Distributed Reinforcement Learning with States Feature Encoding and States Stacking in Continuous Action Space

Tianqi Xu[1], Dianxi Shi[1,2(✉)], Zhiyuan Wang[1], Xucan Chen[1(✉)], and Yaowen Zhang[1]

[1] Artificial Intelligence Research Center (AIRC), National Innovation Institute of Defense Technology (NIIDT), Beijing 100071, China
dxshi@nudt.edu.cn, xcchen18@139.com
[2] Tianjin Artificial Intelligence Innovation Center (TAIIC), Tianjin 300457, China

Abstract. The practical application of reinforcement learning agents is often bottlenecked by the duration of training time. To accelerate training, practitioners often turn to distributed reinforcement learning architectures to parallelize and accelerate the training process. This work, we utilize the distributed reinforcement learning architecture to deal with continuous control tasks. The Importance Weighted Actor-Learner Architectures (IMPALA) decouples the acting and learning process to reduce queuing time. IMPALA attains higher scores on the new DMLab-30 set and the Atari-57 set because of its high performance, good scalability, and high efficiency. We extend IMPALA on the continuous control tasks with three changes. We encoder states into low dimensional data to establish an action distribution function that the agents have the ability to exploit and explore. A queue buffer is used to store a mini-batch data and discard them after training. In order to make the agent take appropriate action in the continuous control environment, we stack the past three steps states that attempt to make the robot moves smoothly. Finally, experiments are carried out on Mujoco tasks. The results show that our work is better than other distributed reinforcement learning algorithms.

Keywords: Distributed reinforcement learning · Importance sampling · Continuous control task · Scalable agent

1 Introduction

Deep reinforcement learning methods have recently achieved great success in a wide variety of domains through trial and error learning [3,9,11,13,14,19].

This work was supported in part by the Key Program of Tianjin Science and Technology Development Plan under Grant No. 18ZXZNGX00120 and in part by the China Postdoctoral Science Foundation under Grant No. 2018M643900.

H. Gao et al. (Eds.): CollaborateCom 2020, LNICST 349, pp. 340–353, 2021.
https://doi.org/10.1007/978-3-030-67537-0_21

While, one of the most important factors restricting the effect of reinforcement learning is the low efficiency of reinforcement learning sampling from the environment, especially in the complex terrain environment, simulation and mechanical control environment. Taking the StarCraft II environment [18] as an example, it consumes 3 to 4 s to start a StarCraft environment or reset a StarCraft environment. The maximum acceleration is about 16 times faster without turning on StarCraft graphics rendering. This works out to about five to ten seconds for training an episode, whereas training an intensive learning model would require tens of thousands or even hundreds of thousands of episodes, which is unacceptable.

In reinforcement learning scenarios [15], there is no static tag information available in advance, usually it's necessary to introduce the actual target system to obtain feedback information, which can be used to judge losses and gains, and then complete the closed-loop feedback of the entire training process [7]. A typical step is to collect feedback information and returns by observing the state of a specific target system. These information are used to adjust the parameters and train the model. According to the new training results, the output can be used to adjust the behavior of the target system and output to the target system. Then it affects the state change of the target system and completes the closed loop. In this way, the ultimate goal is to maximize total returns.

This work, we consider encoder the environment states, outputting the distribution of potential variables through a generation model [8], assuming that the distribution of potential variables is a Gaussian distribution, that the distribution of actions under the corresponding environmental state and updating the model through the probability and the return value of the action. In the environment, only through the data of the states, it can be fully quantified as the corresponding potential variables, so as to select the corresponding action a of the current states. By generating the distribution of potential variables, we can deal with high-dimensional input and continuous action space. We utilize a Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures [6]. The main task of the learner is updating the parameters of each neural network by obtaining the trajectory generated by actors to calculate stochastic gradient descent. Because the process of neural network training can be parallel, the learner uses GPU to calculate. The actor obtains the latest neural network parameters from the learner on a regular basis, and each actor starts up several simulation environments to use the latest policy synchronized by the learner and transmits the acquired trajectories back to the learner to update its neural network parameter. In order to make the agent take appropriate action in the continuous control environment, we stack the past three steps states that attempt to make the robot moves smoothly.

2 Related Work

DeepMind proposed asynchronous methods for deep reinforcement learning (A3C) [10]. It greatly improves the sampling efficiency by learning strategies simultaneously and asynchronously updating parameters through multiple

actors. However, the A3C is generally inefficient in GPU utilization. The average server will have one or several GPUs, but the number of CPUs may reach dozens or even hundreds. Since each distributed Actor in the A3C algorithm needs to learn parameters, and it is obviously unrealistic to equip each actor with a GPU device, the A3C algorithm can only use CPU for training.

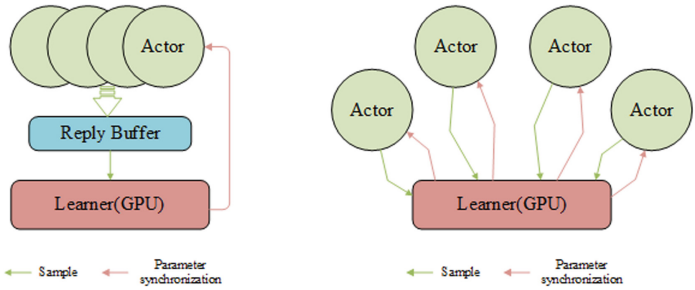
To take full advantage of the GPU, GA3C [1] proposed a hybrid CPU/GPU version of A3C. During training process of A3C, we need to copy networks for each parallel agent to collect samples and calculate the cumulative gradient. When the number of parallel agents is large, it consumes memory. GA3C algorithm module is mainly divided into agent, predictor, and trainer. Like A3C, the agent collects trajectories but does not need to copy a model separately. It only needs to add the current state as a request to the prediction queue before each action is selected. After n steps of the action are executed, the total returns of each step is calculated backward. The $\mathrm{n}(s_t, a_t, R, s_{t+1})$ obtained are added to the training queue. Predictor will queue the request samples in the prediction queue as mini-batch and fill them into the GPU's network model, and return the actions predicted by the model to their respective agents. In order to reduce the delay, multi-threaded parallel multiple predictors can be used. Trainer uses the sample of training queue as mini-batch to fill in the network model of GPU to train and update the model. Similarly, in order to reduce the delay, multiple trainers can be used in parallel with multiple threads.

On the basis of the A3C algorithm, the Importance Weighted Actor-Learner Architectures (IMPALA) [6], a framework of large-scale reinforcement learning and training, which has high performance, good scalability, and high efficiency. Under the framework of large-scale computing, each Actor asynchronously samples, leading to the dislocation of sampled data and the implemented policy, instead of the complete on-policy sampling. In this case, the paper uses V-trace technology to eliminate the error of asynchronously sampled data.

However, these algorithms are limited to problems with a finite number of discrete actions. In control tasks, commonly seen in the robotics domain, continuous action spaces are the norm. For algorithms such as Deep Q Network [11] and A3C, the policy is only implicitly defined in terms of its value function, with actions selected by maximizing this function. In the continuous control domain this would require either a costly optimization step or discretization of the action space. While discretization is perhaps the most straightforward solution, this can prove a particularly poor approximation in high- dimensional settings or those that require finer grained control. Instead, a more principled approach is to parameterize the policy explicitly and directly optimize the long term value of following this policy.

3 Background

The problem of discounted infinite-horizon RL in Markov Decision Processes (MDP) is tough to solve. See [12] where the goal is to find a policy π that maximizes the expected sum of future discounted rewards: $V^{\pi}(x) = \mathbb{E}_{\pi}[\sum_{t \geq 0} \gamma^t r_t]$,



(a) D4PG (b) IMPALA

Fig. 1. Architecture schemes for D4PG and IMPALA. D4PG aggregates actors batch into a large training batch and the learner performs mini-batch SGD. IMPALA actors asynchronously generate data.

the $\gamma \in [0, 1)$ is the discount factor, $r_t = r(x_t, a_t)$ is the reward at time t, x_t is the state at time t (initial state $x_0 = x$), $a_t \sim \pi(\cdot|x_t)$ is the action generated by following some policy π. The goal of off-policy reinforcement learning algorithm is to learn the value function V^π of the target policy π using the trajectories generated by the behavior policy μ, which is usually different from the target policy.

3.1 Importance Weighted Actor-Learner Architectures

IMPALA decouples acting and learning by having the actor threads send actions Fig. 1, observations, and values while the learner thread computes and applies the gradients from a queue of actors experience. This will maximize GPU utilization and allow for increased sample throughput, resulting in high training speed on simpler environments such as Pong. With the increase of the number of learners, the worker policy begins to deviate from the learner policy, resulting in an obsolete policy gradient. The purpose of IMPALA is to get a better estimation of the current state value function $v_{(x)}$ following (1) based on the trajectories $(x_t, a_t, r_t)_{t=s}^{t=s+n}$ and the current state value function network.

$$v_s = V^\pi(x) + \sum_{t=s}^{s+n+1} \gamma^{t-s} \Big(\prod_{i=s}^{t-1} c_i\Big) \rho_t (r_t + \gamma V^\pi(x_t + 1) - V(x_t)) \tag{1}$$

Where V^π is the value network, π is the target network of learner thread, μ is the behavior network of actor thread. ρ_t and c_i are defined as follows: $\rho_t = \min(\overline{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$, $c_i = \min(\overline{c}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$. They are truncated importance sampling (IS) weights.

ρ_t and c_i, as the the (truncated) IS weights, play different roles. ρ_t defines the fixed point of this update rule. Every time, updating state value function $V_\theta(x)$ toward $v(x)$, the state value function converged to is a value function between V^π and V^μ called $V_{\pi_{\overline{\rho}}}$. $\pi_{\overline{\rho}}$ is defined as

$$\pi_{\overline{\rho}}(a|x) = \frac{min(\overline{\rho}\mu(a|x), \pi(a|x))}{\sum_{b \in A} min(\overline{\rho}\mu(b|x), \pi(b|x))} \tag{2}$$

Update the target policy π by sampling under behavior policy μ. But with the limit of the truncated weights added, it cannot exceed $\overline{\rho}$ and $\overline{c}$. Generally, the two weight can be set to ones.

$\pi_{\overline{\rho}}$ is a policy between π and μ. If $\overline{\rho} = \infty$, then $\pi_{\overline{\rho}}$ becomes policy π, if $\overline{\rho} \to 0$ (close to 0), then $\pi_{\overline{\rho}}$ becomes policy μ. (so the larger $\pi_{\overline{\rho}}$ is, the smaller the deviation of off-policy learning is, and the larger the corresponding variance is.

The larger the difference between π and μ, the more obvious the off-policy is, the greater the variance of the results is. The truncation method is used to control the variance. $\overline{\rho}$ affects what value function to converge to, while $\overline{c}$ affects the speed of convergence to this value function.

It should be noted that v_s in the case of on-policy updating (i.e. $\pi = \mu$), and let $c_i = 1$, and $\rho_t = 1$, will become the following formula

$$v_s = V(x) + \sum_{t=s}^{s+n+1} \gamma^{t-s}(r_t + \gamma V(x_t + 1) - V(x_t)) = \sum_{t=s}^{s+n+1} \gamma^{t-s} r_t + \gamma^n V(x_{s+n}) \tag{3}$$

V-trace targets can be calculated in an iterative way (4).

$$v_s - V(x) = \delta_t V + \gamma c_s (v_{s+1} - V(x_{s+1})) \tag{4}$$

3.2 Distributed Distributional DDPG [2]

The method adopted by Distributed Distributional DDPG (D4PG) is from the improvement of DDPG (Deep Deterministic Policy Gradient) algorithm, and contains some enhancement. These extensions include a distributional critic update, the use of distributed parallel actors, N-step returns, and prioritization of the experience replay.

First, and perhaps most crucially, it considers the inclusion of a distributional critic. The return of distributional update is a random variable Z_π, such that $Q_\pi(s, a) = \mathbb{E}Z_\pi(s, a)$. The distributional Bellman operator can be defined as

$$(\mathcal{T}_\pi Z)(s, a) = r(s, a) + \gamma \mathbb{E}[Z(s', \pi(s'))|s, a] \tag{5}$$

In order to use this function within the context of the actor-critic architecture introduced above, They parameterize this distribution and define a loss and write the loss as

$$L(\omega) = \mathbb{E}_\rho[d((T)_{\pi_{\theta'}} Z_{\omega'}(s, a), Z_\omega(s, a)] \tag{6}$$

Next, they utilize a modification to the DDPG update which utilizes N-step returns when estimating the TD error. Finally, they modify the standard training procedure in order to distribute the process of gathering experience.

4 IMPASS Algorithm

Importance Weighted Actor-Learner Architectures with States Stacking (IMPASS algorithm) extends IMPALA algorithm on the continuous control tasks with three changes. We encoder observations into low dimensional data to establish an action distribution function so that the agent has the ability to exploit and explore. A queue buffer is used to store a mini-batch data and discard them after training. In order to make the agent take appropriate action in the continuous control environment, we stack three steps past observations that attempt to make the robot move smoothly. Finally, experiments are carried out on Mujoco tasks. The results show that our work is better than the existing distributed reinforcement learning algorithms.

Algorithm 1. IMPASS

Input: batch size M, number of actors N, sample queue length K, learning rate schedule r, clipped important sample weight $\overline{\rho}$ and $\overline{c}$

Initialize network weights θ, ω at random
Initialize batch buffer m of size M
Launch K actors and replicate network weights θ, ω to each actor
Extract data from K to m
while True **do**
  for $t = 1 \rightarrow T$ **do**
    Get T trajectories $(S_{i:i+T}, a_{i:i+T}, \mu_{i:i+T}, \sigma_{i:i+T}, r_{i:i+T})$ from m
    Establish action distribution with μ and σ and calculate action probability
    Calculate probability of action a with target network
    Calculate the value function loss of critic network
$$\mathbb{E}[\tfrac{\pi(a_t|S_t)}{\mu(a_t|S_t)} \nabla \log \pi Q^{\mu}(S_t, a_t)]$$
    Calculate the policy gradient loss of actor network
$$\nabla_{\omega} \log \pi_{\omega}(a_t|S_t)(r_t + \gamma v_{t+1} - V_{\theta}(S_t))$$
    Calculate the entropy loss
$$-\nabla_{\omega} \sum_a \pi_{\omega}(a|S_t) \log \pi_{\omega}(a|S_t)$$
    Minimize the total loss
    Update network parameters $\theta \leftarrow \theta'$, $\omega \leftarrow \omega'$
    if $t = 0 \bmod t_{params}$ **then**
      Replicate network weights to the actors
    end if
  end for
end while

Actor-i starts synchronously with the Learner

repeat
  Samples action a from action distribution function
  Executes action a, gains reward r
  Transfers (S, a, μ, σ, r) to K
until learner finishes

4.1 States Feature Encoding

We consider to establish the action distribution function from the observations generated by actors, so adopt the method of feature encoding to get the potential variable distribution of input data. The probability distribution of observations is learned by a neural network, referring to the working principle of the Variational Auto-Encoder [8]. This kind of network is called auto-encoder. The probability distribution obtained is the action distribution of the agent in the current state. We assume that the probability distribution is Gaussian distribution.

In practical problems, the action space of many tasks is continuous, such as robot control. If there is a ready-made position controller, the expected output of the policy can be a position, which is naturally continuous. This is also applicable to speed control. In the continuous action space, it's unrealistic to calculate the probability of every action, because of infinite actions. Consequently, the problem can be solved to obtain a probability distribution of action in certain action spaces. At this point, we need to parameterize the policy, which represents the distribution of operations. The action probability distribution of continuous action space is usually a normal distribution, and its probability density function PDF is defined as

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2} \tag{7}$$

A policy can be parameterized by means of mean and variance as

$$\pi(a|s,\theta) \doteq \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp -\frac{(a-\mu(s,\theta)^2}{2\sigma(s,\theta)^2} \tag{8}$$

This policy has two parameters $\theta = [\theta_\mu, \theta_\sigma]^T$, the parameters of the mean and variance, respectively. This two parameters are fitted by stochastic gradient descent. The selection of activation function in the final output layer must be a non-negative mapping, such as a soft-plus activation function.

After generating the probability density function of actions, actors can sample actions from the distributional function like stochastic policy as show in Fig. 3, which is a normal distribution. The mean and variance are all fitted by neural networks. Through this process, the model of mean and variance is optimized, but the operation of "sampling" is not differentiable and the result of sampling is differentiable. That sampling an action a from the distribution $p(x)$ (i.e. $N(\mu, \sigma^2)$) is equivalent to sampling an action a from the distribution $N(0, 1)$, then let $a = \mu + \epsilon * \sigma$.

We change the sampling from $N(\mu, \sigma^2)$ to $N(0, 1)$, and then get the result of sampling from $N(\mu, \sigma^2)$ by parameter transformation. Therefore, the operation of "sampling" does not need to participate in gradient descent. Instead, the result of sampling is involved, so that the whole model can be trained.

4.2 States Stacking

The agent chooses the appropriate action through the current environment, and obtains the reward value of the environmental feedback. Meanwhile, we usually

expect the robot to move more smoothly in continuous control tasks. Having an agent retrospect the environmental observations it has just experienced may help it choose more appropriate actions. So we retain the observations of the past three steps in the state of each step as show in Fig. 2 that learned from DRQN's [4] approach to handling Partially Observable Markov Decision Process (POMDP). So the new environmental observations can be expressed as

$$S_t = S(s_{t-3}, s_{t-2}, s_{t-1}, s_t) \tag{9}$$

Next step, the new environment state s_{t+1} is joined in and the oldest data s_{t-3} will be discarded. There is no historical state to refer to when each environment is reset, so we choose to make four copies of the first step state as the environment initial state stack S_1.

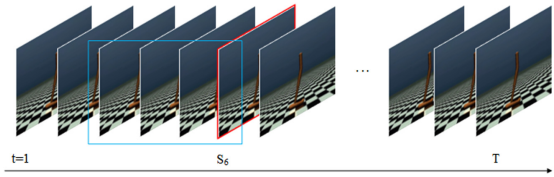


Fig. 2. The architecture of states stack

IMPASS uses a queue buffer to line the samples up that transmitted from different actors. The central learners constantly extract data from the queue to aggregate a batch of data for stochastic gradient descent.

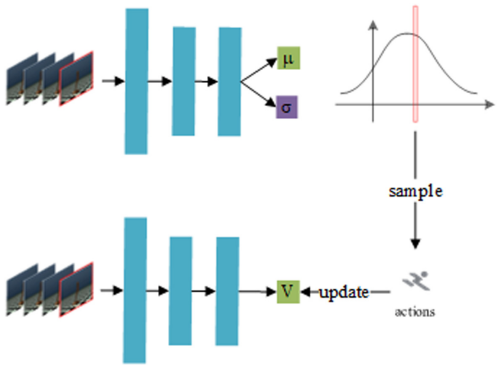


Fig. 3. The architecture of the network of actors and critics. The left figure shows the critic network, which has two layers of forward neural network. The right graph is the actor network, which encodes the observed values through two-layer network and outputs the mean and variance of Gaussian distribution.

4.3 Actor-Critic Method

Consider a parametric representation V_θ, of the value function and the current policy π_ω. Trajectories have been generated by actors following some behavior policy μ. The V-trace targets vs are defined by 1. At time t, the parameter θ of the value network is updated to target v_s by gradient descent on L2 loss. When updating critic network, the gradient direction is

$$(v_s - V_\theta(S_t))\nabla_\theta V_\theta(S_t) \tag{10}$$

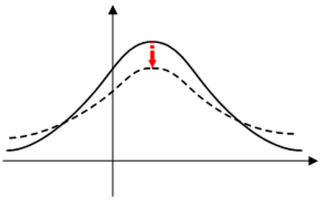


Fig. 4. Schematic diagram of maximizing entropy.

The actor network in Fig. 3 establishes the probability density function of the action distribution by fitting the means and variances of the environment state and obtains the action a of the current state S_t by sampling from the distribution.

The actor network will be updated in the direction of the off-policy learning (i.e. $\mathbb{E}_\mu[\log \mu(a_t|S_t)Q^\mu(S_t, a_t)]$). The goal of our algorithm is to learn target policy π, not behavior policy μ, so we need to replace $\mu \rightarrow (\frac{\mu}{\pi})\pi$ with the coefficient in the bracket, and the π after is the variable, that is

$$\mathbb{E}_\mu[log\mu(a_t|S_t)Q^\mu(S_t, a_t)] = \mathbb{E}[\frac{1}{\frac{\mu}{\pi}\pi}\nabla_\pi Q^\pi(S_t, a_t)] = \mathbb{E}[\frac{\pi(a_t|S_t)}{\mu(a_t|S_t)}\nabla \log \pi Q^\mu(S_t, a_t)] \tag{11}$$

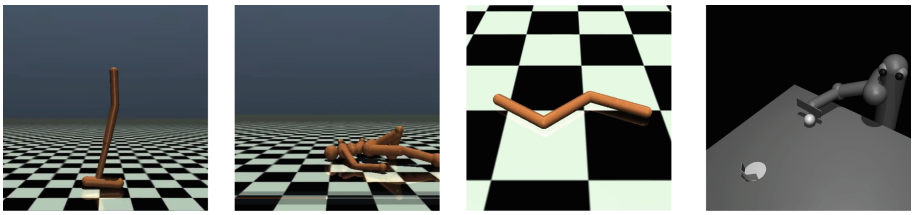
Then, update the actor network in the direction of policy gradient

$$\rho_s = \nabla_\omega \log \pi_\omega(a_s|S_t)(r_t + \gamma v_{t+1} - V_\theta(S_t)) \tag{12}$$

To prevent premature convergence, add a policy entropy penalty to the gradient of the actor

$$-\nabla_\omega \sum_a \pi_\omega(a|S_t) \log \pi_\omega(a|S_t) \tag{13}$$

Before sampling from the distribution, we minimize the entropy multiplied by a coefficient. As show in Fig. 4, the sampling of actions will have more possibilities, which will increase the ability of exploration and avoid falling into local optimum. By adding the three gradients and converting them with appropriate coefficients (which are the hyper parameters of the algorithm), the global update is obtained.



(a) Hopper (b) Humanoid (c) Swimmer (d) Striker

Fig. 5. Experiment environments of Mujoco

5 Experiment

This section will show the performance of the algorithm IMPASS in the continuous action space. Therefore, the training is carried out in each task and the experimental data is recorded regularly.

We first consider evaluating the performance of some simple physical control tasks using a set of benchmark tasks developed by the Mujoco physical simulator [16,17]. Run 1000 steps for each task, and provide instant and intensive rewards according to specific tasks $r_t = [0, 1]$ or sparse reward $r_t = 0, 1$. For each domain, the input presented to the agent consists of reasonable low-dimensional observations, many of which are composed of physical state, joint angle, etc. These observations range from 6 to 60 dimensions, but note that the difficulty of the task is not directly related to its dimensions. For example, Acrobot is one of the lowest dimensional tasks in this suite. Because of its controllability, it is more difficult to learn than other high-dimensional tasks.

We have carried out our experiments on the robot simulation suite Mujoco. Here we show the performance of our algorithm in four simulation scenarios. These four scenarios are Hopper, Humanoid (stand-up), Swimmer and Striker Fig. 5. The task in the Hopper scenario is to make a two dimensional one legged robot hop forward as fast as possible.Humanoid Make a three dimensional biped robot stand as fast as possible. This task involves a 3-link swimming robot in a viscous fluid, where the goal is to make it swim forward as fast as possible, by actuating the two joints. The origins of task can be traced back to Remi Coulom's thesis [5]. The purpose of the Striker task is to control the robot arm and strike a white ball into the goal with a white fence. In the initial state of the environment, the posture of the robot arm and the coordinates of the object are fixed values, and the coordinates of the goal are determined randomly. The task-achievement condition is that the XY coordinates of the object must fall within the goal range.

5.1 Learner-Actor Architecture

We use a scalable learner-actor architecture with multiple distributed actors to train the network. In a model like this, each actor interacts with the environment using a copy of the policy network parameters. The actor will periodically

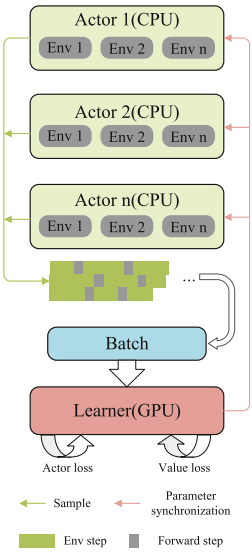


Fig. 6. The whole training process. Learner do back propagation on GPU. Each actor uses a CPU for environment inference, and each actor can run multiple simulation environments meanwhile.

transmits back the sampled data to the central parameter server, without affecting its pause of exploration, and the learners located in the central server will use these tracks to update network. Actors in this framework are not used to calculate policy network gradient. They just collect experiences and pass them on to the central learner. The learner uses a GPU device for stochastic gradient descent Fig. 6. Therefore, in such a architecture, the actor and the learner are work independent. In order to take advantage of the scale advantages of modern computing systems, IMPALA can support a single learner machine or multiple synchronous learner machines. Separating learning and acting in this way can also improve the throughput of the whole system because unlike batch A2C architecture, actors no longer need to wait for learning steps. This helps us train IMPALA in an interesting environment without having to face the difference in frame rendering time or task restart time delay.

Algorithm pseudocode for the IMPASS algorithm which includes all the above-mentioned modifications can be found in Algorithm 1.

5.2 Reinforcement Learning Framework

In order to improve the performance of the algorithm, we deploy the algorithm to a parallel reinforcement learning framework. PARL (Paddlepaddle Reinforcement Learning) is a high performance, flexible reinforcement learning framework. The framework supports large-scale parallel computing, and supports the training of multi-GPUs reinforcement learning model.

PARL provides a compact API for distributed training, allowing users to transfer the code into a parallelized version by simply adding a decorator. Call the decorated function to initialize the parallel communication. The instance obtained in this way has the same function as the original class. Because these classes are running on other computing resources, executing these functions no longer consumes the current thread computing resources. Real actors are running at the cpu cluster, while the learner is running at the local gpu with several remote actors.

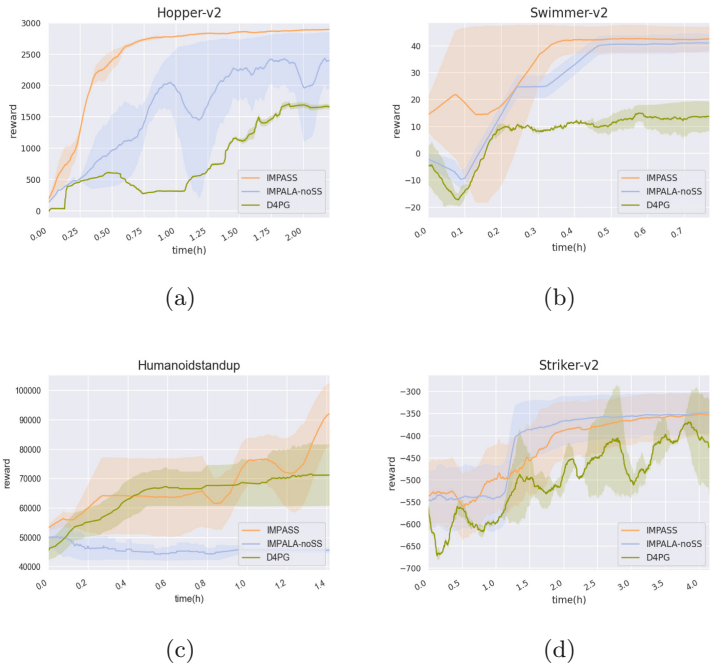


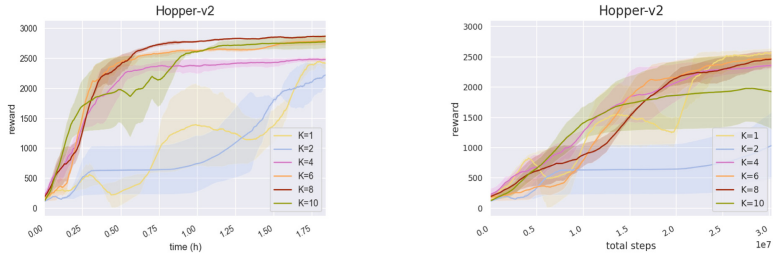
Fig. 7. Performance on different continuous control tasks: hopper, swimmer, striker and HumanStandup. The performance of IMPASS, IMPALA-noSS (IMPALA with states feature encoding and without states stacking) and D4PG. In order to show the fairness of the results, all experiments were run on the same computing platform, and the number of actor was 8.

6 Results

As shown in the Fig. 7, the performance of our algorithm in four Mujoco tasks. We use 8 actors and a single learner, and each actor runs 16 simulators in parallel. Our experiment is carried out on a computer that has one CPU with 16 cores and an Nvidia 1050Ti GPU. At the same time, our algorithm is compared with

D4PG, which is a distributed reinforcement learning algorithm too. In order to ensure the fairness of the experiment, this group of comparison experiments is also run on the same hardware device, and 8 actors are assigned. It can be seen that our agents are better than D4PG in most environments, and they can get a higher reward value. Even in the hopper environment where the performance advantage is not obvious, our agent is the first to reach the convergence state.

The learning speed of the central learner is closely related to the scale of the actor. Theoretically, the faster the sample return rate, the faster the learner can learn. But the computing performance of the GPU is limited. We also tested the effect of the different number of actors on the experimental results (Fig. 8). As expected, the more actors, the faster agents learn. But this growth is not linear. When the number of actors reaches more than 8, the growth of training speed is not obvious. This is because the computing power of GPU limits the training speed of agents. Figure 8(a) shows the correlation between the rewards obtained by the agent and the number of samples used. In the same number of samples, the speed of agent optimization is approximately the same. So increasing sample size and training batch, can increase the training speed. So by referring to Fig. 8(a), and combining your own computer performance, you can set the right number of actors to get the desired effect.



(a) (b)

Fig. 8. Effect of different number of actors on experiments for both sample and time efficiency.

7 Conclusion

We have extended the highly scalable distributed agent, IMPALA, with a method of states feature encoding and states stacking. With its scalable distributed architecture, IMPASS can make efficient use of available computing device at small and large scale. It successfully shows that it has better exploration ability and more flexible expansibility in high-dimensional input space and continuous action space.

Through the error correction of v-trace, the asynchronous environment inference and network parameter synchronization of actor and learner are realized,

which makes IMPASS have higher training speed and is superior to D4PG, which is also a distributed reinforcement learning algorithm optimized on the continuous action space. However, it also brings a problem that IMPASS has low sample utilization efficiency. At the same time, it gets a higher average reward value and with abundant environment frames, D4PG can gain higher average rewards with less sample capacity.

References

1. Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., Kautz, J.: Reinforcement learning through asynchronous advantage actor-critic on a GPU. arXiv Learning (2016)
2. Barthmaron, G., et al.: Distributed distributional deterministic policy gradients. arXiv Learning (2018)
3. Barthmaron, G., et al.: Distributional policy gradients (2018)
4. Chen, C., Ying, V., Laird, D.: Deep q-learning with recurrent neural networks. Stanford Cs229 Course Report 4, 3 (2016)
5. Coulom, R.: Reinforcement learning using neural networks, with applications to motor control. Ph.D. thesis, Institut National Polytechnique de Grenoble-INPG (2002)
6. Espeholt, L., et al.: IMPALA: scalable distributed deep-RL with importance weighted actor-learner architectures. arXiv Learning (2018)
7. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**(1), 237–285 (1996)
8. Kipf, T., Welling, M.: Variational graph auto-encoders. arXiv Machine Learning (2016)
9. Lillicrap, T., et al.: Continuous control with deep reinforcement learning (2016)
10. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning, pp. 1928–1937 (2016)
11. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
12. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (2014)
13. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
14. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (1999)
16. Tassa, Y., et al.: DeepMind control suite. arXiv e-prints arXiv:1801.00690 (2018)
17. Todorov, E., Erez, T., Tassa, Y.: MuJoCo: a physics engine for model-based control, pp. 5026–5033 (2012)
18. Vinyals, O., et al.: StarCraft II: a new challenge for reinforcement learning. arXiv Learning (2017)
19. Zoph, B., Vasudevan, V.K., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition, pp. 8697–8710 (2018)