Hierarchical Approaches for Reinforcement Learning in Parameterized Action Space

Ermo Wei, Drew Wicke, Sean Luke
Department of Computer Science, George Mason University, Fairfax, VA USA
ewei@cs.gmu.edu, dwicke@gmu.edu, sean@cs.gmu.edu

Abstract

We explore Deep Reinforcement Learning in a parameterized action space. Specifically, we investigate how to achieve sample-efficient end-to-end training in these tasks. We propose a new compact architecture for the tasks where the parameter policy is conditioned on the output of the discrete action policy. We also propose two new methods based on the state-of-the-art algorithms Trust Region Policy Optimization (TRPO) and Stochastic Value Gradient (SVG) to train such an architecture. We demonstrate that these methods outperform the state of the art method, Parameterized Action DDPG, on test domains.

Introduction

Deep Reinforcement Learning (DRL) has achieved success in recent years, including beating human masters in Go (Silver et al. 2016), attaining human level performance in Atari games (Mnih et al. 2015), and controlling robots in high-dimensional action spaces (Lillicrap et al. 2015). With these successes, researchers have begun to explore new frontiers in DRL, including how to apply DRL in complex action spaces. Consider for example the real time strategy game StarCraft, where at any time during play we may choose among different types of actions to be able to finish our goals (Vinyals et al. 2017). For example, we may need to choose a building to construct and then select where to build it; or choose a squad of armies and direct them to explore the map. Critically, instead of having a single action set, we may have several sets of actions, either continuous or discrete, and to get a meaningful action to execute, we must choose wisely among these sets.

In this paper, we explore how to apply DRL to tasks with more than one set of actions. Specifically, we consider tasks with parameterized action spaces (Masson, Ranchod, and Konidaris 2016), where at each step the agent must choose both a discrete action and a set of continuous parameters for that action. Tasks with this kind of action space have been proposed in the Reinforcement Learning (RL) community for a long time (Stone et al. 2006) but have not been explored much.

One approach to handle a RL task with a parameterized action space is to do *alternating optimization* (Masson, Ranchod, and Konidaris 2016). Here, we break the task into two separate subtasks by fixing either the parameters or discrete actions and then applying RL algorithms alternating between the induced subtasks. Although this method can work, it has a huge sample complexity because every time we switch the subtask, the previous experience is no longer valid as the environment is different.

Thus, a sample efficient alternative is to train the policies for discrete actions and parameters at the same time. There have already been steps in this direction. Hausknecht and Stone simultaneously train two policies which can produce the values for discrete action and parameters respectively and then select the action to execute based on their values. There are two main drawbacks of this method. The first is that the parameter policy does not know what discrete action is selected at execution time. Thus, the parameter policy needs to output all the parameters for all the discrete actions at every step. As a result, the output size of the parameter policy can explode if we have high dimensional parameters with large discrete action sets. The second problem is that neither the policies nor the training method are aware of the action-selection procedure after the action and parameter values are produced. Therefore, the method may be missing a crucial piece of information for it to succeed.

In this paper, we propose a new architecture for parameterized action space tasks. In our method, we condition our parameter policy on the output of the discrete action policy, thus greatly reducing the output size of the parameter policy. Then we extend the state-of-the-art DRL algorithms to efficiently train the new architecture for parameterized action space tasks. In experiments we show that our methods can achieve better performance than the state of the art method.

Background

Before we delve into the architecture and algorithms, we first present a mathematical formulation of Markov Decision Processes (MDPs) along with some relevant policy gradient algorithms. Then we present Parameterized Action MDPs (PAMDPs). Lastly, we discuss some of the previous work in PAMDPs that is related to our paper.

MDPs and Policy Gradient Methods

Markov Decision Process A Markov Decision Process (or MDP) can be used to model the interaction an agent has with its environment. A MDP is a tuple $\{S, A, T, R, \gamma, H\}$ where S is the set of states; A is the set of actions available to the agent; T is the transition function $T(s, a, s') = P(s'|s, a)$ defining the probability of transitioning to state $s' \in S$ when in state $s \in S$ and taking action $a \in A$; R is the reward function $R : S \times A \mapsto \mathbb{R}$; $0 < \gamma < 1$ is a discount factor; and H is the horizon time of the MDP, that is, the MDP runs for only H steps.[*] An agent selects its actions based on a policy $\pi_\theta(\cdot|s)$, which is a distribution over all possible actions a in state s parameterized by $\theta \in \mathbb{R}^n$.

Policy Gradient Methods One of the major approaches to deal with continuous control problems in MDPs is to apply a policy gradient method. In policy gradient methods, we are trying to use gradient ascent to optimize the following objective

$$\begin{aligned} J(\theta) &= E_{s\sim\rho^{\pi_\theta}}[V^{\pi_\theta}(s)] \\ &= \int_S \rho^{\pi_\theta}(s) V^{\pi_\theta}(s)\, ds, \end{aligned} \tag{1}$$

where s is the state visited, and $\rho^{\pi_\theta}(s)$ is the distribution over all states induced by executing policy π_θ. Many algorithms have been proposed to optimize this objective, including REINFORCE (Williams 1992), GPOMDP (Baxter and Bartlett 2001), and Trust Region Policy Optimization (TRPO) (Schulman et al. 2015), where we collect a set of trajectory samples with the form $\tau = \langle s_0, a_0, s_1, a_1, \ldots, s_H, a_H\rangle$ and use them to evaluate the gradient of $J(\theta)$. It turns out that sometimes, it is beneficial to learn an additional value function $Q(s, a)$ or $V(s)$ to reduce the variance in estimating the gradient of $J(\theta)$. This leads to a family of algorithms named "actor-critic" algorithms where the "actor" is the policy π and the "critic" is the value function. This family of algorithms includes the Stochastic Policy Gradient Theorem (SPG) (Sutton et al. 2000), the Deterministic Policy Gradient Theorem (DPG) (Silver et al. 2014), and so on. In addition, DDPG (Lillicrap et al. 2015) is an extention of DPG to the DRL setting by using a replay buffer to assist off-policy learning.

Parameterized Action MDPs

The MDP notation can be generalized to deal with parameterized tasks, e.g., actions with parameters. Here, instead of having just one set of actions, we have multiple sets of controls: a finite set of discrete actions $A_d = \{a_1, a_2, \ldots, a_n\}$ and for each $a \in A_d$, a set of continuous parameters $X_a \subseteq R^{m_a}$. Thus, an action is a tuple (a, x) in the joint action space,

$$A = \bigcup_{a\in A_d} \{(a, x)|x \in X_a\}.$$

MDPs with this action space are called Parameterized Action MDPs (PAMDPs) (Masson, Ranchod, and Konidaris 2016).

[*]Any infinite horizon MDP with discounted rewards can be ϵ-approximated by a finite horizon MDP using a horizon $H_\epsilon = \frac{\log_\gamma(\epsilon(1-\gamma))}{\max_{s,a}|R(s,a)|}$ (Jie and Abbeel 2010).

Previous Work on Parameterized Action MDPs

Tasks with parameterized actions have been a research topic in RL for a long time (Stone et al. 2006). Zamani et al. considered tasks with a set of discrete parameterized actions (2012). However, their algorithm which is based on Symbolic Dynamic Programming, is limited to MDPs with internal logical relations.

We adopt the Parameterized Action MDP setting from (Masson, Ranchod, and Konidaris 2016). In their work, they train the policy in an alternative fashion. They first fix all the parameter policies, and hence induce an MDP with action set A of only discrete actions. Then they use Q-learning to learn a discrete policy in that MDP, and upon convergence, they fix the discrete policy, and start training the parameter policy. They show that this method can converge to local optima.

Rachelson, Fabiani, and Garcia used parameterized actions to deal with continuous time MDPs where the parameter for all the actions is the waiting time (2009). Thus, they have a unified parameter space. Sharma, Lakshminarayanan, and Ravindran did a similar approach where they extended TRPO to control the repetition of the action, that is, how many steps an action should execute (2017). They argued that the repetition times can be considered as a parameter for their original control signal. However, the repetition times are drawn from a fix set of integers, which is not a continuous signal.

The method that has the closest connection to our work is (Hausknecht and Stone 2015), which extended the DDPG to a parameterized action space. In this algorithm, the policy outputs all the parameters and all the discrete actions, and then selects the (a, x) tuple with the highest Q-value.

Hierarchical Approaches in PAMDPs

In this paper we propose a new, more natural architecture to generate actions for parameterized action tasks. In our algorithm, we have one neural network for the discrete policy and one neural network for the parameter policy. Our parameter policy $\pi(x|s, a)$ takes two inputs, the state s and the discrete action a sampled from discrete action policy $\pi(a|s)$. Then the joint action is given by $(a, x) \sim \pi(a, x|s) = \pi(a|s)\pi(x|s, a)$. Since the action a is known before we generate the parameters, we do not need the post processing step of determining which action tuple (a, x) has the highest Q-values. And since the parameter policy knows the discrete action a, the output size of parameter policy remains constant.

Previously, this architecture was not plausible in policy gradient methods due to the fact that we have to sample the discrete action in the middle of the forward pass, and the gradient cannot flow back to the discrete action policy in the backward pass due to the sampling operation. In this section, we describe two algorithms, Parameterized Action TRPO (PATRPO) and Parameterized Action SVG(0) (PASVG(0)) that solve this problem.

Before we delve into the algorithms, we first introduce some notation. We use $\pi_\Theta(a, x|s)$ to denote our overall policy, where a is the discrete action, x is the the parameter for that action, and Θ is all parameters for the model. Our policy can be broken into two separate policies using conditional probability $\pi_\Theta(a, x|s) = \pi^c_{\theta_x}(x|a, s)\pi^d_{\theta_a}(a|s)$, where θ_a and θ_x are the parameters for discrete action policy $\pi^d(a|s)$ and continuous parameter policy $\pi^c(x|a, s)$ respectively, and $\Theta = [\theta_a, \theta_x]$.

Parameterized Actions TRPO

Among all the policy gradient algorithms, TRPO and DDPG achieve the best performance as they are able to optimize large neural network policies (Duan et al. 2016). Thus, we consider how to apply these two algorithms in PAMDPs.

We first consider how to optimize our policy using TRPO's technique. In the TRPO, we are solving the following optimization problem:

$$\text{maximize}_\theta \;\; L_{\theta'}(\theta) = E_{s\sim\rho_{\theta'}, a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta'}(a|s)} Q_{\theta'}(s, a)\right]$$
$$\text{subject to} \;\; \overline{\text{KL}}_{\theta'}(\theta) = E_{s\sim\rho_{\theta'}}[D_{\text{KL}}(\pi_{\theta'}(\cdot|s)||\pi_\theta(\cdot|s))] < \delta,$$

where θ' and θ are the parameter vectors before and after each policy update respectively, and L is the surrogate loss. $Q_\theta(s, a)$ indicates the Q-function fitted using the samples from policy parameterized by θ. The idea behind TRPO is to optimize the policy in a stable way such that the new policy distribution after each update will not be too different from the old one. This is achieved through the KL-divergence constraint between the policy distributions before and after the parameter update.

A similar idea has been explored before in the natural policy gradient (Kakade 2002), where the objective function is replaced with linear approximation $\frac{\partial L_{\theta'}(\theta)}{\partial\theta}(\theta - \theta')$ and the KL-divergence is replaced with a quadratic approximation $(\theta' - \theta)^T A(\theta' - \theta)$. The positive semidefinite matrix A in the quadratic term is the Hessian matrix of constraint, e.g., $A = \frac{\partial^2}{\partial^2\theta}\overline{\text{KL}}_{\theta'}(\theta)$. However, when the policy model becomes large, A becomes very expensive to compute and store. What is special about TRPO is that it uses a Hessian-free optimization method (Martens 2010; Pearlmutter 1994) and conjugate gradient descent method to avoid the explicit formation of the Hessian matrix. Therefore, TRPO only has a slight increase in the computation cost for optimizing large neural networks.

To apply the TRPO in PAMDPs, we first write down the optimization problem using our notation, which is

$$\text{maximize}_\Theta \;\; E_{s\sim\rho_{\Theta'}, (a,x)\sim\pi_{\Theta'}}\left[\frac{\pi_\Theta(a, x|s)}{\pi_{\Theta'}(a, x|s)} Q_{\Theta'}(s, a, x)\right]$$
$$\text{subject to} \;\; E_{s\sim\rho_{\Theta'}}[D_{\text{KL}}(\pi_{\Theta'}(\cdot|s)||\pi_\Theta(\cdot|s))] < \delta$$

The objective can be further expanded to

$$E_{s\sim\rho_{\Theta'}, (a,x)\sim\pi_{\Theta'}}\left[\frac{\pi^c_{\theta_x}(x|a, s)\pi^d_{\theta_a}(a|s)}{\pi^c_{\theta'_x}(x|a, s)\pi^d_{\theta'_a}(a|s)} Q_{\Theta'}(s, a, x)\right]$$

Notice that, in the objective function, the samples are collecting using Θ' instead of Θ. Thus, in training time, we can just take the gradient of objective function w.r.t Θ to achieve end-to-end training like normal supervised learning, and do not need to use any trick.

However, some changes are needed to meet the constraint of TRPO as there is no closed form solution for computing KL-divergence between two joint distributions. Here, we rewrite the KL-divergence constraint into conditional divergence using the chain rule.

$$\begin{aligned}
&E_{s\sim\rho_{\Theta'}}[D_{\text{KL}}(\pi_{\Theta'}(\cdot|s)||\pi_\Theta(\cdot|s))] \\
=&E_{s\sim\rho_{\Theta'}}\Bigg[D_{\text{KL}}(\pi^d_{\theta'_a}(\cdot|s)||\pi^d_{\theta_a}(\cdot|s)) \\
&\quad + E_{a\sim\pi^d_{\theta'_a}(a|s)}\left[D_{\text{KL}}(\pi^c_{\theta'_x}(\cdot|s, a)||\pi^c_{\theta_x}(\cdot|s, a))\right]\Bigg] \\
=&E_{s\sim\rho_{\Theta'}}\Bigg[D_{\text{KL}}(\pi^d_{\theta'_a}(\cdot|s)||\pi^d_{\theta_a}(\cdot|s))\Bigg] \\
&\quad + E_{s\sim\rho_{\Theta'}}E_{a\sim\pi^d_{\theta'_a}(a|s)}\Bigg[D_{\text{KL}}(\pi^c_{\theta'_x}(\cdot|s, a)||\pi^c_{\theta_x}(\cdot|s, a))\Bigg]
\end{aligned}$$

Thus, we can use samples to estimate both the objective function and KL-divergence. However, we notice that we can further reduce the variance of estimating the KL-divergence by using the analytical form of discrete action policy $\pi(a|s)$. That is, the KL-divergence can be written as

$$\begin{aligned}
&E_{s\sim\rho_{\Theta'}}\Bigg[D_{\text{KL}}(\pi^d_{\theta'_a}(\cdot|s)||\pi^d_{\theta_a}(\cdot|s))\Bigg] \\
&\quad + E_{s\sim\rho_{\Theta'}}\Bigg[\pi(a|s)D_{\text{KL}}(\pi^c_{\theta'_x}(\cdot|s, a)||\pi^c_{\theta_x}(\cdot|s, a))\Bigg]
\end{aligned}$$

Using this form of constraint allows us to estimate the divergence between two policies even when we do not have samples for some discrete actions.

Parameterized Actions SVG(0)

Now we propose our second method based on the reparameterization trick.

One thing that makes the policy gradient methods special is that the samples we need to estimate the gradient come from the policy we are optimizing. That is, the objective usually takes the following form,

$$E_{p_\theta(x)}[f(x)].$$

We can write the gradient of expectation w.r.t θ in this way:

$$\begin{aligned}
\frac{\partial E_{p_\theta(x)}[f(x)]}{\partial\theta} &= \frac{\partial}{\partial\theta}\int_x p_\theta(x)f(x)\,dx \\
&= \int_x \frac{\partial p_\theta(x)}{\partial\theta} f(x)\,dx.
\end{aligned}$$

Since we lost the term $p(x)$ in the integral after we take the gradient, it's no longer an expectation, hence, we can no longer use samples from $p(x)$ to estimate it.

To solve this problem, people made the following changes to the gradient,

$$\begin{aligned}
\frac{\partial E_{p_\theta(x)}[f(x)]}{\partial\theta} &= \int_x \frac{\partial p_\theta(x)}{\partial\theta} f(x)\,dx \\
&= \int_x p(x)\left(\frac{1}{p(x)}\frac{\partial p_\theta(x)}{\partial\theta}\right) f(x)\,dx \\
&= E_{p_\theta(x)}\left[\frac{\partial \ln p_\theta(x)}{\partial\theta} f(x)\right]
\end{aligned}$$

This trick is the foundation for most of the policy gradient methods in RL.

Recently, another trick has been used to attack the same problem in the unsupervised learning community (Kingma and Welling 2013; Rezende, Mohamed, and Wierstra 2014). The idea is that a continuous random variable z can be obtained by first taking a noise variable ϵ and then deterministically transforming it. For example, a gaussian random variable $z \sim \mathcal{N}(\mu, \sigma^2)$ can be reparameterized into a noise random variable $\epsilon \sim \mathcal{N}(0, 1)$ with a deterministc transformation $g_{\mu,\sigma}(z) = \mu + \sigma\epsilon$. By applying this technique, we can optimize an expectation using samples from a noise distribution as follows

$$E_{p_\theta(x)}[f(x)] = \int_x p_\theta(x) f(x)\, dx = \int_\epsilon p(\epsilon) f(g_\theta(\epsilon))\, d\epsilon$$

Then the gradient can be easily written as

$$\frac{\partial E_{p_\theta(x)}[f(x)]}{\partial \theta} = \int_\epsilon p(\epsilon) \left(\frac{\partial f}{\partial g} \frac{\partial g}{\partial \theta} \right) d\epsilon = E_{p(\epsilon)} \left[\frac{\partial f}{\partial g} \frac{\partial g}{\partial \theta} \right]$$

This method has been successfully used in Variational Autoencoders (VAE) for various works (Walker et al. 2016; Sohn, Lee, and Yan 2015). It has also been applied to RL to train Stochastic Value Gradient (SVG) Learners (Heess et al. 2015). Recently, Jang, Gu, and Poole (2016), Maddison, Mnih, and Teh (2016) generalized the reparameterization trick to deal with discrete random variables with the *Gumbel-Softmax* trick. In the discrete case, a random variable x can be drawn from a discrete distribution $\{p(x_1), p(x_2), \ldots, p(x_n)\}$ by the Gumbel-Max trick (Maddison, Tarlow, and Minka 2014),

$$x = \text{argmax}_i [g_i + \ln p(x_i)]$$

where $g_i \sim \text{Gumbel}(0, 1)$. The Gumbel-Softmax trick replace the argmax operator in the above with a continuous differentiable softmax operator. With this change, we can now draw samples as

$$x = \frac{\exp\left[\left((g_i + \ln p(x_i)\right)/t\right]}{\sum_{i=1}^{n} \exp\left[\left((g_i + \ln p(x_i)\right)/t\right]}$$

where t is the "temperature" used to control the tradeoff between bias and variance. This trick has been applied to the RL setting as well, including imitation learning (Baram et al. 2017) and multiagent RL (Mordatch and Abbeel 2017).

For our problem, the key observation is that the two steps of decision making in a parameterized action policy (choosing from a discrete action and then determining the parameters for it), is very much like the paradigm in VAE (2013). In the VAE setting, the encoder of the VAE takes a sample x from the dataset, and generates a latent variable z from it. Then the decoder takes z and reconstructs x out of it. For our situation, the discrete action policy first takes the state s as input and generates a discrete action a, then determines the parameters x based on action a using the continuous parameter policy. Thus, we can roughly think of our discrete action policy and continuous parameter policy as the encoder and decoder in VAE respectively.

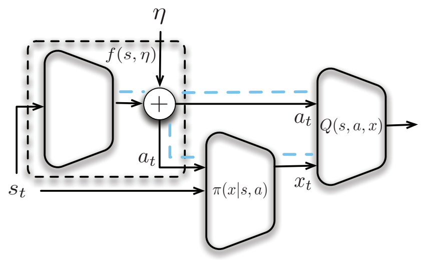

Figure 1: The training flow of the PASVG(0) agent. The black lines indicate the forward pass of the training, and the dash lines indicate the backward pass of the training. The dash box marks the reparameterized policy f.

We start with the objective function in (1) and write it in parameterized action setting.

$$\begin{aligned} J(\Theta) &= \int_s \rho^{\pi_\Theta}(s) V^{\pi_\Theta}(s) ds \\ &= E_{s\sim\rho_\Theta}\left[\sum_a \pi_\Theta(a, x|s) Q(s, a, x)\right] \\ &= E_{s\sim\rho_\Theta}\left[\sum_a \pi_{\theta_a}(a|s) Q(s, a, \pi_{\theta_x}(x|s, a))\right] \end{aligned}$$

For the last step in the previous derivation, we use the DDPG formulation. Then we apply the reparameterization trick. Following the convention in (Heess et al. 2015), we use η to represent the auxiliary noise variable instead of ϵ in the VAE setting.

$$J(\Theta) = E_{s\sim\rho_\Theta}\left[\sum_\eta p(\eta) Q(s, f_{\theta_a}(s, \eta), \pi_{\theta_x}(s, f_{\theta_a}(s, \eta)))\right]$$

where $a = f_{\theta_a}(s, \eta)$ is the discrete action policy after reparameterization. Then the gradient w.r.t Θ is simply

$$\frac{\partial J(\Theta)}{\partial \Theta} = E_{\rho_\Theta} E_{p(\eta)}\left[\frac{\partial}{\partial \Theta} Q(s, f_{\theta_a}(s, \eta), \pi_{\theta_x}(s, f_{\theta_a}(s, \eta)))\right]$$

Since we are reparameterizing our stochastic policy for a 0-step value function (Q-function), similar to Heess et al.'s method, thus we name our algorithm Parameterized Action SVG(0) (See Figure 1 for the training flow).

However, there is one critical difference between our work and Heess et al.. In our work, we do not need to infer the noise variable since we are not using any dynamic model. To see this, we rewrite the gradient estimation using the Bayes' rule, following the method from (Heess et al. 2015),

$$\begin{aligned} \frac{\partial J(\Theta)}{\partial \Theta} = {} & E_{\rho_\Theta} E_{\pi(a,x|s)} E_{p(\eta|a,x,s)} \\ & \left[\frac{\partial}{\partial \Theta} Q(s, f_{\theta_a}(s, \eta), \pi_{\theta_x}(s, f_{\theta_a}(s, \eta)))\right] \end{aligned} \qquad (2)$$

Heess et al. use this method to infer the noise ζ of their reparameterized approximate dynamic model $s' =$

Figure 2: Platform domain

$g(s, a, \zeta)$. Thus, they need to learn the $p(\zeta|s, a, s')$ which is similar to $p(\eta|a, x, s)$ in our case. However, for us, we use the sample η and generate a, x from it. Hence, we do not need to learn the model $p(\eta|a, x, s)$. Instead, we can just record the value of η when we are collecting the training samples.

The last part of the algorithm is to make the gradient estimation not depend on the samples collected by $\pi(a, x|s)$, as the policy is constantly changing. We use the replay buffer from DDPG to solve this issue and turn our algorithm into an off-policy algorithm to improve sample efficiency.

Experiments

We conducted our experiments using the Platform domain from (Masson, Ranchod, and Konidaris 2016) (See Figure 2). In this domain, we control the agents (cyan block) to jump across several platforms while avoiding enemies (red blocks) and falling off the platforms. This domain has three discrete actions to choose from: run, jump, and leap. A jump moves the agent over its enemies, while a leap propels the agent to the next platform. Each of the actions take one parameter which determines the speed along the x-axis. More details of the domain can be found in the original paper.

We implemented the Parameterization Action DDPG[†] (PADDPG) algorithm from (Hausknecht and Stone 2015) as our comparison baseline which is considered as the state of the art. Specifically, we implemented the PADDPG algorithm following the settings and parameters from the original paper except for the size of the hidden layers. In the original paper, PADDPG used a huge network with four hidden layers with size $\{1024, 512, 256, 128\}$, which is rare in DRL community for tasks with continuous signals. We followed the DDPG paper (Lillicrap et al. 2015), which used two hidden layers with sizes $\{400, 300\}$ for the neural networks. We also implemented their invert-gradient trick, as they claimed that this was the only way to make the learning work in a bounded parameter space.

For our PATRPO agent, we adopted the setting from (Duan et al. 2016), where we had three hidden layers with sizes $\{200, 100, 50\}$ for the policies. We used ReLU for activation, and Softmax and Tanh for the output layers of the discrete action policy and continuous parameter policy respectively. For our PASVG(0) agent, we also used neural networks with two hidden layers of sizes $\{400, 300\}$ and ReLU for activation. For the output layer, we used Gumbel Softmax for the discrete action policy and Tanh for the continuous parameter policy.

[†]This is the DDPG algorithm for parametereized action spaces, not to be confused with the DDPG algorithm from (Lillicrap et al. 2015) for continuous control.

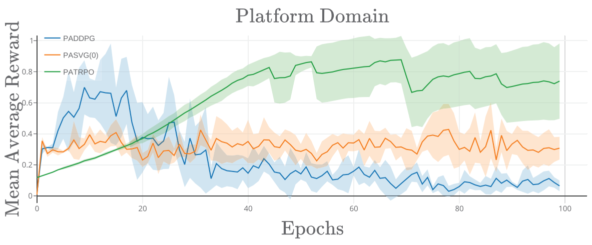

Figure 3: Comparison on Platform domain of three learners. The x-axis shows the training epochs. The y-axis shows the average reward. Solid lines are average value over five random seeds. Shaded regions are standard deviation.

We trained all the agents using 100 epochs with 10000 samples per epoch. For the online method, we had a replay buffer with size 10^7 and we did not start the training until we had 10^4 samples in the replay buffer, which is a standard setting in DRL experiments. We used 0.005 as the step size for PATRPO agent and 10^{-3} and 10^{-5} as the learning rate for the value function and policies respectively for the PASVG(0) agents. We fixed the temperature to 1.0 for the Gumbel-Softmax layer and kept it for the entire training process.

The experiment results are shown in Figure 3. The plot of PADDPG is very interesting: we found that it can learn to successfully finish the game at an early stage of learning, but would quickly unlearn that policy and converge to something else.

We then noted that, although we are using Tanh to bound the output of our parameter policy, which corresponded to the squash-gradient method in (Hausknecht and Stone 2015), we managed to make it work for our methods, which suggests that there are more training options than the invert-gradient method suggested by (Hausknecht and Stone 2015). Our PATRPO method achieved the best performance among all three learners, and unlike PADDPG learner, it maintained its performance after obtaining its best learned policy. The PASVG(0) learner converged to a local optimum with average reward of around 0.4. By examining the game, we found that this corresponded to avoiding the first enemy but failing to land on the second platform. One of the possible reasons was that the learner was conducting joint-learning, which is very similar to cooperative multiagent learning, and thus may converge to a local optima.

We further investigated this joint-learning issue by trying two different step sizes for the PATRPO agent. Figure 4 shows the result of using larger step size parameters. As we can see, both of the agents can achieve a good performance in relatively short period of time with much lower variance. But once they learn the optimal policy, their performance starts to drop and the variance becomes much larger. However, PATRPO still manages to maintain a decent performance which is far better than the PADDPG algorithm. This shows that a more stable method is desirable for learning in

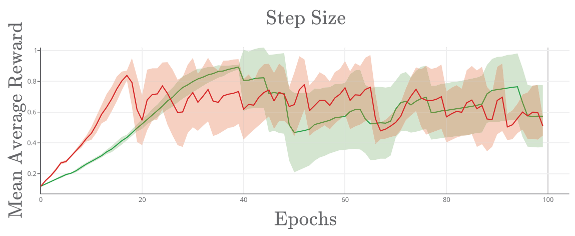

Figure 4: Different step size parameters for PATRPO agents. $\delta = 0.05$ in red and $\delta = 0.01$ in green.

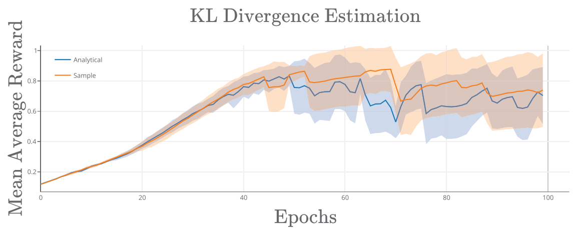

Figure 5: Comparison on the Platform domain for different KL-Divergence estimation methods.

the parameterized action space.

Last, we conducted an experiment using different techniques for estimating the KL divergence in PATRPO. The experiment as illustrated in Figure 5 showed that none of them makes much of a difference in this small domain.

We also tested our algorithm in the HFO domain introduced by (Hausknecht and Stone 2015). In this domain (Figure 6), we controlled an agent to score a goal. For the simplicity, we did not have a goalie. We had three actions in this domain, **dash** with parameters *power* and *direction*, **turn** with parameter *direction* and **kick** with parameters *power* and *direction*. Thus, different actions required different numbers of parameters. For our agents, if we outputed more parameters than we actually needed, we just took the first part of the output and ignored the remainder. This domain had a 59-dimensional state space, which was much larger compared to the platform domain, and thus we trained our agents using larger neural networks and with more samples. Due to time constraints, we only trained our PATRPO agent and PADDPG in this domain for 100 epochs with 50000 steps per epoch. We used three hidden layers with size $\{400, 300, 200\}$ for both PATRPO and PADDPG agents.

Figures 7 shows the result on this domain. As we can see, again, the PATRPO agent achieved stable performance in this domain while PADDPG demonstrated a large variance in its performance. We also note that the performance of the PADDPG algorithm is far worse than in the original paper. One of the possible reasons for this is due to the difference in the neural network sizes. But since our PATRPO agent can achieve stable learning in this domain with a much smaller neural network, this suggests that a large neural network is not necessary in this domain.

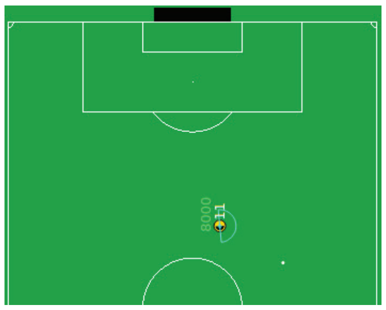

Figure 6: An example of Half Field Offense Domain, with no goalie.

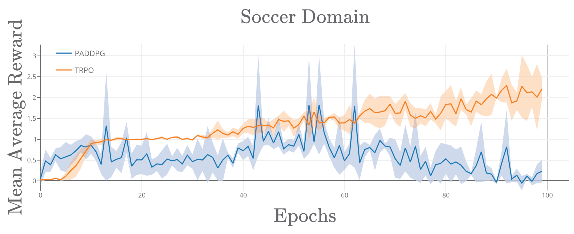

Figure 7: Comparison on Soccer domain for PATRPO and PADDPG agents on three different random seeds.

Conclusion and Future Work

We presented two algorithms for learning effective control in parameterized action space. We demonstrated that our method can learn better policy in these setting compared to PADDPG method. However, we found that learning could be unstable due to the joint-learning between the discrete action policy and parameter policy. An interesting future direction would be to find more stable methods for this domain. We would like to study these methods in the context of more complex domains (in soccer for example) particularly involving more agents.

Acknowledgments

The research in this paper was conducted with the support of research infrastructure developed under NSF grant 1727303.

References

Baram, N.; Anschel, O.; Caspi, I.; and Mannor, S. 2017. End-to-end differentiable adversarial imitation learning. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, 390–399.

Baxter, J., and Bartlett, P. L. 2001. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research* 319–350.

Duan, Y.; Chen, X.; Houthooft, R.; Schulman, J.; and Abbeel, P. 2016. Benchmarking deep reinforcement learning for continuous control. In *Proceedings of The 33rd International Conference on Machine Learning*, 1329–1338.

Hausknecht, M., and Stone, P. 2015. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*.

Heess, N.; Wayne, G.; Silver, D.; Lillicrap, T.; Erez, T.; and Tassa, Y. 2015. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, 2944–2952.

Jang, E.; Gu, S.; and Poole, B. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

Jie, T., and Abbeel, P. 2010. On a connection between importance sampling and the likelihood ratio policy gradient. In *Advances in Neural Information Processing Systems*, 1000–1008.

Kakade, S. M. 2002. A natural policy gradient. In *Advances in neural information processing systems*, 1531–1538.

Kingma, D. P., and Welling, M. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Maddison, C. J.; Mnih, A.; and Teh, Y. W. 2016. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*.

Maddison, C. J.; Tarlow, D.; and Minka, T. 2014. A* sampling. In *Advances in Neural Information Processing Systems 27*, 3086–3094.

Martens, J. 2010. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 735–742.

Masson, W.; Ranchod, P.; and Konidaris, G. 2016. Reinforcement learning with parameterized actions. In *AAAI*, 1934–1940.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mordatch, I., and Abbeel, P. 2017. Emergence of grounded compositional language in multi-agent populations. *CoRR* abs/1703.04908.

Pearlmutter, B. A. 1994. Fast exact multiplication by the hessian. *Neural computation* 6(1):147–160.

Rachelson, E.; Fabiani, P.; and Garcia, F. 2009. Timdppoly: An improved method for solving time-dependent mdps. In *Tools with Artificial Intelligence, 2009. ICTAI'09. 21st International Conference on*, 796–799. IEEE.

Rezende, D. J.; Mohamed, S.; and Wierstra, D. 2014. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 1278–1286.

Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M.; and Moritz, P. 2015. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 1889–1897.

Sharma, S.; Lakshminarayanan, A. S.; and Ravindran, B. 2017. Learning to repeat: Fine grained action repetition for deep reinforcement learning. *arXiv preprint arXiv:1702.06054*.

Silver, D.; Lever, G.; Heess, N.; Degris, T.; Wierstra, D.; and Riedmiller, M. 2014. Deterministic policy gradient algorithms. In *ICML*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Sohn, K.; Lee, H.; and Yan, X. 2015. Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems*, 3483–3491.

Stone, P.; Kuhlmann, G.; Taylor, M. E.; and Liu, Y. 2006. Keepaway soccer: From machine learning testbed to benchmark. In Noda, I.; Jacoff, A.; Bredenfeld, A.; and Takahashi, Y., eds., *RoboCup-2005: Robot Soccer World Cup IX*, volume 4020. Berlin: Springer Verlag. 93–105.

Sutton, R. S.; McAllester, D. A.; Singh, S. P.; and Mansour, Y. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, 1057–1063.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Walker, J.; Doersch, C.; Gupta, A.; and Hebert, M. 2016. An uncertain future: Forecasting from static images using variational autoencoders. In *European Conference on Computer Vision*, 835–851. Springer.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8(3-4):229–256.

Zamani, Z.; Sanner, S.; Fang, C.; et al. 2012. Symbolic dynamic programming for continuous state and action mdps. In *AAAI*.