Match Plan Generation in Web Search with Parameterized Action Reinforcement Learning

Ziyan Luo[1,2,*,†], Linfeng Zhao[1,3,*,†], Wei Cheng[1,4,*,†], Sihao Chen[1,5,†], Qi Chen[1], Hui Xue[1]
Haidong Wang[1], Chuanjie Liu[1], Mao Yang[1], Lintao Zhang[1]
[1] Microsoft
[2] University of California, San Diego
[3] Northeastern University
[4] North China Electric Power University
[5] University of California, Berkeley
discat@foxmail.com,zhao.linf@northeastern.edu,weicheng5993@foxmail.com,sihao@berkeley.edu
{cheqi,xuehui,haidwa,chuanli,maoyang,lintaoz}@microsoft.com

ABSTRACT

To achieve good result quality and short query response time, search engines use specific match plans on Inverted Index to help retrieve a small set of relevant documents from billions of web pages. A match plan is composed of a sequence of match rules, which contain discrete match rule types and continuous stopping quotas. Currently, match plans are manually designed by experts according to their several years' experience, which encounters difficulty in dealing with heterogeneous queries and varying data distribution. In this work, we formulate the match plan generation as a Partially Observable Markov Decision Process (POMDP) with a parameterized action space, and propose a novel reinforcement learning algorithm Parameterized Action Soft Actor-Critic (PASAC) to effectively enhance the exploration in both spaces. In our scene, we also discover a skew prioritizing issue of the original Prioritized Experience Replay (PER) and introduce Stratified Prioritized Experience Replay (SPER) to address it. We are the first group to generalize this task for all queries as a learning problem with zero prior knowledge and successfully apply deep reinforcement learning in the real web search environment. Our approach greatly outperforms the well-designed production match plans by over 70% reduction of index block accesses with the quality of documents almost unchanged, and 9% reduction of query response time even with model inference cost. Our method also beats the baselines on some open-source benchmarks[1].

CCS CONCEPTS

• **Information systems → Search engine architectures and scalability**.

[1]Our code is available at https://github.com/RL-matchplangeneration/Match-Plan-Generation-in-Web-Search

[*] The authors contributed equally to this research.
[†] This work was primarily completed during the author's internship at Microsoft.

KEYWORDS

Deep Reinforcement Learning, Information Retrieval, Parameterized Action Soft Actor-Critic, Search Engine

ACM Reference Format:
Ziyan Luo[1,2,*,†], Linfeng Zhao[1,3,*,†], Wei Cheng[1,4,*,†], Sihao Chen[1,5,†], Qi Chen[1], Hui Xue[1] and Haidong Wang[1], Chuanjie Liu[1], Mao Yang[1], Lintao Zhang[1]. 2021. Match Plan Generation in Web Search with Parameterized Action Reinforcement Learning. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449862

1 INTRODUCTION

Nowadays, most keyword-based search engines use Inverted Index to help retrieve relevant results from billions of documents in order to achieve sub-second response time [27]. Inverted Index maps each keyword to a list of relevant documents (posting list), which provides initial candidates for a query. These candidates are then re-ranked by some ranking models and finally presented to the users [15]. With the explosive growth of documents on the Web, the collection of documents related to a popular keyword becomes so large that it cannot be processed in a limited amount of time. To further improve the search efficiency, search engines begin to separate the documents from different fields into different posting lists, organize these posting lists according to the document quality and apply a *match plan* to the online candidate generation procedure in which posting lists from different keywords and different fields are merged. Figure 1 shows the execution of a match plan.

A *match plan* is composed of a sequence of *match rules* which are executed one after the other. A match rule defines how the search engine matches documents over a period of time. For example, match N candidates that keywords appear in document body, URL or title fields from the beginning of the posting lists ($rule_A$), match M candidates that keywords only appear in document URL or title fields from the current scan position of the posting lists ($rule_B$), etc. Each *match rule* has a discrete match rule type (e.g., $rule_A$) and several continuous stopping quotas (e.g., matched candidate count N). Note that different match rules have different execution costs. In figure 1, $rule_A$ is expensive since the number of candidates that need to be checked in $rule_A$ is larger than other rules. Once the

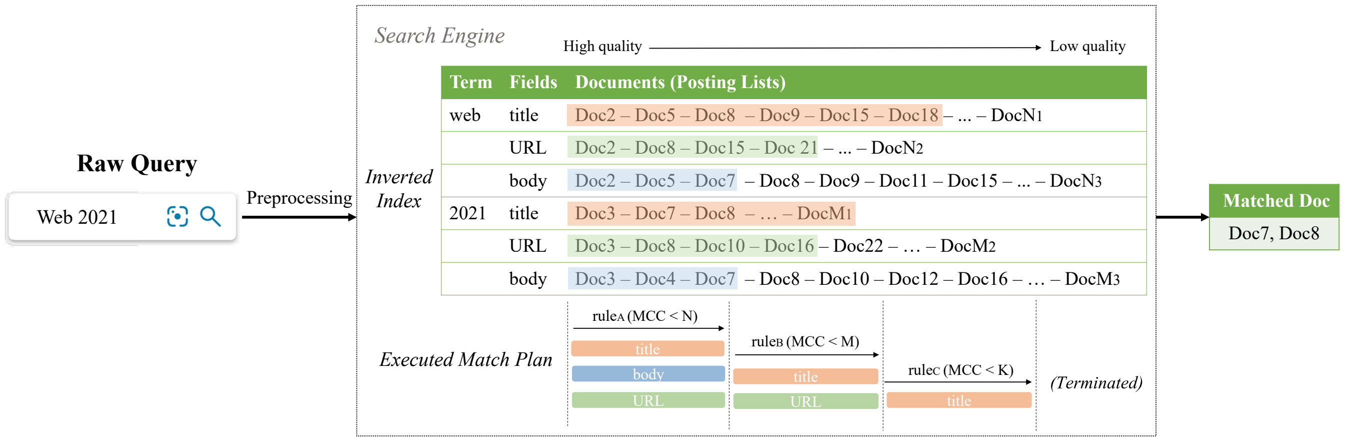

Search Engine
High quality
Low quality
Raw Query
Web 2021
Preprocessing
Inverted Index
Term
Fields
Documents (Posting Lists)
web
title
Doc2 – Doc5 – Doc8 – Doc9 – Doc15 – Doc18 – ... – DocN1
URL
Doc2 – Doc8 – Doc15 – Doc 21 – ... – DocN2
body
Doc2 – Doc5 – Doc7 – Doc8 – Doc9 – Doc11 – Doc15 – ... – DocN3
2021
title
Doc3 – Doc7 – Doc8 – ... – DocM1
URL
Doc3 – Doc8 – Doc10 – Doc16 – Doc22 – ... – DocM2
body
Doc3 – Doc4 – Doc7 – Doc8 – Doc10 – Doc12 – Doc16 – ... – DocM3
Matched Doc
Doc7, Doc8
Executed Match Plan
ruleA (MCC < N)
title
body
URL
ruleB (MCC < M)
title
URL
ruleC (MCC < K)
title
(Terminated)

Figure 1: An example of match plan execution. After preprocessing, multiple posting lists representing different terms and different fields are retrieved from the inverted index in which documents are organized in the descending order of quality. The search engine scans the posting lists by executing a match plan which is composed of a sequence of match rules. A match rule defines how the search engine matches documents over a period of time. It is made up of a discrete match rule type (e.g. $rule_A$) and several continuous stopping quotas (e.g. $MCC < N$, where "MCC" stands for matched candidate count). Once the quality of the remaining documents is too low to be considered, the search engine will choose to terminate the execution of a match plan.

quality of the remaining documents is too low to be considered, the search engines may terminate the execution of a match plan.

Traditionally, the match plan is carefully hand-crafted by engineers according to their expertise in order to make a reasonable trade-off between result quality and response time. However, engineers are still facing several challenges when doing match plan design. Firstly, with the increasing number of match rule types and stopping quota types, it is much more difficult for engineers to design a good match plan that works well for most queries. Secondly, since the data distribution of the Inverted Index varies among different machines and over time, it is also a challenge for engineers to design different match plans for different machines periodically. Finally, hand-crafted static design cannot dynamically revise the match rule based on intermediate system states because of its open-loop nature, which is less adaptive to some corner cases.

Researchers begin to leverage learning algorithms to help with the match plan generation. It is difficult to directly apply traditional supervised learning algorithms to our task because the best match plans (labels) are unknown. The previous work [20] tries to automatically generate the match plan using tabular Q-learning with discretized state space and predefined action parameters (stopping quotas). It learns a policy for a specific query class each time and solves it only in discretized spaces with tabular methods, which limits its generalizability to broader queries in real search scenarios.

In this paper, we extend the match plan generation to the general case, such that the state signals and match plans are fully parameterized and learned from scratch without any predefined knowledge (e.g., stopping quotas). We formulate the match plan generation problem as a Reinforcement Learning (RL) problem. The state consists of dynamic system runtime signals (e.g. current matched documents) and static query features (e.g. query embedding). The action space is called *parameterized* or *discrete-continuous* *hybrid*, where an action has a discrete *action-type* and continuous *action-parameters*. The reward is a function of *result quality* and *query response time*. The formulation is similar to *Parameterized Action* RL (PARL) [14], while our setting requires all actions to *share* the same action-parameters. Moreover, our environment is complex. The runtime signals we can observe are limited, which are only an epitome of the intermediate system states. The reward can only be obtained after the entire match plan is executed, and only very few match plans can match desired documents, which results in sparse reward signals.

We propose a novel deep reinforcement learning algorithm, *Parameterized Action Soft Actor-Critic* (PASAC), to address these challenges, which learns to act on parameterized action space environments and maximize both the expected return and the entropy in the parameterized action space. We propose *Stratified Prioritized Experience Replay* (SPER) to solve the *skewed prioritizing* issue of the original PER by introducing "buffer stratifying" to optimize all queries in different reward ranges. To handle the inherent partial observability, we apply *recurrent* policies [9] on variable-length match plans. We present an agent to integrate these techniques. The agent is trained and evaluated on the production environment with a query dataset collected from Bing[2]. The match plans generated by the agent outperform the well designed production match plans, especially in latency: our approach achieves over 70% reduction of index block accesses on test dataset with the quality of results almost unchanged and 9% reduction of query response time even with model inference cost. We conduct ablation studies to validate the effectiveness of the components. We also test on a few existing PARL benchmarks, where our agent beats our baseline methods and performs the state-of-the-art results in the comparable methods.

[2]https://www.bing.com

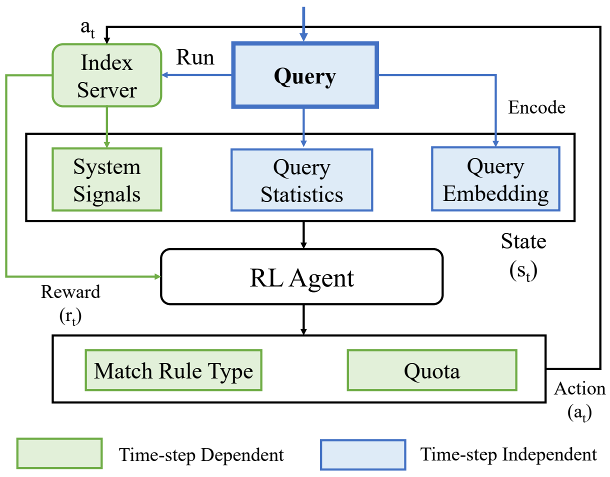


Figure 2: Overview of an *episode* of the generation process. Green boxes indicate quantities updated in sequentially predicting discrete-continuous actions at each (time) step, while blue boxes are fixed in each episode. Rounded rectangles are procedures producing corresponding quantities. (1) Given a randomly sampled *query*, we collect some *query statistics*, such as its length, and extract a *query embedding*. (2) We execute the query using all previously computed actions $\{a_1, .., a_t\}$ on an index server and take the *system signals*, such as latency, as feedback to the RL agent. We concatenate all three quantities and use them (and past observations) as input to the agent. The agent computes a match rule type and quota parameters and run it again on the index server to get the next system signals, and repeat until a *stop* action or server-side terminal signal. The entire *action sequence* $\{a_1, .., a_N\}$ is a complete *match plan*.

2 BACKGROUND

2.1 Processing Procedure in Search Engine

When a user types a query in the website, the following procedure [4] will be executed by the search engine: Firstly, the query will be parsed and preprocessed by natural language processing techniques, like word segmentation, stop word removal, etc. Then, the posting list of a given query, which may contain millions of documents, will be scanned. In this *matching procedure*, high-quality relevant documents need to be recalled from a large number of candidates in a short response time, which makes this procedure time-sensitive. In the *ranking procedure*, the recalled documents will be sorted by relevance scores. This means *ranking procedure* pays more attention to the quality of results instead of latency.

In our work, we focus on the problem that efficiently recalls candidates in *matching procedure*. We perform our experiments on the production environment deployed at Bing. We briefly describe the current baseline system in Bing and the development of match optimization as following.

2.2 Matching Procedure Optimization in Bing

As increasingly numerous amount of documents on the Internet are crawled, the posting list in Inverted Index becomes too long to retrieve in a limited time. The match optimization is built on the assumption that documents with high quality are more likely to appear in the front of the index. It suggests us to stop early and skip low-quality documents at the long tail of the posting list.

Engineers designed some matching policy manually and it could effectively prune the set of candidate documents, which called *match plan*. Typically, a match plan consists of a sequence of match rules, where each match rules can be controlled by several *quotas*. Note that different match rules have different execution costs since some *index block accesses* ("IBA") are needed for checking candidates. Usually, high-cost strategies always correspond to high-quality results. In Figure 1, $rule_A$ is expensive since the number of candidates that need to be checked in $rule_A$ is larger than other rules. Based on that, we proposed a learning method that could automatically balance the trade-off between the quality (i.e. *relevance score* graded by Bing's production environment) and latency (i.e. *index block accesses*).

In the production environment, the query is classified into one of few predefined categories by simple statistics information (i.e. the length of the query), and then a match plan is selected based on hand-crafted rules. Therefore, the existing rough classification still has great potential in optimizing the *matching procedure*. In our work, we also include queries' statistical features and query embedding to identify different queries and then generate corresponding match plans.

3 PROBLEM STATEMENT

In [20], the action-parameters are predefined for the match plan generation, where the action space is a set of discrete match rules (actions) and the state space is also discretized. We fully parameterize a match plan to allow the agent to generate any valid plans.

3.1 Problem Formulation

We model the match plan generation problem as a discrete-time Partially Observable Markov Decision Process (POMDP) [11], as we assume that the runtime signals we can observe are limited, which are far from enough to describe the complete intermediate system state. We use a *parameterized* (discrete-continuous hybrid) action space to represent any possible choices of each step.

A POMDP is given by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma)$, and the underlying Markov Decision Process (MDP) is defined by $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ the *parameterized* action space, $\mathcal{T}$ the transition function $\mathcal{T} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. The set of observations is given by Ω and the the observation function mapping underlying states to probability distributions over observations is given by $\mathcal{O}$. The agent interacts with an environment to maximize the cumulative reward with a discount factor $\gamma \in [0, 1)$.

In previous *Parameterized Action RL* works like [2, 14], the action space is denoted as:

$$\mathcal{A} = \bigcup_{k \in \mathcal{A}_d} \{(k, x_k) | x_k \in \mathcal{X}_k\},$$

where each discrete action $a \in \mathcal{A}_d = [K] = \{k_1, ..., k_K\}$ has a corresponding continuous action-parameter space $\mathcal{X}_a$. We introduce a slightly different formulation according to the nature of match plan generation:

$$\mathcal{A} = \{(k, x) | k \in \mathcal{A}_d, x \in \mathcal{X}\} = \mathcal{A}_d \times \mathcal{X},$$

where the discrete action space (for match rules) $\mathcal{A}_d$ share the action-parameter space $\mathcal{X} \in \mathbb{R}^N$, i.e. the continuous action space (for quotas). Such formulation results in a disentangled action space between discrete and continuous actions which a class of parameterized action (P-DQN [26] and MP-DQN [2]) may not be trivially applied. We have a large action space and such formulation is over-complicated, making the agent impossible to converge. Thus, ours only contains a discrete action space and a shared parameter space.

Parameterized v.s discretized action space. An alternative way of dealing with our problem is to discretize the aforementioned continuous action space. We use the parameterized action space instead of the discretized one, mainly due to the following motivations: 1) Controlling with coarse-grained quotas can lead to imprecise control. Conversely, the action space is too large, making the models hard to converge. It is difficult to balance the two issues; 2) Generating discretized quotas cannot utilize the prior knowledge of numerical ordering. Besides, in Section 5, experiments show that using the parameterized action space performs better than the discretized one.

Environment. The environment communicates with *Bing's index servers* to send action sequences and receive states and rewards. At the beginning of an episode, a query is randomly chosen, and the goal is to find an optimal match plan in terms of the reward. For every step, the agent generates a match rule and sent it to the index. After execution, the index returns the state signals to the agent. Note that, the environment only contains one shred of Bing's index, which is the epitome of the whole index. Because the index is evenly distributed in different machines, we assume that if we achieve optimal performance at one machine, the whole cluster of machines would be also in the optimal settings. We also test our algorithm and models on other shreds of the index, and the results remain nearly unchanged.

3.2 Environment Modeling

The interactions between the agent and the environment are depicted in Figure 2. The key components of the POMDP are as follows:

- **State.** At each time step, the agent receives a state with the *time-step dependent* features and the *time-step independent* features. As shown in Figure 2, the time-step dependent part contains selected run-time system signals (e.g. the index position, matched document count, IBA, matched document quality, etc.) from the Inverted Index system. The time-step independent part includes some statistical features and semantic embedding of queries to allow the agent to identify different queries. Such statistical features are used in the current production system, such as length and popularity of a query. Details about state features are included in Appendix.
- **Action.** The agent selects a parameterized action a_t including a *match rule* type and the allocated *quotas* at each step. There are 29 types of match rules in $\mathcal{A}_d$ and also a parameter space of *quotas* $\mathcal{X}$ for each match rule. All outputted quotas are only effective for the current step. There is also a special action to note: *stop* (by agent). The environment may terminate and return terminal signal, but it only happens in extreme cases, such as query latency is too long which exceeds the budget. To allow a better balance of latency and performance, we allow the agent to choose stop or not as another type of discrete action, which is additional to the other 29 match rules.
- **Reward.** We use two criteria in the reward: *latency* and *quality*. For latency consideration, since *execution time* is noisy because of caching, we use *index block accesses* ("IBA") which is a constant in different external circumstances. The performance is indicated by *relevance scores* ("RS") of top K returned documents weighted by the decreasing weights which are graded by the ranker model built by Bing's engineers. RS is trained to be an approximation of NDCG[20]. Our match plan generation models training lies on one machine. However, there are few (e.g. zero/one) labeled documents for each query on one machine. The labeled query set is too small to train models. Therefore, instead of NDCG, we use RS to indicate the relevance, since RS has a more smooth and continuous value space than NDCG and is stabler and easier for models to learn.
  The reward r_t is calculated by IBA and RS:
  $$r_t = (\lambda_1 \mathrm{RS}_t - \lambda_2 \mathrm{IBA}_t) - (\lambda_1 \mathrm{RS}_{t-1} - \lambda_2 \mathrm{IBA}_{t-1}), \quad (1)$$
  where $\lambda_{1,2}$ the weight for two evaluation metrics, which are used to make a trade-off between relevance and efficiency. Specifically, the initial values RS_0 and IBA_0 are all zeros.

4 METHOD

It is a challenging task to use RL for match plan generation in search engines. Firstly, the action space in our environment contains both discrete and continuous actions, while most reinforcement learning algorithms focus on problems that the action space is either discrete or continuous. Secondly, the environment of the search engine is complex in terms of heterogeneous query representation and states. In our scene, queries are heterogeneous which needs a high dimension representation; many intermediate signals used in our states have a large and continuous value range as well. Finally, sparse reward is also challenging for training to converge [1]. Especially, at the beginning of the training stage, positive rewards are rare because of insufficient training of the policies.

4.1 Parameterized Action Soft Actor-Critic

Soft Actor-Critic (SAC) [6] is a state-of-the-art control RL algorithm that has proven its sample efficiency and learning stability as well as hyper-parameter robustness. However, SAC is only is well studied on continuous action and can be adopted to discrete action. Therefore, PASAC is proposed for the parameterized action space. The pseudocode of PASAC is included in Algorithm 1 and networks architecture in Appendix.

Maximum entropy in parameterized action policy. PASAC optimizes stochastic policies in an off-policy way and optimizes policies that maximize both the expected future return and the maximum entropy objectives:

$$J(\pi) = \sum_t \mathbb{E}_{(s_t, x_t, k_t) \sim \tau_\pi} \left[\gamma^t \left(r\left(s_t, x_t, k_t\right)\right) + \alpha \mathcal{H}\left(\pi\left(\cdot | s_t\right)\right) \right] \quad (2)$$

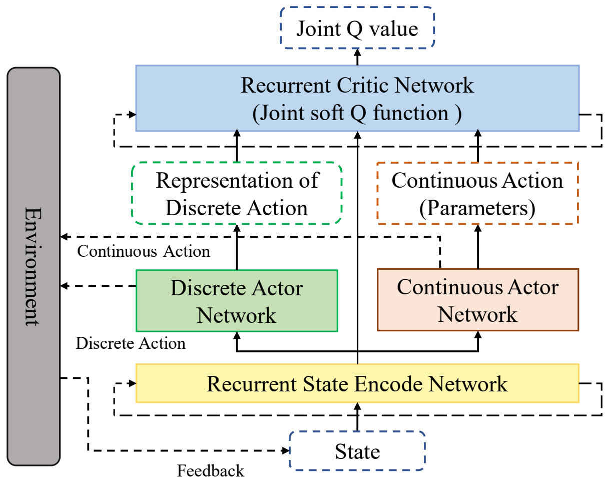


Figure 3: The framework of PASAC and its interaction with the environment of the search engine.

where π is a policy, T is the number of time steps, r is the reward function, $\gamma \in [0, 1]$ is the discount rate, $s_t \in \mathcal{S}$ is the state at time step t, $a_t \in \mathcal{A}$ is the action at time step t, τ is the distribution of trajectories induced by policy π, α is the temperature parameter which determines the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy. Here $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy of the policy π at state s_t.

Architecture of PASAC. Original SAC contains a critic network to evaluate the Q value of continuous action and an actor network to estimate a mean and variance of a Gaussian distribution for the continuous actions. In PASAC, there are discrete and continuous actor branches (network), and they are used to generate discrete and continuous actions at the same time. These two actor networks have parameters that are not exactly the same, and share the first few layers to encode the state information. PASAC contains one critic network for estimating Q values of *both* discrete and continuous actions. The architecture of PASAC is shown in Figure 2.

Policy network with parameterized action space. To generate the discrete actions, the discrete actor network estimates a categorical policy π_ϕ for all discrete actions, and the discrete action to execute is sampled from the categorical distribution $\pi_\phi(k|s)$ [23]. Continuous action is sampled from a Gaussian distribution generated by the mean and variance outputted by the continuous actor network (as SAC typically does). The continuous actor network generates the stochastic policy π_ψ for continuous actions by outputting the mean and variance of a Gaussian distribution for each of the parameters. They are updated separately by maximizing their respective entropy objective. The objective for the discrete policy π_ϕ is given by:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D}\left[\mathbb{E}_{k_t \sim \pi_\phi}\left[\alpha_d \log\left(\pi_\phi\left(k_t|s_t\right)\right) - Q_\theta\left(s_t, k_t, x_t\right)\right]\right] \tag{3}$$

and similarly the objective for the continuous policy π_ψ is:

$$J_\pi(\psi) = \mathbb{E}_{s_t \sim D}\left[\mathbb{E}_{x_t \sim \pi_\psi}\left[\alpha_c \log\left(\pi_\psi\left(x_t|s_t\right)\right) - Q_\theta\left(s_t, k_t, x_t\right)\right]\right] \tag{4}$$

where D denotes experience replay and θ denotes parameters of the joint soft Q network, ϕ and ψ denote the parameters of discrete and continuous actor networks, α_d is discrete temperature parameters of entropy of discrete policy, and α_c is continuous temperature parameters of entropy of continuous policy.

Soft Q network. Each *complete* action composes of both discrete and continuous parts $a_t = (k_t, x_t)$. Therefore, in PASAC, the critic network estimates the joint soft Q function $Q_\theta(s_t, k_t, x_t)$, where θ is the parameters of critic network. This strategy keeps the discrete and continuous actions of the output of two different actors relevant. The discrete actor network outputs categorical probabilities as the representation $[f(k_t^1), f(k_t^2), \ldots, f(k_t^n)]$ for n dimension discrete actions, where $f(\cdot)$ is the $softmax$ function. The joint soft Q function takes the continuous action x_t and the representation vector $[f(k_t^1), f(k_t^2), \ldots, f(k_t^n)]$ as input. This Q function estimates a joint Q value of discrete and continuous action instead of a Q values for only continuous actions, which is different from P-DQN. It can be trained to minimize the soft Bellman residual:

$$J_Q(\theta) = \mathbb{E}_{(s_t, x_t, k_t) \sim D}\left[\frac{1}{2}\left(Q_\theta^t\left(s_t, x_t, k_t\right) - y_t\right)^2\right], \tag{5}$$

where the target y_t is

$$y_t = r_t + \gamma\left(Q_{\bar{\theta}}^{t+1}\left(s_{t+1}, x_{t+1}, k_{t+1}\right) - h_t\right), \tag{6}$$

and the terms from maximum entropy

$$h_t = \alpha_d \log \pi_\phi\left(k_{t+1}|s_{t+1}\right) + \alpha_c \log \pi_\psi\left(x_{t+1}|s_{t+1}\right), \tag{7}$$

where $\bar{\theta}$ denotes parameters of the target joint soft Q network.

Double alpha tuning. [7] use an alpha loss to adjust temperature of entropy automatically. This method enables the policy to explore adaptively during the training and across environments. Inspired by SAC, we propose *two alpha loss* to adjust the temperatures of discrete and continuous policies' entropy *respectively*. This method allows independent exploration of discrete and continuous policies. Furthermore, it provides a more efficient and balanced exploration strategy. To train the temperature parameters α_d and α_c, the gradient descent is applied to the following objectives:

$$J\left(\alpha_d\right) = \mathbb{E}_{k_t \sim \pi_\phi^t}\left[-\alpha_d\left(\log\left(\pi_\phi\left(k_t|s_t\right)\right) + \bar{\mathcal{H}_d}\right)\right] \tag{8}$$

$$J\left(\alpha_c\right) = \mathbb{E}_{x_t \sim \pi_\psi^t}\left[-\alpha_c\left(\log\left(\pi_\psi\left(x_t|s_t\right)\right) + \bar{\mathcal{H}_c}\right)\right] \tag{9}$$

where $\bar{\mathcal{H}_d}$ is the target entropy of the discrete policy, $\bar{\mathcal{H}_c}$ is the target entropy of the continuous policy.

Recurrent state head. To apply PASAC to the match plan generation task, we apply recurrent neural networks to handle the partial observability in our setting [9], since we empirically find that the system signals cannot provide sufficient information of next state. The training of RNN parameters uses backpropagation through time (BPTT) for entire trajectories.

Algorithm 1 Parameterized Action Soft Actor-Critic

```
input: Initial parameters $\theta_1, \theta_2, \phi, \psi$
  ⋄ Initialize target networks' weights    $\bar{\theta_1} \leftarrow \theta_1, \bar{\theta_2} \leftarrow \theta_2$
  ⋄ Initialize an experience replay buffer    $\mathcal{D} \leftarrow \emptyset$
  for each episode do
    for each environment step do
      ⋄ Sample a discrete action    $k_t \sim \pi_\phi(k_t|s_t)$
      ⋄ Sample parameters    $x_t \sim \pi_\psi(x_t|s_t)$
      ⋄ Store the transition    $\mathcal{D} \leftarrow \mathcal{D} \cup (s_t, f(k_t), x_t, r_t, s_{t+1})$
    end for
    for each gradient step do
      ⋄ Sample a mini-batch from replay buffer $\mathcal{D}$
      ⋄ Update the joint soft Q-function parameters
        $\theta_i \leftarrow \theta_i - \lambda_Q \nabla_{\theta_i} J_Q(\theta_i)$ for i $\in [1, 2]$
      ⋄ Update discrete policy weights
        $\phi \leftarrow \phi - \lambda_{\pi_\phi} \nabla_\phi J_\pi(\phi)$
      ⋄ Update continuous policy weights
        $\psi \leftarrow \psi - \lambda_{\pi_\psi} \nabla_\psi J_\pi(\psi)$
      ⋄ Adjust temperature of discrete policy's entropy
        $\alpha_d \leftarrow \alpha_d - \lambda_{\alpha_d} \nabla_{\alpha_d} J(\alpha_d)$
      ⋄ Adjust temperature of continuous policy's entropy
        $\alpha_c \leftarrow \alpha_c - \lambda_{\alpha_c} \nabla_{\alpha_c} J(\alpha_c)$
      ⋄ Update target networks' weights
        $\bar{\theta_i} \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta_i}$ for i $\in [1, 2]$
    end for
  end for
output: Optimized parameters $\theta_1, \theta_2, \phi, \psi$
```

4.2 Stratified Prioritized Experience Replay

In the off-policy RL algorithms, experiences (s_t, a_t, r_t, s_{t+1}) are stored in a replay memory, and the agent samples a mini-batch from the memory uniformly instead of using the current experience for training. Recently, many off-policy RL algorithms use Prioritized Experience Replay [21] (PER) and achieve better results than ordinary experience replay. In PER, the sampling from a replay memory can be prioritized with a probability p_i proportional to TD-errors to increase the sample efficiency.

Skewed prioritizing. In our complex environment, we discover a *skewed prioritizing* property of PER that: the highly prioritized samples often center on a small range of the reward space, as shown in Figure 4. In such a condition, the experiences whose rewards are in certain ranges are more likely to be sampled, which makes the agent behave poorly in some state subspaces. Specifically, in our settings, it causes insufficient training on some corner queries. In order to alleviate the *skewed prioritizing* issue, we propose Stratified Prioritized Experience Replay (SPER) to replace the original PER.

Buffer stratifying. The replay buffer is divided into several bins (strata) of the same capacity. Each bin stores transitions within a certain range of the reward space. When sampling from SPER, the same number of transitions are fetched from each bin by prioritized sampling. Each bin can be regarded as a PER and the sampling procedure in each bin is the same as in PER [21].

Priority with TD-error and policy loss. In SAC, policy loss is the Kullback-Leibler divergence of the policy and the scaled exponential soft Q function [6] implying the potential improvement of the policy. Inspired by this, we adapt prioritized sampling to PASAC by modifying the priority to:

$$p(s_t, a_t) = |\delta(s_t, a_t)| + \lambda\xi(s_t, a_t) + \epsilon_d, \tag{10}$$

where $\delta(s_t, a_t)$ is the TD-error, $\xi(s_t, a_t)$ is the loss applied to the actor, e.g. policy loss, and the hyperparameter λ is the weight of policy loss, ϵ_d is a base priority to ensure transactions are sampled with appropriate probabilities at the beginning of training even if the losses of new samples are high. To optimize prioritized replay for a PASAC agent, SPER adds the policy loss to the priority in PER, which makes transactions with larger improvement potential more likely to be sampled than merely using the TD-error. DDPGfD [24] adopts similar method but with a different motivation.

In our settings, we store trajectories of experiences instead of transactions in SPER to train the recurrent neural networks [12].

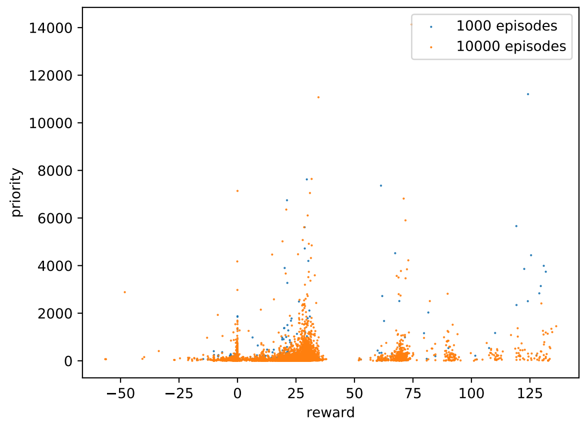


Figure 4: Scatter plot of distributions of query samples' priority in the original PER at episode 1000 and 10000.

5 EXPERIMENTS

We study the purposed agent on the match plan generation problem on a self-made dataset based on real queries in Bing, addressing the following questions:

Q1. Does the proposed algorithm work better than the heuristic hand-crafted method tuned by engineers, or other RL algorithms?

Q2. Is it more appropriate that we formulate the problem into a PARL problem, instead of discretizing the action space?

Q3. How is the improvement of our method in real search scenes?

Q4. How is the effect of applying SPER, and its components?

Q5. Does the proposed agent work well on other PARL benchmarking baselines?

5.1 Experimental Settings

Dataset Preprocessing. For match plan generation, we experiment on a dataset that has about 100,000 queries sampled from Bing's search log. We filter the dataset in the following aspects:

- We skip a few special queries that do not have embedding or need additional operations beyond match plans.

- We skip all the advanced queries (e.g. raw queries contain "site:"). Because such advanced features cannot be expressed in the query embedding.
- We only use queries in the market designed for English-speaking countries to avoid including some skewed data.
- After filtering, the dataset size is about 36,000. In a real search scenario, we never know what the users want to search for. To simulate this scenario, we randomly choose 3,000 queries as the test dataset, and the remaining queries as the training dataset.

Baseline. For the match plans in the currently implemented system in Bing, each query is classified into a predefined query class which has some heuristic hand-crafted rules. We use each query's *production match plan* as the *baseline*. The production works well for most queries, thus it is not trivial to outperform the production on many queries.

Evaluation Metrics for Match Plan Generation. The following metrics are used in experiments:

- *Average Relative Improvement (ARI).* ARI is designed for evaluating different RL agents. It represents the mean improvement on final returns (sum of rewards in one episode) of all test queries. It is the most important metric in our experiments because it links firmly with our optimizing target (final return). A given agent's *ARI* is computed as follows:

$$ARI = \frac{\sum_{i=1}^{|D|}(R^i_{agent} - R^i_{baseline})}{|D|} \tag{11}$$

  where $|D|$ represents test dataset size, R^i_{agent} and $R^i_{baseline}$ represents the given agent's and baseline's final returns on the i^{th} query in the test dataset.
- *Better* and *Equal. Better* means the percentage of queries in the test dataset that the match plan generated by our approach is better than the baseline. *Equal* is defined in a similar way.

Model Training. The system state alternates between the training and test stages at fixed intervals. All of our reported experiment results are from the test stage. At the test stage, it sequentially evaluates the agent on the whole test dataset. At the training stage, the agent randomly chooses a query in the training dataset per episode. We evaluate the agents per 5,000 episodes of training on the test dataset. Each experiment is repeated three times with different random seeds.

Implementation Details. Each comparison experiment and PASAC is trained on one Tesla P100 GPU using PyTorch. In all the experiments, the $\lambda_{1,2}$ in Equation 1 are set to 1. More details about experiment implementation can be found in Appendix.

5.2 Comparison Approaches

We compare PASAC agent with several state-of-the-art methods. The methods are slightly adjusted to adapt to our scene. To answer Q2, some state-of-the-art RL methods with discrete action settings are adopted. All the agents in the comparison are implemented with LSTM [9, 10] and without advanced components like SPER for a fair comparison.

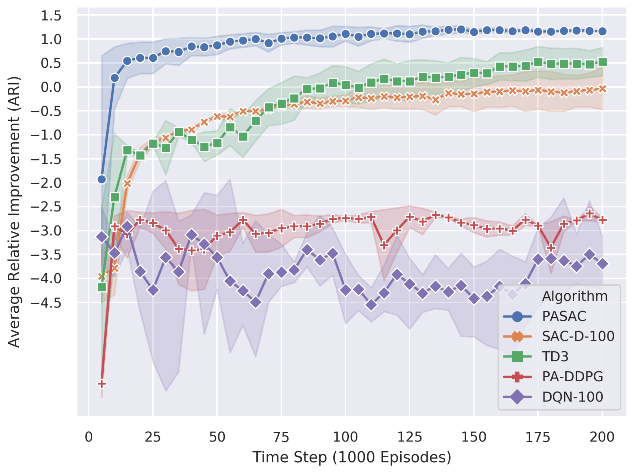


Figure 5: Comparison experiments on Average Relative Improvement (ARI). The X-axis is the number of thousand episodes. The Y-axis denotes ARI.

Note that, to answer Q1, all the following approaches are compared with our *baseline*, i.e. the current heuristic hand-crafted production rules used in Bing, as is mentioned in Section 5.1.

- **DQN**. Deep Q-Learning (DQN) [16] is a well-known Q-learning-based, off-policy method dealing with discrete action space. In our scene, we uniformly discretize our continuous action space (*quota*) into 20 (DQN-20) and 100 discrete values (DQN-100) respectively.
- **PA-DDPG**. Deep Deterministic Policy Gradients (DDPG) [13] is a state-of-the-art method that is off-policy and Q-learning based but learns a deterministic policy. To adapt to our scene, we mainly refer to PA-DDPG [8] to apply DDPG in our problem.
- **TD3**. Twin Delayed DDPG (TD3) [5] improves the stability and efficiency of DDPG. We adopt a similar method in PA-DDPG to adapt to our scene. Details are included in Appendix.
- **SAC-Discrete**. SAC-Discrete (SAC-D) [3] is an alternative version of SAC that can be applied to discrete action settings. We adopt the same method in DQN to discretize the continuous action space to make SAC-D available in our settings (SAC-D-20 and SAC-D-100).

5.3 Performance Comparison

Figure 5 shows the *ARI* on test dataset of all comparison approaches. Table 1 shows *Better*, *Equal* and *ARI* of all mentioned approaches. DQN-20 and SAC-D-20 are not plotted, because the performances are similar to DQN-100 and SAC-D-100 which are not satisfactory.

As seen in Figure 5 and Table 1, addressing this problem in a discrete action setting would lead to worse results. On average, DQN performs the worst in all comparison experiments and demonstrates a large variance. It empirically shows that DQN's epsilon-greedy exploration strategy cannot efficiently explore the large discretized action space. It could also suffer from Q-value overestimation. SAC-D outperforms DQN as expected, because not only the algorithm

Table 1: Better, Equal and ARI of all mentioned approaches. Here we only show each agent's best performance (on ARI) in duplicate experiments.

	DQN-20	DQN-100	D-SAC-20	D-SAC-100	PA-DDPG	TD3	**PASAC**	**PASAC+SPER**
ARI	-1.500	-1.957	0.210	0.460	-2.492	0.8384	1.280	**1.912**
Better	26.70%	27.43%	40.47%	49.30%	39.97%	41.90%	50.53%	**60.10%**
Equal	3.70%	4.50%	10.10%	6.03%	3.17%	4.67%	6.07%	**11.60%**

itself shows superiority as many works concluded, but its exploration controlled by tuned alpha is more effective in our settings as well, which leads to better results here. However, the discretization also gives rise to a loss of accuracy in controlling *quotas*.

As for methods for discrete-continuous hybrid action space settings, PA-DDPG performs the worst. TD3 solves the overestimation issue in DDPG and performs better than PA-DDPG in our experiments as we expect. However, both PA-DDPG and TD3 fail to sufficiently explore the large action and state space in our settings, because learning alpha to control stochasticity is more adaptive than fixed action space noise in deterministic policy learning. Taking all the issues above into consideration, as expected, PASAC performs the best on both stability and efficiency.

5.4 Performance Evaluation

Table 2: Improvement of (Average) IBA, (Average) RS, documents recalling latency (Latency), documents recalling latency considering reference time (Latency+inference) of our approach compared to the baseline. Note: "+" represents improvement.

	IBA	RS	Latency	Latency+inference
Improvement	+75.77%	-1.47%	+28.14%	+8.97%

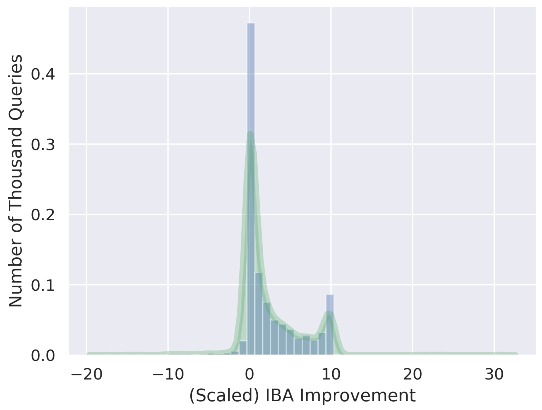
To answer Q3, we first examine the performance of our approach on each part of the reward mentioned in Section 3. In the test stage, we do statistics about scaled *index block accesses* (IBA) improvement and scaled *relevance score* (RS) improvement on each query.

In Figure 6 and Table 2, we discover our method achieves only 1/4 IBA with the quality of documents almost unchanged. Because the production match plans are manually defined by engineers, which cannot flexibly control the *quotas*. Moreover, we find that using our approach, match plan sequences are 3.78% shorter than using the production rules. Thus, much time is wasted due to unsuitable *quotas* and redundant *match rules* for match rules.

Besides, due to the high cache hit rate in Bing, our approach only reduces the *documents recalling latency* of the production by 28.14% on average per query. In consideration of our models' inference time, our approach still improves the latency by 8.97%. We only use the models without any optimization for inference (e.g. pruned to save computation, compressed using distillation or quantized to 16-bits or 8-bits weights), which can further be compressed and optimized. Thus, this result already shows our method's feasibility and superiority applying to the real production of Bing.

5.5 Ablation Study

In response to Q4, we compare results of applying different replay buffer with PASAC. Policy loss is applied in the priority calculation in all these experiments. We further compare results of applying SPER with or without policy loss in the priority calculation.



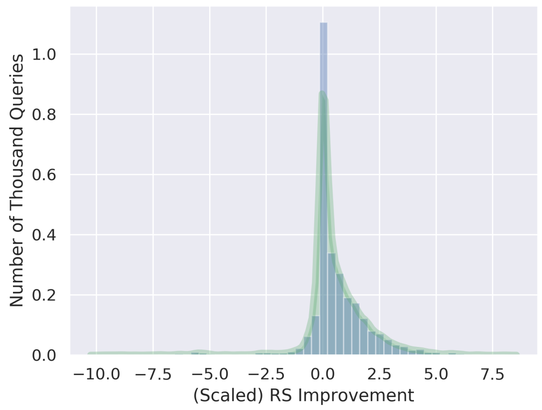
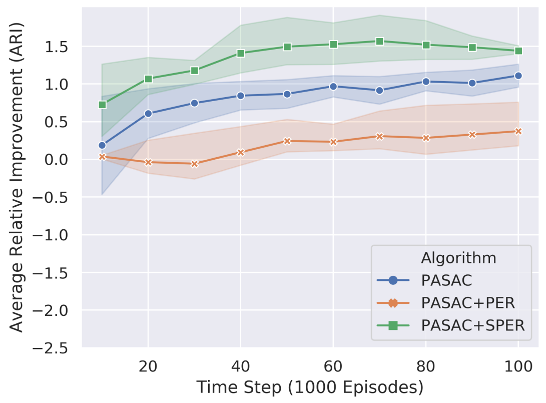


Figure 6: Histograms of distributions of IBA and RS improvement in test stage. Since the actual range of RS improvement is fairly large, we only show [-10, 10] because the data is too sparse outside of this range.

From Figure 7 (upper), we can observe that PER performs even the worse than the original experience replay buffer. A reason could be, PER fails to consider the *skewed prioritizing* issue, and always samples a small range of queries with similar "difficulty" to find relevant documents. The other queries, however, are insufficiently trained. Thus, simply applying PER in our settings is unreasonable. After conducting our improvement in SPER, the agent performs better than PER and the original experience replay buffer.

From Figure 7 (lower), we find that SPER with policy loss in priority performs better. One reason could be, using priority with



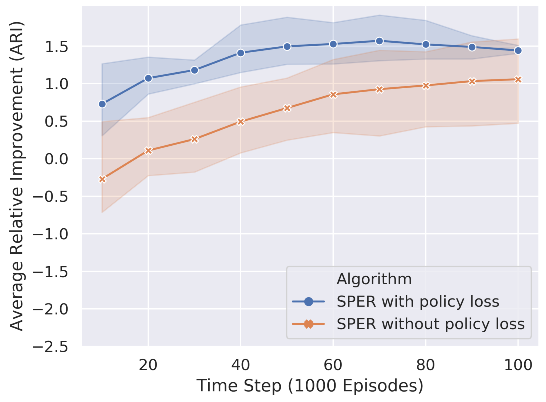


Figure 7: Experiments on ARI using different experience replay buffers and SPER with or without policy loss.

policy loss considers another circumstance that a sample has large potential to improve: policy loss is high and parameters of policy can be improved.

5.6 Benchmark Games

To evaluate our agent in a broader context and answer Q5, we experiment on two open benchmark games to evaluate our algorithm, `Platform-v0` and `Goal-v0` from [2]. These games follow a slightly different formulation [2] that each discrete action has a separate continuous action-parameter space. We assume the environments are fully observable, thus recurrent networks are not used in PASAC agent in benchmarks.

For benchmarks, we use Microsoft NNI[3] to choose the hyperparameters. The search space and best trial curves are included in the Appendix. We release the code for reproducibility (see abstract).

We report the best final metric. PASAC significantly outperforms PA-DDPG on both benchmarks. For Platform-v0, the return has a range of $[0, 1]$. Goal-v0 has a maximum return of 50, and PASAC can almost constantly achieve that after half of training. Our observation meets the reported poor performance of PA-DDPG in [2]. We observe that PASAC+SPER has almost the same performance

[3]https://github.com/microsoft/nni

Table 3: Average evaluation results (the average of all training rewards and final evaluation reward (repeated 100 times)) on benchmarks Platform-v0 and Goal-v0 with PA-DDPG [8].

Average Eval Return	PASAC	PASAC+SPER	PA-DDPG[8]
Platform-v0	0.9723	**0.9727**	0.3113
Goal-v0	43.11	**43.85**	-6.208

as PASAC. This verifies that the stratified sampling may better fit the environment with *skewed prioritizing* issue if PER is applied.

6 RELATED WORK

Our closely related work includes [20]. It works on discretized state and action space with tabular RL method. It learns a policy for a specific query class each time and solves it only in discretized space which limits its generalizability in heterogeneous queries.

Our algorithm is based on Soft Actor-Critic [6], an off-policy Q-learning based policy gradient method. SAC optimizes an energy-based stochastic policy. A previous work Deep Deterministic Policy Gradients (DDPG) [13] is also off-policy and Q-learning based, but learns a deterministic policy. Twin Delayed DDPG (TD3) [5] improves the stability and efficiency of DDPG.

Parameterized Action Reinforcement Learning (PARL) refers to the RL setting that the action space is parameterized (discrete-continuous hybrid). Current PARL methods mainly originated from DQN and DDPG, including two classes of methods: PA-DDPG [8] based on DDPG and P-DQN [26] based on DQN. They use different strategies to combine discrete and continuous actions. P-DQN [26] learns multiple continuous action policy network for each discrete action. MP-DQN [2] extends P-DQN to tackle the problem that Q-value is a function of the joint action-parameter vector $Q(s', k', \mathbf{x}^Q(s'))$ in normal PAMDP, which may results in false gradients. However, such a problem does not exist in our slightly modified setting, since the action-parameter space $\mathcal{X}$ in the match plan generation is inherently defined to be shared for each $k \in \mathcal{A}_d$. [14] purposes a method to iteratively optimizing discrete and continuous actions by alternating between them.

We focus on off-policy algorithms and utilize past experience by storing them in a replay memory [17, 22]. *Prioritized Experience Replay* [21] improves the efficiency with reusing existing experience by prioritizing transitions with high TD-error. There are different methods on applying RL on POMDPs [11], while we focus on using recurrent networks following [9] (RDPG) with backpropagation through time (BPTT) [25].

7 CONCLUSIONS

To automatically generate good match plans for different queries, we formulate the match plan generation to the general PARL framework. We propose a novel algorithm, Parameterized Action Soft Actor-Critic (PASAC), for such RL formulation and corresponding applications to maximize both expected return and exploration entropy in the parameterized action space. To address the *skewed prioritizing* issue of PER, Stratified Prioritized Experience Replay (SPER) is applied. Experiment results show that our learned match plan significantly outperforms the production baseline in terms of resource-saving. We also test our algorithm on some existing open benchmarks. The results demonstrate our agent performs the state-of-the-art returns in the comparable baselines. Our future works include further optimize the model reference time, and inventing more delicate strategies in exploring the parameterized action space.

8 ACKNOWLEDGEMENTS

We would like to thank Lifei Zhu for his help on an earlier draft.

REFERENCES

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. 2017. Hindsight experience replay. In *Advances in neural information processing systems*. 5048–5058.

[2] Craig J Bester, Steven D James, and George D Konidaris. 2019. Multi-Pass Q-Networks for Deep Reinforcement Learning with Parameterised Action Spaces. *arXiv preprint arXiv:1905.04388* (2019).

[3] Petros Christodoulou. 2019. Soft Actor-Critic for Discrete Action Settings. arXiv:cs.LG/1910.07207

[4] J Shane Culpepper, Charles LA Clarke, and Jimmy Lin. 2016. Dynamic cutoff prediction in multi-stage retrieval systems. In *Proceedings of the 21st Australasian Document Computing Symposium*. 17–24.

[5] Scott Fujimoto, Herke van Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477* (2018).

[6] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *International Conference on Machine Learning*. 1861–1870.

[7] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. 2018. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905* (2018).

[8] Matthew J. Hausknecht and Peter Stone. 2016. Deep Reinforcement Learning in Parameterized Action Space. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[9] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. 2015. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455* (2015).

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[11] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence* 101, 1-2 (1998), 99–134.

[12] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. 2018. Recurrent experience replay in distributed reinforcement learning. (2018).

[13] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[14] Warwick Masson, Pravesh Ranchod, and George Konidaris. 2016. Reinforcement learning with parameterized actions. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[15] Irina Matveeva, Chris Burges, Timo Burkard, Andy Laucius, and Leon Wong. 2006. High accuracy retrieval with multiple nested ranker. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 437–444.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[18] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. 2018. Overcoming Exploration in Reinforcement Learning with Demonstrations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 6292–6299. https://doi.org/10.1109/ICRA.2018.8463162 ISSN: 2577-087X.

[19] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. 2017. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905* (2017).

[20] Corby Rosset, Damien Jose, Gargi Ghosh, Bhaskar Mitra, and Saurabh Tiwary. 2018. Optimizing query evaluations using reinforcement learning for web search. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. ACM, 1193–1196.

[21] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[22] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms.

[23] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[24] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. 2018. Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards. *arXiv:1707.08817 [cs]* (Oct. 2018). arXiv: 1707.08817.

[25] Paul J Werbos et al. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[26] Jiechao Xiong, Qing Wang, Zhuoran Yang, Peng Sun, Lei Han, Yang Zheng, Haobo Fu, Tong Zhang, Ji Liu, and Han Liu. 2018. Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space. *arXiv preprint arXiv:1810.06394* (2018).

[27] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM computing surveys (CSUR)* 38, 2 (2006), 6.

APPENDIX

A DETERMINISTIC POLICY VERSION WITH TD3

In addition to the stochastic policy learning with SAC, we further purpose a parameterized action agent with a deterministic actor based on Twin Delayed DDPG (TD3, [18]), to study the effect of deterministicity to discrete and continuous policy branches.

Since TD3 learns deterministic policy, the continuous action branch outputs a deterministic value, and the discrete branch still follows categorical output. We adopt similar techniques such as the experience replay and recurrent state head. It turns out that the difference is mainly the exploration on both discrete and continuous action space.

We thus apply *parameter space noise* [19] on the deterministic policy network, where the threshold of empirical action noise is computed by the sum for discrete and continuous actions. It increases the exploration performance empirically by controlling the exploration on both spaces.

The update for the policy network is

$$\nabla L(\mu) = \nabla \mathbb{E}_{s\sim D}\left[Q_\theta(s,\mu(s))\right] = \nabla \frac{1}{|D|}\sum_{s\sim D} Q_\theta(s,k,x) \qquad (12)$$

The update for the Q-value network is given by

$$\nabla L(\theta_Q) = \nabla \mathbb{E}\left[\frac{1}{2}\left(y - Q_\theta(s,k,x)\right)^2\right], \qquad (13)$$

where $y = r + \gamma Q'_{\tilde{\theta}}(s', x', k')$.

B IMPLEMENTATION DETAILS

B.1 State Feature List

Important features for match plan generation are:

Time-dependent features (intermediate system signals when executing match plans): matched document count, index block accesses (IBA), position on Inverted Index, accumulated page count (APC), ...

Time-independent features (query's depictions): query embedding, query length, query popularity, minimum document frequency (MinDF), ...

B.2 Networks Architecture

Both policy and value networks use two fully connected layers with 512 hidden units. For the recurrent case, both networks use the output from an LSTM layer [10]. The training of recurrent state uses backpropagation through time (BPTT) [9] for entire trajectories. We also apply the Clipped Double Q-learning algorithm in TD3 [5] to reduce overestimation.

Besides, in our implementation for match plan generation, the policy network structure is slightly different from the original PASAC. The outputted categorical distribution of match rules is used as an input of estimating quotas to help the agent generate corresponding quotas for match rules.

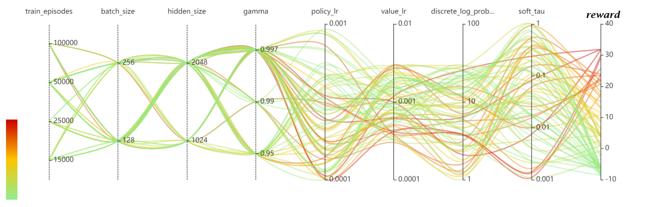


Figure 8: Relationship between tuning parameters and reward in Goal-v0 experiment. The redder the curve, the higher the reward.

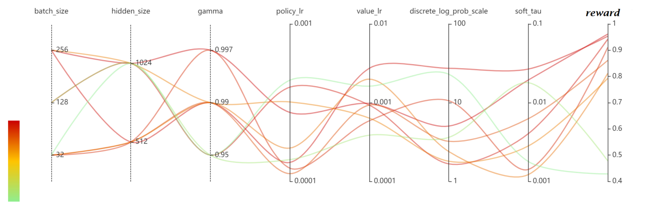


Figure 9: Relationship between tuning parameters and reward in Platform-v0 experiment. The redder the curve, the higher the reward.

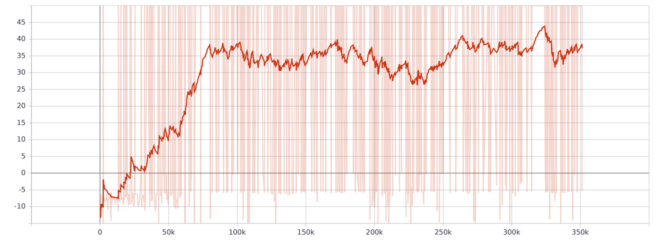


Figure 10: The learning curve of PASAC+SPER with best parameters in Goal-v0.

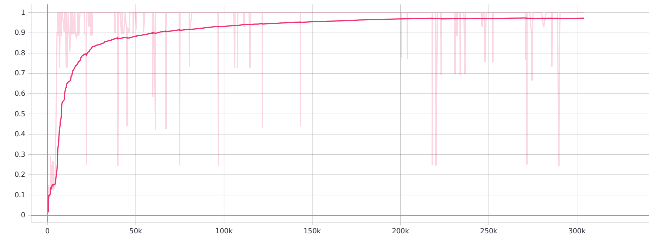


Figure 11: The learning curve of PASAC+SPER with best parameters in Platform-v0.

C HYPERPARAMETER TUNING FOR BENCHMARKS

We use Microsoft NNI[4] and OpenPAI[5] to search hyperparameters. The final metric to report to NNI is set to the sum of (1) average training rewards (per episode) and (2) final evaluation reward (repeated 100 times). The intermediate metric is set to the evaluation

[4]https://github.com/microsoft/nni
[5]https://github.com/microsoft/pai

reward per 10 episodes. We also use the early stop assessor. We summarize the search space for each hyper-parameter below.

Table 4: Search space of hyperparameters for benchmarks.

Name	Type	Range
batch_size	choice	{32,128,256}
hidden_size	choice	{512,1024}
gamma	choice	{0.95,0.99,0.997}
policy_lr	loguniform	[1e-4,1e-3]
value_lr	loguniform	[3e-4,3e-3]
soft_tau	loguniform	[1e-3,3e-2]

For *loguniform*, the range shows the minimal and maximum values. For the type *choice*, it can only take the value in the set.

We run the PASAC+SPER, PASAC, and the PA-DDPG experiments for tuning parameters for about 24 hours. Figure 8 and Figure 9 show the relationship between parameters and rewards in PASAC+SPER experiment. Figure 10 and Figure 11 show the learning curve of PASAC+SPER with best parameters in Goal-v0 and Platform-v0 environment. These results are consistent with Table 3.