Combining Soft-Actor Critic with Cross-Entropy Method for Policy Search in Continuous Control

Hieu Trung Nguyen, Khang Tran, Ngoc Hoang Luong
University of Information Technology, Ho Chi Minh City, Vietnam
Vietnam National University, Ho Chi Minh City, Vietnam
{18520750, 18520072}@gm.uit.edu.vn, hoangln@uit.edu.vn

2022 IEEE Congress on Evolutionary Computation (CEC) | 978-1-6654-6708-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/CEC55065.2022.9870209

***Abstract*—In this paper, we propose CEM-SAC – a hybridization between the cross-entropy method (CEM), i.e., an estimation-of-distribution algorithm, and the soft-actor critic (SAC), i.e., a state-of-the-art policy gradient algorithm. Our work extends the evolutionary reinforcement learning (ERL) line of research on integrating the robustness of population-based stochastic black-box optimization, that typically assumes little to no problem-specific knowledge, into the training process of policy gradient algorithms, that exploits the sequential decision making nature for efficient gradient estimation. Our hybrid approach, CEM-SAC, exhibits both the stability of CEM and the efficiency of SAC in training policy neural networks of reinforcement learning agents for solving control problems. Experimental result comparisons with the three baselines CEM, SAC, and CEM-TD3, a recently-introduced ERL method that combines CEM and the twin-delayed deep deterministic policy gradient (TD3) algorithm, on a wide range of control tasks in the MuJoCo benchmarks confirm the enhanced performance of our proposed CEM-SAC. The source code is available at https://github.com/ELO-Lab/CEM-SAC.**

***Index Terms*—Reinforcement learning, Evolutionary computation, Cross-entropy method, Soft actor-critic, Policy search.**

I. Introduction

Reinforcement learning (RL) is a machine learning paradigm to train artificial intelligence (AI) agents to accomplish a certain task when there does not exist a dataset of training examples with corresponding ground truth labels. It is not straightforward, and might be not realizable, to effectively construct such a dataset in many control problems, such as robotics, inventory management, real-time bidding. Designated tasks often involve the optimization of a reward/objective function. RL algorithms train the agents via an iterative exploration-exploitation mechanism [1]. At each iteration, experiences, often in the form of (state, action, reward, next state), are obtained by interacting directly with the environment. These collected training signals are then employed to favorably adjust the trial-and-error interactions of the agents at subsequent iterations. In recent years, the couplings with deep neural networks (DNNs) and other deep learning (DL) techniques have enabled RL to achieve successful applications in a wide range of complicated problems with high-dimensional state spaces and action spaces. Policy neural networks, that recommend which action agents should perform given each input state, are often trained by a class of deep reinforcement learning (DRL) algorithms called policy gradient methods. Stochastic gradients of reward functions with respect to the policy parameters can be used to update network weights at each gradient ascent iteration.

DRL algorithms, however, still suffer from some essential obstacles: credit assignment with sparse/delayed reward signals, premature convergence due to difficulties in maintaining a meaningful diverse exploration, sensitivity to hyperparameter settings [2], [3]. Salisman et al. [4] proposed to treat the training of an agent's policy network as a black-box optimization problem, which can then be solved by evolution strategies (ES), in particular natural evolution strategies [5], with competitive results compared to state-of-the-art DRL algorithms. Notable advantages of evolutionary computation (EC) as a scalable alternative to RL have been reported in [4]: 1) EC methods can be easily parallelized when parallel computing resource is available; 2) EC manages to maintain a beneficial exploration, and the agents acquire novel behaviors that are not observed when being trained with policy gradient methods; 3) EC methods are typically robust, and competitive results can be obtained with the same fixed hyperparameters in multiple environments rather than having to perform intensive problem-specific hyperparameter tunings. However, zeroth-order optimization techniques like (classic) EC methods might have slow convergence speed and exhibit sample inefficiency when the problem dimensionality is large [6]. It is the case that DRL agents' policy neural networks often have a huge number of parameters that need to be trained. Recent studies have been made to combine the strengths of both DRL and EC methods to efficiently and reliably train policy networks in solving control problems [2], [3].

II. Background

The core of an agent is its policy (or controller) π that dictates which action $a_t \in \mathcal{A}$ to take given a state (or an observation) $s_t \in \mathcal{S}$ at each time step t. An instant reward r_t is fed back to the agent, and the environment is transitioned to the next state s_{t+1}. The sequence $\tau = (s_0, a_0, r_0, s_1, \ldots, s_t, a_t, r_t, s_{t+1}, \ldots)$ forms a trajectory of the agent through the state space up to a horizon $t = H$, and the corresponding return $R(\tau)$ can be obtained $R(\tau) = \sum_{t=0}^{H} \gamma^t r_t$, where $\gamma \in (0, 1]$ is a discount factor [7]. Because there typically exists certain randomness in the environment (e.g., transitions and rewards follow some probability distribution $\Pr(s_{t+1}, r_t | s_t, a_t)$), and the agent can have a stochastic

978-1-6654-6708-7/22/$31.00 ©2022 IEEE

policy (i.e., a random distribution over the action space conditioned on the state space $\pi(a_t|s_t) = \Pr(a_t|s_t)$), the agent's trajectories would be different from time to time. RL aims to train the agent to obtain the maximum expected return $J(\pi) = \mathbb{E}_{\tau\sim\pi}[R(\tau)]$.

The agent's policy can be parameterized by a parameter vector θ, and in DRL, θ would be the weight vector representing a deep neural network π_θ. The training process in RL is then equivalent to employing a search algorithm to find a policy parameter vector θ^* yielding an optimal policy π_{θ^*} that is able to maximize the expected return [7], [8]

$$\theta^* = \arg\max_\theta J(\pi_\theta) = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{1}$$

$$= \arg\max_\theta \sum_\tau \Pr(\tau|\pi_\theta)R(\tau) \tag{2}$$

where the expectation is computed by sampling, and $\Pr(\tau|\pi_\theta)$ indicates the probability of a trajectory τ given a policy π_θ. Next, we describe two popular families of policy search algorithms, policy gradient and evolutionary computation.

A. Policy gradient and soft actor-critic

Policy gradient methods [9], [10] are a family of RL algorithms that assume the policy gradient $\nabla_\theta J(\pi_\theta)$ can be computed as follows:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \sum_\tau \Pr(\tau|\pi_\theta)R(\tau) \tag{3}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log\pi_\theta(a_t|s_t)R(\tau)\right] \tag{4}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log\pi_\theta(a_t|s_t)Q^{\pi_\theta}(s_t,a_t)\right] \tag{5}$$

where $Q^{\pi_\theta}(s,a)$ is the action-value function of policy π_θ, which indicates the expected return that can be achieved if the agent performs action a in state s and then following policy π_θ thereafter [10]. The policy parameter vector θ can be updated via first-order optimizers such as gradient ascent or Adam [11]. Policy gradient methods are usually combined with Q-learning, where $Q^{\pi_\theta}(s,a)$ can be estimated by function approximation, giving rise to the actor-critic framework [12]:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log\pi_\theta(a_t|s_t)Q_\phi(s_t,a_t)\right] \tag{6}$$

where $Q_\phi(s_t,a_t)$ is typically a neural network parameterized by ϕ. Algorithms that belong to this family can be implemented in an *on-policy* setting (i.e., experiences sampled under policy π_θ are discarded after being used to approximate $\nabla_\theta J(\pi_\theta)$) [13], which is sample inefficient but offers a higher stability, or *off-policy* setting (i.e., experiences sampled under different polices are stored in a buffer and are later used to estimate $\nabla_\theta J(\pi_\theta)$) [12], [14], [15], which has a higher sample efficiency but sensitive to many hyperparameters [16]. Soft actor-critic (SAC) [17] is an off-policy algorithm that is built on the actor-critic framework. SAC introduces an entropy regularization term into the RL objective function and a policy should learn to maximize the following function:

$$\begin{aligned} J(\pi_\theta) &= \mathbb{E}_{\pi_\theta}[\sum_{t\geq 0}\gamma^t\Big(r_t + \alpha H(\pi_\theta(\cdot|s_t))\Big)] \\ &= \mathbb{E}_{\pi_\theta}[\sum_{t\geq 0}\gamma^t\Big(r_t - \alpha\log\pi_\theta(\cdot|s_t)\Big)] \end{aligned} \tag{7}$$

To this end, SAC simultaneously learns two Q-functions (Equation 9) and uses them to optimize the policy function (Equation 10). Policy function, or actor, $\pi_\theta(a_t|s_t)$ used in SAC is a stochastic actor [17], in contrast to the deterministic actor used by deterministic policy gradient methods such as DDPG [12] and TD3 [15]. While a deterministic actor directly estimates the action given a state vector, a stochastic actor maintains a Gaussian distribution over the action space, where its mean and covariance are given by the policy neural network. In every decision step t, a state vector is forwarded through the actor policy network, and the resulting distribution is used to sample a corresponding action vector to be carried out. The interaction experience, in the form of the environment feedback, (state s, action a, reward r, next state s'), is added to the replay buffer. At each update step, a mini-batch of experiences is randomly sampled from the buffer to be used for estimating gradient vectors. The pseudo-code for SAC is summarized in Algorithm 1.

Algorithm 1 Soft Actor-Critic [17], adapted from [18]

Input: initial parameters $\theta_{init}, \phi_1, \phi_2$, replay buffer $\mathcal{D}$
Initialize: actor π_θ with θ_{init}, two critics Q_{ϕ_1}, Q_{ϕ_2} with ϕ_1, ϕ_2, two target critics $Q_{\phi'_1}$, $Q_{\phi'_2}$ with parameters $\phi'_1 \leftarrow \phi_1$ and $\phi'_2 \leftarrow \phi_2$
for $t = 1$ to `max_steps` **do**
  Execute action $a_t \sim \pi_\theta(\cdot|s_t)$, observe next state s_{t+1}, and reward r_t
  Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  for $j = 1$ to `update_step` **do**
    Sample from $\mathcal{D}$ a mini-batch $\mathcal{B}$ of experiences (s_k, a_k, r_k, s'_k). For each (s_k, a_k, r_k, s'_k), if s'_k is a terminal state, then $y_k = r_k$. Otherwise, compute y_k:

$$y_k = r_k + \gamma\left(\min_{j=1,2} Q_{\phi'_j}(s'_k, a'_k) - \alpha\log\pi_\theta(a'_k|s'_k)\right) \tag{8}$$

    where $a'_k \sim \pi_\theta(\cdot|s'_k)$.
    Update critic parameters with 1 gradient descent step:

$$\nabla_{\phi_i}\frac{1}{|\mathcal{B}|}\sum_{k=0}^{|\mathcal{B}|}\left(Q_{\phi_i}(s_k,a_k) - y_k\right)^2 \text{ for } i = 1,2 \tag{9}$$

    Update actor parameters with 1 gradient ascent step:

$$\nabla_\theta\frac{1}{|\mathcal{B}|}\sum_{k=0}^{|\mathcal{B}|}\Big(\min_{i=1,2} Q_{\phi_i}(s_k,\tilde{a}_k) - \alpha\log\pi_\theta(\tilde{a}_k|s_k)\Big) \tag{10}$$

    where $\tilde{a}_k \sim \pi_\theta(\cdot|s_k)$.
    Update target critic parameters with the current critic parameters:

$$\phi'_i \leftarrow \rho\phi'_i + (1-\rho)\phi_i \text{ for } i = 1,2$$

  end for
end for

B. Cross Entropy Method (CEM)

Evolutionary algorithms (EAs) are metaheuristic search algorithms with operating mechanisms inspired by biological evolution processes in nature [19]. EAs consider each candidate solution of the optimization problem under concern as an individual in a population. Each individual is represented by a chromosome where each gene is associated with a decision variable. Regarding the context of policy search, an individual would be a specific policy parameter vector θ (of an actor network). The fitness of each individual, indicating the quality of the corresponding solution, is assessed through a fitness function, which is constructed based on the objective function of the optimization problem (e.g., sum of rewards in RL problems). A typical EA has two main operators: 1) *selection* to select a subset of elite individuals that have better fitness values than the others in the population and 2) *variation* to generate new offspring from currently existing individuals. The role of the selection operator is to maintain beneficial traits exhibiting in higher-fitness individuals of the current population so that these traits can be inherited to and further improved in future generations. The variation operator can be implemented via *crossover*, that recombines selected individuals (i.e., current candidate solutions) to create offspring (i.e., novel candidate solutions), and *mutation*, that randomly alters current individuals with a small probability. The exploration and exploitation of the search are governed via these selection and variation operators. Together, the population is drawn toward better regions in the solution space after each iteration.

Estimation-of-distribution algorithms (EDAs) are a family of EAs that explicitly represent the dependencies between decision variables via probability distribution models (e.g., Gaussian distributions, Bayesian networks, hidden Markov models) and exploit these dependencies to generate new solutions. Cross-entropy method (CEM [20]) can be considered as an EDA that employs a multivariate Gaussian parameterized by μ and Σ to encode the distribution of the current population. After every generation g, CEM generates new N individuals $\{\theta_i\}_{i=1}^{N}$ by sampling from $\mathcal{N}(\mu_g, \Sigma_g)$. All these individuals are evaluated regarding the fitness function F and a subset of N_e elite individuals with the highest fitness values are then selected $\{z_i\}_{i=1}^{N_e}$. This subset is used to update the parameters of the Gaussian distribution for the next generation as follows:

$$\mu_{g+1} = \sum_{i=1}^{N_e} w_i z_i \tag{11}$$

$$\Sigma_{g+1} = \sum_{i=1}^{N_e} w_i (z_i - \mu_{g+1})(z_i - \mu_{g+1})^T \tag{12}$$

where weight coefficients $w_i = \frac{1}{N_e}$. However, updating covariance matrix Σ_{g+1} according to Equation 12 tends to drastically reduce the population diversity, potentially leading to premature convergence. To fix this problem, Pourchot et al. [3] proposed the following modifications: 1) replacing the current mean μ_{g+1} with previous mean μ_g when updating Σ_{g+1}, 2) using weight coefficients $w_i = \frac{\log(N_e+1)-\log(i)}{\sum_{i=1}^{N_e}\log(N_e+1)-\log(i)}$, and 3) adding ϵ to diagonal elements of Σ_{g+1}. Besides, naively applying CEM to high-dimensional problems (e.g., policy search for DRL) would not be efficient owing to large memory requirements $\mathcal{O}(dim^2)$ and sampling complexity $\mathcal{O}(dim^{2.3})$. One workaround is to simplify Σ_g to be diagonal [3]. The pseudo-code for this CEM variant is in Algorithm 2.

Algorithm 2 CEM [3]

Hyperparameters: extra variance ϵ, $\sigma_{init} \in R_{>0}$
Input: `interaction_budget`, mean policy $\mu_0 \in \mathbb{R}^d$,
    population size N, number of elite policies N_e
Initialize: Covariance matrix $\Sigma_0 = \sigma_{init}\mathbf{I}_d$
    Weight coefficients $(w_i)_{i=1,\dots,N_e}$, where
$$w_i = \frac{\log(N_e+1)-\log(i)}{\sum_{i=1}^{N_e}\log(N_e+1)-\log(i)}$$
$g = 0$, `steps` $= 0$
while `steps` $<$ `interaction_budget` **do**
    Sample N policy parameter vectors $\theta_1, \dots, \theta_N$ from $\mathcal{N}(\mu_g, \Sigma_g)$
    Evaluate all policy parameter vectors on the control task.
    Increase `steps` by # interaction steps used in policy evaluations.
    Select a top subset of N_e policy parameter vectors $(z_i)_{i=1,\dots,N_e}$ with better performance scores.
    Estimate new mean policy parameter vector μ_{g+1} and new covariance matrix Σ_{g+1} using $(z_i)_{i=1,\dots,N_e}$

$$\mu_{g+1} = \sum_{i=1}^{N_e} w_i z_i$$

$$\Sigma_{g+1} = \left(\sum_{i=1}^{N_e} w_i \odot (z_i - \mu_g)^2\right) \odot \mathbf{I}_d + \epsilon \mathbf{I}_d$$

    $g \leftarrow g + 1$
end while

C. CEM-RL

Pourchot et al. [3] propose CEM-RL, a hybridization scheme to combine CEM with an off-policy DRL algorithms, and the performance of CEM-DDPG and CEM-TD3 have been investigated. In CEM-RL, a multivariate Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ over the policy parameters space is maintained and updated to fit the region where high-performing policy parameters locate. On each generation, a population of policy parameters is sampled by adding some Gaussian noises to the mean of CEM distribution. Half of these policy parameters in this population are then trained by the DRL algorithm, using gradient information estimated using experiences that are stored in a replay buffer. At the end of a generation, all policy parameters are evaluated for their corresponding performance scores (i.e., fitness values). CEM distribution parameters μ, Σ are re-computed from the $N_e = N/2$ top-performing policy parameters. The pseudocode of CEM-RL with CEM-TD3 version is given in Algorithm 3.

III. Proposed approach

While CEM-TD3 [3] is a state-of-the-art ERL algorithm, the soft actor-critic (SAC [17]) has been recently shown to achieve better results in multiple continuous control benchmarks compared to TD3. In this paper, we describe a modified CEM-RL hybridization scheme and then employ this framework to integrate CEM and SAC. Our proposed CEM-SAC utilizes the stability of the population-based nature of CEM and the efficiency of SAC gradient updates.

Algorithm 3 CEM-TD3 [3]

Input: `interaction_budget`, initial policy parameter μ, population size N
Initialize: Actor network π and target actor network π_t,
  Critic networks Q_1^π, Q_2^π and target critic networks $Q_{t,1}^\pi, Q_{t,2}^\pi$
  Covariance matrix $\Sigma = \sigma_{\text{init}}\mathbf{I}$ and mean μ, number elite $N_e = N/2$
  Interaction steps made by actors `actor_steps` $= 0$, replay buffer $\mathcal{D}$
while `total_steps` $<$ `interaction_budget` **do**
  Sample N policy parameter vectors $(\theta_i)_{i=1,\dots,N}$ from $\mathcal{N}(\mu, \Sigma)$
  for $i = 1$ to N_e **do**
    Set policy parameters of actor π and target actor π_t to θ_i
    Update Q^π using gradients estimated on $(2 \times$ `actor_steps`$)/N$ mini-batches drawn from $\mathcal{D}$ following TD3 update rule [15]
    Update π and π_t using gradients estimated on `actor_steps` mini-batches drawn from $\mathcal{D}$ following TD3 update rule [15]
    Set θ_i to parameters of π
  end for

  `actor_steps` $= 0$
  for $i = 1$ to N **do**
    Set policy parameters of actor π to θ_i
    (fitness f, #interaction steps s) $\leftarrow$ EVALUATE(π)
    Fill $\mathcal{D}$ with experiences from evaluating π
    `actor_steps` $=$ `actor_steps` $+ s$
  end for
  `total_steps` $=$ `total_steps` $+$ `actor_steps`
  Select top N_e policies to update distribution $\mathcal{N}(\mu, \Sigma)$ of CEM
end while

A. CEM-SAC

The idea of combining EC and RL is to maintain a population of candidate policy parameter vectors and iteratively improve the population based on the evolution mechanism of EC and the gradient-based policy update of RL. Given that our choice of EA is CEM and our choice of RL algorithm is SAC, CEM-SAC's population is encoded by a multivariate Gaussian $\mathcal{N}(\mu, \Sigma)$ over SAC's stochastic policies π_θ.

First, the policy and critic network architectures of SAC (e.g., the number of layers, the number of neurons in each layer) need to be chosen, and the parameters of CEM (μ and Σ) are initialized. In every generation, a *CEM population* of candidate policy parameter vectors are sampled from $\mathcal{N}(\mu, \Sigma)$ and evaluated to obtain their fitness scores f_i^{CEM}. Experience tuples (s_t, a_t, r_t, s_{t+1}) from the evaluation trajectories are stored in the replay buffer. The CEM population is then cloned to form a *SAC population*. For every policy parameter vector in the SAC population, a number of SAC training episodes (e.g., at most five training episodes in this paper) are performed one after another. Each SAC training episode is a complete execution of Algorithm 1 but the `max_steps` is limited to one episode, and the initial policy parameter vector θ_{init} is set as the one copied from the CEM population (in the first training episode) or the resulting policy from the preceding training episode. During this training episode, a trajectory of experiences (s_t, a_t, r_t, s_{t+1}) are collected when the agent interacts with the environment, the critics are updated according to Equation 9, and the actor (i.e., the policy) is updated according to Equation 10 using gradients estimated based on mini-batches $\mathcal{B}$ of experiences sampled from the replay buffer. After a training episode, the updated policy is assessed by executing one evaluation episode to obtain its new fitness score f_i^{RL}. If SAC's gradient-based updates yield an improvement, i.e., $f_i^{\text{RL}} > f_i^{\text{CEM}}$, the algorithm moves on to train the policy with the next policy parameter in the SAC population. Otherwise, if the budget of RL training steps for that policy is not used up yet, another SAC training episode is performed with θ_{init} set to the current policy parameter vector θ_i^{RL}. While gradient-based training is generally sample efficient, gradients in off-policy RL are computed using samples from the replay buffer, and might thus suffer from overestimation bias. We would like to prevent such excessive gradient-based policy updates.

At the end of every generation, CEM population (containing gradient-free policies) and SAC population (containing policies updated with gradient information) are merged together. Top $N_e = N/2$ policies with highest fitness values are selected to update parameters μ and Σ of CEM. The algorithm is terminated when the total budget of interaction steps with the environment is used up. Fig. 1 illustrates and Algorithm 4 provides a pseudo-code for CEM-SAC.

For SAC training episodes in our CEM-SAC framework, the hyperparameter `update_step`, i.e., the number of gradient-based policy updates per interaction step (line 6 in Algorithm 1), is set to 2 initially. When experience data fill 10% of the replay buffer, we let `update_step` $= 3$. Elitism is implemented for CEM here, i.e., we preserve the best performing policy from the previous generation.

Algorithm 4 CEM-SAC

Input: `interaction_budget`, `budget_per_actor`
Initialize: parameters of CEM (Algorithm 2) and SAC (Algorithm 1)
`total_steps` $= 0$
while `total_steps` $<$ `interaction_budget` **do**
  Sample policy parameter vectors $(\theta_i^{\text{CEM}})_{i=1,\dots,N/2}$ from $\mathcal{N}(\mu, \Sigma)$
  for $i = 1$ to $N/2$ **do**
    (fitness f_i^{CEM} , interaction steps `s`) $\leftarrow$ EVALUATE($\pi_{\theta_i^{\text{CEM}}}$)
    Fill $\mathcal{D}$ with experiences from evaluating $\pi_{\theta_i^{\text{CEM}}}$
    `total_steps` $=$ `total_steps` $+$ `s`
  end for
  $(f_i^{\text{RL}}, \theta_i^{\text{RL}}) \leftarrow (f_i^{\text{CEM}}, \theta_i^{\text{CEM}}), \quad i = 1, \dots, N/2$
  for $i = 1$ to $N/2$ **do**
    `steps` $= 0$
    while $f_i^{\text{RL}} \leq f_i^{\text{CEM}}$ **and** `steps` $\leq$ `budget_per_actor` **do**
      Update θ_i^{RL} and $Q_{i=1,2}^\phi$ (SAC training)
      Increase `steps`, `total_steps` by #interaction steps used in SAC training
      (fitness f_i^{RL} , #interaction steps `s`) $\leftarrow$ EVALUATE($\pi_{\theta_i^{\text{RL}}}$)
      `total_steps` $=$ `total_steps` $+$ `s`
      Fill $\mathcal{D}$ with experiences from training and evaluation
    end while
  end for
  Construct a population $\mathcal{P}$

$$\mathcal{P} \leftarrow \Big\{(f_1^{\text{CEM}}, \theta_1^{\text{CEM}}), \dots, (f_{N/2}^{\text{CEM}}, \theta_{N/2}^{\text{CEM}}), (f_1^{\text{RL}}, \theta_1^{\text{RL}}), \dots, (f_{N/2}^{\text{RL}}, \theta_{N/2}^{\text{RL}})\Big\}$$

  Select top $N/2$ policies from $\mathcal{P}$ to update $\mathcal{N}(\mu, \Sigma)$
end while

B. Comparison with previous works

The works that are closest to our CEM-SAC are CEM-RL [3] and ESAC [21]. While most ERL algorithms share

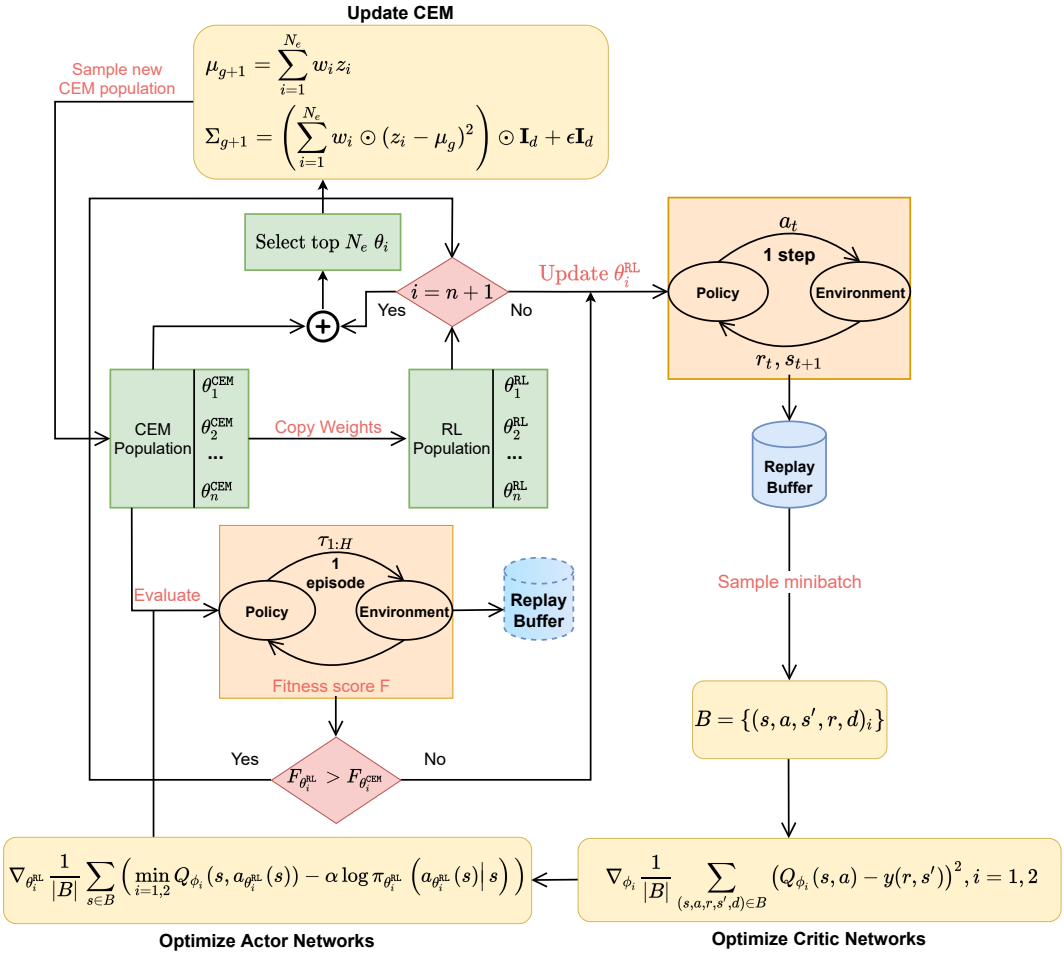

Fig. 1. Illustration of CEM-SAC algorithm

certain similarities in the idea of combining EC with RL to enhance both convergence stability and sample efficiency for policy search, CEM-SAC is different from the other two algorithms as follows.

Suri et al. [21] aim to combine SAC with evolution strategies (ES). In ESAC, ES and SAC work independently of each other. For each generation of ES, there is a probability that an actor (separated from ES population) receives update from the SAC training procedure. This actor is only added to the ES population at the end of each generation. On the other hand, in CEM-SAC, communication between CEM and SAC is much more frequent when SAC training processes are executed directly on policies sampled from CEM distribution.

CEM-SAC differs from CEM-TD3 in the way it updates the newly-sampled policies at the start of each generation. While CEM-TD3 assigns a fixed number of gradient updates to half of its CEM population every generation, we vary the number of SAC training episodes per policy. We compare the performance of an updated policy with its original policy sampled from CEM's Gaussian distribution to decide when to terminate the gradient-based updating process. This is based on our observation that excessive usage of gradient estimation information can introduce instability into the agent training process, hence worsens performance scores. Besides, CEM-TD3 only uses the update rules of TD3 to estimate gradients based on mini-batches drawn from the replay buffer, i.e., no interaction with the environment happens during lines 8-12 in Algorithm 3. CEM-TD3 interacts with the environment only when evaluating actors in the CEM population, and experience data generated during this phase are put into the replay buffer (lines 16-21 in Algorithm 3). In contrast, CEM-SAC runs actual SAC training episodes with environment interactions (line 15 in Algorithm 4). CEM-SAC saves to the buffer all experience tuples generated in every interaction of the learning agent with the environment (lines 8 and 19 of Algorithm 4).

IV. Experimental Setup

A. MuJoCo Benchmark

MuJoCo [22] is a simulation physics engine commonly used to construct benchmarks to evaluate RL agents. Table I lists six MuJoCo OpenAI Gym [23] locomotion tasks used in our experiments in the increasing order of the dimensions of state spaces and action spaces. These tasks have the same objective of the trained agents moving forward as far as possible using

their body parts. To take action in the simulated environments, agents are equipped with a policy that outputs a continuous action vector (-1 to 1) based on each input state vector. Scores are awarded as the agents move further from the starting point. It is possible to get negative scores when the agents fall to the ground or perform excessive actions.

TABLE I
MUJOCO OPENAI GYM ENVIRONMENTS [23]

Environment	State Dimensions	Action Dimensions
Hopper-v3	11	3
Walker-v3	17	6
HalfCheetah-v3	17	6
Ant-v3	111	8
Humanoid-v3	376	17
HumanoidStandup-v2	376	17

B. Implementation details

We use the source code of CEM-RL[1] to run the experiments for CEM and CEM-TD3. We implement our CEM-SAC using the Tianshou Library [24]. Results for SAC are obtained from the Tianshou library's data [2].

For SAC and CEM-SAC, we use two hidden layers with 256 nodes as a feature extractor. The output vector is then fed to 2 other hidden layers that represents the parameters μ and σ of an unbounded Gaussian distribution over the action space. Here we use the ReLU activation function after each hidden layers. For CEM and CEM-TD3, we mostly use the same settings as reported in [3]. The only difference is the feature extractor size is changed from (400,300) to (256, 256) to match with SAC and CEM-SAC. Note that TD3 uses a deterministic actor instead of a parameterized Gaussian distribution for stochastic policies as in SAC. Therefore, the output vector from CEM and CEM-TD3's feature extractor can be directly transformed into an action vector. Table II summarizes our hyperparameter settings.

TABLE II
HYPERPARAMETER VALUES

Hyper-parameter	CEM-SAC	SAC	CEM-TD3
Population size	10	N/A	10
Alpha	0.05	0.05	N/A
Batch size	256	256	100
Actor learning rate	$1e^{-}3$		
Critic learning rate	$1e^{-}3$		
Gamma	0.99		
Optimizers	Adam		
Buffer size	1000000		

C. Evaluation Protocol

In each environment, for each algorithm (i.e., CEM-SAC, CEM, SAC, and CEM-TD3), we execute 30 independent runs. Each run is provided with a computational budget of 10^6 interaction steps with the environment. For CEM, CEM-SAC, and CEM-TD3, at the end of every generation, we extract the current mean policy parameter vector μ from the Gaussian distribution maintained by CEM, and evaluate its performance by executing N_{eval} evaluation episodes. For SAC, after every 5,000 interaction steps, we extract the current policy π_θ and evaluate its performance by executing N_{eval} evaluation episodes. We use the average accumulated un-discounted reward (i.e., $R(\tau_i) = \sum_{t=0}^{T_i} \gamma^t r_{t,i}$ with $\gamma = 1$) as the performance metric, which can be computed as follows:

$$\texttt{average sum of rewards} = \frac{1}{N_{\text{eval}}} \sum_{i=1}^{N_{\text{eval}}} \sum_{t=1}^{T_i} r_{t,i} \quad (13)$$

where N_{eval} is the number of evaluation episodes ($N_{\text{eval}} = 10$ here), T_i is the total number of time steps in episode i, and $r_{t,i}$ is un-discounted reward returned at time step t in episode i. These performance results are then averaged over 30 independent runs for each algorithm in each environment.

V. RESULTS AND DISCUSSIONS

A. Results

Table III shows performance scores achieved by resulting policies obtained after training for one million interaction steps. We highlight the results that are statistically significant (Student's t-test with p-value $< \alpha = 0.05$). Experimental results show that our proposed CEM-SAC significantly outperforms both the constituent algorithms CEM and SAC in terms of final scores in all environment. CEM-TD3 performs slightly better than our CEM-SAC in the first environment Hopper-v3, but as the control tasks become more complicated with larger state spaces and action spaces, CEM-SAC obtains better policies with much higher performance scores.

Fig. 2 exhibits the learning curves of the four algorithms, where the x-axis and y-axis indicate the number of interaction steps and the simulation scores (i.e., average sum of rewards as in Eq. 13), respectively. The standard deviation of performance scores (over 30 runs) of CEM is considerably small compared to other algorithms, indicating the stability of the method; however, it does not manage to train an agent that can solve the control tasks within the budget of 10^6 steps. The learning curves of CEM here are similar with the results in [3]. Black-box (gradient-free) optimization methods might need up to $10^7 - 5.10^7$ interaction steps to solve these control tasks as reported in [25]. In contrast, SAC succeeds to obtain high-performance policies with the computation budget of 10^6 steps, but there exists a considerable variability during the training process of SAC. This lack of consistency results in large standard deviations of performance scores of the obtained policies. CEM-TD3, a hybridization between an EC method and a DRL algorithm, manages to reduce the typical instability of off-policy DRL algorithms. However, it does not maintain its performance when dealing with environments of large state spaces and action spaces like Ant-v3, Humanoid-v3, and HumanoidStandup-v2. We note that the flat learning curves of CEM-TD3 for Humanoid-v3 and HumanoidStandup-v2 here are similar to the one of TD3 when solving Humanoid-v1 as reported in [17].

[1] https://github.com/apourchot/CEM-RL
[2] https://github.com/thu-ml/tianshou

TABLE III
FINAL PERFORMANCE OF CEM-SAC, CEM-TD3, SAC, AND CEM.

	CEM-SAC (ours)		CEM-TD3		SAC		CEM	
Environment	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Hopper-v3	3543.060	303.548	**3687.163**	137.589	3406.688	419.595	1034.035	33.317
Walker2d-v3	**4893.221**	414.913	4492.048	377.041	4461.056	629.585	988.697	161.123
HalfCheetah-v3	**12393.059**	1353.351	10389.799	1003.154	11491.836	1798.925	2893.837	914.66
Ant-v3	**6280.306**	292.451	3745.099	907.568	5643.622	680.005	949.163	56.111
Humanoid-v3	**5779.767**	163.132	213.375	0.840	5415.815	734.711	213.522	0.846
HumanoidStandup-v2	**195598.457**	38089.379	29716.539	6.109	164047.701	30366.302	29716.976	5.367

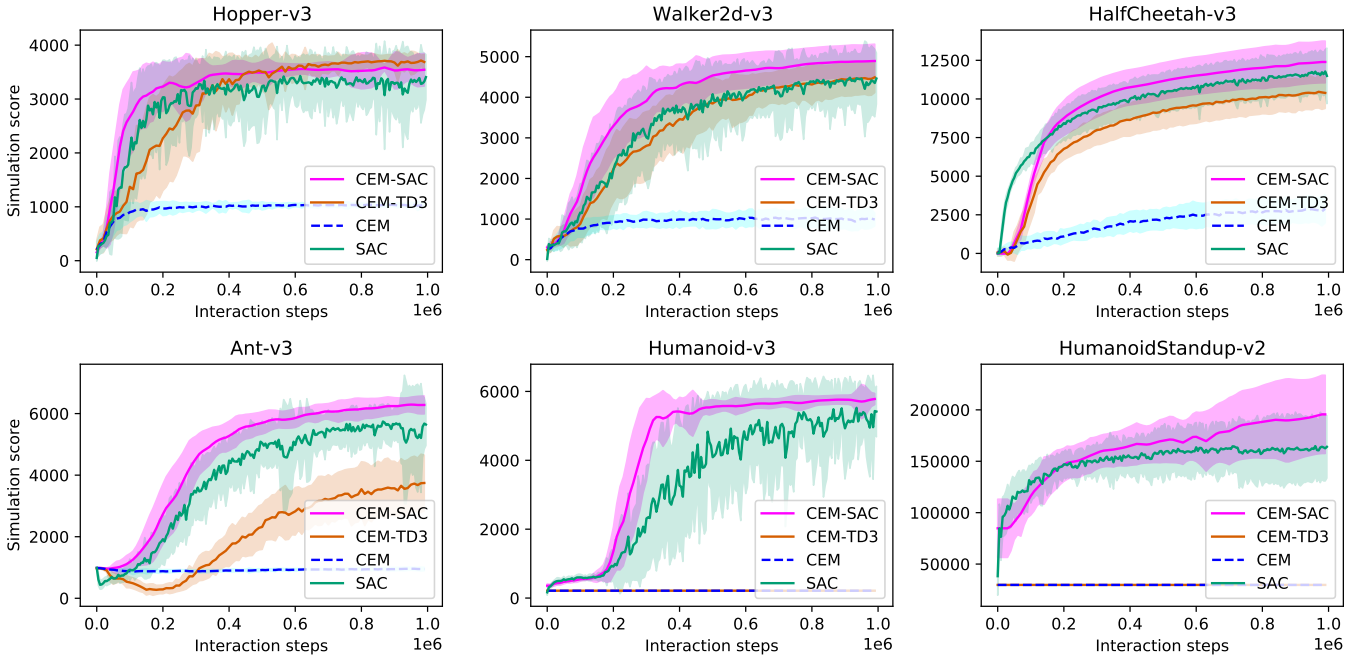

Fig. 2. Average performance results on MuJoCo-v3 environments of CEM-SAC, CEM-TD3, CEM, and SAC over 30 independent runs

Fig. 2 exhibits that our CEM-SAC is better than CEM-TD3 in combining the efficiency of DRL algorithms with the stability of EC methods. Control tasks with large action vector representation in environments with large state vector representation, i.e., Ant-v3, Humanoid-v3, and HumanoidStandup-v2, require agent policies that are more complex than others. While CEM-TD3 does not perform well in these environments, CEM-SAC still maintains its performance and stability as the problem complexity increases, indicating the scalability of CEM-SAC. Furthermore, CEM-SAC discovers competitive policies much earlier than the other algorithms, i.e., using fewer interaction steps, in almost all environments.

B. Discussions

To achieve top-performing results, it typically involves a lot of hyperparameter tunings, especially important ones such as learning rates, network architectures, gradient updates per interaction step, or population sizes, for each environment dedicatedly. However, our work here does not focus on achieving state-of-the-art performance, but on investigating a potential hybridization between CEM and SAC instead. We thus employ only one configuration of hyperparameters, obtained from CEM-TD3 in CEM-RL framework [3] and SAC in Tianshou library [24], for all experiments in this paper. Experimental results (in Fig. 2 and Tables III) show that CEM-SAC can work properly with one reasonable setting across all environments. Impacts of hyperparameter control and tuning on performance of ERL methods should be studied in future works.

Empirical results show that CEM-SAC efficiently learns top-performing policies for control problems with high-dimensional state spaces. Off-policy RL algorithms update policy parameters using gradients that are estimated based on experiences tuples sampled from the replay buffer. However, finding out the proper buffer size and the sampling strategy for each specific environment, which influence the quality of gradient estimations, is a non-trivial task. To mitigate detrimental effects when using sub-optimal gradient estimations, we need to frequently evaluate policies that are being updated by SAC (line 17 in Algorithm 4). CEM-SAC also controls the right amount of gradient-based policy updates on each generation with a simple condition on the fitness of the actors (line 14 in Algorithm 4) so that the policy search process does

not severely suffer from overestimation bias. Besides, similar to other ERL methods, CEM-SAC maintains high-performing policies and discards worse policies every generation (line 23 in Algorithm 4), thereby stabilizing the overall training. Compared to SAC, our CEM-SAC is much more stable and suffers less from sudden faulty gradient estimates that stem from approximation bias.

For future works, further improvements can be made as follows. First, CEM-SAC has not utilized the easily parallelizable nature of EAs, thus designing an efficient multi-threaded version of CEM-SAC would be a straightforward improvement. Second, CEM distribution parameters μ and Σ are updated only after all policies in the CEM population are evaluated with full episodes (i.e., lines 10 and 13 in Algorithm 2, lines 18 and 23 in Algorithm 3, lines 7 and 23 in Algorithm 4). A full unroll of an episode returns a final performance score, but might also be very long with many interaction steps, consuming a lot of the computation budget. It would be more efficient if computation techniques of partial unrolls (as in [26]) can be employed so that CEM distribution parameters μ and Σ can be updated with a faster rate that uses fewer interaction steps. Designing such efficient policy evaluation mechanisms to be used with CEM-SAC would be an important topic for future works.

VI. Conclusions

In this paper, we continue to build on the research line of employing EC methods to enhance the performance of DRL algorithms in policy search by introducing CEM-SAC, a hybridization between an estimation-of-distribution algorithm (i.e., CEM) and a policy gradient method (i.e., SAC). CEM-SAC encapsulates the working principle of CEM in maintaining a Gaussian distribution over promising candidate policy parameter vectors to promote meaningful diversity and exploration, together with a stable optimization process, while utilizing SAC's off-policy gradient-based updates that promote exploitation and sample efficiency. We calibrate a mechanism to exploit SAC's gradient updates to a beneficial extent for each policy in CEM's population. Experimental results on a wide range of simulated continuous control tasks support our claim that, compared to strong baselines such as SAC and CEM-TD3, our proposed CEM-SAC is capable of stabilizing the policy search process and achieving resulting policies with excellent performance within the same computational budget.

Acknowledgment

Ngoc Hoang Luong was funded by Vingroup JSC and supported by the Postdoctoral Scholarship Programme of Vingroup Innovation Foundation (VINIF), Vingroup Big Data Institute (VinBigdata), code VINIF.2021.STS.22.

References

[1] N. D. Nguyen, T. T. Nguyen, H. Nguyen, and S. Nahavandi, "Review, analyze, and design a comprehensive deep reinforcement learning framework," *CoRR*, vol. abs/2002.11883, 2020. [Online]. Available: https://arxiv.org/abs/2002.11883

[2] S. Khadka and K. Tumer, "Evolution-guided policy gradient in reinforcement learning," in *NeurIPS*, 2018.

[3] A. Pourchot and O. Sigaud, "CEM-RL: Combining evolutionary and gradient-based methods for policy search," in *ICLR*, 2019.

[4] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017.

[5] D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber, "Natural evolution strategies," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2008.

[6] N. Maheswaranathan, L. Metz, G. Tucker, D. Choi, and J. Sohl-Dickstein, "Guided evolutionary strategies: augmenting random search with surrogate gradients," in *ICML*, 2019.

[7] M. P. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," *Found. Trends Robotics*, vol. 2, no. 1-2, pp. 1–142, 2013.

[8] K. I. Chatzilygeroudis, V. Vassiliades, F. Stulp, S. Calinon, and J. Mouret, "A survey on policy search algorithms for learning robot controllers in a handful of trials," *IEEE Trans. Robotics*, vol. 36, no. 2, pp. 328–347, 2020.

[9] A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "A closer look at deep policy gradients," in *ICLR*, 2020.

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *ICLR*, 2016.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[14] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," in *ICLR*, 2017.

[15] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning (ICML)*, 2018.

[16] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, 2018.

[17] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *ICML*, 2018.

[18] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

[19] A. E. Eiben and J. E. Smith, *What Is an Evolutionary Algorithm?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 25–48.

[20] R. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodology and computing in applied probability*, vol. 1, no. 2, pp. 127–190, 1999.

[21] K. Suri, X. Q. Shi, K. N. Plataniotis, and Y. A. Lawryshyn, "Evolve to control: Evolution-based soft actor-critic for scalable reinforcement learning," *CoRR*, vol. abs/2007.13690, 2020. [Online]. Available: https://arxiv.org/abs/2007.13690

[22] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[23] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[24] J. Weng, H. Chen, D. Yan, K. You, A. Duburcq, M. Zhang, H. Su, and J. Zhu, "Tianshou: a highly modularized deep reinforcement learning library," *arXiv preprint arXiv:2107.14171*, 2021.

[25] K. Choromanski, A. Pacchiano, J. Parker-Holder, Y. Tang, and V. Sindhwani, "From complexity to simplicity: Adaptive es-active subspaces for blackbox optimization," in *NeurIPS*, 2019, pp. 10 299–10 309.

[26] P. Vicol, L. Metz, and J. Sohl-Dickstein, "Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies," in *ICML 2021*. PMLR, 2021, pp. 10 553–10 563.