PARAMETERIZED ACTION REINFORCEMENT LEARNING FOR INVERTED INDEX MATCH PLAN GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Match plan generation in the inverted index at Microsoft Bing is used to be based on hand-crafted rules. We formulate the generation process as a Parameterized Action MDP with sharing parameters and purpose a reinforcement learning algorithm on such formulation. We combine deterministic policy learning on discrete and continuous action spaces and several recent advances in deep reinforcement learning. For exploring in the parameterized action space, the agent outputs *softmax* values for discrete actions and applies *Parameter Space Noise* on policy network to unify the exploration direction in both spaces. We apply *prioritized recurrent* replay on match plan sequences and *pad* short match plans. We also use *invertible value function rescaling* and *n-step return* to stabilize the training. The agent is evaluated on our environment and some benchmarks. It outperforms the well-designed production match plan and beats the baselines on the benchmarks.

1 INTRODUCTION

Using machine learning to optimize and accelerate software and hardware systems is an emerging field in the past few years (Mirhoseini et al. (2017); Rosset et al. (2018)). The process of decision-making in those systems is usually hand-crafted by human engineers, which is less explored to be automatically done by learning algorithms. A promising direction is to formulate such sequential decision-making problems as Reinforcement Learning (RL) problems, such as search plan generation problem in inverted indexes (Rosset et al. (2018)).

The inverted index is a specialized data structure that is commonly used in search engines, including Microsoft Bing[1]. An inverted index provides an inverted mapping from term to documents. Each term has a posting list which contains all the *(document, location)* pairs that the term appears. By combining the posting lists of the terms in a user query, the initial document candidates are generated (Witten et al. (1999); Zobel & Moffat (2006)).

In Bing, documents are scanned with specified match plans, either predefined or generated in real-time. A match plan has a sequence of *rewrites* (match rules, e.g., it treats the query as a phrase which should appear in the document exactly as it is), where each rewrite can be controlled by several *quotas* (stopping criteria). There are several types of rewrites, and all these types share the same continuous quotas. A pair of a rewrite type and the quotas forms a *single* action which determines if a document will be a ranking candidate.

We formulate the generation problem as a RL problem. The state consists of system runtime signals and semantic embeddings of queries. The action space is called *parameterized* or *discrete-continuous hybrid*, where an action has a discrete *action* and continuous *action-parameters*. The reward is weighted by *result quality* and *query latency*. It is similar to *Parameterized Action* RL (Masson et al. (2016)) (PARL), while our setting requires all actions to *share* same parameters.

Previously, the match plans are predefined manually for each query. It hardly utilizes the rich information in the state to dynamically adjust the rewrites and quotas for specific query online. Rosset et al. (2018) tries to automatically generate match plans using tabular Q-learning with discretized state space and predefined action-parameters. They learn the generation policy for a specific query class each time and solve it only in discretized spaces with tabular methods. We extend the genera-

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

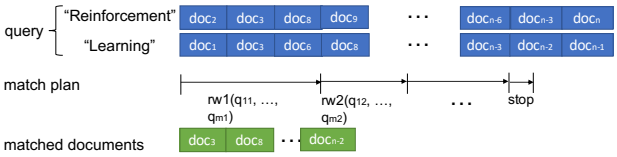

Figure 1: Match plan example: for the query "Reinforcement Learning", the search engine firstly gets a document posting list for each term from the inverted index, and then scans the document candidates following a serial of rewrites (match rules) according to the match plan.

tion process to the general case, such that the match plans are fully parameterized and learned from scratch without any predefined knowledge (e.g., limited match rules).

In this paper, we purpose a parameterized action RL algorithm that learns to act on parameterized action space environments, such as the match plan generation process. We introduce normalized softmax values of discrete actions to form a categorical distributions to enable gradients backpropagation. We also combine several recent advances in RL to accelerate and stabilize the training. We investigate *parameter space noise* (Plappert et al. (2017)) on parameters of the policy for unifying directions in exploring the structured action space. We explore *recurrent* deterministic policies (Heess et al. (2015)) with *prioritized replay buffer* (Schaul et al. (2015)) on sequence, due to the inherent nature of partial observability and sparse feedback in our setting. We also use *n-step return* and *invertible value function rescaling* (Kapturowski et al. (2018)) for further stability.

We present an agent to integrate these techniques and evaluate on offline training on a query dataset collected from Bing search. When training on the dataset with uniformly sampling, the agent outperforms the currently deployed production match plan, which is generated based on well-designed hand-crafted rules. We study the performance of training on the dataset or only with several selected complicated queries individually, as well as ablation studies for the various components. We test on a few existing PARL benchmarks, where our agent beats our baseline methods and performs the state-of-the-art results.

2 Background

2.1 Reinforcement Learning

We address the problem using reinforcement learning (RL) framework with a parameterized action space. An agent interacts with an environment to maximize the accumulated reward with discount factor 2 $[0;1)$. We model the environment with a discrete-time Partially Observable Markov Decision Process (POMDP). A POMDP is given by a tuple $(S;A;T;R;\ ;O)$, and the underlying Markov Decision Process (MDP) is defined by $(S;A;T;R)$, where S is the state space, A the *parameterized* action space, T the transition function $T : S \quad S \quad A \ ! \ R_+$, and $R : S \quad A \ ! \ R$ is the reward function. The set of observations is given by and the the observation function mapping underlying states to probability distributions over observations is given by O.

Specifically, the environment in the match plan generation has a parameterized action space. In Masson et al. (2016); Bester et al. (2019), such formulation in fully observable settings is referred as *Parameterized Action MDP* (PAMDP), where the action space is denoted as

$$A = \bigcup_{k2A_d} f(k;x_k)jx_k \ 2 \ X_k g;$$

where each discrete action $a \ 2 \ A_d = [K]$ has a corresponding continuous action-parameter space X_a. However, our setting requires a slightly different formulation:

$$A = f(k;x)jk \ 2 \ A_d; x \ 2 \ X g = A_d \quad X;$$

where the discrete action space (rewrite) A_d share the action-parameter space X. Such formulation results in a disentangled action space between discrete and continuous actions which a class of parameterized action (P-DQN (Xiong et al. (2018)) and MP-DQN (Bester et al. (2019))) may not be trivially applicable to. Considering the notational simplicity in partially observable environment (POMDP), we do not explicitly denote it when there is no ambiguity in action settings.

2.2 FORMULATION OF MATCH PLAN GENERATION

In Rosset et al. (2018), the action-parameters are predefined for the match plan generation, where the action space is then a set of discrete rewrites (actions). We fully parameterize a match plan to allow the agent to generate any valid plans.

State. At each time step, the agent receives a state with two parts. First part has selected run-time system signals from the inverted index system. Since the query is uniformly sampled in different episode, we include some statistical features and semantic embeddings of queries to allow the agent to identify different queries and then generate corresponding match plans. Such statistical features are used in the current production system, such as the length of a query. All signals and embeddings are normalized to a reasonable range around $[-1, 1]$ based on empirical estimations.

Action. The agent selects a parameterized action including a rewrite id and its allocated quotas at each step. There are 29 types of rewrites A_d in and also a 5-dimensional quota parameter space X for each rewrite in int64 type. However, the range of each quota is empirically set to valid values and then normalized to $[-1, 1]$. All outputted quotas are effective for the current step.

There is also a special action to notstop (by agent). Normally, the environment will return a terminal signal, but it happens in extreme cases, such as low system resources. 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 the same level as other 29 rewrites.

Reward. We use two criterions in the design: latency and performance. For latency consideration, we use a signal Seek Count that is constant for a same query in different runs with same match plan, instead of executed time. We weight the Ranking Scores of top five returned documents as an indicator of performance with weights $[0.4, 0.2, 0.2, 0.1, 0.1]$. When there are less than five documents returned, the scores of missing documents are treated as a minimum possible ranking score. The final scalar reward is weighted by these two objectives to balance them. We do not use the division in Rosset et al. (2018), since the loss surfaces may become more non-convex for optimizing.

We also have punishments (negative rewards) for some special cases. One is to assign a punishment when the agent selects to illegally stop at the first step or some unsupported rewrites (for some special queries). However, unsupported rewrites do not necessarily cause a terminal state, since the production system has same hand-crafted match plans for a class of queries, in which the unsupported rewrites are simply omitted.

3 METHOD

For generating match plans, we introduce a method that works on the aforementioned formulation with slightly different parameterized action space than standard PAMDPs in the literature. We start from the intuition with deterministic policy learning (Silver et al. (2014)) and how it relates to parameterized actions. PAMDP often refers to the formulation that each individual discrete action $k \in A_d = [K]$ has a corresponding continuous action-parameter space. Thus, there are K continuous action-parameter spaces corresponding to discrete action (Xiong et al. (2018)). In our shared-parameter PAMDP, the Bellman equation incorporated both discrete action k and continuous action-parameters x is given by:

$$Q(s,k,x) = \mathop{E}_{r,s'}\left[r + \max_{k'} \sup_{x' \in X} Q(s',k',x') \,\middle|\, s,k,x \right]. \tag{1}$$

P-DQN (Xiong et al. (2018)) tackles PAMDP by incorporating multiple action-parameter policies $x_k(s;\theta_x) : S \to X_k$ for each action k and update them with $\max_{k'} \sup_{x_{k'} \in X_{k'}} Q(s',k',x_{k'})$ simultaneously. However, since all parameters are shared for actions k in our case, we do not require multiple policies. The policy loss in P-DQN (Xiong et al. (2018)) given by

$$L_x(\theta_x) = \mathop{E}_{s \sim D}\left[\sum_{k=1}^{K} Q(s,k,x_k(s;\theta_x);\theta_Q) \right] \tag{2}$$

Figure 2: The diagram of the process.

naturally degenerates to a single policy in next case, where the gradient $\nabla_x Q_x(\cdot|\cdot)$ exactly is the deterministic policy gradient (DPG, Silver et al. (2014)). MP-DQN (Bester et al. (2019)) states a problem regarding to the erroneous gradients of Q-network Q_θ in P-DQN caused by multiple action-parameter policies backpropagating gradients at the same time. However, it does not exist in such single network situation. This is a desired architecture since separating multiple action-parameter policies loses the knowledge that they share same potential meanings.

Following this intuition, we directly parameterize the policy with two heads $\pi_\theta : \mathcal{S} \to \mathbb{R}^K \times \mathcal{X}$ with $\text{as} : k_t^{\text{soft}}, x_t = a_t^{\text{soft}} = \pi_\theta(s_t, \cdot)$, where k_t^{soft} and a_t^{soft} refers to action-parameters or actions with softmax values. One head outputs the softmax values for all discrete actions. For another head, it outputs normalized action-parameter values with hyperbolic tangent $\tanh(\cdot)$ activation after the continuous head. The environment will denormalize it to a valid range. This corresponds to the Squashing Gradients method for bounded-continuous action spaces in Hausknecht & Stone (2015).

Exploration with Parameter Space Noise and action sampling. In DQN (Mnih et al. (2015)), the behavioral policy is given by ϵ-greedy strategy for exploration purpose, where an agent randomly selects a discrete action with probability $\epsilon < 1$. In DDPG (Lillicrap et al. (2015)), although the policy learned is deterministic, to explore the action space, the behavioral policy needs to be different because of the off-policy nature. It uses action space noise such as uncorrelated Gaussian noise or correlated Ornstein-Uhlenbeck process (Lillicrap et al. (2015)).

Directly combining such two exploration strategies is straightforward. However, it may induce a problem that the exploration in two action spaces $\mathcal{A} = \mathcal{A}_d \times \mathcal{X}$ in different paces. For example, the match plan may need another rewrite with larger quotas in a step. However, if the exploration strategy is to use ϵ-greedy for a rewrite k and Gaussian noise for its quotas x, the agent may require more samples to discover the potential rewards (positive documents). We instead adapt parameter space noise (Plappert et al. (2017)) on parameterized action space to tackle such issue.

We denote the discrete and continuous action heads as $(k_k^{\text{soft}}(s), \pi_x(s)) = \pi(s, \cdot)$. To compute the distance $d(\cdot, \cdot) = D_{KL}(\cdot \| \cdot)$ of non-perturbed and perturbed policies $(\pi(s, \cdot); \tilde{\pi}(s, \cdot))$, we use weighted sum of the distance of discrete and continuous actions. For the continuous actions $\tilde{\pi}_x(s)$, the distance is given by $\frac{1}{n} \sum_i^n (\pi_x(s)_i - \tilde{\pi}_x(s)_i)^2$ to estimate KL-divergence empirically. For the discrete actions, we use outputted softmax probabilities to compute $D_{KL}(k_k^{\text{soft}}(s) \| \tilde{k}_k^{\text{soft}}(s))$. The state and action pairs are sampled from a replay memory. The variance of the noise is updated after a policy update based on the distance and threshold (Plappert et al. (2017)). The policy and target networks with layer normalization (Ba et al. (2016)) are perturbed per episode.

Prioritized replay and recurrent policies. We use recurrent architecture to obtain the underlying system state of the POMDP for both value and policy networks (Heess et al. (2015)). To avoid recurrent state staleness (Kapturowski et al. (2018)), we store and replay a sequence $(s, k^{\text{soft}}, x, r)$ with softmax values into a replay memory. Since a match plan is usually short, we set a maximum length and pad shorter episodes with zero or randomly sampled states and actions.

The sampling from a replay memory can be prioritized with a probability proportional to TD-errors (Schaul et al. (2015)) to increase the sample efficiency in such structured action space. To prioritize the transitions with well-matched documents, we slightly modify the sampling strategy. With a half probability, the agent uses regular prioritization, otherwise partitions the memory to multiple bins and retrieve a transition (sequence) with max reward from each bin. Our strategy provides a more efficient and balanced exploration strategy of the evaluation.

Policy update. In the update, we used clipped double Q-learning and delayed policy update (Fujimoto et al. (2018)) to stabilizing the training. After the agent samples a batch of (sequences), it perturbs the target policy network and computes the perturbed action target for policy smoothing. The parameter space noise variance is then updated with the sampled states and actions.

The policy is still deterministic since it is learned in off-policy and only the exploration involves stochasticity. The update is given by deterministic policy gradients theorem (Silver et al. (2014)):

$$r_{\theta} L(\theta) = r_{\theta} E_{s \sim D}[Q(s, \pi(s, \theta), \theta_Q)] = r_{\theta} \frac{1}{|D|} \sum_{s \in D} Q(s, k^{soft}, x, \theta_Q), \quad (3)$$

where D is a replay memory. We found this policy architecture is similar to PA-DDPG (Hausknecht & Stone (2015)), while it does not including softmax activation for discrete actions and other advanced techniques. We also build upon other useful techniques such as invertible value function rescaling $h(x) = \text{sign}(x)(\sqrt{|x|+1} - 1) + \epsilon x$ (Kapturowski et al. (2018)):

$$y = h\left(\sum_{k=0}^{n-1} r_{t+k} \gamma^k + \gamma^n h^{-1}\left(Q(s_{t+n}, a, \theta_Q)\right)\right), \quad (4)$$

where $a = (\max k_t^{soft}, x_t) = (k_t, x_t)$ is the greedy action, y the target for updating Q-network, and θ_Q denotes the target Q-network. The policy network outputs differentiable softmax values and is updated by the gradients backpropagated from the Q-network:

$$r_{\theta} L_Q(\theta_Q) = r_{\theta} E\left[\frac{1}{2}(y - Q(s, k^{soft}, x, \theta_Q))^2\right], \quad (5)$$

where the expectation is took over samples from a memory.

The pseudocode for the algorithm with more details can be found in the Appendix.

4 EXPERIMENTS

In this section, we apply the purposed agent on the inverted index match plan generation problem. We create a dataset which contains a set of queries and corresponding query embeddings. We perform various ablation study on how each component interacts and the benefits of them. We also test on other Paramterized Action RL benchmarking baselines.

4.1 EXPERIMENT SETTINGS

For match plan generation, we experiment on a dataset that has about 100,000 queries sampled from Bing search log. In the current production system, each query is classified to a predefined query classes online based on a set of rules which are related to statistical features of the query. Each query class has some hand-crafted rules which outputs a match plan for the classified query. We use each query's production match plan as baseline. The match plan for each query is generated by the hand-crafted rules and does not change over each running. We only skip a few special queries that do not have embeddings or need additional operations beyond match plans.

The generated match plan is evaluated by delta values of both Ranking Score and Seek Count and overall reward. We provide learning curves of delta values of evaluation rewards. Note that the reward are not symmetrical around 0, since the ranking scores has a minimum value and is sparse since a few queries are hard to find good documents and will be assigned a very low score. It may significantly pull down the average rewards shown in the curves, thus we present histograms for fair comparison. For the benchmarks, we report the results on evaluation reward curves during tuning.

4.2 MATCH PLAN GENERATION PERFORMANCE EVALUATION

We first visualize the query embeddings in Figure 3 using UMAP (McInnes et al. (2018)) to demonstrate our learned models. We evaluate 6,000 queries and draw the colormap based on the difference between evaluation reward and production baseline reward. Although most queries perform similar to the well-designed production rules, there are some clusters of queries and some patterns exist. We

(a) MLP vs. RNN.

(b) Different Prioritization.

Figure 3: Visualization of query embeddings.

Figure 4: Different techniques.

(a) Delta Reward. (b) Delta Ranking Score. (c) Delta Seek Count.

Figure 5: Delta Reward Distribution in Evaluation Phrase.

notice that there is a main cluster at the center (about $(7, 1)$) that gathers most queries on which our agent does not perform well. That cluster contains some random inputs from the users, which may have some typos. For the queries that the agent outperforms the baseline, we find they are quite scattered. We guess the reason is that the baseline with hand-crafted rules do not consider the embeddings, thus it is less affected by embeddings. This may suggest us to look for more informative embeddings of queries to better distinguish some clustered queries.

In the plots shown in Figure 6, we visualize the distribution of delta rewards, ranking scores and seek count values comparing to the well-designed production rules. For most of the queries the agent learns promising match plan without any prior knowledge just based on the reward signals. There are also some hard queries to find a pattern that have poor reward (about -100) and usually lies in the main cluster and around a few more small clusters. Other than some meaningless queries around the center, they also include some non-English queries, such as French and Chinese, which are possibly clustered around their centroid and are challenging to learn the policy for all of them.

Reward design. We consider different weights to trade-off between the query latency from the input and the quality of the returned documents. The quality is evaluated by Ranking Scores of top five documents. We found that it is sparse since the good documents are hard to match. When we weight more on ranking score, the agent tends to stop the search early.

We also investigate a few punishments and try to use less of them to avoid manual design. We found the agent learns some common patterns guided by the reward and punishments. It tries to avoid "stop" at the first step because we set a huge punishment on such invalid stop. This punishment is necessary since it prevents the agent stops at first step for better value than trying more steps but get no documents matched. In other words, the agent is encouraged to explore different rewrites instead of using empty match plans. Without such punishment, the performance significantly drops

	Overall	Ranking Score (Quality)	Seek Count (Ef ciency)
OU Noise	+1.00	+1.20	-0.20
Param Noise	+1.69	+0.82	-0.87

Table 1: Improved Scores of Different Noise Compared to RNN Gaussian Noise. Ranking Score and Seek Count are scaled to match $Overall = RankingScore - SeekCount$ for easy comparison.

	Overall	Ranking Score (Quality)	Seek Count (Ef ciency)
MLP + Prioritized	+14.54	+8.56	-5.98
RNN + Prioritized	+32.63	+14.44	-18.19
RNN + Prioritized Sample	+32.67	+15.29	-17.38

Table 2: Improved Scores of Different Model Compared to MLP

since it is hard to explore good documents at rst. We also punish the agent to avoid unsupported rewrites, since the production system uses designed rules for each class of the queries and simply omits a rewrite without any useful feedback. We test punishing repeated rewrites, however, it helps seek count (more ef cient) but may harm ranking scores.

4.3 Ablation Study

We compare different exploration noise, including action space noise and parameter space noise. For the action space noise, the noise on discrete and continuous actions are applied separately. We test Gaussian and OrnsteinUhlenbeck noise on continuous actions, while the discrete actions only use -greedy. To keep the comparison fair, we also test greedy strategy on continuous actions by uniformly sampling a point with probability. The parameter noise is directly applied on policy network $(s;)$. We found the parameter space noise performs more stable than all types of action space noise. The uniform sampling on both discrete and continous action-space with -greedy failed for sometimes, thus we did not include it in comparison. We give the relative improvement in Table 1 compared to RNN with Gaussian Noise. The results show that parameter space noise has best performance overall.

The results in Figure 6(a) and Table 2 show that RNN got signi cantly better performance than MLP with or without parameterization on our environment. It indicates that the environment is highly non-Markov. We guess one obvious possibility is that the system signals of the state space cannot include all the information, while recurrent networks try to extract latent state from the history. Note that each episode will sample one query, thus the query embedding is constant for all steps in one episode. Another possible reason is that some rewrites may not be supported by special queries or have too small change for the state signals.

We compare different parameterization methods on recurrent networks. The results are shown in Figure 4(b) and Table 2. We empirically found that our modi ed strategy using reward bin is more stable from the beginning. It is possible that the agent repeatedly replays not only good experience on matching documents, but also learns to avoid punishment we set, as we expect. In general prioritized replay, the sampling probability is just based on TD-error and may be biased to worse samples since value function may not update towards possible results.

4.4 Benchmarking Games

We experiment on Platform-v0 and Goal-v0 from Bester et al. (2019). Note that, in all these games, each discrete action has a separate continuous action-parameter space. Our algorithm is designed for shared action-parameters and does not utilize such prior, thus we do not compare with P-DQN-style algorithms. We apply n-step return since the episode in these games is much longer. We assume the environment is fully observable and do not use recurrent networks in comparison.

Hyperparameters. We examine the games with different combinations of hyperparameters, since they are easy to parallize on each training nodes without the need to connect to an production environment emulator. With NNI, the learning rates for value and policy network are set to log-uniform in $[3 \quad 10^{-4}; 3 \quad 10^{-3}]$ and $[1 \quad 10^{-4}; 1 \quad 10^{-3}]$. The in parameterized replay is set to log-uniform in $[0:2; 1:0]$. The action noise threshold in parameter space noise is set to log-uniform between $[0:05; 0:8]$. Tn soft parameter update, the is set to log-uniform in $[1 \quad 10^{-3}; 1 \quad 10^{-2}]$.

In Platform-v0 and Goal-v0, we found the agent is not very sensitive to the range we set, such as , and . The values of top 20% trials vary between $[0:05; 0:3]$, while and values are evenly scattered in the de ned range. In these settings, usually the agent prefers slightly larger value learning rate than policy learning rate.

Average Eval Return	Our	PA-DDPG
Platform-v0	0.9573	0.3113
Goal-v0	34.20	-6.208

Table 3: Average evaluation results (the average of all training rewards and nal evaluation reward) on benchmarks Platform-v0 and Goal-v0 with PA-DDPG (Hausknecht & Stone (2015)), MP-DQN (Bester et al. (2019)) and P-DQN (Xiong et al. (2018)). We use reported number from the papers, while last two methods report another metric on Goal-v0.

5 Related Work

While the aforementioned algorithms and techniques work on discrete or continuous action spaces, it is not trivial to apply them on parameterized action space, since such discrete-continuous hybrid action space is hard to parameterized by a single distribution. A related series of work is to combine DDPG and DQN to optimize Q-value function on parameter actions. There are two classes of methods that belong to them: PA-DDPG (Hausknecht & Stone (2015)) based on DDPG and P-DQN (Xiong et al. (2018)) based on DQN. They are Q-Learning-based methods which select the best action by maximizing the Q-value function on discrete action space or learning a deterministic policy outputting best continuous action. However, they use different strategies to combine discrete and continuous actions. Bester et al. (2019) (MP-DQN) extends P-DQN to tackle the problem that Q-value is a function of the joint action-parameter vector $Q(s^0, k^0, x^Q(s^0))$ in normal PAMDP, which may results in fault gradients. However, such problem does not exist in our slightly modi ed setting, since the action-parameter space in the match plan generation is inherently de ned to be shared for each $k \in A_d$. Masson et al. (2016) purposes a method to iteratively optimizing discrete and continuous actions by alternating between them. Another perspective (Klimek et al. (2017); Wei et al. (2018); Fu et al. (2019)) for a parameterized action space is to regard it as a two-hierarchy action space, where an agent selects discrete action rst and continuous parameter later. However, we do not consider this direction in current scheme because we share the same parameters for all discrete action in not very large scale. Therefore, the hierarchical methods may not bring a signi cant performance gain.

6 Discussions

In this paper, we present a parameterized action RL match plan generation method which extends the plan generation to the general case without any prede ned knowledge. Key to address the problem are normalized softmax values of discrete actions to enable gradients backpropagation, parameter space noise on parameters of the policy for unifying the exploration direction in both discrete and continuous spaces, and recurrent deterministic policies with prioritized replay buffer to accelerate and stabilize the training. Our algorithm can be applied to not only the match plan generation environment, but also other similar parameteried action environments. The experiment results demonstrate our method outperforms the well-designed hand-crafted rules in Bing and serveral baseline results in some existing PARL benchmarks. In this paper, we mainly discuss about of ine training procedure. In the future, we plan to apply learned policy to the production environment.

REFERENCES

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.

Craig J Bester, Steven D James, and George D Konidaris. Multi-pass q-networks for deep reinforcement learning with parameterised action spaces. arXiv preprint arXiv:1905.04388, 2019.

Haotian Fu, Hongyao Tang, Jianye Hao, Zihan Lei, Yingfeng Chen, and Changjie Fan. Deep multi-agent reinforcement learning with discrete-continuous hybrid action spaces. arXiv preprint arXiv:1903.04959, 2019.

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

Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. arXiv preprint arXiv:1511.04143, 2015.

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

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

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

Maciej Klimek, Henryk Michalewski, Piotr Mi, et al. Hierarchical reinforcement learning with parameters. In Conference on Robot Learning, pp. 301–313, 2017.

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

Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In Thirtieth AAAI Conference on Arti cial Intelligence, 2016.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426, 2018.

Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 2430–2439. JMLR. org, 2017.

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. Human-level control through deep reinforcement learning. Nature, 518(7540):529, 2015.

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

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

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

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

Ermo Wei, Drew Wicke, and Sean Luke. Hierarchical approaches for reinforcement learning in parameterized action space. In 2018 AAAI Spring Symposium Series, 2018.

Ian H Witten, Ian H Witten, Alistair Moffat, Timothy C Bell, Timothy C Bell, and Timothy C Bell. Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann, 1999.

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

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

A APPENDIX

A.1 PSEUDOCODE

Algorithm pseudocode for the algorithm is provided in Algorithm 1 which includes all aforementioned details.

Algorithm 1

Input: Empty replay buffer $\mathcal{D}$, init parameter noise std σ_{param}, action noise threshold σ_{action}
Initialize policy parameters ϕ, value parameters θ_1, θ_2
Set target networks' parameters to original parameters $\phi_{target} \leftarrow \phi$, $\theta_{Q_{target}} \leftarrow \theta_Q$
for each episode do
  Perturb policy parameters $\tilde{\phi} \leftarrow \phi + \mathcal{N}(0, \sigma_{param})$ and target policy parameters $\tilde{\phi}_{target}$
  Observe state s_t, output a disturbed action embedding $a_t^{soft} = \tilde{\mu}(s_t; \tilde{\phi})$
  Compute executing action a_t by taking max over k
  Execute parameterized action a_t in the environment server
  Observe next state s_{t+1}, reward r_t, done signal d_t denoting if s_{t+1} is terminal
  Store transition tuple (including action embedding) $(s_t, a_t^{soft}, r_t, s_{t+1}, d_t)$ to the buffer $\mathcal{D}$
  If s_{t+1} is terminal, reset to an initial state s_0
  for each update if update-condition do
    Sample a minibatch from prioritized replay buffer $\mathcal{D}$ with specific priorities
    Compute and transform target actions with disturbed target policy network
    Update parameter noise std σ_{param} using empirical distance $d(\phi, \tilde{\phi})$
    Compute targets for TD-error with min Q-value in the twin Q-networks
    Update Q-network parameters using gradient descent
    if policy update frequency then
      Update policy network parameters with
      Update target parameters with polyak averaging
    end if
  end for
end for

A.2 FURTHER EXPERIMENTAL DETAILS

Hyperparameter search. We use Microsoft NNI[2] and OpenPAI[3] to search hyperparameters. The final metric to report to NNI is set to the sum of average training reward and final evaluation reward (repeated 1,000 times) `final_metric=(avg_reward+eval_reward)/2`. The intermediate metric is set to evaluation reward per 1,000 episodes. We also use the early stop assessor. Each GPU server node connects to a production environment emulator with ethernet.

Model architecture. Both policy and value networks use two fully connected layers with 512 hidden units and a LSTM layer (Hochreiter & Schmidhuber (1997)). Each layer also uses a layer normalization (LayerNorm, Ba et al. (2016)) as suggested by Plappert et al. (2017) in consideration of stability for noise applied on parameters, and follows a ReLU activation. Both output heads of the policy network has a hidden layer with $tanh$ activations.

A.3 FURTHER ENVIRONMENTAL DETAILS

Accumulated values. Note that, for accumulated values in received states, rewards and outputted action-parameters, we use the difference (delta values) from the last step. For states and rewards, it is $s_t = s'_t - s_{t-1}$, $r_t = r'_t - r_{t-1}$, where where s' and r' denote raw state and reward. For actions, the agent outputs raw action-parameter output a'_t, and the emulator converts it to accumulated value $a_t = a'_t + a_{t-1}$.

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