An Efficient Asynchronous Method for Integrating Evolutionary and Gradient-based Policy Search

Kyunghyun Lee **Byeong-Uk Lee** **Ukcheol Shin** **In So Kweon**
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Korea
{kyunghyun.lee, byeonguk.lee, shinwc159, iskweon77}@kaist.ac.kr

Abstract

Deep reinforcement learning (DRL) algorithms and evolution strategies (ES) have been applied to various tasks, showing excellent performances. These have the opposite properties, with DRL having good sample efficiency and poor stability, while ES being vice versa. Recently, there have been attempts to combine these algorithms, but these methods fully rely on synchronous update scheme, making it not ideal to maximize the benefits of the parallelism in ES. To solve this challenge, asynchronous update scheme was introduced, which is capable of good time-efficiency and diverse policy exploration. In this paper, we introduce an Asynchronous Evolution Strategy-Reinforcement Learning (AES-RL) that maximizes the parallel efficiency of ES and integrates it with policy gradient methods. Specifically, we propose 1) a novel framework to merge ES and DRL asynchronously and 2) various asynchronous update methods that can take all advantages of asynchronism, ES, and DRL, which are exploration and time efficiency, stability, and sample efficiency, respectively. The proposed framework and update methods are evaluated in continuous control benchmark work, showing superior performance as well as time efficiency compared to the previous methods.

1 Introduction

Reinforcement Learning (RL) algorithms, one major branch in policy search algorithm, were combined with deep learning and showed excellent performance in various environments, such as playing simple video games with superhuman performance [1, 2], mastering the Go [3], and solving continuous control tasks [4–6]. Evolutionary methods, another famous policy search algorithm, were applied to the parameters of deep neural network and showed compatible results as Deep Reinforcement Learning (DRL) [7].

These two branches of policy search algorithms have different properties in terms of sample efficiency and stability [8]. DRL is sample efficient, since it learns from every step of an episode, but is sensitive to hyperparameters [9–12]. Evolution Strategies (ES) are often considered as the opposite because they are relatively stable, learning from the result of the whole episode [8], yet they require much more steps in the learning process [8, 13, 14].

ES and DRL are often considered as competitive approaches in policy search [7, 15], and relatively few studies have tried to combine them [16–18]. Recently, some works tried to utilize the useful gradient information of DRL into ES directly. Evolutionary Reinforcement Learning (ERL) has an independent RL agent that is periodically injected into the population [13]. In Cross-Entropy Method-Reinforcement Learning (CEM-RL), half of the population is trained with gradient information, and new mean and variance of the population are calculated with better performing individuals [19].

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

In both ES and RL, parallelization methods were introduced for faster learning and stability [2, 7, 15, 20, 21]. Most of these parallelization methods take synchronous update scheme, which aligns the update schedule of every agents to the one with the longest evaluation time. This causes crucial time inefficiency because agents with shorter evaluation should wait until the whole agents finish their job.

The asynchronous method is one of the direct solutions to this problem, as one agent can start the next evaluation immediately without waiting other agents [22–25]. Another advantage of the asynchronous method is that updates occur more often than those of synchronous methods, which can encourage diverse exploration [25, 26].

In this paper, we propose a novel asynchronous framework that efficiently combines both ES and DRL, alongside with some effective asynchronous update schemes by thoroughly analyzing the property of each. The proposed framework and update schemes are evaluated on the continuous control benchmark, underlining its superior performance and time-efficiency compared to previous ERL approaches.

Our contributions include the following:

- We propose a novel asynchronous framework that efficiently combines both Evolution Strategies (ES) and Deep Reinforcement Learning algorithms.
- We introduce several asynchronous update methods for the population distribution. We thoroughly analyze all update methods and their properties. Finally, we propose the most effective asynchronous update rule.
- We demonstrate the time and sample efficiency of the proposed asynchronous method. The proposed method reduces the entire training time about 75% wall clock on the same hardware configuration. Also, the proposed method can achieve up to 20% score gain with given time steps through effective asynchronous policy searching algorithm.

2 Background and Related work

2.1 Evolution Strategies

Evolutionary Algorithm (EA) is a type of black-box optimization inspired by natural evolution, that aims to optimize a fitness value. All individuals in EA are evaluated to calculate fitness function f, which is similar concept to the reward in RL. Evolutionary Strategy (ES) is one of the main branches in EA that one individual remains for each generation. When the population is represented by mean and covariance matrix, these algorithms are called estimation of distribution. The most famous algorithms in this category are the Cross-Entropy Method (CEM) [27] and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [28].

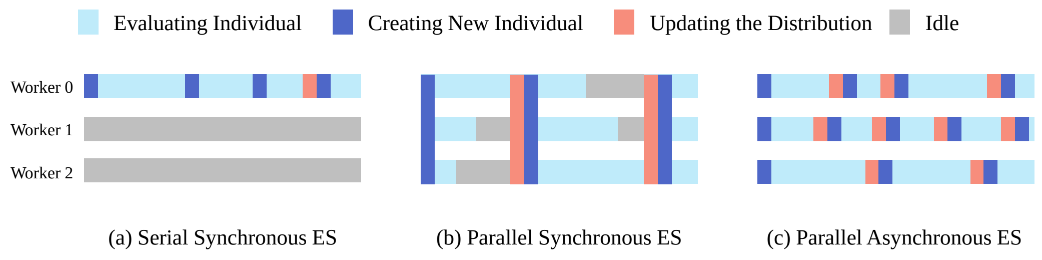
In CEM, individuals are sampled based on the distribution $\mathcal{N}(\mu, \mathbf{\Sigma})$. All individuals are evaluated and the distribution is updated based on the fixed number of best-performing individuals. CMA-ES is similar to CEM, except it considers the "evolutionary path" that collects the direction of consecutive generations.

2.2 Parallel, Synchronous and Asynchronous ES

Parallelization is the most useful way to increase the learning speed of the ES and DRL, by enabling concurrent execution. Many existing studies reported improvements using parallelization [23, 26, 29–31]. For example, Salimans et al. [7] solved MuJoCo humanoid with a simple ES algorithm in 10 minutes by using 1440 parallel cores. In particular, changing from serial to synchronous parallel requires no modification on algorithmic flow, while enabling substantial improvement in execution speed.

The synchronization process is a step that updates the population with the results of all individuals for the next generation. Therefore, when the evaluation times of all individuals differ, all workers should wait until the last worker is finished [25], thereby increasing idle time. This is more critical when the number of cores increases, since it is hard to expect linear increase of time efficiency [24].

Asynchronous algorithms are introduced to address this problem. They propose a modified process to allow all individuals to update the population immediately after their evaluation, thereby reducing



Evaluating Individual
Creating New Individual
Updating the Distribution
Idle
Worker 0
Worker 1
Worker 2
(a) Serial Synchronous ES
(b) Parallel Synchronous ES
(c) Parallel Asynchronous ES

Figure 1: **Parameter Update Scheme Comparison.** (a) One worker is used in serial synchronous ES. The worker evaluate the individuals sequentially. (b) In parallel ES, all workers evaluate the individuals in parallel. However, some workers are in idle until the last worker is finished in each generation. (c) All workers update the distribution immediately after the evaluation and evaluate the next generation. Note that the number of updates per evaluation is much more frequent in (c) compare to the other methods, which encourages the exploration.

meaningless idle time. Also, more frequent updates increase the exploration in the parameter space [25, 24]. Concepts and differences are illustrated in Fig. 1.

2.3 Update methods in ES

The update method can be divided into two main approaches depending on how the fitness value is used: the rank-based methods and the fitness-based methods. The rank-based methods sort the fitness values of all individuals and only the rank information is used for update. The fitness-based methods use the fitness value itself. The rank-based methods are more conservative and robust to inferiors. However, it also ignores the useful information from superiors [32].

In synchronous and asynchronous perspective, rank-based methods are more suitable to synchronous methods because it needs the results of all individuals in the population. On the other hand, fitness-based methods do not have this restriction and can utilize the superior individual sufficiently. The main problem in fitness-based methods is fitness value normalizing, because the evolution pressure is directly proportional to the fitness value.

In CEM, the population is updated with the rank-based method, as

$$\mu_{t+1} = \sum_{i=1}^{K_e} \lambda_i z_i, \qquad \mathbf{\Sigma}_{t+1} = \sum_{i=1}^{K_e} \lambda_i (z_i - \mu_t)(z_i - \mu_t)^\intercal + \epsilon \mathcal{I}, \tag{1}$$

where K_e is a number of elite individuals, $(z_i)_{i=1,\ldots,K_e}$ are parameters of selected individuals according to the fitness value, and $(\lambda_i)_{i=1,\ldots,K_e}$ are weights of selection for the individuals. These weights are generally either $\lambda_i = \frac{1}{K_e}$ or $\frac{\log(1+K_e)/i}{\sum_{i=1}^{K_e}\log(1+K_e)/i}$ [28]. The former is the method of giving the same weights for the selected individuals, and the latter is the method of giving higher weights according to rank.

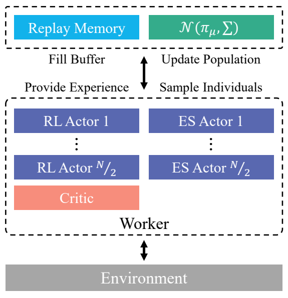
The covariance matrix calculation is sometimes reduced to variance because the calculation complexity grows exponentially as the number of parameters increases [19]. Then, Eq. (1) is simplified as

$$\mathbf{\Sigma}_{t+1} = \sum_{i=1}^{K_e} \lambda_i (z_i - \mu_t)^2 + \epsilon \mathcal{I}. \tag{2}$$

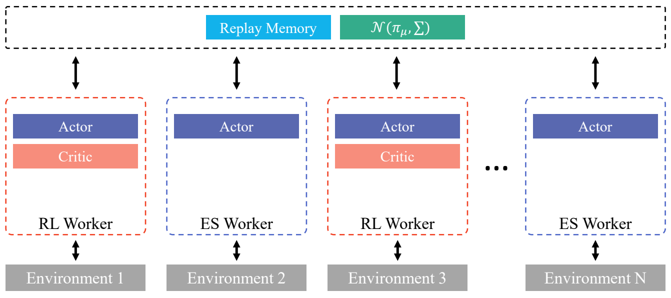
2.4 Evolutionary Reinforcement Learnings

As mentioned in the introduction section, EA and DRL have opposite property in sample efficiency and stability. Based on these characteristics, there have been a few approaches to combine these two, taking advantages from both [13, 33, 14, 19].

The evolutionary reinforcement learning (ERL) [13] is the pioneering trial to merge two approaches: population-based evolutionary algorithm and sample efficient off-policy deep RL algorithm (DDPG) [4]. However, the sample efficiency issue remains an important problem, which triggered various enhanced version of ERL to be suggested. Collaborative Evolutionary Reinforcement Learning



(a) Serial Sync ES (CEM-RL)



(b) Parallel Asynchronous ES (ours)

Figure 2: **Framework Comparison with Serial Synchronous (CEM-RL) and Parallel Asynchronous method (ours).** CEM-RL has one worker and evaluates all individuals sequentially. Ours has several distributed workers and evaluates individuals asynchronously

(CERL) [33] collects different time-horizons episodes by leveraging a portfolio of multiple learners to improve sample diversity and sample efficiency. Further, Proximal Distilled Evolutionary Reinforcement Learning (PDERL) [14] points out that the standard operators of GA, the base evolutionary algorithm of ERL, are destructive and cause catastrophic forgetting of the traits. All ERL variants share the same architecture, an independent RL agent. That is, an RL agent is trained along with the ES, and injected into the population periodically to leverage the RL. Otherwise, CEM-RL [19] trains the half of individuals with RL algorithm, utilizing the gradient information directly.

3 Methods

In this section, we present a novel asynchronous merged framework of ES and RL. Any kind of the off-policy actor-critic algorithm with a replay buffer can be applied to the RL part. Also, we present effective new update method that balances between exploration and stability. A pseudocode for the whole algorithm is described in Appendix B.

3.1 Asynchronous Framework

As described in Fig. 1, the parallel asynchronous update scheme is more time-efficient and encourages exploration compared to the synchronous update schemes. The framework of our baseline, CEM-RL, has half ES actors and half RL actors as shown in Fig. 2-(a). Only after all evaluations of the actors are finished, CEM-RL updates population distribution represented by the mean μ and the variance $\mathbf{\Sigma}$. On the other hand, our proposed parallel asynchronous framework allows each worker to have an ES or RL actor and individually update the population distribution as shown in Fig. 2-(b). Also, in CEM-RL, the shared critic network learns after synchronizing step and takes time for a while. In contrast, the shared critic network continuously learns in parallel with the other actors in our framework. Based on the change of this update scheme, the overall training time reduction and active exploration can be expected.

However, there are some problems to convert the serial synchronous update scheme to the parallel asynchronous update scheme. The first problem occurs in creating new individuals. CEM-RL creates a fixed number of ES and RL actors in new generation only after all actors are terminated. It is hard to apply the method directly to the asynchronous update because the actors have different ending moments. This problem is handled in Sec. 3.2. Secondly, we need an asynchronous method to update population distribution effectively. Since each actor has its own fitness value, a novel update method to adaptively exploit this is required. This problem is handled in Sec. 3.3 and Sec. 3.4. Our overall algorithm pseudo-code is prested in Appendix B

3.2 RL and ES Population Control

To solve the new individual creating problem, we introduce a probability based population control method for the asynchronous scheme. This method probabilistically allocates a new individual as an ES or RL actor so that the ratio between RL and ES actor is maintained as a desired ratio. The probability of being an RL actor is represented as p_{rl}, and is adaptively determined by the cumulative number of RL and ES actors, described as follows:

$$p_{\text{rl}} = \text{clip}\left(-K_{rl}\left[\frac{n_{\text{rl}}}{n_{\text{rl}} + n_{\text{es}}} - p_{\text{desired}}\right] + 0.5, 0, 1\right) \tag{3}$$

where K_{rl} is a P controller gain, p_{desired} is the desired rate of RL agent, and n_{rl} and n_{es} are the total number of RL and ES agents, respectively.

3.3 Fitness-based Mean Update

In rank-based methods, the valid weights of individuals for updating distribution are solely based on the rank, therefore the amount of "how much better is the agent" is ignored. To address this problem, we consider fitness-based methods. Fitness-based methods use the fitness value itself rather than rank, making it available to fully utilize the superior individual. One of the early works, (1+1)-ES [34], compares the fitness values of a current and new individual. Although it is not considered as a fitness-based method, we can consider it as an extremely aggressive fitness-based method, since it moves the mean μ entirely to the new individual when the fitness value of the new one is better than the current one. For smoother update, we design the mean update algorithm as Eq. (4),

$$\mu_{t+1} = (1-p)\mu_t + p \cdot \mathbf{z} \tag{4}$$

where p is an update ratio between two parameters, and $\mathbf{z}$ is a parameter of newly evaluated individual. Regarding Eq. (4), p in (1+1)-ES can be expressed as

$$p = \begin{cases} 1 & \text{if } f(\mu_t) < f(\mathbf{z}) \\ 0 & \text{if } f(\mu_t) \geq f(\mathbf{z}) \end{cases} \tag{5}$$

In order to restrain the aggressiveness of (1+1)-ES, while maintaining the capability of obtaining much information from superior individuals, we propose two novel fitness-based update methods.

Fixed Range First one is the fixed range method. This method updates the distribution relative to the current fitness value. The update factor is determined by the fixed range around the fitness value of current mean. We evaluate two variants of the fixed range methods, it sets the update rate by clipped linear interpolation over a fixed range hyperparameter r.

$$p = p_{\text{sg}} \cdot \text{clip}\left(\frac{f(\mathbf{z}) - f(\mu_t)}{r}, -1, 1\right) \tag{6}$$

p_{sg} is determined as follow,

$$p_{\text{sg}} = \begin{cases} p_{\text{positive}} & \text{if } f(\mu_t) < f(\mathbf{z}) \\ p_{\text{negative}} & \text{if } f(\mu_t) \geq f(\mathbf{z}) \end{cases} \tag{7}$$

where p_{positive} and p_{negative} are hyperparameters. Note that it is possible to move μ only in positive direction by setting $p_{negative}$ as 0, or to let it move backward by setting $p_{negative}$ other than 0.

Additionally, we introduce Sigmoid function for more smooth update:

$$p = p_{\text{sg}} \cdot \left[\text{Sigmoid}\left(\frac{f(\mathbf{z}) - f(\mu_t)}{r}\right)\right] \tag{8}$$

where p_{sg} is the same as in Eq. (7), and $\text{Sigmoid}(x) = 1/(1+e^{-x})$

Fitness Baseline Secondly, we propose the baseline methods for more conservative updates compare to the fixed range methods. In the fixed range methods, the update ratio changes drastically with moving averages. Therefore, we apply baseline techniques to make it less optimistic, as numerous RL methods [2, 5] advantage from. There are two variants in this category: absolute baseline and relative baseline. The absolute baseline sets the baseline for the entire steps, while the relative method moves the baseline with the current mean value. We also clip the update ratio from -1 to 1. The former method follows the update rule in Eq. (9),

$$p = \text{clip}\left(\frac{f(\mathbf{z}) - f_{\text{b}}}{[(f(\mu) - f_{\text{b}}) + (f(\mathbf{z}) - f_{\text{b}})]}\right) \tag{9}$$

where f_{b} is a hyperparameter for the fitness baseline.

The relative baseline is defined by $f_{\text{rb}} = f(\mu) - f_{\text{b}}$, which makes an update ratio p and a modified version of p_{sg} to be expressed as

$$p = p_{\text{sg}} \cdot \text{clip}\left(\frac{f(\mathbf{z}) - f_{\text{rb}}}{f_b + (f(\mathbf{z}) - f_{\text{rb}})}, -1, 1\right), \qquad p_{\text{sg}} = \begin{cases} p_{\text{positive}} & \text{if } f(\mathbf{z}) \geq f_{rb} \\ p_{\text{negative}} & \text{if } f(\mathbf{z}) < f_{rb} \end{cases} \tag{10}$$

3.4 Variance Matrix Update

We update the variance matrix σI instead of the covariance matrix $\mathbf{\Sigma}$ to reduce time complexity, following the assumption used in CEM-RL [19]. As described in Sec. 2.3, the variance of the synchronous method can be calculated directly from a set of individuals, but not in the asynchronous update method. Therefore, different methods such as asynchronous ES [26] and Rechenberg's 1/5th success rule [34] were introduced, where the first one utilizes pre-stored individuals, and the second one increases or decreases variance based on a success ratio p_s and threshold ratio p_{th}. However, the former simply adapted the method of rank-based synchronous, and the latter naively controlled the variance. In this section, we propose two asynchronous variance update methods that effectively encourages exploration in proportion to the update ratio of p.

Online Update with Fixed Population In the asynchronous update scheme, the mean and variance of all individuals are unknown, and only the values of the current population distribution and a single individual are known. Welford's online update rule [35] is an update method that can be used in this situation, described as follows:

$$\sigma_{\mathbf{t}}{}^2 = \sigma_{\mathbf{t-1}}{}^2 + \frac{(\mathbf{z} - \mu_{\mathbf{t-1}})^{\intercal}(\mathbf{z} - \mu_{\mathbf{t}}) - \sigma_{\mathbf{t-1}}{}^2}{n} \tag{11}$$

This equation was initially designed to save the memory space without storing the past data by calculating the new variance from the current variance. For this purpose, the original n in Eq. (11) is increased each time new data comes in, to represent the total number of data. Also, past data and present data are equally valuable in the original equation. However, since the past and present individuals should not be treated equally in the ES, we need to calculate the variance by giving more weight to the values of the recent individuals. Thus, we modified n in the equation to have constant value inspired by the concept of rank-based variance update method [26] which removes the oldest individual from the population. The influence of old individuals would gradually disappear with the fixed n.

Online Update with Adaptive Population In the fixed population update rule, the number of the past individuals n is fixed during the update. It means that every new individual added to the calculation will have the same influence in every update. To make this process more adaptive to each individual, we consider Rechenberg's method [34]. It makes its variance increase for better exploration when success rate is high, and decrease for more stability when success rate is low.

We design n by utilizing the update ratio p from the mean calculation in Sec. 3.3, to replace 'success rate' in [34] to be more suitable for our algorithm, as well as to smooth and stabilize the update process. p implies how 'important' the new individual is.

To elaborate, assume p of the newly evaluated individual is high. This means that the individual is relatively 'important' than the other individuals. We want to increase the influence of this individual in the variance update, by reducing the value of n, which reduces the influence of the previous variance σ_{t-1} in Eq. (11). If the new individual is relatively not 'important', i.e. it has small p, then n is set to a larger value. When $p = 0$, then the new individual has no importance, the variance remains unchanged. With a clipping operation added, we can express the update process of n as

$$n = \max\left(\frac{1-p}{p}, 1\right) \tag{12}$$

Note that the mean and the variance update rules are independent, so any combination of the rules can be used.

Table 1: **Time efficiency comparison results in HalfCheetah-v2, Walker2d-v2 and Hopper-v2.** In HalfCheetah-v2, the episode length is fixed to 1000 steps, and evaluation time for all agents are theoretically identical. In Walker2d-v2 and Hopper-v2, the episode length varies. The percentage indicates the amount of *reduced* time compared to CEM-RL. More detailed results about the number of workers are provided in Appendix F.

	CEM-RL	P-CEM-RL	AES-RL	
Sequence Update	Serial Sync.	Parallel Sync.	Parallel Async.	
Workers	1	5	5	9
HalfCheetah-v2 (min)	467	187 (59.9%↓)	83 (82.1%↓)	54 (88.3%↓)
Walker2d-v2 (min)	487	205 (57.9%↓)	105 (78.4%↓)	77 (84.1%↓)
Hopper-v2 (min)	505	188 (62.6%↓)	133 (73.6%↓)	91 (82.0%↓)

Table 2: **Performance analysis for various combinations of our proposed mean-variance update methods.** Results are measured with average scores of ten test runs within a total of 1M step from the summation of all worker steps, averaged with ten random seeds. Full results for various environments are presented in Appendix D.

	Category	Previous algorithms		Proposed algorithms							
		(1+1)-ES	Rank-Based	Fixed Range		Fitness Baseline		Fixed Range		Fitness Baseline	
	μ	Full Move	Oldest	Linear	Sigmoid	Absolute	Relative	Linear	Sigmoid	Absolute	Relative
	Σ	Success Rule		Online Update - Adaptive				Online Update - Fixed			
HalfCheetah-v2	Mean	11882	10010	10279	12053	12224	**12550**	10870	12031	11767	12128
	Std.	385	746	1044	398	422	**187**	409	604	458	821
Walker2D-v2	Mean	2347	4230	5020	5360	5137	**5474**	3419	5121	5039	5070
	Std.	320	254	799	683	223	**223**	1676	965	471	557

4 Experiment

In this section, we compare the results of provided methods with the synchronous algorithm in the perspective of time-efficiency and performance. We evaluate the algorithms in several simulated environments which are commonly used as benchmarks in policy search: HalfCheetah-v2, Hopper-v2, Walker2d-v2, Swimmer-v2, Ant-v2, and Humanoid-v2 [36]. The presented statistics were calculated and averaged over 10 runs with the same configuration. As our aim is to show the efficiency of the asynchronous methods, we tried to use architecture and hyperparameters as similar as possible of those in CEM-RL [19]. We used TD3 [5] in RL part. Detailed architecture and hyperparameters for all methods are shown in Appendix A and C, respectively. [1]

4.1 Time Efficiency Analysis

We compare the time efficiency of three architectures, serial-synchronous, parallel-synchronous, and parallel-asynchronous. CEM-RL from the original author[2] is based on a serial-synchronous method. We implement parallel-synchronous version of CEM-RL, P-CEM-RL. Finally, AES-RL is our proposed parallel-asynchronous method. All three algorithms are evaluated in HalfCHeetah-v2, which has the fixed episode steps, and Walker2d-v2 and Hopper-v2, which has varying episode steps, in the same hardware configuration, two Ethernet-connected machines of Intel i7-6800k and three NVidia GeForce 1080Ti; a total of 24 CPU cores and 6 GPUs.

The results are shown in Tab. 1. It shows that parallelization brings significant time reduction compared to the serial methods. By simply switching from serial to parallel, time efficiency is greatly improved, reducing around 60% of total training time compared to the synchronous serial method. Additional efficiency of around 10-20% is achieved from asynchronism. We also scale workers up to nine, then 80-90% of the training time is reduced compared to the serial version. The result shows that the inefficiency of the synchronous methods is significant with varying episode length, as described in Fig. 1.

[1]The source code of our implementation is available at `https://github.com/KyunghyunLee/aes-rl`

[2]`https://github.com/apourchot/CEM-RL`

4.2 Performance Analysis of Mean-Variance Update Rules

We propose the asynchronous update methods including 4 mean and 2 variance update rules, as described in Sec. 3. Therefore, 8 combinations are available, and we thoroughly analyze their properties. Tab. 2 shows the comparison results, along with the previously proposed algorithms, (1+1)-ES [34], and asynchronous version of rank-based update [26]. All algorithms are evaluated in HalfCheetah-v2 and Walker-v2 environment, where former is easier and latter is harder, relatively.

As we expected in Sec. 3, (1+1)-ES shows the instability because of its extreme update rule. It successfully solved the HalfCheetah-v2, but failed in Walker-v2. In contrast to the (1+1)-ES, Rank-based method shows lower performance in both environment. They saturated at the lower score and was not able to explore further.

The fixed range algorithms show lower performance compared to the baseline algorithms, but are better than the rank-based method. Sigmoid is better than linear for all configuration, since it leverages more advantages from superior individuals. The baseline methods show higher performance and stability, showing higher mean and lower variance, consistently. Here, stability is defined as how consistent the results are for various random seeds with the std value per mean σ/μ. In the absolute baseline method, it updates conservatively at the latter part of training because of the scaling effect mentioned in Sec. 2.3. Whereas, the relative baseline method effectively utilizes the information from superior individuals, thus showing the best results. In terms of the variance update, the adaptive method shows better performance, compare to the fixed population method. The fixed population method shows higher variance and lower mean, because it sometimes fails to explore enough compared to the adaptive method. As a result, the relative baseline with adaptive variance method shows the best results as expected. Detailed result for combinations are presented in Appendix D.

4.3 Performance Analysis of Other Algorithms

We compare our proposed algorithm to the previous RL and ERL algorithms: TD3, SAC [6], CEM, ERL and CEM-RL. The results are summarized in Tab. 3. Except for our results, we cite the results of the RL and ERL algorithms reported in [19]. Excluding the ERL, the other algorithms are trained for 1M steps. Training steps of the ERL is the same as what were given in the original paper [13]:

Table 3: **Performance analysis of TD3, SAC, CEM, ERL, CEM-RL, and AES-RL in six MuJoCo benchmarks.** Our algorithm outperforms the other methods in most of environments except for Ant-v2 and Swimmer-v2. Results of other algorithms are from their original report, except for Humanoid-v2. Improvements are compared to the baseline CEM-RL.

Environment	Statistics	TD3	SAC	CEM	ERL	CEM-RL	AES-RL	Improvement
Algorithm		RL		ES	RL+ES		RL+ES	
Update Method		-		-	Sync.		Async.	
HalfCheetah-v2	Mean	9630	11504	2940	8684	10725	**12550**	**17.02 %**
	Std.	202	183	353	130	397	**187**	
	Median	9606	11418	3045	8675	11539	**12571**	
Hopper-v2	Mean	3355	3239	1055	2288	3613	**3751**	**3.83 %**
	Std.	171	18	14	240	105	**58**	
	Median	3626	3230	1040	2267	3722	**3746**	
Walker2D-v2	Mean	3808	4268	928	2188	4711	**5474**	**16.19 %**
	Std.	339	435	50	328	155	**223**	
	Median	3882	4354	934	2338	4637	**5393**	
Ant-v2	Mean	4027	**5985**	487	3716	4251	5120	**20.44 %**
	Std.	403	**114**	33	673	251	170	
	Median	4587	**6032**	506	4240	4310	5071	
Swimmer-v2	Mean	63	46	**351**	350	75	161	-
	Std.	9	2	**9**	8	11	100	
	Median	47	45	**361**	360	62	128	
Humanoid-v2	Mean	5496	5505	-	5170	5579	**6136**	**9.98 %**
	Std.	187	108	-	130	228	**444**	
	Median	5465	5539	-	5120	5673	**6133**	

2M on HalfCheetah-v2, Swimmer-v2 and Humanoid-v2, 4M on Hopper-v2, 6M on Ant-v2 and 10M on Walker2d-v2.

Our algorithm outperforms most of the other methods consistently, except for Ant-v2 and Swimmer-v2. As reported in CEM-RL, Swimmer-v2 is a challenging environment for all RL algorithms. However, our methods show better performance than TD3, CEM-RL in Swimmer-v2, because of better exploration. In detail, our algorithm gets two high scores and eight low scores out of 10 trials, in contrast to CEM-RL and TD3 that got low scores for all trials. The results for failed trials are **111.97** $\pm$ **24.10**, and **355.95** $\pm$ **1.95** for the succeeded trials. This shows that even in our 8 failed attempts, we outscore CEM-RL, while showing the comparable results in 2 succeeded trials.

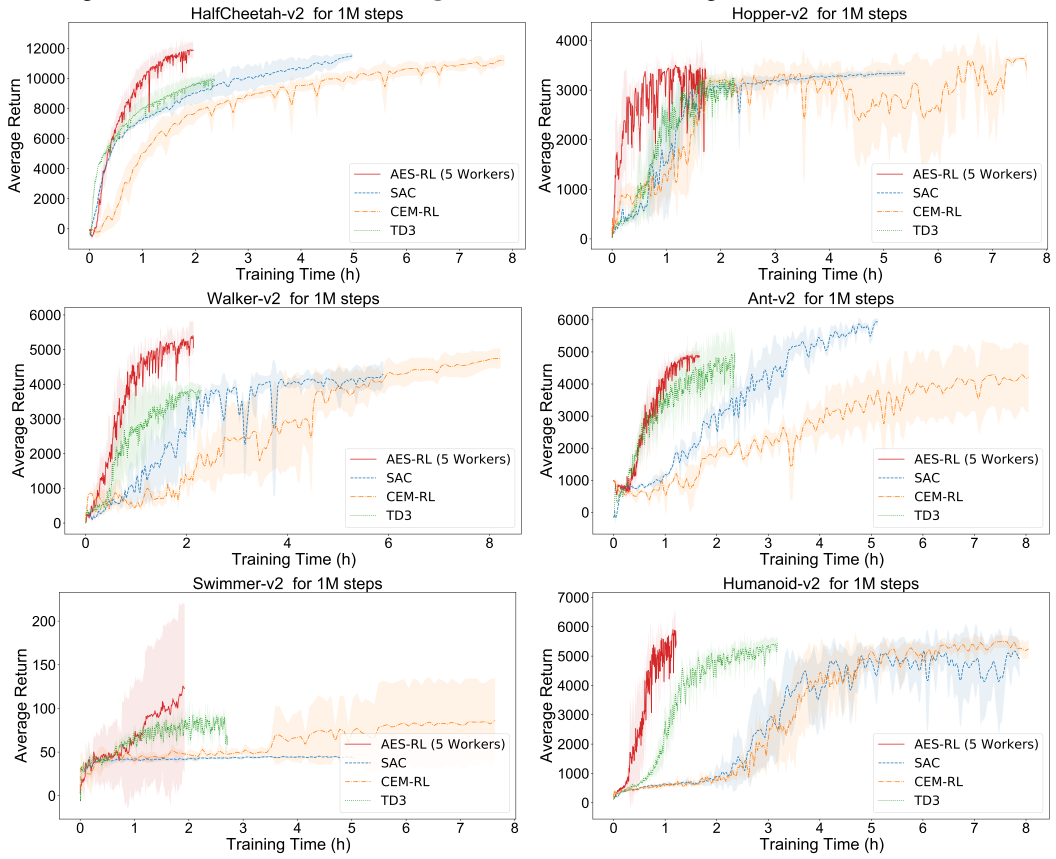
The learning curves for each environment are presented in Fig. 3. It shows that the AES-RL can achieve better performance with less time and also within a fixed amount of steps.

5 Discussion

In this paper, we proposed the asynchronous parallel ERL framework to address the problems of previous synchronous ERL methods. We also provided several novel mean-variance update rules for updating the distribution of the population and analyzed the influence of each method. Our proposed framework and update rules were evaluated in six MuJoCo benchmarks, and showed outstanding results with 80% of time reduction with five workers and 20% of performance improvement at its best.

With our framework, it is possible to use any off-policy RL algorithms and many ask-and-tell ES algorithms, which can be used with up to 200k parameters. We remain this for the future work.

Figure 3: **Wallclock-wise Learning Curve** Curves are averaged with three random seed



Broader Impact

To apply RL in the real world problem, efficiency and stability are essential, because the cost of interacting with the environment is much higher than the simulation. Evolutionary Reinforcement Learning is an attempt to combine ES and RL for efficiency and stability. However, the previous works used synchronous evolutionary algorithms that cannot be scaled up.

This paper focuses on the asynchronous combination method of RL and ES. We introduce several efficient update rules for applying asynchronism and analyze the performance. We confirm that the asynchronous methods are much more time efficient, and it also have more exploration property for searching effective policy. With the proposed algorithm, AES-RL, branch of stable and efficient policy search algorithms can be extended to numerous workers. We expect the policy search algorithms to be actively applied to real world problems.

Acknowledgments and Disclosure of Funding

This work was in partial supported by the Industrial Strategic Technology Development Program(Development of core technology for advanced locomotion/manipulation based on high-speed/power robot platform and robot intelligence, 1 0070171) funded By the Ministry of Trade, industry & Energy(MI, Korea) This work was also in partial supported by the Technology Innovation Program funded by the Ministry of Trade, Industry and Energy, South Korea, under Grant 2017-10069072.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

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

[5] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[6] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870, 2018.

[7] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[8] Olivier Sigaud and Freek Stulp. Policy search in continuous action domains: an overview. *CoRR*, abs/1803.04706, 2018.

[9] Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S. Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In *NIPS*, pages 1204–1212, 2009.

[10] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016.

[11] Chen Tessler, Nadav Merlis, and Shie Mannor. Stabilizing deep reinforcement learning with conservative updates, 2019.

[12] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017.

[13] Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1188–1200, 2018.

[14] Cristian Bodnar, Ben Day, and Pietro Lio'. Proximal distilled evolutionary reinforcement learning, 2019.

[15] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567, 2017.

[16] Rein Houthooft, Richard Y. Chen, Phillip Isola, Bradly C. Stadie, Filip Wolski, Jonathan Ho, and Pieter Abbeel. Evolved policy gradients, 2018.

[17] David Ackley and Michael Littman. Interactions between learning and evolution. In C. G. Langton, C. Taylor, C. D. Farmer, and Rasmussen S., editors, *Artificial Life II, SFI Studies in the Sciences of Complexity*, volume X, pages 487–509. Addison-Wesley, Reading, MA, USA, 1992.

[18] Madalina M. Drugan. Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms. *Swarm and Evolutionary Computation*, 44:228 – 246, 2019. ISSN 2210-6502. doi: https://doi.org/10.1016/j.swevo.2018.03.011.

[19] Pourchot and Sigaud. CEM-RL: Combining evolutionary and gradient-based methods for policy search. In *International Conference on Learning Representations*, 2019.

[20] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

[21] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *CoRR*, abs/1803.00933, 2018.

[22] Bernard P. Zeigler and Jinwoo Kim. Asynchronous genetic algorithms on parallel computers. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 660, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. ISBN 1558602992.

[23] Juan José Durillo, Antonio J. Nebro, Francisco Luna, and Enrique Alba. A study of master-slave approaches to parallelize nsga-ii. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[24] Matthew A. Martin, Alex R. Bertels, and Daniel R. Tauritz. Asynchronous parallel evolutionary algorithms: Leveraging heterogeneous fitness evaluation times for scalability and elitist parsimony pressure. In *GECCO Companion '15*, 2015.

[25] Eric O. Scott and Kenneth A. De Jong. Understanding simple asynchronous evolutionary algorithms. In *FOGA '15*, 2015.

[26] Tobias Glasmachers. A natural evolution strategy with asynchronous strategy updates. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 431–438, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319638. doi: 10.1145/2463372.2463424. URL `https://doi.org/10.1145/2463372.2463424`.

[27] Reuven Y. Rubinstein and Dirk P. Kroese. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-Carlo Simulation (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2004. ISBN 038721240X.

[28] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772, 2016. URL `http://arxiv.org/abs/1604.00772`.

[29] Erick Cantú-Paz. A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10, 1998.

[30] Enrique Alba and José M. Troya. A survey of parallel distributed genetic algorithms. *Complex.*, 4(4):31–52, March 1999. ISSN 1076-2787.

[31] Gabriel Luque and Enrique Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer Publishing Company, Incorporated, 2013. ISBN 3642268684.

[32] Lance D. Chambers. *Practical Handbook of Genetic Algorithms*. CRC Press, Inc., USA, 1995. ISBN 0849325196.

[33] Shauharda Khadka, Somdeb Majumdar, Tarek Nassar, Zach Dwiel, Evren Tumer, Santiago Miret, Yinyin Liu, and Kagan Tumer. Collaborative evolutionary reinforcement learning. In *International Conference on Machine Learning*, pages 3341–3350, 2019.

[34] I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Number 15 in Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.

[35] Author(s) B. P. Welford and B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, pages 419–420, 1962.

[36] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IROS*, pages 5026–5033. IEEE, 2012. ISBN 978-1-4673-1737-5.

A Network Architecture

For a fair comparison, our network follows the same structure as CEM-RL [19]. The architecture is originally from Fujimoto et al. [5], the only difference is using tanh instead of RELU. Pourchot and Sigaud [19] reported the difference between the RELU and tanh. We use $(400, 300)$ hidden layer for all environment except Humanoid-v2. For Humanoid-v2, we used $(256, 256)$ as in TD3 [5].

Table 4: **Network architectures** The architecture from the input layer to the output layer

Layer Type	Actor	Critic
Linear	(state_dim, 400)	(state_dim + action_dim , 400)
Activation	tanh	leaky RELU
Linear	(400, 300)	(400, 300)
Activation	tanh	leaky RELU
Linear	(300, action_dim)	(300, 1)
Activation	tanh	

B AES-RL pseudo-code

Algorithm 1: AES-RL

```
1  Initialize: the mean of the population $\pi_\mu$, shared critic $Q^\pi$ and target critic $Q^{\pi'}$
2  Initialize: the covariance matrix $\mathbf{\Sigma} = \sigma_{\text{init}}\mathcal{I}$, the emtpy replay buffer $\mathcal{R}$, $total\_steps = 0$
   /* Start training of the shared critic */
3  critic_worker.start_critic_training()
4  while $total\_steps < max\_steps$ do
5      $N_{idle\_worker}$ = num_idle_worker()
6      while $N_{idle\_worker}$ > 0 do
           /* Create new individual */
7          new_individual = population.sample($\pi_\mu$, $\mathbf{\Sigma}$)
8          Create new individual by sampling from $\mathcal{N}(\pi_\mu, \mathbf{\Sigma})$
9          actor_worker = get_idle_worker()
10         actor_worker.set_actor_network(new_individual)
11         Calculate $p_{\text{rl}}$ according to Eq. (3)
12         if $rand() < p_{rl}$ and $total\_step >= rl\_start\_step$ then
               /* Train the actor with shared critic */
13             critic_weight = critic_worker.get_critic_weight()
14             actor_worker.set_critic_network(critic_weight)
15             actor_worker.train_actor_network()
16             $n_{\text{rl}} \leftarrow n_{\text{rl}} + 1$
17         else
               /* Evaluate immediately */
18             $n_{\text{es}} \leftarrow n_{\text{es}} + 1$
19         end
           /* Evaluate the individual, with filling the replay buffer */
20         actor_worker.evaluate($\mathcal{R}$)
21         $N_{idle\_worker} \leftarrow N_{idle\_worker} - 1$
22     end
23     $N_{finished\_worker}$ = num_finished_worker()
24     while $N_{finished\_worker} > 0$ do
25         finished_worker = get_finished_worker()
26         fitness, individual, current_steps = finished_worker.get_evaluation_information()
           /* Update population with the finished individual */
27         population.update(fitness, individual)
28         $total\_steps \leftarrow total\_steps + current\_steps$
29         $N_{finished\_worker} \leftarrow N_{finished\_worker} - 1$
30     end
31 end
```

```
Algorithm 2: actor.evaluate($\mathcal{R}$)
Require: hyperparameter action noise $a_{\text{noise}}$
state = env.reset()
done = False
steps = 0
total_reward = 0
while not done do
    action = actor_network.forward(state)
    if $a_{noise} \neq 0$ then
        action = clip(action + $a_{\text{noise}}$ * random.normal(0, 1), -1, 1)
    end
    next_state, reward, done, info = env.step(action)
    $\mathcal{R}$.append(state, next_state, reward, done)
    steps ← steps + 1
    total_reward ← total_reward + 1
    state = next_state
end
return total_reward, steps
```

C Hyperparameters

Most of hyperparameters are the same value as CEM-RL [19]. However, the size of replay buffer is modified to $2e5$, we analyzed the effect on Appendix C.1. Also, we used action noise of 0.1 as suggested by Khadka and Tumer [13]. Pourchot and Sigaud [19] reported the action noise is not useful for their algorithm, we found that the action noise improves exploration as the original ERL paper. We use $a_{\text{noise}} = 0.1$ for all environments. We discuss the effect on Appendix C.2. The K_{rl} in population control is set to 50. p_{negative} for fixed range algorithms are set to 0. We use '1/5' for success rate of $(1+1)$-ES. Also, we use $p_{\text{desired}} = 0.5$ for all environment except Swimmer-v2. CEM-RL reported that RL algorithms provide deceptive gradients, therefore most of the RL algorithms fail to solve. Therefore we use $p_{\text{desired}} = 0.1$, which means that the population of the ES is 9 times larger than the RL. Consequently, algorithms are become more dependent on the ES part.

Other values related to the range are presented in Tab. 5. We noticed that the corresponding values are about 1/6 of the maximum reward. For example, the reward value reaches up to 12000 in HalfCheetah, then the 1/6 of the maximum is about 2000.

Table 5: **Hyperparameters** A list of the hyperparameters that vary with the environment

		Fixed Range	
		Linear	Sigmoid
Range r	HalfCheetah-v2	2000	2000
	Hopper-v2	600	600
	Walker2D-v2	860	860
	Ant-v2	960	960
	Swimmer-v2	48	48
	Humanoid-v2	960	960
		Fitness Baseline	
		Absolute	Relative
Baseline f_b	HalfCheetah-v2	-2000	2000
	Hopper-v2	-600	600
	Walker2D-v2	-860	860
	Ant-v2	-960	960
	Swimmer-v2	-48	48
	Humanoid-v2	-960	960

C.1 Replay Buffer Size

The replay buffer size of 200k improves the performance of the proposed algorithm. The performance of CEM-RL also increased, but not as much as the asynchronous algorithm. A possible hypothesis for this phenomenon is as follows. In CEM-RL, the critic learning steps are guaranteed in the synchronous stage. However, in AES-RL, the critic is trained in a parallel with the other workers; thus samples should be more productive. With the reduced size of the replay buffer, it is filled with more recent steps and replace the oldest experiences which is nearly useless. Therefore, the samples are more informative.

Tab. 6 shows the comparison results. In Walker-v2, the size of the replay buffer does not significantly affect the performance of AES-RL. However, the performance gap is relatively more significant in HalfCheetah-v2. One of the differences of the two environment is that the learning saturates earlier in Walker-v2. Therefore, the replay buffer is productive enough. Differently, actors in HalfCheetah-v2 still learn at the end of 1M steps. Here, We may use other techniques like importance sampling to enhance the efficiency of a mini-batch sample; however, we remain it as future work.

Table 6: **Effect of the Replay Buffer Size**

	μ / Σ	Relative Range Adaptive		CEM-RL	
	Replay	200k	1M	200k	1M
HalfCheetah-v2	Mean	**12550**	11472	11515	10725
	Std.	**187**	467	203	354
Walker2D-v2	Mean	**5474**	5468	4503	4711
	Std.	**223**	690	388	155

C.2 Action Noise

CEM-RL reported that the action noise is not useful, as opposed to ERL. However it consistently improves the performance a little in our experiments. We hypothesize that the π_μ of CEM-RL moves with the weighted average of individuals; therefore, the effect of action noise is reduced. Otherwise, in AES-RL, the action noise increases the chance of better policy which affects directly to the π_μ.

Table 7: **Effect of the Action Noise**

	μ / Σ	Relative Range Adaptive	
	a_{noise}	0.1	0.0
HalfCheetah-v2	Mean	**12550**	12095
	Std.	**187**	338
Walker2D-v2	Mean	**5474**	5244
	Std.	**223**	670

D Full Results of Proposed Methods

We compare all combination of methods proposed in Sec. 3. Except for Swimmer-v2, the relative baseline is the best in both performance and stability. In Hopper-v2 and Ant-v2, the relative baseline scores are slightly lower than the best methods, but it is comparable. Therefore we use the relative baseline methods when compare with the previous algorithms: TD3, CEM, ERL, and CEM-RL.

In Swimmer-v2, CEM, pure evoluationary algorithm, was the best, ERL, mostly evolutionary algorithm, was also able to solve the environment [19]. Among the asynchronous algorithms, $(1 + 1)$-ES, which has the most aggressive update rule, is consistently successful. From this result we can infer that aggressive exploration is more important in Swimmer-v2. Also, other methods have a chance to solve the environment, but not always.

In conclusion, we proposed various update rules for asynchronous algorithms. We started from the Rank-Based asynchronous algorithm and the $(1 + 1)$-ES update algorithm, which are at the extremes.

Our design purpose is to achieve a balance between the two. The relative baseline method with adaptive variance showed the best performance, which effectively balances between aggressive and conservative updates.

Table 8: **Full results of our proposed mean-variance update methods** Results are measured with average scores of ten test runs within a total of 1M step from the summation of all worker steps, averaged with ten random seeds.

	Category	Previous algorithms		Proposed algorithms							
		(1+1)-ES	Rank-Based	Fixed Range		Fitness Baseline		Fixed Range		Fitness Baseline	
	μ	Full Move	Oldest	Linear	Sigmoid	Absolute	Relative	Linear	Sigmoid	Absolute	Relative
	Σ	Success Rule		Online Update - Adaptive				Online Update - Fixed			
HalfCheetah-v2	Mean	11882	10010	10279	12053	12224	**12550**	10870	12031	11767	12128
	Std.	385	746	1044	398	422	**187**	409	604	458	821
Walker2D-v2	Mean	2347	4230	5020	5360	5137	**5474**	3419	5121	5039	5070
	Std.	320	254	799	683	223	**223**	1676	965	471	557
Hopper-v2	Mean	2588	3729	3506	3764	**3789**	3751	2996	3769	3788	3423
	Std.	748	53	179	40	**26**	58	1068	20	30	626
Ant-v2	Mean	5098	3917	3015	**5140**	3883	5120	3406	5007	4947	4613
	Std.	715	514	1233	**546**	1047	170	944	891	543	712
Swimmer-v2	Mean	**347**	59	99	128	97	161	191	81	107	120
	Std.	**19**	8	43	65	32	100	123	12	34	63
Humanoid-v2	Mean	600	3476	5697	5958	5770	**6136**	5662	5695	5774	5837
	Std.	35	1980	177	301	229	**444**	107	209	267	239

E Ablation Study

AES-RL algorithm mainly consists of three novel methods; an asynchronism, the mean update rule, and the variance update rule. In this section, we evaluate the effectiveness of each methods.

E.1 Asynchronism

To compare the effectiveness of asynchronism, we adopt an update rule of CEM-RL based on the simple asynchronous methods in [26]. Therefore, the resulting algorithm is an asynchronous version of CEM-RL, namely ACEM-RL. As in [26], previous results are stored in a separate list with its score. When a new individual is evaluated, it is stored on the list, and the oldest one is removed to maintain the population size. However, there should be a multiplication factor $1/n$ because the update occurs n times frequently. Therefore the mean update in Eq. (1) is modified to

$$\mu_{t+1} = \frac{1}{n} \sum_{i=1}^{K_e} \lambda_i z_i \tag{13}$$

E.2 Mean and Update Rule

In the mean update, we already compared various update rules in Sec. 4.2 and Appendix D. The result of ACEM-RL, which applies asynchronism to CEM-RL, is in column “Rank-based"-“Oldest" in Tab. 8 In addition, we fix the variance to the constant value. Here, We expect that the fixed variance prevents exploration.

The overall results are displayed in Tab. 9. As a result, simply applying asynchronism to the previous method without proper mean and variance update reduces the performance.

Table 9: **Ablation study** The overall result of the ablation study. From the baseline algorithm CEM-RL, ACEM-RL adopts asynchronism. AES-RL with constant variance means that only the mean is updated. Finally, AES-RL results include all features, asynchronism, mean update, and variance update.

		CEM-RL	ACEM-RL	AES-RL $\sigma^2 = \text{const.}$		**AES-RL**
				0.0001	0.001	
HalfCheetah-v2	Mean	10725	10010	11636	12306	**12550**
	Std.	397	746	209	233	**187**
Walker2D-v2	Mean	4711	4230	5167	5302	**5474**
	Std.	155	254	251	458	**223**

F Training Time According to the Number of workers

We compare the execution time for a various number of workers, from 2 to 9. Training time of CEM-RL is measured with original author's implementation. P-CEM-RL is our implementation of parallel version of CEM-RL, which has parallelized actors with synchronous update scheme. For AES-RL,

Table 10: **Training time according to the number of workers** Training time is measured in minutes with Ethernet-connected two machines of Intel i7-6800k with three NVidia GeForce 1080Ti each.

	CEM-RL	P-CEM-RL	AES-RL							
Workers	1	5	2	3	4	5	6	7	8	9
HalfCheetah-v2	467.17	187.42	225.63	136.32	103.15	83.43	72.38	65.52	58.88	**54.47**
Walker-v2	487.25	205.17	275.90	163.10	131.18	105.42	97.12	84.23	81.02	**77.33**
Hopper-v2	504.63	188.48	305.23	199.55	144.77	133.23	122.58	97.35	92.52	**90.75**

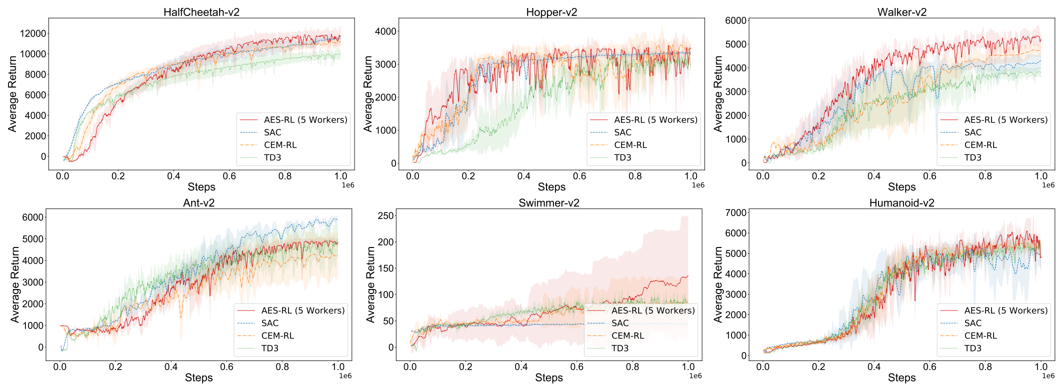
G Contribution of RL and ES in Learning Process

We approximately compare the contribution of RL and ES agents. To measure the contributio,n we used the update ratio p in mean update rules. Higher p indicates the new agent moves the mean of the distribution more. We recorded the p value for all updates, and the values are accumulated for total experiment. The result in Tab. 11 shows that the ratio of each agents differs in each environments. For Swimmer-v2, our agent fails to find good solution (reward higher than 300) 8 out of 10, we only measured in successful trials.

Table 11: **Contribution of RL and ES**

	HalfCheetah-v2	Walker2D-v2	Hopper-v2	Ant-v2	Swimmer-v2	Humanoid-v2
ES (%)	37.1	50.3	64.2	22.5	79.5	50.4
RL (%)	62.9	49.7	35.8	77.5	20.5	49.6

H Step-wise Learning Curve


HalfCheetah-v2
Hopper-v2
Walker-v2
Ant-v2
Swimmer-v2
Humanoid-v2
Average Return
Steps
1e6
AES-RL (5 Workers)
SAC
CEM-RL
TD3