feature of the next worker is $\bar{f}_{w_{i+1}} = \sum_{w_{i+1}} Pr(w_{i+1}|g) f_{w_{i+1}}$, the expectation of future state feature is $\bar{f}_{s_{i+1}} = [\bar{f}_{w_{i+1}}, f_{T_{i+1}}]$ and the loss function and updating equation are given by Eq. 3 and Eq. 4, respectively.

E. Explorer

Exploration is an important step to find the relationship between workers and tasks or the correlation among currently available tasks. The most straightforward strategy to conduct exploration is ϵ-greedy [20]. This approach randomly selects a task or sorts tasks with a probability of ϵ, or follows

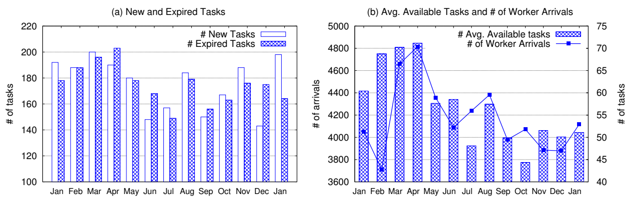


Figure 6: New/Expired/Available Tasks and Worker Arrivals

$Q(s_i, t_j)$ to recommend a list of tasks with probability $1 - \epsilon$. This is suitable for recommending one task but does not perform well in recommending a list of tasks because it is too random. Instead of ignoring $Q(s_i, t_j)$ totally, we add a random value v into $Q(s_i, t_j)$ with a probability of ϵ. We generate v as a normal distribution where the mean is zero and the standard deviation is the same as that of the current Q values ($Q(s_i, t_j), \forall t_j \in \{T_i\}$). Besides, we also use a decay factor to multiply the standard deviation, in order to reduce randomness when the Q-network is relatively mature.

V. Experiments

A. Experimental Setup

1) Dataset: We conduct experiments on a real dataset collected from the commercial crowdsourced platform CrowdSpring [2]. This platform helps requesters publish tasks to obtain high-quality custom logos, names, designs, etc. Most of the tasks are public, i.e., we can see all the information including start date and deadline, category, sub-category, domain and the relationship of workers who completed it. We use a web crawler to obtain all the information about public tasks ranging from Jan 2018 to Jan 2019. There are totally 2285 tasks created and 2273 tasks expired. There are about 1700 active workers during the entire process. We show the number of new and expired tasks per month in Fig. 6(a), which are around 180. Besides, Fig. 6(b) shows that there are about 4200 arrivals of workers per month. When a worker comes, s/he can see 56.8 available tasks on average.

We also generated a synthetic dataset, simulating the real dataset using factors considered in [32]. We consider the arriving density of workers, the distribution of qualities of workers and scalability of updating time.

2) Settings: We restore the process of the arrival of workers, creation or expiration of tasks as time goes by. The collected dataset records each timestamp i when a worker w_i completes a task. We assume that one completion corresponds to one worker arrival. The completed task is considered to be interesting to w_i. The remaining available but uncompleted tasks are assumed not to be interesting to w_i at timestamp i.

During the experiments, we run different algorithms to recommend a task t_j from the currently available tasks for w_i at timestamp i. Considering the benefit of workers, the reward/label (for reinforcement/supervised learning) is 1 if the recommended task is the completed task. For the benefit of requesters, the reward/label is the quality gain of t_j. We set $p = 2$ and utilize Eq. 5 to compute: $q_{t_j}^{\text{new}} - q_{t_j}^{\text{old}} = ((q_{t_j}^{\text{old}})^p + (q_{w_i})^p)^{1/p} - q_{t_j}^{\text{old}}$. For each worker w_i, there is a score in $\in [0, 100]$ in CrowdSpring. We extract this value and divide it by 100 to derive a normalized (in $[0, 1]$) quality q_{w_i} for each worker.

When we consider the action to recommend a list of tasks, we assume that the worker looks through the list in order until s/he finds the completed task. The reward/label for the skipped tasks which are in front of the selected one is 0. The reward/label for the completed task is 1 or the quality gain. The skipped and the completed tasks are considered to be the feedback when updating/training the model.

3) Evaluation Measures: Depending on whether the agent recommends one task or a list of tasks, and considering the benefit of workers or requesters, we use the following measures to evaluate the performance of methods.

For the benefit of workers:

- Worker Completion Rate (**CR**). At timestamp i the worker w_i comes, the agent recommends a task t_j. We compute the cumulative number of completions rate where $y_{ij} = 1$ means that the task is completed and $y_{ij} = 0$ means that the task is skipped.

$$CR = \frac{\sum_i y_{ij}}{\text{number of total timestamps}} \tag{8}$$

- **nDCG-CR**. Instead of one task, the agent recommends a list of tasks. We apply the standard Normalized Discount Cumulative Gain proposed in [12] to evaluate the success of the recommended list $L = \{t_{j_1}, t_{j_2}, ..., t_{j_{n_i}}\}$ for all available tasks at timestamp i. r is the rank position of tasks in the list, n_i is the number of available tasks. We assume that w_i looks through the tasks in order and completes the first task t_{j_r} s/he is interested in. $y_{ij_r} = 1$ indicates that t_{ij_r} is completed; all other $y_{ij_{r'}}$ are 0.

$$nDCG - CR = \frac{\sum_i \sum_{r=1}^{n_i} \frac{1}{log(1+r)} y_{ij_r}}{\text{number of total timestamps}} \tag{9}$$

- Top-k Completion Rate (**kCR**). We limit the length of the list to k, i.e., the agent recommends k tasks $\{t_{j_1}, t_{j_2}, ..., t_{j_k}\}$ for the worker w_i. We assume that k tasks also have an order and that w_i looks through the tasks in order and completes the first interesting task t_{j_r}.

$$kCR = \frac{\sum_i \sum_{r=1}^{k} \frac{1}{log(1+r)} y_{ij_r}}{\text{number of total timestamps}} \tag{10}$$

For the benefit of requesters:

- Task Quality Gain (**QG**). At timestamp i, worker w_i comes and the agent recommends a task t_j. We compute the cumulative gain of the qualities of tasks. If the task is skipped, $g_{ij} = 0$. Otherwise, g_{ij} is the difference of the task quality q_{t_j} before and after w_i finishes t_j.

$$QG = \sum_i g_{ij} = \sum_i q_{t_j}^{\text{new}} - q_{t_j}^{\text{old}} \tag{11}$$

- **nDCG-QG**. Same as **nDCG-CR**, we apply nDCG to give different weights for rank positions of tasks. y_{ij_r}

indicates whether t_{j_r} is completed, and g_{ij_r} is the gain in the quality of t_{j_r}.

$$nDCG-QG = \sum_{i}\sum_{r=1}^{n_i} \frac{1}{log(1+r)} y_{ij_r} g_{ij_r} \quad (12)$$

- Top-k Task Quality Gain (**kQG**). Similarly, we limit the recommended list into k tasks $\{t_{j_1}, t_{j_2}, ..., t_{j_k}\}$ for the worker w_i.

$$kQG = \sum_{i}\sum_{r=1}^{k} \frac{1}{log(1+r)} y_{ij_r} g_{ij_r} \quad (13)$$

4) Competitors: We compared our approach with five alternative methods. The worker and task features of all these methods are updated in real-time. The methods using supervised learning (Taskrec(PMF)/Greedy+Cosine Similarity/Greedy+Neural Network) predict the completion probability and the quality gain of tasks and select one available task or sort the available tasks based on predicted values. The parameters of the models are updated at the end of each day. For the reinforcement learning methods (LinUCB/DDQN), the parameters are updated in real-time after one recommendation.

- **Random**. For each worker arrival, one available task is picked randomly, or a list of tasks is randomly sorted and recommended.
- **Taskrec (PMF)**. Taskrec [34] is a task recommendation framework for crowdsourcing systems based on unified probabilistic matrix factorization. Taskrec builds the relationship between the worker-task, worker-category and task-category matrices and predicts the worker completion probability. It only considers the benefit of workers.
- **SpatialUCB/LinUCB**. SpatialUCB [10] adapts the Linear Upper Confidence Bound [17] algorithm in online spatial task assignment. We adapt SpatialUCB in our setting by replacing the worker and task features. SpatialUCB selects one available task or sorts the available tasks according to the estimated upper confidence bound of the potential reward. For the benefit of requesters, we add the quality of workers and tasks as features and then predict the gain quality of the tasks.
- **Greedy+Cosine Similarity**. We regard the cosine similarity between the worker feature and task feature as the completion rate, and select or sort tasks greedily according to the completion rate. For the benefit of requesters, we use the actual value of the quality gain by multiplying the completion probability of each task to pick or rank the available tasks.
- **Greedy+Neural Network**. We input the worker and task features into a neural network of two hidden-layers to predict the completion rate. For the benefit of requesters, we add the quality of workers and tasks as features and then predict the gain quality of the tasks.
- **DDQN**. Double Deep Q-Network is our proposed framework, In the first two experiments, we use a version of DDQN that only considers the benefit of workers or requesters when comparing it with the other approaches. For the experiments regarding MDP(w), we test two versions of DDQN with simple state(w) and complex state(r) respectively, as described in Sec. II.

B. *Experimental Results (real dataset)*

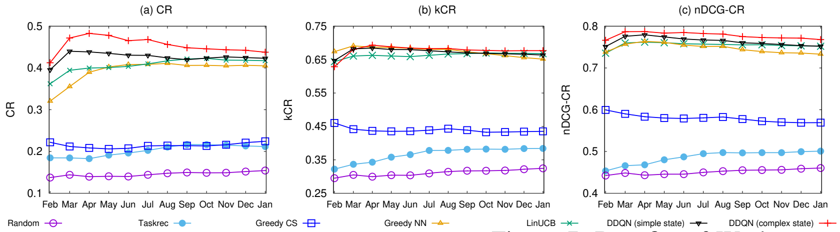
1) Implementation details: We utilize the data in the first month (Jan 2018) to learn initial parameters of the model. The entire updating/testing process runs from Feb 2018 to Jan 2019. To solve the cold-start problem of a new worker, we use the first five tasks s/he completed to initialize her/his features.

We use Pytorch to implement all the algorithms and used a GeForce GTX 1080 Ti GPU. The dimensionality of output features in each layer of Q-Network is set to 128. The buffer size for DDQN is 1000 and we copy parameters $\widetilde{\theta}$ from θ after each 100 iterations. The learning rate is 0.001 and the batch size is 64. We set the discount factor $\gamma = 0.5$ for the benefit of requesters and $\gamma = 0.3$ for workers. To do the exploration, we set the initial $\epsilon = 0.9$, and increase it until $\epsilon = 0.98$ for assigning a task. To recommend the task list, ϵ is always 0.9, and the decay factor for standard deviations is set as 1 initially and decreases into 0.1 with further learning.

2) Considering the benefit of workers: We show QR, kQR and nDCG-QR for each method at the end of each month in Fig. 7. Random performs the worst since it never predicts the worker completion probability. The reason behind the bad performance of Taskrec is that it only uses the category of tasks and workers and ignores the domain or award information. Because of the simple model to compute the similarity of tasks for a certain worker, Greedy CS also performs badly. Greedy NN uses the neural network to predict the relationship between tasks and workers, and updates the parameters every day. However, it only considers the immediate reward. Thus it performs worse than LinUCB and DDQN. LinUCB utilizes all information of features of workers and tasks, estimates the upper confidence bound of the reward and updates parameters after each worker feedback. So its performance is second to DDQN. Our proposed model, DDQN, not only uses the neural network to model the complex relationship between workers and tasks, but also predicts the immediate and future reward and updates the parameters after each worker feedback. Since DDQN (complex state) considers the relationship among the available tasks, DDQN (complex state) outperforms all competitors, including DDQN (simple state).

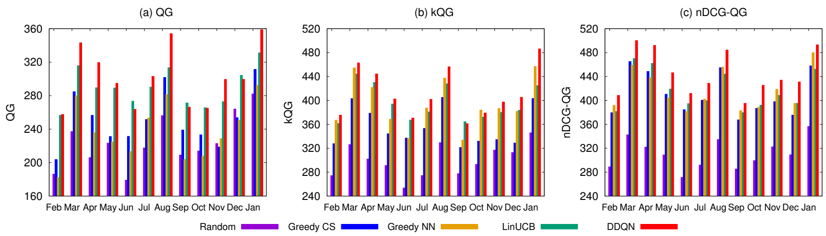
The table lists the final value of CR, kCR and nDCG-CR of each method; our approach is around 2% better than other models.

3) Considering the benefit of requesters: We show the separate quality gain of tasks in each month in Fig. 8. Note that the gain is not consistently increasing but it is related to the number of worker requests at each month in Fig. 6(b). The random method again performs the worst. Although we give the real value of the quality gain of each task, Greedy CS still cannot recommend tasks with the high gain which are completed by workers. Greedy NN and LinUCB perform similarly (in kQR and nDCG-QR). Greedy NN achieves a better estimation than LinUCB when aggregating the quality



	CR	kCR	nDCG-CR
Random	0.154	0.325	0.460
Taskrec	0.212	0.384	0.501
Greedy CS	0.224	0.435	0.569
Greedy NN	0.405	0.651	0.733
LinUCB	0.417	0.668	0.752
DDQN (simple state)	0.423	0.664	0.752
DDQN (complex state)	**0.438**	**0.677**	**0.768**

Figure 7: Benefits of Workers



	QR	kQR	nDCG-QR
Random	2697.96	3598.05	3733.52
Greedy CS	3017.46	4269.64	4929.46
Greedy NN	2854.58	4716.83	4998.76
LinUCB	3474.04	4731.97	4999.67
DDQN	**3625.34**	**4943.29**	**5350.98**

Figure 8: Benefits of Requesters

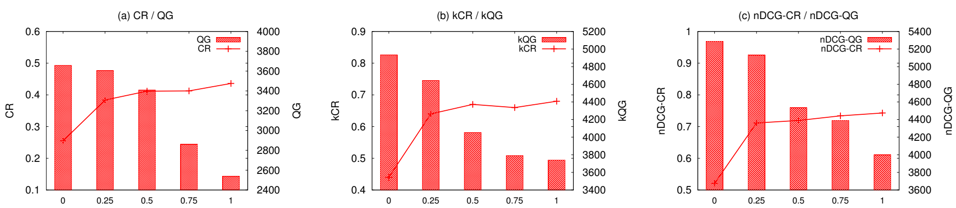


Figure 9: Balance of Benefits

	Time (Sec)
Taskrec	3.193
Greedy NN	7.476
LinUCB	0.073
DDQN	0.042

Table I: Efficiency

gain and completion rate of each task, while LinUCB could update the model more timely. Still, the performance of DDQN is the best because it utilizes the nonlinear and complex Q-network to approximate, predict and integrate the gain and completion rate of tasks in the long term.

The table lists the final value of QR, kQR and nDCG-QR of each method; our method is at least 4.3% better than its competitors.

4) Balance of benefits: We integrate the two benefits of workers and requesters using the weighed sum model $Q(s_i,t_j) = wQ_w(s_i,t_j) + (1-w)Q_r(s_i,t_j)$ and show the result in Fig. 9. We test the cases of $w = 0, 0.25, 0.5, 0.75$ and 1.0. From the trend of CR and QG in Fig. 9(a), we find that the change of QG is small from $w = 0$ to 0.25 while the shift in CR is small from $w = 0.25$ to 1. Thus, the weight that achieves holistic maximization is around 0.25. This analysis also holds for kCR / kQG and nDCG-CR / nDCG-QG.

5) Efficiency: We show the updating time of each method in Table I. Random and Greedy CS are not included because they do not have a model to update. Taskrec and Greedy NN are supervised learning-based methods which update the whole model with incremental data. During the entire process, although we train them with newly collected data once at the end of each day, the average updating time during the whole process is still longer than 3s. LinUCB and DDQN are reinforcement learning-based methods, which update the existing model quickly after collecting every new feedback. The average updating time is in the order of milliseconds.

C. Experimental Results (synthetic dataset)

1) Arriving density of workers: We change the number of worker arrivals (50k) in the real dataset using sampling with replacement. We range the sampling rate of worker arrivals from 0.5 to 2.0, resulting in 25k to 100k arrivals. For the same arrival which is sampled multiple times, we add a delta time following a normal distribution where the mean and std are 1 day, to make their arrival times distinct.

Fig. 10(a) and 10(b) show the change of CR / QG with a different sampling rate of worker arrivals. Because CR is divided by the number of timestamps (i.e., the number of worker arrivals), the values of all the methods are similar at different sampling rates. QG is the absolute value, so the values of all the methods increase at a high sampling rate. The performance of our algorithm DDQN is typically better than that of others for both CR and QG in the different cases.

2) Distribution of qualities of workers: We change the qualities of workers by adding noise. We generate the noise from a normal distribution and add it to the original quality of workers randomly. We tried four distributions: $\mathcal{N}(-0.4, 0.2)$, $\mathcal{N}(-0.2, 0.2)$, $\mathcal{N}(0.0, 0.2)$ and $\mathcal{N}(0.2, 0.2)$. The result is shown in Fig. 10(c). Since the quality of workers only affects the quality gain of tasks, we show the change of QG for various worker qualities. Obviously, the sum of qualities of tasks becomes larger as the quality of workers increases. Moreover, DDQN always performs better than its competitors, no matter whether the worker qualities are low or high.

3) Scalability: There are two types of time in DDQN: one is the time for deciding actions, and the other is the time for

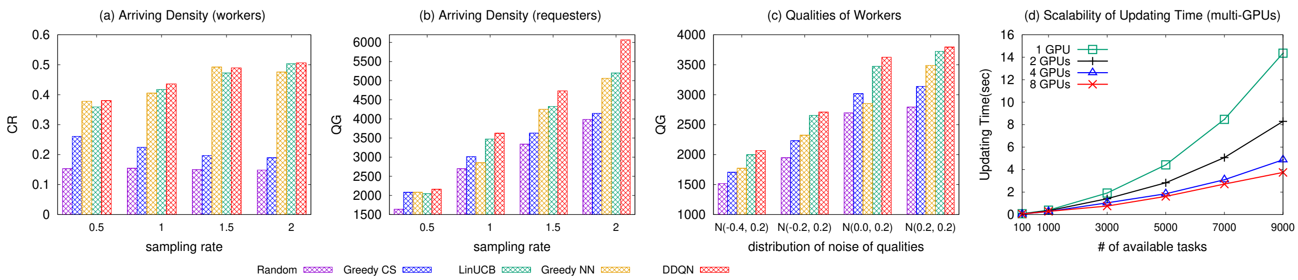


Figure 10: Synthetic Results

updating parameters in the model. There exists a real-time requirement for deciding actions. For updating the model, the smaller the update cost is, the better the performance is. The reason is that the updated model could be more accurate for the following recommendations. DDQN can decide actions in real-time (< 0.01s) even if there are 10k available tasks at a time. To measure the update cost, we vary the number of the currently available tasks from 100 to 9k on 1, 2, 4 and 8 GPUs in Fig. 10(d). Generally speaking, the cost is approximately linear to the number of available tasks. Besides, the running time can be reduced by one third if we use twice as many GPUs. DDQN updates 1k tasks in 0.5s using one GPU. Parallel computation (8-32 GPUs) can be used for larger platforms.

VI. Related Work

A. Reinforcement learning and deep reinforcement learning

Unlike supervised learning which requires labeled training data and infers a classification or a regression model, reinforcement learning (RL) learns how agents should take sequences of actions in an unknown environment in order to maximize cumulative rewards. The environment is formulated as a Markov Decision Process [4], and the agent makes a tradeoff between exploring untouched space and exploiting current knowledge. RL methods are mainly divided into three categories, model-free, model-based and policy search, based on the assumption of MDPs. In this paper, we utilize the model-free method, Q-learning [31], which estimates a Q-function iteratively using Bellman backups [27] and acts greedily based on Q-functions until convergence.

Deep reinforcement learning is a combination of RL and deep learning. Deep RL has experienced dramatic growth recently in multiple fields, such as games (AlphaGo) [20], [28], [30], robotics [9], natural language processing [21], computer vision [18], finance [7], computer systems [16], [19], [36] and recommender systems [37], [38]. Deep Q-Network (DQN) is an improved version of Q-learning with a neural network. [36] is the first work that applies DQN to solve the database configuration problem. However, the definitions of state, action, and reward in [36] are different and cannot directly be transferred to our problem. The applications of DQN in recommender systems [37], [38] are the most related. Instead of recommending items to users, we arrange tasks to workers. But recommender systems only consider the benefit of users, which is just one objective of our framework.

In summary, to solve the task arrangement problem, we integrate two MDPs, design a new representation of states and revise Q-value computing equations with probabilistic future states compared with other previous DQN works.

B. Task Recommendation and Assignment in Crowdsourcing

Crowdsourcing is an effective way to harness human effort to address tasks which the machine cannot solve automatically [15], [25], [26]. Some learning-based methods have been developed to recommend or assign tasks into workers.

1) supervised learning: Content-based recommendation methods [3], [23], [33] match task profiles to worker profiles. They firstly define features of workers and tasks (e.g., a bag of words from user profiles) and the task selection history for workers. Then they calculate similarity and recommend based on these features. Collaborative filtering has also been used in crowdsourcing. For example, [34] builds the task-worker, worker-category and task-category relationship matrices, and applies probabilistic matrix factorization to capture workers' preferences. [22] uses category-based matrix factorization and kNN algorithms to recommend top-k tasks to workers.

2) reinforcement learning: Several studies have applied reinforcement learning for spatial crowdsourcing [10], [32]. [10] proposes a multi-armed bandit approach for online spatial task assignment. The task acceptance rate of the worker is modeled as a linear model of the travel distance and task type, and the goal is to maximize the cumulative success rate of assignments. In [32], an RL-based algorithm is proposed to solve a dynamic bipartite graph matching problem. However, a simple state representation is used, i.e., the number of available nodes in the bipartite graph, which limits the power of RL.

VII. Conclusions

In this paper, we proposed a Deep Reinforcement Learning framework. We consider the benefits of workers and requesters simultaneously, helping platforms to maximize their profit. We used novel and effective representations of state, action, reward, state transition and future state, and new equations for deriving Q values. Experiments on both real and synthetic datasets verify the effectiveness of our framework.

Acknowledgement. We would like to thank the reviewers for the insightful comments. Caihua Shan and Reynold Cheng were supported by the Research Grants Council of HK (RGC Projects HKU 17229116, 106150091, and 17205115), HKU (Projects 104004572, 102009508, and 104004129), and the Innovation and Technology Commission of HK

(ITF project MRP/029/18). Nikos Mamoulis has been co-financed by the European Regional Development Fund, Research–Create–Innovate project "Proximiot" (T1EDK-04810). Guoliang Li was supported by NSF of China (61925205, 61632016), Huawei, and TAL education. Xiang Li is the corresponding author.

APPENDIX

Definition 1: (Permutation-invariant Function) Let $\{\sigma\}$ be the set of all permutations of indices $\{1, .., n\}$. A function of $f : X^n \to Y^n$ is permutation-invariant iff for any permutation in $\{\sigma\}$, $f(\sigma x) = \sigma f(x)$.

Proof 1: (**rFF function is Permutation-invariant.**)

Let $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_n} \end{bmatrix}$, where each row is the feature of an item in the set. Then, rFF($\boldsymbol{X}$) = relu$(\boldsymbol{X}W+b) = \begin{bmatrix} \text{relu}(\boldsymbol{x_1}W+b) \\ \vdots \\ \text{relu}(\boldsymbol{x_n}W+b) \end{bmatrix}$.

The value in row i of rFF($\boldsymbol{X}$) only depends on $\boldsymbol{x_i}$ and is independent to $\boldsymbol{x_j}$ where $\forall j \neq i$.

Proof 2: (**MultiHead Self-Attention Layer is Permutation -invariant.**)

First of all, we prove that each head$_j$ = Att(XW_j^Q, XW_j^K, XW_j^V) is permutation-invariant.

Similarly, let $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_n} \end{bmatrix}$ and $W_j^Q(W_j^K)^T = W_j'$, then $XW_j^Q(XW_j^K)^T = XW_j'X^T = \begin{bmatrix} \boldsymbol{x_1}W_j'\boldsymbol{x_1^T}, \cdots, \boldsymbol{x_1}W_j'\boldsymbol{x_n^T} \\ \vdots \\ \boldsymbol{x_n}W_j'\boldsymbol{x_1^T}, \cdots, \boldsymbol{x_n}W_j'\boldsymbol{x_n^T} \end{bmatrix}$

. After multiplying XW_j^V and scaling by $\omega(\cdot)$, head$_j$ becomes $\begin{bmatrix} \sum_{i=1}^n \omega(\boldsymbol{x_1}W_j'\boldsymbol{x_i^T})\boldsymbol{x_i}W_j^V \\ \vdots \\ \sum_{i=1}^n \omega(\boldsymbol{x_n}W_j'\boldsymbol{x_i^T})\boldsymbol{x_i}W_j^V \end{bmatrix}$. Each value in row i of head$_j$ depends on $\boldsymbol{x_i}$ and weighed sum of $\boldsymbol{x_j}, \forall j$, which is also permutation-invariant.

Next we consider MultiHead(X, X, X). Because of Concat(head$_1$, ..., head$_h$) and multiplying W^O are both row-wise, we can prove the permutation-invariance in the same way as for the rFF function.

REFERENCES

[1] Amazon Mechanical Turk. https://www.mturk.com/mturk/.
[2] CrowdSpring. https://www.crowdspring.com/explore/.
[3] V. Ambati, S. Vogel, and J. Carbonell. Towards task recommendation in micro-task markets. In *AAAI Workshop*, 2011.
[4] R. Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 1957.
[5] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv*, 2014.
[6] N. Craswell, O. Zoeter, M. Taylor, and B. Ramsey. An experimental comparison of click position-bias models. In *WSDM*, 2008.
[7] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems*, 2016.
[8] A. K. Dixit and J. E. Stiglitz. Monopolistic competition and optimum product diversity. *The American economic review*, 1977.
[9] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *ICRA*, 2017.
[10] U. U. Hassan and E. Curry. A multi-armed bandit approach to online spatial task assignment. In *UIC-ATC-ScalCom*, 2014.
[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997.
[12] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *TOIS*, 2002.
[13] N. Kaufmann, T. Schulze, and D. Veit. More than fun and money. worker motivation in crowdsourcing-a study on mechanical turk. In *AMCIS*, 2011.
[14] J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *ICML*, 2019.
[15] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *TKDE*, 2016.
[16] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: a query-aware database tuning system with deep reinforcement learning. *PVLDB*, 2019.
[17] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, 2010.
[18] F. Liu, S. Li, L. Zhang, C. Zhou, R. Ye, Y. Wang, and J. Lu. 3dcnn-dqn-rnn: A deep reinforcement learning framework for semantic parsing of large-scale 3d point clouds. In *ICCV*, 2017.
[19] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *HotNets*, 2016.
[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.
[21] K. Narasimhan, T. Kulkarni, and R. Barzilay. Language understanding for text-based games using deep reinforcement learning. In *EMNLP*, 2015.
[22] M. Safran and D. Che. Efficient learning-based recommendation algorithms for top-n tasks and top-n workers in large-scale crowdsourcing systems. *TOIS*, 2018.
[23] B. Satzger, H. Psaier, D. Schall, and S. Dustdar. Stimulating skill evolution in market-based crowdsourcing. In *BPM*, 2011.
[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv*, 2015.
[25] C. Shan, N. Mamoulis, G. Li, R. Cheng, Z. Huang, and Y. Zheng. T-crowd: Effective crowdsourcing for tabular data. In *ICDE*, 2018.
[26] C. Shan, N. Mamoulis, G. Li, R. Cheng, Z. Huang, and Y. Zheng. A crowdsourcing framework for collecting tabular data. *TKDE*, 2019.
[27] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[28] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, 2016.
[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, 2017.
[30] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *ICML*, 2016.
[31] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 1992.
[32] W. Yansheng, T. Yongxin, L. Cheng, X. Pan, X. Ke, and L. Weifeng. Adaptive dynamic bipartite graph matching: A reinforcement learning approach. In *ICDE*. 2019.
[33] M.-C. Yuen, I. King, and K.-S. Leung. Task recommendation in crowdsourcing systems. In *CrowdKDD*, 2012.
[34] M.-C. Yuen, I. King, and K.-S. Leung. Taskrec: A task recommendation framework in crowdsourcing systems. *Neural Processing Letters*, 2015.
[35] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. Deep sets. In *NIPS*, 2017.
[36] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, 2019.
[37] X. Zhao, L. Zhang, Z. Ding, L. Xia, J. Tang, and D. Yin. Recommendations with negative feedback via pairwise deep reinforcement learning. In *SIGKDD*, 2018.
[38] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li. Drn: A deep reinforcement learning framework for news recommendation. In *WWW*, 2018.