Deep Reinforcement Learning in Large Discrete Action Spaces

Gabriel Dulac-Arnold
Rich Evans
Hado van Hasselt
Peter Sunehag
Google DeepMind

1. Introduction

Advanced AI systems will likely need to reason with a large number of actions at every step. Recommender systems used in large systems such as YouTube and Amazon must reason about millions of items every second, and control systems for large industrial processes may have millions of possible actions that can be applied at every time step. All of these systems are fundamentally reinforcement learning (Sutton & Barto, 1998) problems, but current algorithms are difficult or impossible to apply.

In this paper, we present a new policy architecture which operates efficiently with a large number of actions. We achieve this by leveraging prior information about the actions to embed them in a continuous space upon which the actor can generalize. This embedding also allows the policy’s complexity to be decoupled from the cardinality of our action set. Our policy produces a continuous action within this space, and then uses an approximate nearest-neighbor search to find the set of closest discrete actions in logarithmic time. We can either apply the closest action in this set directly to the environment, or fine-tune this selection by selecting the highest valued action in this set relative to a cost function. This approach allows for generalization over the action set in logarithmic time, which is necessary for making both learning and acting tractable in time.

We begin by describing our problem space and then detail our policy architecture, demonstrating how we can train it using policy gradient methods in an actor-critic framework. We demonstrate the effectiveness of our policy on various tasks with up to one million actions, but with the intent that our approach could scale well beyond millions of actions.

2. Definitions

We consider a Markov Decision Process (MDP) where $\mathcal{A}$ is the set of discrete actions, $\mathcal{S}$ is the set of discrete states, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the transition probability distribution, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1]$ is a discount factor for future rewards. Each action $a \in \mathcal{A}$ corresponds to an $n$-dimensional vector, such that $a \in \mathbb{R}^n$. This vector provides information related to the action. In the same manner, each state $s \in \mathcal{S}$ is a vector $s \in \mathbb{R}^m$.

The return of an episode in the MDP is the discounted sum of rewards received by the agent during that episode: $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r(s_i, a_i)$. The goal of RL is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ which maximizes the expected return over all episodes, $\mathbb{E}[R_1]$. The state-action value function $Q^\pi(s, a) = \mathbb{E}[R_1 | s_1 = s, a_1 = a, \pi]$ is the expected return starting from a given state $s$ and taking an action $a$, following $\pi$ thereafter. $Q^\pi$ can be expressed in a recursive manner using the Bellman equation $Q^\pi(s, a) = r(s, a) + \gamma \sum_{a'} P(s' | s, a)Q^\pi(s', \pi(s'))$. In this paper, both $Q$ and $\pi$ are approximated by parametrized functions.

3. Problem Description

There are two primary families of policies often used in RL systems: value-based, and actor-based policies.

For value-based policies, the policy’s decisions are directly conditioned on the value function. One of the more common examples is a policy that is greedy relative to the value function:

$$\pi_Q(s) = \arg \max_{a \in \mathcal{A}} Q(s, a). \quad (1)$$

In the common case that the value function is a parameterized function which takes both state and action as input, $|\mathcal{A}|$ evaluations are necessary to choose an action. This quickly becomes intractable, especially if the parameterized function is costly to evaluate, as is the case with deep neural networks. This approach does, however, have the desirable property of being capable of generalizing over actions when using a smooth function approximator. If $a_i$ and $a_j$ are similar, learning about $a_i$ will also inform us about $a_j$. Not only does this make learning more efficient, it also allows value-based policies to use the action features to reason about previously unseen actions. Unfortunately,
Deep Reinforcement Learning in Large Discrete Action Spaces

![Wolpertinger Architecture](image)

Figure 1. Wolpertinger Architecture

execution complexity grows linearly with $|A|$ which renders this approach intractable when the number of actions grows significantly.

In a standard actor-critic approach, the policy is explicitly defined by a parameterized actor function: $\pi_\theta: S \rightarrow A$. In practice $\pi_\theta$ is often a classifier-like function approximator, which scale linearly in relation to the number of actions. However, actor-based architectures avoid the computational cost of evaluating a likely costly Q-function on every action in the $\arg\max$ in Equation (1). Nevertheless, actor-based approaches do not generalize over the action space as naturally as value-based approaches, and cannot extend to previously unseen actions.

Sub-linear complexity relative to the action space and an ability to generalize over actions are both necessary to handle the tasks we interest ourselves with. Current approaches are not able to provide both of these, which motivates the approach described in this paper.

4. Proposed Approach

We propose a new policy architecture which we call the Wolpertinger architecture. This architecture avoids the heavy cost of evaluating all actions while retaining generalization over actions. This policy builds upon the actor-critic (Sutton & Barto, 1998) framework. We define both an efficient action-generating actor, and utilize the critic to refine our actor’s choices for the full policy. We use multi-layer neural networks as function approximators for both our actor and critic functions. We train this policy using Deep Deterministic Policy Gradient (Lillicrap et al., 2015). The Wolpertinger policy’s algorithm is illustrated in Figure 1. We will detail these in the following sections.

Our architecture reasons over actions within a continuous space $\mathbb{R}^n$, and then maps this output to the discrete action set $A$. We will first define $f_\theta^*: S \rightarrow \mathbb{R}^n$ with $f_\theta^*(s) = \hat{a}$. $f_\theta^*$ is a function parametrized by $\theta^*$, mapping from the state representation space $\mathbb{R}^m$ to the action representation space $\mathbb{R}^n$. This function provides a proto-action in $\mathbb{R}^n$ for a given state, which will likely not be a valid action, i.e. it is likely that $\hat{a} \notin A$. Therefore, we need to be able to map from $\hat{a}$ to an element in $A$. We can do this with $g: \mathbb{R}^n \rightarrow A$ where:

$$g_k(\hat{a}) = \arg\min_{a \in A} |a - \hat{a}|_2.$$

$g_k$ is a $k$-nearest-neighbor mapping from a continuous space to a discrete set. It returns the $k$ actions in $A$ that are closest to $\hat{a}$ by $L_2$ distance. In the exact case, this lookup is of the same complexity as the $\arg\max$ in the value-function derived policies described in Section 3, but each step of evaluation is an $L_2$ distance instead of a full value-function evaluation. This task has been extensively studied in the approximate nearest neighbor literature, and the lookup can be performed in an approximate manner in logarithmic time (Muja & Lowe, 2014). This step is described by the bottom half of Figure 1, where we can see the actor network producing a proto-action, and the $k$-nearest neighbors being chosen from the action embedding.

4.1. Action Refinement

Depending on how well the action representation is structured, actions with a low $Q$-value may occasionally sit closest to $\hat{a}$ even in a part of the space where most actions have a high $Q$-value. Additionally, certain actions may be near each other in the action embedding space, but in certain states they must be distinguished as one has a particularly low long-term value relative to its neighbors. In both of these cases, simply selecting the closest element to $\hat{a}$ from the set of actions generated previously is not ideal.

To avoid picking these outlier actions, and to generally improve the finally emitted action, the second phase of the algorithm, which is described by the top part of Figure 1, refines the choice of action by selecting the highest-scoring action according to $Q_{\theta^*}$:

$$\pi_\theta(s) = \arg\max_{a \in g_k f_\theta^*(s)} Q_{\theta^*}(s, a). \quad (2)$$

This equation introduces $\pi_\theta$ which is the full Wolpertinger

---

1For $k = 1$ this is a simple nearest neighbor lookup.
policy. The parameter $\theta$ represents both the parameters of the action generation element in $\theta^*$ and of the critic in $\theta^Q$.

As we demonstrate in Section 6, this second pass makes our algorithm significantly more robust to imperfections in the choice of action representation, and is essential in making our system learn in certain domains. The size of the generated action set, $k$, is task specific, and allows for an explicit trade-off between policy quality and speed.

4.2. Training with Policy Gradient

Although the architecture of our policy is not fully differentiable, we argue that we can nevertheless train our policy by following the policy gradient of $f_{\theta^*}$. We will first consider the training of a simpler policy, one defined only as $\pi_\theta = g \circ f_{\theta^*}$. In this initial case we can consider that the policy is $f_{\theta^*}$ and that the effects of $g$ are a deterministic aspect of the environment. This allows us to maintain a standard policy gradient approach to train $f_{\theta^*}$ on its output $\hat{a}$, effectively interpreting the effects of $g$ as environmental dynamics. Similarly, the $\arg \max$ operation in Equation (2) can be seen as introducing a non-stationary aspect to the environmental dynamics.

4.3. Wolpertinger Training

The training algorithm’s goal is to find a parameterized policy $\pi_\theta$ which maximizes its expected return over the episode’s length. To do this, we find a parametrization $\theta^*$ of our policy which maximizes its expected return over an episode: $\theta^* = \arg \max_{\theta} \mathbb{E}[R_1|\pi_\theta]$.

We perform this optimization using Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015) to train both $f_{\theta^*}$ and $Q_{\theta^Q}$. DDPG draws from two stability-inducing aspects of Deep Q-Networks (Mnih et al., 2015) to extend Deterministic Policy Gradient (Silver et al., 2014) to neural network function approximators by introducing a replay buffer (Lin, 1992) and target networks. DPG is similar to work introduced by NFQCA (Hafner & Riedmiller, 2011) and leverages the gradient-update originally introduced by ADHDP (Prokhorov et al., 1997).

The goal of these algorithms is to perform policy iteration by alternatively performing policy evaluation on the current policy with Q-learning, and then improving upon the current policy by following the policy gradient. The critic is trained from samples stored in a replay buffer (Mnih et al., 2015). Actions stored in the replay buffer are generated by $\pi_\theta^*$, but the policy gradient $\nabla_\theta Q_{\theta^Q}(s,a)$ is taken at $\hat{a} = f_{\theta^*}(s)$. This allows the learning algorithm to leverage the otherwise ignored information of which action was actually executed for training the critic, while taking the policy gradient at the actual output of $f_{\theta^*}$. The target action in the Q-update is generated by the full policy $\pi_\theta$ and not simply $f_{\theta^*}$.

5. Related Work

There has been limited attention in the literature with regards to large discrete action spaces within RL. Most prior work has been concentrated on factorizing the action space into binary subspaces. Generalized value functions were proposed in the form of H-value functions (Pazis & Parr, 2011), which allow for a policy to evaluate $\log(|\mathcal{A}|)$ binary decisions to act. This learns a factorized value function from which a greedy policy can be derived for each subspace. This amounts to performing $\log(|\mathcal{A}|)$ binary operations on each action-selection step.

A similar approach was proposed which leverages Error-Correcting Output Code classifiers (ECOCs) (Dieterich & Bakiri, 1995) to factorize the policy’s action space and allow for parallel training of a sub-policy for each action subspace (Dulac-Arnold et al., 2012). In the ECOC-based approach case, a policy is learned through Rollouts Classification Policy Iteration (Lagoudakis & Parr, 2003), and the policy is defined as a multi-class ECOC classifier. Thus, the policy directly predicts a binary action code, and then a nearest-neighbor lookup is performed according to Hamming distance.

Both these approaches effectively factorize the action space into $\log(|\mathcal{A}|)$ binary subspaces, and then reason about these subspaces independently. These approaches can scale to very large action spaces, however, they require a binary code representation of each action, which is difficult to design properly. Additionally, the generalized value-function approach uses a Linear Program and explicitly stores the value function per state, which prevents it from generalizing over a continuous state space. The ECOC-based approach only defines an action producing policy and does not allow for refinement with a Q-function.

These approaches cannot naturally deal with discrete actions that have associated continuous representations. The closest approach in the literature uses a continuous-action policy gradient method to learn a policy in a continuous action space, and then apply the nearest discrete action (Van Hasselt et al., 2009). This is in principle similar to our approach, but was only tested on small problems with a uni-dimensional continuous action space (at most 21 discrete actions) and a low-dimensional observation space. In such small discrete action spaces, selecting the nearest discrete action may be sufficient, but we show in Section 6 that a more complex action-selection scheme is necessary to scale to larger domains.

Recent work extends Deep Q-Networks to ‘unbounded’ action spaces (He et al., 2015), effectively generating action representations for any action the environment provides,
and picking the action that provides the highest Q. However, in this setup, the environment only ever provides a small (2-4) number of actions that need to be evaluated, hence they do not have to explicitly pick an action from a large set.

A similar policy architecture has also been leveraged for learning to attend to actions in MDPs which take in multiple actions at each state (Slate MDPs) (Sunehag et al., 2015).

6. Experiments

We evaluate our architecture on three environments: discretized control, multi-step planning and a sequential recommender system task. The first two environments demonstrate the general behavior of the learning algorithm and the scaling abilities of our approach, going up to one million actions. The third environment serves as a validation of our approach on real-world data.

6.1. Discretized Control

We use a discretized control task to demonstrate the speedups possible using the Wolpertinger architecture. We use a cart-pole task defined in the MuJoCo (Todorov et al., 2012) physics simulator to evaluate how the agent’s performance and learning speed relate to the number of discrete actions. We discretize this one-dimensional control task into \( i \) equally spaced values.

We use this environment as a demonstration both that our agent is able to reason with both a small and large number of actions efficiently, especially when the action representation is well-formed. In these tasks, actions are represented by the force to be applied on the cart. In the cart-pole case, this is along a single dimension, so actions are represented by a single number.

The cart-pole task was generated with a discretization of one million actions. On this task, our algorithm is able to converge to a good policy. Each item also has a reward \( r \) associated with it if accepted by the user. The current state \( s \) will accept recommendation \( j \) if \( W_{i,j} \) defines the probability that a user will accept recommendation \( j \) given that the last item they accepted was item \( i \). Each item also has a reward \( r \) associated with it if accepted by the user. The current state is the item the user is currently consuming, and the previously recommended items do not affect the current transition. Not all items are accessible from a given state, so it is necessary to sometimes recommend a low-reward item to subsequently access a state which leads to higher-reward items. This formulation means that greedy approaches are

![Figure 2. Agent performance for various settings of \( k \). Top figure is for exact lookup, and bottom with FLANN. Top figure is as a function of number of steps, and bottom figure as a function of wall-time on one million action cart-pole. We can see that with 0.5% of neighbors, training time is prohibitively slow.](image)

where 39 nearest neighbors at the leaf nodes are checked, ‘Fast’ corresponds to a randomized K-d tree with 1 nearest neighbor at the leaf node checked. These settings were obtained with FLANN’s auto-tune mechanism. In this particular case, FLANN performance impacts nearest-neighbor lookup negatively for all settings except ‘fast’ as we are looking for a nearest neighbor in a single dimension. We will see in the next section that for more action dimensions this is no longer true.

6.2. Recommender Environment

To demonstrate how the agent would perform on a real-world large action space problem we constructed a simulated recommendation system utilizing data from a live large-scale recommendation engine. This environment is characterized by a set of items to recommend, which correspond to the action set \( \mathcal{A} \) and a transition probability matrix \( W \), such that \( W_{i,j} \) defines the probability that a user will accept recommendation \( j \) given that the last item they accepted was item \( i \). Each item also has a reward \( r \) associated with it if accepted by the user. The current state is the item the user is currently consuming, and the previously recommended items do not affect the current transition. Not all items are accessible from a given state, so it is necessary to sometimes recommend a low-reward item to subsequently access a state which leads to higher-reward items. This formulation means that greedy approaches are

![Figure 2.](image)
Experiments were run on 3 different recommender tasks involving 49 elements, 835 elements, and 13,138 elements. These tasks’ dynamics are quite irregular, with certain actions being good in many states, and certain states requiring a specific action rarely used elsewhere. This has the effect of rendering agents with \( k = 1 \) quite poor at this task. Additionally, although initial exploration methods were purely uniform random with an epsilon probability, to better simulate the reality of the running system — where state transitions are also heavily guided by user choice — we restricted our epsilon exploration to a likely subset of good actions provided to us by the simulator. This subset is only used to guide exploration; at each step the policy must still choose amongst the full set of actions if not exploring. Learning with uniform exploration converges, but in the larger tasks performance is typically 50% of that with guided exploration.

For each environment, we vary the number of nearest neighbors \( k \) from \( k = 1 \), to \( k = |\mathcal{A}| \). For \( k = 1 \), we demonstrate the performance of the nearest-neighbor element of our policy \( g \circ f_{\Theta^*} \). This is the fastest policy configuration, but as we see in the section, is not always sufficiently expressive. For \( k = |\mathcal{A}| \), we demonstrate the performance of a policy that is greedy relative to \( Q \), always choosing the true maximizing action from \( \mathcal{A} \). This gives us an upper bound on performance, but we will soon see that this approach is often computationally intractable. Intermediate values of \( k \) are evaluated to demonstrate the performance gains of partial re-ranking.

Figure 3 shows performance on the 835-element task using exact lookup for varying values of \( k \) as a percentage of the total number of actions. We can see a clear progression of performance as \( k \) is increased in this task. Although not displayed in the plot, these smaller action sizes have much less significant speedups, with \( k = |\mathcal{A}| \) taking only twice as long as \( k = 83 \) (1%).

<table>
<thead>
<tr>
<th>( k )</th>
<th>Exact</th>
<th>Slow</th>
<th>Medium</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>1% – 131</td>
<td>23</td>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>5% – 656</td>
<td>10</td>
<td>13</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>10% – 1,313</td>
<td>7</td>
<td>7.5</td>
<td>7.5</td>
<td>7</td>
</tr>
<tr>
<td>100% – 13e3</td>
<td>1.5</td>
<td>1.6</td>
<td>1.5</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 1. Median steps/second as a function of \( k \) & FLANN settings on the 13k recommender task.

Results on the 13, 138 element task are visualized in Figures 4-top for varying values of \( k \), and in Figure 4-bottom with varying FLANN settings. We note that the agent using all actions (in yellow) does not train as many steps due to slow training speed. It is training approximately 15 times slower in wall-time than the 1% agent.

Results on the first two tasks show that our method can scale to more than a million actions, and results on the real-world recommender tasks show that our approach can scale to real-world MDPs with large number of actions.

7. Conclusion

In this paper we introduce a new policy architecture able to efficiently learn and act in large discrete action spaces. We
describe how this architecture can be trained using DDPG and demonstrate good performance on a series of tasks with a range from tens to one million discrete actions.

Architectures of this type give the policy the ability to generalize over the set of actions with sub-linear complexity relative to the number of actions. We demonstrate how considering only a subset of the full set of actions is sufficient in many tasks and provides significant speedups. Additionally, we demonstrate that an approximate approach to the nearest-neighbor lookup can be achieved while often impacting performance only slightly.

References


