Comparing Deep Reinforcement Learning and Evolutionary Methods in Continuous Control

Shangtong Zhang
Dept. of Computing Science
University of Alberta
shangtong.zhang@ualberta.ca

Osmar R. Zaiane
Dept. of Computing Science
University of Alberta
osmar.zaiane@ualberta.ca

Abstract

Reinforcement learning and evolutionary strategy are two major approaches in addressing complicated control problems. Both have strong biological basis and there have been recently many advanced techniques in both domains. In this paper, we present a thorough comparison between the state of the art techniques in both domains in complex continuous control tasks. We also formulate the parallelized version of the Proximal Policy Optimization method and the Deep Deterministic Policy Gradient method.

1 Introduction

The biological basis of reinforcement learning (RL) is the behavioral learning process of animals, where an individual learns knowledge by *trial-and-error*. However, the evolutionary strategy (ES) comes from the evolution of the species, where randomness happens at every time and only individuals with positive mutations can survive. Both mechanisms widely exist in nature and both are crucial to the development of the species. Similarly, both RL and ES methods have gained huge success in solving difficult tasks. However, different from nature, where individual learning and specie evolution are combined in a reasonable way, we have not had a unified framework to combine RL and ES until now. As a result, we need to understand the advantages and the weaknesses of both methods in order to select the proper approach when faced with different tasks. Our main contribution is a systematic comparison between the state of the art techniques in both domains in continuous control problems. Control problems can be coarsely divided into continuous control and discrete control. In discrete control like Atari games (Bellemare et al. 2013), the action is usually finite and low dimensional. In continuous control, however, the action is infinite and often lies in the high dimensional vector space, which makes effective exploration much more difficult. Continuous control problems widely exist in the real world, e.g. robot control, and are challenging for both RL and ES methods. Deep learning has recently achieved great success in various domains, so in our comparison, we always use neural networks as function approximators, i.e. deep RL. Moreover, all the algorithms are implemented in a parallelized manner across multiple processes to exploit the advance of modern computation resources. We showcase that in general, ES methods have better exploration while deep RL methods are better at rich dynamics and more efficient.

Although the earliest method of RL is value-based, e.g. the Temporal Difference (TD) method (Sutton 1988), people always turn to policy-based methods when addressing continuous control tasks. The simplest policy gradient method (Sutton et al. 2000) is the REINFORCE algorithm (Williams 1992). Williams (1992) also reduced the variance of the REINFORCE algorithm with a baseline, resulting later on in the actor-critic method. Combining the actor-critic method with the advantage function and asynchronous gradient descent (Recht et al. 2011), Mnih et al. (2016) proposed the Asynchronous Advantage Actor-Critic (A3C) method. For continuous control, a common approach is to represent the action with a Gaussian distribution, resulting in the Continuous A3C (CA3C) method. Instead of

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

a stochastic policy gradient, Silver et al. (2014) proposed the Deterministic Policy Gradient (DPG) and Lillicrap et al. (2015) integrated DPG with neural networks, resulting in the Deep Deterministic Policy Gradient (DDPG) algorithm, which is one of the most popular algorithms in continuous control. Trust Region Policy Optimization (TRPO, Schulman et al. 2015) is another kind of policy gradient method, where the update of the policy is more concisely controlled. The main drawback of TRPO is its computation complexity. To address this issue, Schulman et al. (2017) proposed the Proximal Policy Optimization (PPO), which truncates the policy ratio in TRPO and discards the constraint term. Our other contribution in this paper is that we formulate the parallelized version of PPO (Parallelized PPO, aka P3O) and DDPG (Distributed DDPG, aka D3PG) and showcase their power in continuous control tasks. In conclusion, we consider in our comparison CA3C, P3O and D3PG as the representatives of the state-of-the-art deep RL methods.

The Covariance Matrix Adaptation Evolution Strategy (CMAES, Hansen and Ostermeier 1996) is one of the most well-known evolutionary methods, after which, Stanley and Miikkulainen (2002) proposed the NeuroEvolution of Augmenting Topologies (NEAT) to effectively evolve both the architecture and the weight of the neural network. HyperNEAT (Stanley et al. 2009) is a successful extension of NEAT to help discover geometry information. However, HyperNEAT was mainly designed for image input, so it is not included in our comparison. More recently, Salimans et al. (2017) proposed a fairly simple but effective version of Natural Evolution Strategy (NES), which is highly scalable. In our comparison, we consider CMAES, NEAT and NES as the representatives of the state of art evolutionary methods.

We conducted our comparison in terms of both wall time and environment time steps, i.e. the amount of interactions with the environment, to gain a better empirical understanding of the running speed and data efficiency of all the algorithms.

2 Related Work

Taylor et al. (2006) compared NEAT with SARSA (Rummery and Niranjan 1994, Singh and Sutton 1996), but their work was limited on the football game Keepaway (Stone et al. 2005), which is a discrete control task. This comparison was extended later on by Whiteson et al. (2010), where new criterion was involved but the comparison was still limited to primitive RL methods where deep neural networks were not included. Duan et al. (2016) compared various deep RL methods with some evolutionary methods. However they did not include the NEAT paradigm and there has been exciting progress in both RL and ES, e.g. Schulman et al. 2017 and Salimans et al. 2017. Therefore these comparisons are not representative enough of the current state of the art. Moreover our comparison is conducted mainly on different tasks and measurements and focused on parallelized implementations.

3 Compared Algorithms

Considering a *Markov Decision Process* with a state set $\mathcal{S}$, an action set $\mathcal{A}$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and a transition function $p : \mathcal{S} \times \mathcal{A} \to (\mathcal{S} \to [0, 1])$, the goal of the agent is to learn an optimal policy to maximize the expected discounted return $G_t = \sum_{k=0}^{T-1} \gamma^k R_{t+k+1}$ at every time step, where T denotes the terminal time and γ is the discount factor. In the continuous control domain, an action $\boldsymbol{a}$ is usually a vector, i.e. $\boldsymbol{a} \in \mathbb{R}^d$, where d is the dimension of the action. In some methods, e.g. CA3C and P3O, the policy π is stochastic: $\mathcal{S} \to \mathcal{P}(\mathcal{A})$, where $P(\mathcal{A})$ is a probability distribution over $\mathcal{A}$. While in other methods, e.g. D3PG, CMAES, NEAT and NES, the policy μ is deterministic: $\mathcal{S} \to \mathcal{A}$. Here we use π and μ to represent a stochastic and a deterministic policy respectively, and we assume the policy is parameterized by θ in the following sections, which is a neural network.

3.1 Deep Reinforcement Learning Methods

In deep RL methods, the structure of the network is predefined, θ simply represents the numeric weights of the neural network.

3.1.1 Continuous A3C

The goal of the actor-critic method is to maximize the value of the initial state $v_\pi(s_0)$, where $v_\pi(s)$ is the value of the state s under policy π, i.e. $v_\pi(s) \doteq \mathbb{E}_\pi[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \mid S_t = s]$. Let $q_\pi(s, \boldsymbol{a}) \doteq \mathbb{E}_\pi[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \mid S_t = s, A_t = \boldsymbol{a}]$ denote the value of the state action pair $(s, \boldsymbol{a})$, we have $v_\pi(s) = \sum_{\boldsymbol{a} \in \mathcal{A}} \pi(\boldsymbol{a}|s) q_\pi(s, \boldsymbol{a})$. Our objective, therefore, is to maximize $J(\theta) = v_\pi(s_0)$. According to the policy gradient theorem (Sutton et al. 2000), we have

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta log \pi(A_t|S_t, \theta)(R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(S_t))]$$

The value function v_π is updated in a semi-gradient TD fashion (Sutton and Barto 1998). In continuous action domain, the policy π is often parameterized in the form of the probability density function of the multivariate Gaussian distribution, i.e.

$$\pi(\boldsymbol{a}|s, \theta) \doteq \frac{exp(-\frac{1}{2}(\boldsymbol{a} - \boldsymbol{\mu}(s, \theta))^T \boldsymbol{\Sigma}(s, \theta)^{-1} (\boldsymbol{a} - \boldsymbol{\mu}(s, \theta)))}{\sqrt{|2\pi \boldsymbol{\Sigma}(s, \theta)|}}$$

In practice, setting the covariance matrix $\boldsymbol{\Sigma}(s, \theta) \equiv \boldsymbol{I}$ is a good choice to increase stability and simplify the parameterization. Beyond the standard advantage actor-critic method, Mnih et al. (2016) introduced asynchronous workers to gain uncorrelated data, speed up learning and reduce variance, where multiple workers were distributed across processes and every worker interacted with the environment separately to collect data. The computation of the gradient was also in a non-centered manner.

3.1.2 Parallelized PPO

Schulman et al. (2015) introduced an iterative procedure to monotonically improve policies named Trust Region Policy Optimization, which aims to maximize an objective function $L(\theta)$ within a trust region, i.e.

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & L(\theta) = \mathbb{E}[\frac{\pi_\theta(\boldsymbol{a}_t|s_t)}{\pi_{\theta_{old}}(\boldsymbol{a}_t|s_t)} A_t] \\ \text{subject to} \quad & \mathbb{E}[D_{KL}(\pi_\theta(\cdot\,|s_t), \pi_{\theta_{old}}(\cdot\,|s_t))] \leq \delta \end{aligned}$$

where A_t is the advantage function, D_{KL} is the KL-divergence and δ is some threshold. In practice, solving the corresponding unconstrained optimization problem with a penalty term is more efficient, i.e.

$$\underset{\theta}{\text{maximize}} \quad L^{KL}(\theta) = L(\theta) - \beta \mathbb{E}[D_{KL}(\pi_\theta(\cdot\,|s_t), \pi_{\theta_{old}}(\cdot\,|s_t))]$$

for some coefficient β. Furthermore, Schulman et al. (2017) proposed the clipped objective function, resulting in the PPO algorithm. With $r_t(\theta) \doteq \frac{\pi_\theta(\boldsymbol{a}_t|s_t)}{\pi_{\theta_{old}}(\boldsymbol{a}_t|s_t)}$, the objective function of PPO is

$$L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta) A_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t)]$$

where ϵ is a hyperparameter, e.g. $\epsilon = 0.2$. In practice, Schulman et al. (2017) designed a truncated generalized advantage function, i.e.

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1} \delta_{T-1}$$

where T is the length of the trajectory, δ_t is the TD error, i.e. $\delta_t = r_t + v_\pi(s_{t+1}) - v_\pi(s_t)$, and λ is a hyperparameter, e.g. $\lambda = 0.97$. The value function is updated in a semi-gradient TD manner.

We parallelize the PPO mainly following the training protocol of A3C and the Distributed PPO method (DPPO, Heess et al. 2017). In P3O, we distribute multiple workers across processes like A3C. Following DPPO, each worker in P3O also has its own experience replay buffer to store its transitions. The optimization that happens in the worker is solely based on this replay buffer. Different from DPPO, where the objective function was $L^{KL}(\theta)$, we use $L^{CLIP}(\theta)$ in P3O, as it was reported to be able to outperform $L^{KL}(\theta)$ (Schulman et al. 2017). Heess et al. (2017) reported that synchronized gradient update can actually outperform asynchronous update in DPPO. So in P3O we also used the synchronized gradient update. However our synchronization protocol is quite simple. We use A3C-style update with an extra shared lock for synchronization and all the gradients are never dropped, while DPPO adopted a quite complex protocol and some gradients may get dropped. Moreover, a worker in P3O only does one batch update based on all the transitions in the replay buffer as we find this can increase the stability. In DPPO, however, each worker often did multiple mini-batch updates. The full algorithm of P3O is elaborated in **Algorithm 1**.

Algorithm 1: A worker of P3O

Input:
shared global parameters θ^-
shared global lock $\mathcal{L}$
number of rollouts N

Initialize worker-specific parameter θ
Initialize environment
while *stop criterion not satisfied* **do**
  Initialize replay buffer $\mathcal{M}$
  Sync $\theta = \theta^-$
  for $i = 1$ *to* N **do**
    Sample a *rollout* with the environment
    Compute advantages adv and returns g following PPO method
    Store states s, actions a, adv, g into replay buffer $\mathcal{M}$
  end
  Sync $\theta = \theta^-$
  Compute gradients $d\theta$ w.r.t. θ following PPO method using all the trajectories in $\mathcal{M}$
  Acquire shared lock $\mathcal{L}$
  Update global parameter θ^- with $d\theta$
  Release shared lock $\mathcal{L}$
end

3.2 Distributed DDPG

Similar to CA3C, the goal of DDPG is also to maximize $J(\theta) = v_\mu(s_0)$. According to the deterministic policy gradient theorem (Silver et al. 2014), we have

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mu'}[\nabla_a q_\mu(s, \boldsymbol{a})|_{s=s_t, \boldsymbol{a}=\mu(s_t|\theta)} \nabla_\theta \mu(s|\theta)|_{s=s_t}]$$

where μ' is the behavior policy. The behavior policy usually combines the target policy with some noise, i.e. $\mu'(s) = \mu(s) + \mathcal{N}$, where $\mathcal{N}$ is some random noise. Following Lillicrap et al. (2015), we use an Ornstein-Uhlenbeck process (Uhlenbeck and Ornstein 1930) as the noise. To stabilize the learning process, Lillicrap et al. (2015) introduced experience replay (Lin 1993) and target network (Mnih et al. 2015), resulting in the DDPG algorithm.

We formulate the distributed version of DDPG in analogy to P3O, except the experience replay buffer is shared among different workers. Each worker interacts with the environment, and acquired transitions are added to a shared replay buffer. At every time step, a worker will sample a batch of transitions from the shared replay buffer and compute the gradients following the DDPG algorithm. The update is synchronized and the target network is shared. The full algorithm is elaborated in **Algorithm 2**.

3.3 Evolutionary Methods

Evolutionary methods solve the control problem by evolving the control policy. Different evolutionary methods adopt different mechanisms to generate *individuals* in a *generation*, where individuals with better performance are selected to produce the next generation.

3.3.1 CMAES

In CMAES, the structure of the neural network is predefined, θ here only represents the weights of the network. Each generation $\mathcal{G}$ consists of many candidate parameters, i.e. $\mathcal{G} = \{\theta_1, \ldots, \theta_n\}$. Each θ_i is sampled from a multivariate Gaussian distribution in the following fashion:

$$\theta_i \sim \boldsymbol{\mu} + \sigma\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$$

where $\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ is a multivariate Gaussian distribution with zero mean and covariance matrix $\boldsymbol{\Sigma}$, $\boldsymbol{\mu}$ is the mean value of the search distribution and σ is the overall standard deviation (aka step size). All the candidates in $\mathcal{G}$ are evaluated by the environment. According to the performance, a certain selection mechanism will select some candidates to update the sample distribution, i.e. update $\boldsymbol{\mu}$, σ and $\boldsymbol{\Sigma}$, and the next generation is sampled from the updated distribution.

Algorithm 2: A worker of D3PG

Input:
shared global parameters θ^-
shared global target network θ^-_{target}
shared global lock $\mathcal{L}$
shared global replay buffer $\mathcal{M}$ number of rollouts N

Initialize worker-specific parameter θ
Initialize environment
while *stop criterion not satisfied* **do**
  Sync $\theta = \theta^-$
  Interact with the environment, get the one step transition e
  Add e to the shared replay buffer $\mathcal{M}$
  Sample a batch of transitions $\mathcal{E}$ from $\mathcal{M}$
  Compute gradients $d\theta$ w.r.t. θ with $\mathcal{E}$ following the DDPG algorithm
  Acquire shared lock $\mathcal{L}$
  Update global parameters θ^- with $d\theta$
  Release shared lock $\mathcal{L}$
  Update shared target network θ^-_{target} with θ in a soft-update manner
end

3.3.2 NEAT

NEAT has shown great success in many difficult control tasks and is the basis of many modern evolutionary methods like HyperNEAT. The basic idea of NEAT is to evolve both the structure and the weights of the network. Thus θ now represents both the weights and the structure of the network. At every generation, NEAT selects several best individuals to crossover. And their descendants, with various mutations, will form the next generation. NEAT introduced a *genetic encoding* to represent the network efficiently and used the *historical markings* to perform a crossover among networks in a reasonable way, which helped avoid expensive topological analysis. Furthermore, NEAT incorporates an *innovation number* to protect innovations, which allowed the network to evolve from the simplest structure.

3.3.3 NES

In NES, the structure of the network is given, θ here only represents the weights. NES assumes the population of θ is drawn from a distribution $p_\phi(\theta)$, parameterized by ϕ, and aims to maximize the expected fitness $J(\phi) = \mathbb{E}_{\theta\sim p_\phi} F(\theta)$, where F is the fitness function to evaluate an individual θ. Salimans et al. (2017) instantiated the population distribution p_ϕ as a multivariate Gaussian distribution, with mean ϕ and a fixed covariance $\sigma^2 I$. Thus J can be rewritten in terms of θ directly, i.e. $J(\theta) = \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\theta + \sigma\epsilon)$. In a fashion similar to policy gradient, we have

$$\nabla_\theta J(\theta) = \frac{1}{\sigma}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\theta + \sigma\epsilon)\epsilon$$

In practice, at every generation, the population $\{\epsilon_1, \ldots, \epsilon_n\}$ is drawn from $\mathcal{N}(0, I)$. The update rule for θ is, therefore,

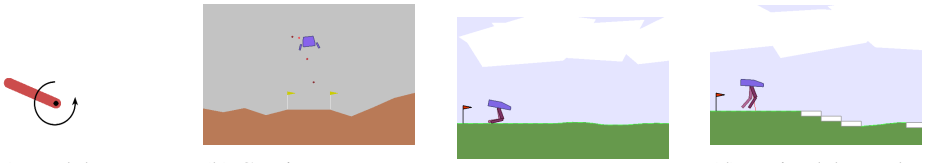
$$\theta \leftarrow \theta + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F(\theta + \sigma\epsilon_i)\epsilon_i$$

where α is the step size and n is the population size.

4 Experiments

4.1 Testbeds

The *Pendulum* task is a standard toy problem in the continuous control domain, where the state space is $\mathbb{R}^3$ and the action space is $[-2, 2]$. It is usually a good start point to verify an implementation.



(a) Pendulum (b) Continuous Lunar Lander (c) Bipedal Walker (d) Bipedal Waler Hardcore

Figure 1: Four tasks. The background of the *Continuous Lunar Lander* was edited for better display.

Recently *MuJoCo* (Todorov et al. 2012) has become popular as a benchmark in the continuous control domain and there have been many empirical results on *MuJoCo*. Although not systematic, the community can still grasp some intuition. In our comparison, we considered the *Box2D* (Catto 2011) environment, especially the *Continuous Lunar Lander* task, the *Bipedal Walker* task and the *Bipedal Waler Hardcore* task. *Continuous Lunar Lander* is a typical task that needs careful exploration. Negative rewards are continuously given during the landing so the algorithm can easily get trapped in a local minima, where it avoids negative rewards by doing nothing after certain steps until timeout. Its state space is $\mathbb{R}^8$ and action space is $[-1, 1]^2$. *Bipedal Walker* is a typical task that includes rich dynamics and *Bipedal Walker Hardcore* increases the difficulty for the agent to learn how to run by putting some obstacles in the path. They have the same state space $\mathbb{R}^{24}$ and action space $[-1, 1]^4$. The three *Box2D* tasks have comparable complexity with *MuJoCo* and all the four tasks are free and available in OpenAI Gym (Brockman et al. 2016). Some screenshots of the tasks are shown in **Figure 1**.

4.2 Performance Metrics

For evolutionary methods, we performed 10 test episodes for the best individual in each generation and averaged the test results to represent the performance at those time steps and wall time. For CA3C and P3O, an additional test process run deterministic test episodes continuously. We performed 10 independent runs for each algorithm and each task without any fixed random seed. Our training was based on episodes (generations), while the length (environment steps and wall time) of episodes is different. So the statistics are not aligned across different runs. We used linear interpolation to average statistics across different runs.

4.3 Policy Representation

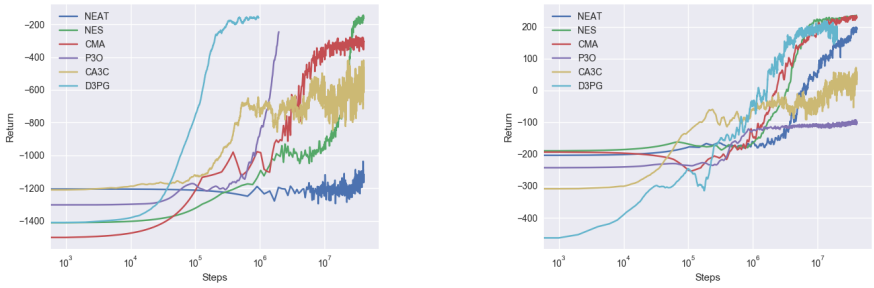
For P3O, CA3C and D3PG, following Schulman et al. (2017), we used two two-hidden-layer neural networks to parameterize the policy function and the value function. Each hidden layer had 64 hidden units with the hyperbolic tangent activation function and the two networks did not have shared layers. The output units were linear to produce the mean of the Gaussian distribution (for P3O and CA3C) and the action (for D3PG). Moreover, we used the identity covariance matrix for P3O and CA3C, thus the entropy penalty was simply set to zero. For CMAES and NES, we used a single *ReLU* (Nair and Hinton 2010) hidden layer network with 200 hidden units. The output units were also linear. Our preliminary experiments showed that the single hidden layer network achieved the same performance level as the architecture in deep RL methods with a shorter training time. For NEAT, the initial architecture had only one hidden unit, because NEAT allows the evolution to start from the simplest architecture.

4.4 Wall Time

We implemented a multi-process version of CMAES and NEAT based on *Lib CMA* [1] and *NEAT-Python* [2], while all other algorithms were implemented from scratch in *PyTorch* for a like-to-like comparison. For deep RL algorithms, the most straightforward parallelization method is to only parallelize data collection, while the computation of the gradients remains centralized. However this approach needs careful balance design between data collection and gradients computation. Moreover, it is unfriendly to data efficiency. So we adopted algorithm dependent parallelization rather than this

[1] `https://github.com/CMA-ES/pycma`

[2] `https://github.com/CodeReclaimers/neat-python`



(a) Pendulum

(b) Continuous Lunar Lander

(c) Bipedal Walker

(d) Bipedal Waler Hardcore

Figure 2: Performance in terms of environment steps. X-axis is in *log* scale.

uniform solution. We run all the algorithms in the same server [3] and GPU was not involved. Although the relative wall time is highly dependent on the implementation, the comparison we did here can still give us some basic intuition. We made all the implementations publicly available [4] [5] for further study.

4.5 Hyperparameter Tuning

For NEAT and CMAES, we used the default parameters in the package. For D3PG, we use hyperparameters reported by Lillicrap et al. 2015. For all other algorithms, we tuned the parameters empirically, some were based on the parameters reported in OpenAI baselines (Dhariwal et al. 2017). We normalized the state and the reward by running statistics, all the statistics were shared and updated across parallelized workers and all the algorithms were parallelized with 8 processes.

4.6 Results

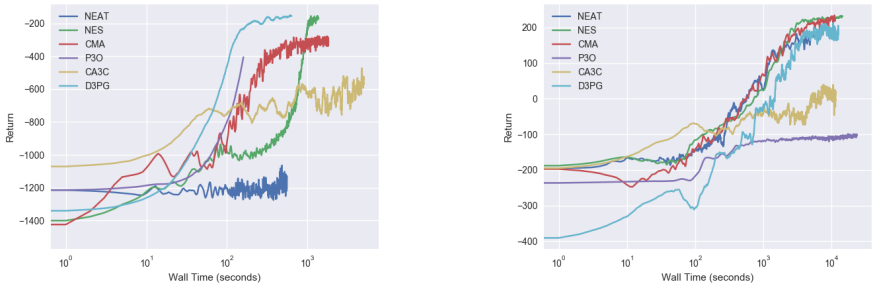
We reported the performance in terms of both the environment steps and wall time in **Figure 2** and **Figure 3**. All the curves were averaged over 10 independent runs. As deep RL methods had larger variance than evolutionary methods, all the curves of CA3C, P3O and D3PG were smoothed by a sliding window of size 50. To make the difference among the algorithms clearer, we transformed the x-axis into the *log* scale.

We can hardly say which algorithm is the best, as no algorithm can consistently outperform others across all the tasks. NEAT achieved a good performance level in the three *Box2D* tasks, but failed the simplest *Pendulum* task. One interesting observation is that all the evolutionary methods solved the *Continuous Lunar Lander* task, but most deep RL methods appeared to get trapped in some local minima, which denotes that evolutionary methods are better at exploration than the deep RL methods. However when it comes to the two *Walker* tasks, where rich dynamics are involved, most deep RL methods worked better and the evolutionary methods appeared to get trapped in some

[3] Intel® Xeon® CPU E5-2620 v4

[4] `https://github.com/ShangtongZhang/DeepRL`

[5] `https://github.com/ShangtongZhang/DistributedES`



(a) Pendulum

(b) Continuous Lunar Lander

(c) Bipedal Walker

(d) Bipedal Waler Hardcore

Figure 3: Performance in terms of wall time. X-axis is in *log* scale.

local minima, which denotes that the deep RL methods can handle rich dynamics better than the evolutionary methods. In terms of data efficiency, the deep RL methods can solve tasks with much less interaction with the environment than the evolutionary methods, and the wall time is usually shorter, at worst comparable. In conclusion, the performance of the parallelized algorithms appears to be task dependent. But in general, NEAT, D3PG and P3O are safe choices.

5 Discussion

We compared parallelized deep RL and evolutionary methods systematically in different kinds of tasks. There was not a consistent best algorithm and the performance seemed task dependent. In general, deep RL methods can deal with rich dynamics better. But when we need careful exploration, evolutionary methods seem to be a better choice. Due to the lack of exploration, deep RL methods appear to have larger variance among different runs than evolutionary methods. Deep RL methods also outperform evolutionary methods in both data efficiency and wall time. However, evolutionary methods are fairly easy to parallelize, while parallelizing deep RL methods needs careful implementation.

Although our testbeds include several representative tasks, our comparison is still limited in tasks with low dimensional vector state space. With the popularity of the Atari games, image input has become a new norm in the discrete action domain. Our future work will involve continuous control tasks with image input.

6 Acknowledgment

The authors thank G. Zacharias Holland and Yi Wan for their thoughtful comments.

References

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Catto, E. (2011). Box2d: A 2d physics engine for games.

Dhariwal, P., Hesse, C., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2017). Openai baselines. `https://github.com/openai/baselines`.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338.

Hansen, N. and Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 312–317. IEEE.

Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, A., Riedmiller, M., et al. (2017). Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*.

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*.

Lin, L.-j. (1993). Reinforcement Learning for Robots Using Neural Networks. *Report, CMU*, pages 1–155.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701.

Rummery, G. A. and Niranjan, M. (1994). *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering.

Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

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)*, pages 1889–1897.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Recent Advances in Reinforcement Learning*, pages 123–158.

Stanley, K. O., D'Ambrosio, D., and Gauci, J. (2009). A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Stone, P., Sutton, R. S., and Kuhlmann, G. (2005). Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

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*, pages 1057–1063.

Taylor, M. E., Whiteson, S., and Stone, P. (2006). Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1321–1328. ACM.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE.

Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Physical review*, 36(5):823.

Whiteson, S., Taylor, M. E., and Stone, P. (2010). Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 21(1):1–35.

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