Learning Complex Policy Distribution with CEM Guided Adversarial Hypernetwork

Shi Yuan Tang
Nanyang Technological University
Alibaba Group
shiyuan002@e.ntu.edu.sg

Athirai A. Irissappane
University of Washington
athirai@uw.edu

Frans A. Oliehoek
Delft University of Technology
f.a.oliehoek@tudelft.nl

Jie Zhang
Nanyang Technological University
zhangj@ntu.edu.sg

ABSTRACT

Cross-Entropy Method (CEM) is a gradient-free direct policy search method, which has greater stability and is insensitive to hyper-parameter tuning. CEM bears similarity to population-based evolutionary methods, but, rather than using a population it uses a distribution over candidate solutions (policies in our case). Usually, a natural exponential family distribution such as multivariate Gaussian is used to parameterize the policy distribution. Using a multivariate Gaussian limits the quality of CEM policies as the search becomes confined to a less representative subspace. We address this drawback by using an adversarially-trained hypernetwork, enabling a richer and complex representation of the policy distribution. To achieve better training stability and faster convergence, we use a multivariate Gaussian CEM policy to guide our adversarial training process. Experiments demonstrate that our approach outperforms state-of-the-art CEM-based methods by 15.8% in terms of rewards while achieving faster convergence. Results also show that our approach is less sensitive to hyper-parameters than other deep-RL methods such as REINFORCE, DDPG and DQN.

KEYWORDS

Cross-Entropy Method; Hypernetworks; Generative Adversarial Networks; Reinforcement Learning

ACM Reference Format:
Shi Yuan Tang, Athirai A. Irissappane, Frans A. Oliehoek, and Jie Zhang. 2021. Learning Complex Policy Distribution with CEM Guided Adversarial Hypernetwork. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), Online, May 3–7, 2021*, IFAAMAS, 9 pages.

1 INTRODUCTION

In Reinforcement Learning (RL), Cross-Entropy Method (CEM) [24] is a gradient-free optimization approach used to directly search in the policy space. Different from value-based and policy gradient-based approaches like Deep Q-Networks (DQN) [27], Asynchronous Advantage Actor-Critic (A3C) [26], and Trust Region Policy Optimization (TRPO) [33], CEM has several advantages [21, 32, 36]: 1) it is easy to implement; 2) the evolutionary nature of CEM in population selection leads to fast convergence and the sampling procedure is easy to parallelize for scalability; and 3) its training is more stable and insensitive to hyper-parameters. However, the performance of CEM is limited due to the inadequacy in policy distribution representation especially for large and complex problems.

To explain, CEM is similar in spirit to population-based search methods such as evolutionary algorithms, but rather than using a population it uses a distribution over candidate policies. This parameterized distribution, p, is iteratively updated: during every training iteration, a batch of policies are sampled from p and are ranked based on their episodic rewards (returns). Then the distribution p is updated towards *elite* policies (those with high return). While CEM has proven effective in many settings [19, 29, 36], a limitation in the way it is employed in RL settings lies in the representation of p: commonly, a natural exponential family distribution (NEF) such as a multivariate Gaussian is used. This, however, imposes severe constraints on the distributions that can be represented (e.g., only uni-modal), which limits the ability of p to properly guide the search. Mixture Gaussian [20] distribution can be used instead, but it requires user-defined assumptions on the number of modes.

In this paper, we address the restrictive policy representation issue typically observed in multivariate Gaussian CEM without imposing any prior assumptions on the number of modes. We use a hypernetwork [40] to represent the policy distribution, from which the parameters/weights for a main policy network are sampled. We optimize the policy distribution (hypernetwork) using an adversarial process [14]. Further, we use the elite policies sampled from a separate multivariate Gaussian CEM as supervision during adversarial training. This hybrid combination of CEM guided adversarial hypernetwork allows us to effectively learn the complex policy distribution (via hypernetwork), as well as achieve faster convergence and reduce hyper-parameter sensitivity (via guiding multivariate Gaussian CEM). Thus, our approach can achieve optimal results faster with limited tuning. We list our main contributions below:

- Our key innovation is in using the adversarially-trained hypernetwork architecture to address the issue of restrictive policy distribution in CEM. Hypernetworks can model complex multi-modal distributions [23, 28, 40] and provide better generalization by introducing uncertainty in policy network weights [1], thereby, more suitable to represent a policy distribution than multivariate Gaussian. Existing CEM-based deep-RL techniques [19, 29, 36] do not analyze the expressive power of CEM in representing the policy distribution and only focus on addressing the issues of

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, Online.

hyper-parameter sensitivity and training instability. Whereas, we aim to achieve both in our proposed work.

- In multivariate Gaussian CEM, the policy distribution updates are skewed heavily towards the first few elite policies during the early iterations, resulting in sub-optimal results. While one may wonder if the multivariate Gaussian CEM is fit to guide the hypernetwork, we address this issue by using a replay buffer to store *only* the elite policies from the guiding CEM and use them to train the hypernetwork. The elite policies are also carefully assessed before being added to the replay buffer to prevent premature sub-optimal convergence. It also improves sample efficiency [29] compared to discarding samples after every iteration.
- We provide a simple incremental approach to learn the policy distribution without explicitly learning a Q-function or using policy gradients which are highly sensitive to hyper-parameters. The guiding CEM helps to achieve stable and faster convergence.
- We conduct experiments on discrete and continuous action problems. Results shows that: 1) our approach enables richer policy representation, outperforming other state-of-the-art CEM-based approaches: Qt-Opt [19], Qt-Opt+DDPG [36] at least by 15.8% in rewards; 2) our approach achieves faster convergence than Qt-Opt and Qt-Opt+DDPG (62.9% less training time) and is less sensitive to hyper-parameters tuning than deep-RL methods such as REINFORCE [41], DDPG [22] and DQN [27].

2 RELATED WORK

In RL, the appeal of CEM-based direct policy search methods largely comes from its hyper-parameter insensitive, stability and highly parallelizable characteristics. Deep Q-learning methods [27] often suffer from hyper-parameter sensitivity and instability across different training runs. Techniques to address these issues have been proposed: [26, 39] reduce training instability to an extent; TRPO [33] and Proximal Policy Optimization (PPO) [34] reduce variance in policy gradient estimation; Deep Deterministic Policy Gradients (DDPG) [22], Twin Delayed Deep Deterministic Policy Gradients (TD3) [12], Soft Actor-Critic (SAC) [16] offer better training stability; however, all of them have more room for improvement [36].

Researchers have used CEM [24] together with deep-RL techniques to reduce hyper-parameter sensitivity and training instability. CEM-RL [29] combines CEM and TD3. Qt-Opt [19] is a CEM guided Q-learning method for continuous actions. In Qt-Opt, there is no policy being learned directly and the role of CEM is only for action selection based on the estimated Q-value. Qt-Opt+DDPG [36] extends Qt-Opt by learning a policy to approximate the CEM action selection process for reducing inference time. All of them use multivariate Gaussian to represent the CEM policy distribution. Our method shares the idea of using CEM to guide the training process from Qt-Opt, but, we use hypernetwork instead of multivariate Gaussian. Unlike Qt-Opt and Qt-Opt+DDPG, our method does not require a Q-function estimator as we update in the policy space directly. Further, instead of using CEM for action selection as in Qt-Opt, we use it to search the policy parameter space.

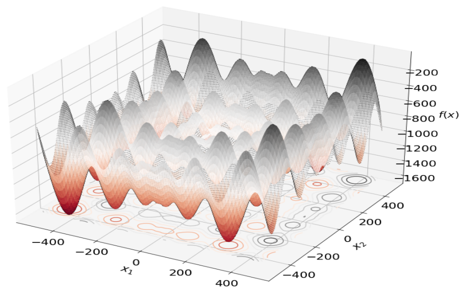
Multivariate Gaussian CEM policies tend to be easily trapped in local optima due to their inability to handle multiple modes and updates tend to skew towards lucky episodes during early iterations. Some works aim to suppress this overly greedy and optimistic update behaviour in CEM, such as applying noisy CEM [37] and introducing a smoothing parameter while updating the distribution [6]. Another stream of work focuses on addressing the fundamental bottleneck of representing and maintaining a complex parametric distribution [20]. They use a multi-modal CEM with mixture Gaussian models to alleviate the influence of elites in the first few iterations. However, such multi-modal Gaussian mixture may not outperform its uni-modal counterpart at all times [13], as additional assumption on the number of components in the mixture is needed. In our approach, we use hypernetwork (without assumptions on modality) to represent the complex policy distribution.

Adversarially-trained hypernetworks were used for image classification [18], where the hypernetwork generates weights for multiple classification models. In our work, we use an adversarially-trained hypernetwork to generate weights for policy network. However, unlike the supervised image classification problem, in our case, there is no ground-truth available and we rely on elite samples from a separate multivariate Gaussian CEM to guide the training process.

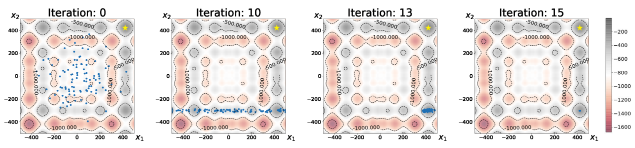
3 MULTIVARIATE GAUSSIAN LIMITATION

Existing theory on CEM applied in combinatorial optimization problems [5, 25, 31] show that CEM can find the optimal solution when sampling distribution converges to a unit mass with probability of 1. These works use the Bernoulli distribution, which can be viewed as a special case of bi-modal or joint distribution of two different Dirac delta functions [2] to sample discrete binary solutions. CEM-based approaches [19, 29, 36] in RL, however, use *uni-modal* multivariate Gaussian distribution while searching for policy parameters. The constrain of parametric distribution becomes more pronounced when the parameters are continuous, for example, neural network policies whose parameters, i.e., weights are continuous. In addition to the restrictive representation, the rare-event probability maximization nature of CEM updates causes the policy distribution to skew heavily towards the first few elite policies determined during the early iterations. This behaviour quickly limits the search space to a much smaller neighbourhood, allowing CEM to be easily trapped in a local optimum, mainly due to the lack of exploration and the inability to handle multiple modes [11].

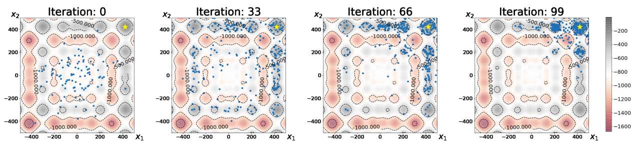
To illustrate these shortcomings, a maximization problem is performed on (the negative of) Schwefel function [35] using multivariate Gaussian CEM. The function $f(x)$ is shown in Fig. 1a with parameters x_1 and x_2, where black regions represent maxima and dark red regions represent minima. $f(x)$ is non-convex and complex, with multiple geometrically distant local maxima. Fig. 1b shows the contour plots across different CEM update iterations. The global optimum is marked with yellow star and the blue dots represent sampled values from the distribution. The (uni-modal) multivariate Gaussian distribution quickly collapses into a narrow distribution along the x_2-axis during the 10th iteration. This quickly eliminates any further exploration along x_2, and subsequently converges to a point mass distribution at a local optimum within 15 iterations. Further, multivariate Gaussian assumes i.i.d (independent and identically distributed) parameters, implying that the convergence favours a parameter value which leads to high $f(x)$ score



(a) Schwefel Objective Function



(b) Uni-modal Multivariate Gaussian



(c) Multi-modal Hypernetwork

Figure 1: Learning the Schwefel Objective Function using Multivariate Gaussian CEM Vs Hypernetworks CEM-AH

independent of other parameters. In Fig. 1b, this means that the relatively high $f(x)$ scores when $x_1 \in [375, 425]$ and $x_2 \in [-200, 200]$ will be eliminated early from candidate solutions and will not be explored. The low $f(x)$ scores from $x_1 \in [-425, -375]$ decreases the favourability of selecting $x_2 \in [-200, 200]$.

Fig. 1c shows the contour plots when using hypernetwork (our proposed approach CEM-AH) instead of multivariate Gaussian. Our approach does not require assumptions on the number of modes or the underlying probability distribution. We see that multiple modes can be maintained. Across different iterations, the sampled values are concentrated near different local maxima and slowly move towards the global maximum (yellow star) by iteration 99.

4 METHODOLOGY

We first describe our guiding multivariate Gaussian CEM in Sec. 4.1. We then explain our adversarially-trained hypernetwork and how CEM[1] is used to guide the adversarial training process in Sec. 4.2.

4.1 Guiding Cross-Entropy Method (CEM)

CEM tries to optimize the policy's parameters towards those with higher episodic rewards. Each policy parameter is sampled from a distribution and it is customary that these distributions belong to the same family (we collectively call the set of distributions as policy distribution). CEM searches the policy's parameter space by sampling multiple policies from these distributions. During CEM update, these distributions are updated towards the elite policies, i.e., policies with higher episodic rewards. Specifically, we use neural network policies, whose parameters[2] are continuous in nature.

The policy parameters represented using $\boldsymbol{\phi} \subset \mathbb{R}^d$ is a vector of dimension $d = |s| \times |a|$, where $|s|$ and $|a|$ denote the dimensions of state and action spaces. The parameters are sampled from a multivariate Gaussian distribution, $\boldsymbol{\phi} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mu, \Sigma)$, where μ and Σ represent the mean and covariance matrices. We assume Σ to be a diagonal matrix, signifying independence between each uni-modal Gaussian. Every CEM policy $\pi^{\boldsymbol{\phi}}_{\text{CEM}}$ is evaluated based on the episodic rewards obtained using the policy. After ranking the policies, the elite policies $\boldsymbol{\pi}_{\text{CEM[elite]}}$ are identified by choosing the top ρ percentile (or the elite fraction) of the sorted list of policies, based on which the multivariate Gaussian distribution is updated.

Algorithm 1 shows the detailed CEM training process. We first initialize a d-dimensional Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ with zero mean and unit covariance. At each iteration i (Line 2), the parameters $\boldsymbol{\phi}_i$ are sampled from the policy distribution and added to the population (Line 3). For each policy $\pi^{\boldsymbol{\phi}_i}_{\text{CEM}}$, corresponding to the parameters $\boldsymbol{\phi}_i$, we sample actions and obtain rewards until termination (Lines 4-7). The function $\pi^{\boldsymbol{\phi}_i}_{\text{CEM}}$ is a non-linear combination of weights and the corresponding state s, analogous to a neural network. For a given state s, $\pi^{\boldsymbol{\phi}_i}_{\text{CEM}}$ outputs continuous values. Depending on the implementation setup, the output could be interpreted as control values for all actions (in continuous control tasks) or probability distribution over actions (stochastic policy for discrete actions). Thus, the policy can be adapted for both discrete and continuous actions tasks (see experiments in Sec. 5).

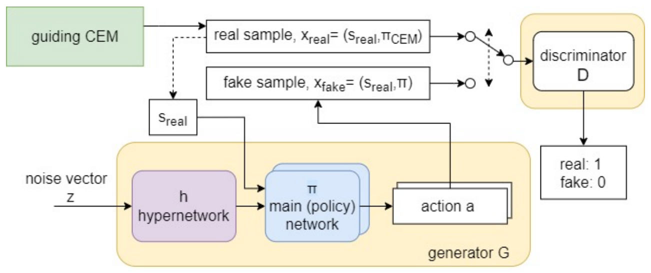
The total episodic reward R_i for the policy $\pi^{\boldsymbol{\phi}_i}_{\text{CEM}}$ is recorded (Line 7). A total of N episodes are sampled, each corresponds to a different policy. The policies are sorted based on $R \in \mathcal{R}$ and the ρ top-percentile are marked as elite policies (Lines 8-9). The policy distribution is updated based on the mean and covariance of

Algorithm 1: Cross-Entropy Method (CEM)

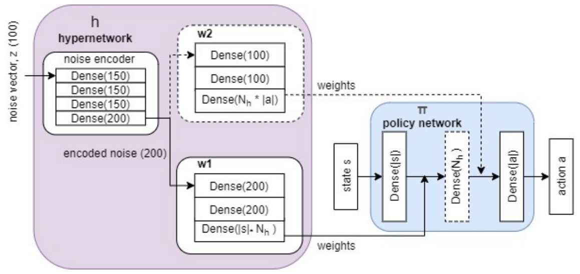
```
Input: population size $N$, elite fraction $\rho$, CEM policy $\pi^{\boldsymbol{\phi}}_{\text{CEM}}(a|s)$
Initialize: $\boldsymbol{\mu} \leftarrow 0^d, \Sigma \leftarrow diag(1)^{d\times d}$
1  Population $\Phi = \{\}$
2  for $i = 1 \rightarrow N$ do
       /* Sample policy parameters $\boldsymbol{\phi}_i$ */
3      $\boldsymbol{\phi}_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, Append $\boldsymbol{\phi}_i$ to $\Phi$
       /* Sample episodes following policy $\pi^{\boldsymbol{\phi}_i}_{\text{CEM}}$ */
4      while not terminal state do
5          $a \leftarrow \pi^{\boldsymbol{\phi}_i}_{\text{CEM}}(s)$
6          Observe $s'$ and obtain reward $r$
7          $R_i += r$
   /* Filter the elite policy parameters */
8  $I \leftarrow \{\text{sort}(\mathcal{R})_i : i \in [1, \ldots, k]\}, k = \rho$ top-percentile
9  $\boldsymbol{\pi}_{\text{CEM[elite]}} = \{\pi^{\boldsymbol{\phi}_i}_{\text{CEM}}, i \in I\}$
   /* Update CEM policy using elite policy parameters */
10 $\boldsymbol{\mu} \leftarrow \frac{1}{k}\sum_{i\in I}\boldsymbol{\phi}_i$
11 $\Sigma \leftarrow \text{diag}(\text{var}_{i\in I}(\boldsymbol{\phi}_i))$
```

[1]We refer to the guiding multivariate Gaussian CEM (Alg. 1) as CEM and our proposed approach as CEM-AH.

[2]Since we are referring to the context of neural networks, we use the terms "parameter" and "weight" interchangeably in the paper.



(a) CEM-AH Architecture



(b) Weights Generation by Hypernetwork

Figure 2: CEM-AH: Adversarially-Trained Hypernetwork

elite policies (Lines 10-11). While Algorithm 1 shows the training process, during evaluation/testing, we sample one set of parameters ϕ_{test} per episode, and use the policy $\pi_{\text{CEM}}^{\phi_{test}}$ for action selection.

4.2 CEM Guided Adversarial Hypernetwork (CEM-AH)

Due to the uni-modal assumption in multivariate Gaussian, the CEM approach (Sec. 4.1) heavily constrains the policy distribution representation. Further, the rare-event probability maximization nature of CEM updates is overly greedy and optimistic in favour of lucky episodes. These often result in CEM being heavily influenced by elite policies during early iterations, amplifying the likelihood of converging to different local optima across different runs.

Instead of multivariate Gaussian, we use a hypernetwork [15] to represent a complex multi-modal policy distribution. The main idea is to use a neural network (hypernetwork), to generate weights for another network. In our approach, the hypernetwork is used to generate weights for a main (policy) network, bearing analogy to sampling a policy from a distribution. Hypernetwork can learn high-dimensional distributions mapped from a lower-dimensional weight space [7, 38] and can be used to model complex multi-modal weight posteriors [23, 28, 40]. Such flexibility allows us to maintain multiple modes in the policy distribution across updates, thereby reducing the influence of lucky episodes during the early iterations and encouraging more exploration. Also, adopting such a generative hypernetwork model enables the use of ensembles during testing, where we use a collection of policies which are sampled to obtain a reliable action. Ensembles have been widely shown to provide stabilizing effect and reliable performance gain [8, 10, 17].

We adopt an adversarial training process for the hypernetwork. Unlike typical Generative Adversarial Network (GAN) [14] training, the challenge in RL setting is the unavailability of ground-truth. To address this, we use the actions from multivariate Gaussian CEM elite policies described in Sec. 4.1 as supervision to guide the training. This avoids learning Q-values or using policy gradients, both of which require extensive hyper-parameter tuning. Thereby, the proposed CEM guided Adversarial Hypernetwork (CEM-AH) enables faster and stable convergence and is less sensitive to hyper-parameters while retaining richness in policy distribution.

The architecture for our proposed CEM-AH is shown in Fig. 2a. CEM-AH is implemented as a GAN, consisting of generator G and discriminator D components. The generator consists of two neural networks, the hypernetwork and the main policy network. The hypernetwork, denoted by $h(\theta|z;\alpha)$ is conditioned by Gaussian input noise z along with network parameters α, and represents the policy distribution. It generates the network weights θ for the (main) policy network, denoted by $\pi(a|s;\theta)$. The policy network in turn predicts the action probabilities for a given state s.

Fig. 2b shows how the hypernetwork $h(\theta|z;\alpha)$ generates the weights θ for a policy network π. The hypernetwork consists of a noise encoder (with 4 dense layers) which takes an input Gaussian noise vector z of size 100. The output of the noise encoder is then passed through two weight-generators ($w1$, $w2$) which generate the weights for the policy network π. The policy network π consists of an input layer (with state space dimension $|s|$ nodes), a hidden dense layer with $N_h = 200$ nodes and an output layer (with action space dimension $|a|$ nodes). All layers are followed by RELU activation except for the last layer which uses tanh activation. $w1$ generates weights for the connections between the input layer and the hidden layer and $w2$ generates weights for the connections between the hidden layer and the output layer. The policy network π is then used to infer action probabilities for a given state s. We conduct experiments (see Fig. 6a and 6c) using 2 neural network architectures in different sizes: a 3-layer policy network as shown in Fig. 2b as well as a 2-layer policy network, for which we remove the hidden layer in the policy network as well as $w2$ in the hypernetwork. We then choose the architecture which gives the best results in Sec. 5.

The training data for CEM-AH is a set of tuples of the form $x = \langle s, a \rangle$. The discriminator $D(x;\beta)$, with parameters β is tasked to identify whether the tuple x is real or fake. CEM policy π_{CEM} (see Sec. 4.1) is used to guide the training of CEM-AH. For this, real tuples are generated using Algorithm 1, i.e., $x_{real} = \langle s, \pi_{\text{CEM}}(s) \rangle$. The fake tuples are produced using randomly sampled policies from the hypernetwork $X_{fake} = \langle s, \pi(s; h(z)) \rangle$ with the same set of states used to generate the real tuples. Note that for a given state s, both x_{real} and x_{fake} will be generated and used during training.

4.2.1 Training Procedure for CEM-AH. Algorithm 2 shows the training routine of CEM-AH. As a part of this training process, we sequentially train the guiding CEM and CEM-AH. During every training iteration (Line 3), we first learn and update our guiding CEM policy, π_{CEM} (Line 4) by running Algorithm 1 for a given set of initial states. During this process, we create several real tuples $\langle s_{real}, \pi_{\text{CEM}}(s_{real}) \rangle$ using the states s_{real} observed and actions sampled $\pi_{\text{CEM}}(s_{real})$. These real tuples are sampled only from the elite

Algorithm 2: CEM-AH

```
Require: hypernetwork $h(\theta|z;\alpha)$, policy network $\pi(a|s;\theta)$,
         discriminator $D(x;\beta)$, standard Gaussian $\mathcal{N}_z(0,1)$
Initialize: $\alpha$, $\beta$
Input: NTrain, $M$, $\eta$, G-MSE, D-Adv, G-Adv
/* Main Function to train the hypernetwork */
1  Function Main:
2      $\Phi_{buf} = \{\}$
3      for $t = 1 \rightarrow NTrain$ epochs do
4          Update $\pi_{\text{CEM}}$ using Algorithm. 1
           /* Add high reward CEM real tuples to replay buffer */
5          if $\mathcal{R}_i >= \mathcal{R}_{\Phi_{buf}}$ then
6              $\Phi_{buf} = \{\Phi_{buf} \cup \langle s_{real}, \pi_{\text{CEM}}(s_{real})\rangle\}$[:capacity]
           /* Pre-train generator */
7          for G-MSE epochs do
8              Sample a set of $S_{real}$ states from $\Phi_{buf}$
9              $X_{real} \leftarrow \{\langle s_{real}, \pi_{\text{CEM}}(s_{real})\rangle, \forall s_{real} \in S_{real}\}$
10             $X_{fake} \leftarrow$ GenerateFake$(S_{real}, M)$
11             $L_{pre} \leftarrow$MSE$(X_{real}, X_{fake} | s_{real} \in S_{real})$
12             $\alpha \leftarrow \alpha - \eta\nabla L_{pre}$
           /* Adversarial Training for Discriminator */
13         for D-Adv epochs do
14             Sample a set of $S_{real}$ states from $\Phi_{buf}$
15             $X_{real} \leftarrow \{\langle s_{real}, \pi_{\text{CEM}}(s_{real})\rangle, \forall s_{real} \in S_{real}\}$
16             $X_{fake} \leftarrow$ GenerateFake$(S_{real}, M)$
17             $\nabla L_D \leftarrow \mathbb{E}_{real}\nabla_\beta[\log D_\beta(X_{real})]$
                   $+ \mathbb{E}_{fake}\nabla_\beta[\log(1 - D_\beta(X_{fake})]$
18             $\beta \leftarrow \beta - \eta\nabla L_D$
           /* Adversarial Training for Generator */
19         for G-Adv epochs do
20             Use $X_{fake}$ generated during discriminator training
21             $\nabla L_G \leftarrow \mathbb{E}_{fake}\nabla_\alpha[-\log(D_\beta(X_{fake})]$
22             $\alpha \leftarrow \alpha - \eta\nabla L_G$
23     return
/* Function to generate fake tuples using hypernetwork */
24 Function GenerateFake($S_{real}$, $M$):
25     Split $S_{real} = \{S^i_{real}, i = 1, \dots, M\}$
26     $X_{fake} = \{\}$
27     for $i = 1 \rightarrow N$ do
28         $i = 1, \dots, M$
29         $z_i \leftarrow$sample noise, $z_i \sim \mathcal{N}_z(0,1)$
30         $X_{fake} \leftarrow X_{fake} \cup \langle s_{real}, \pi_i(s_{real}; h(z_i))\rangle, \forall s_{real} \in S^i_{real}$
31     return $X_{fake}$
```

policies $\boldsymbol{\pi}_{\text{CEM[elite]}}$ with high episodic rewards (top ρ percentile of policies). These real tuples are then added to a replay buffer Φ_{buf} (Line 5-6). It is to be noted that, for a given train iteration t, the real tuples are added to the replay buffer only when the mean reward of the elite policies $\boldsymbol{\pi}_{\text{CEM[elite]}}$ used to sample these tuples is greater than or equal to mean reward of the policies for the tuples already present in Φ_{buf}. Such a selective addition of real tuples provides CEM-AH a diverse buffer of real (elite) tuples to learn from even when the guiding CEM converges to a narrow policy (and/or a local optima). Using a replay buffer also improves sample efficiency by re-using previously generated elite tuples without discarding them immediately. Φ_{buf} follows first-in-first-out ordering by removing old tuples when its maximum capacity is reached.

We pre-train the generator in Lines 7-12. For this, we sample a set of real states S_{real} from Φ_{buf} (Line 8) from which we create a set of real tuples X_{real} using the guiding CEM policy π_{CEM} (Line 9). We also create a set of fake tuples X_{fake} (Line 10). To generate fake tuples, we use the function GenerateFake (Lines 24-31). Here, we divide the set of real tuples S_{real} into M sets. We create M different policy networks $\pi_i, i = 1, \dots, M$ by feeding M different noise vectors z_i to the hypernetwork $h(z_i)$. Each policy network π_i generates fake tuples $\langle s_{real}, \pi_i(s_{real}; h(z_i))\rangle$ for a set of S^i_{real} states. Once the fake tuples are generated, we use Mean Squared Error (MSE) loss between the real and corresponding fake tuples, i.e., the error between $\pi_{\text{CEM}}(s_{real})$ and $\pi(s_{real}; h(z))$ to update the generator parameters with learning rate η (Lines 11-12). Pre-training the generator every time the replay buffer Φ_{buf} changes after updating the guiding π_{CEM} stabilizes the adversarial training process, as demonstrated by our experimental results in Fig. 6h.

During adversarial training, the discriminator D and generator G are trained one after the other. The discriminator (Lines 13-18), parameterized with β is trained using the cross-entropy loss (Eqn. 1) which maximizes the log probability for categorising the corresponding real tuples X_{real} and fake tuples X_{fake} (generated using Lines 24-31). For stable convergence, the discriminator is updated more times than the generator, i.e., D-Adv > G-Adv during initial training. We also add a decaying noise to discriminator input to improve its generalization capability [30]. The generator is trained (Lines 19-22) using the loss L_G (Eqn. 2), which is based on how well the discriminator identifies real and fake tuples.

$$\nabla L_D \leftarrow \mathbb{E}_{real}\nabla_\beta[\log D_\beta(X_{real})] + \mathbb{E}_{fake}\nabla_\beta[\log(1 - D_\beta(X_{fake})] \quad (1)$$

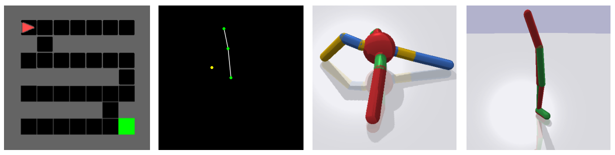
$$\nabla L_G \leftarrow \mathbb{E}_{fake}\nabla_\alpha[-\log(D_\beta(X_{fake})] \quad (2)$$

The attentive reader may wonder how the multivariate Gaussian π_{CEM} is fit to guide CEM-AH, as it suffers from drawbacks mentioned earlier. The reason is that the replay buffer Φ_{buf} is only filled with global elite samples from $\boldsymbol{\pi}_{\text{CEM[elite]}}$ encountered throughout the training process when the distribution is being updated. We reiterate that the generative hypernetwork $h(\theta|z;\alpha)$ learns a mapping from a sampled noise vector $z_i \sim \mathcal{N}_z(0,1)$ to a target policy distribution such that the sampled policies produce optimal actions (equivalent to those suggested by $\boldsymbol{\pi}_{\text{CEM[elite]}}$). This allows multiple high reward policies to be captured, or in other words, it learns multiple ways of accomplishing a task. Such an approach is different from the uni-modal guiding CEM (Algorithm 1), which learns a single policy (or single way) towards convergence. Learning multiple high reward policies in CEM-AH allows more robust policy behaviour by using a bagging ensemble method during testing.

4.2.2 *Testing (Evaluating) CEM-AH policy.*

For evaluating the learnt CEM-AH policy, we will only use the generator component of CEM-AH after it is trained using Algorithm 2. A batch of K policies is randomly sampled $\Pi_i = \{\pi_i(a|s;\theta_i); \theta_i = h(z_i), i = 1, \dots, K\}$ by feeding K random noise vectors $z_i, i = 1, \dots, K$ to the hypernetwork h of the generator. The final policy is obtained by implementing an ensemble of the K policies, $\pi_{\text{CEM-AH}} = ensemble(\Pi_i)$. We define $ensemble(\Pi_i)$ as $softmax[\sum_i \pi_i(a|s)]$ for discrete action problems and $\frac{1}{M}\sum_i \pi_i(a|s)$ when continuous actions are considered. While we use simple averaging as the ensemble method, more sophisticated techniques such as weighted average and ensemble networks could be used. The spirit is to show the complementary benefit of learning a distribution of policies, which could be sampled cheaply and be collectively used to boost the test performance.

5 EXPERIMENTS



(a) Maze (b) Reacher (c) Ant (d) Walker

Figure 3: Environments

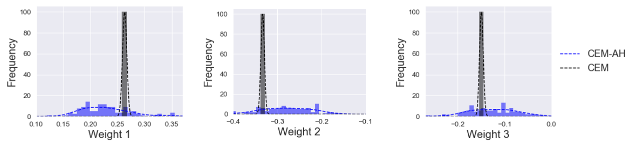
We conduct experiments on discrete and continuous action problems. For discrete actions, we use a fully observable maze [3] with sparse rewards setting. The maze is a 7×7 grid as shown in Fig. 3a, with state space dimension of 243 and action space dimension of 3. The agent in red is required to reach the goal (shown in green). The state encodes information about the presence of a wall, empty space, presence of the agent/goal, color and agent's orientation on different channels of the image grid. The agent's actions include moving forward, turning left and right, each having a small negative reward of -0.1. Reaching the goal gives a large reward of 1000. We set the maximum time-steps per episode to be $10,000$.

For continuous actions, we use reacher and two environments (ant, walker2d) from Pybullet-Gym [9][3], all of which have dense rewards. For the reacher environment, a robotic arm with 2 joints is centered in the environment and the goal is to move the end-effector (top end of the arm) to reach the yellow target (shown in Fig. 3b). The reward is calculated using the inverse distance between end-effector and target at every step, with a maximum of 100 time-steps per episode. The state space dimension is 8, consisting of x, y coordinates of end-effector's initial and target locations. The action space dimension is 2, representing a continuous angle value for the 2 joints. For the ant and walker environments, we use Pybullet-Gym's default settings with maximum of 1000 time-steps per episode. Both having the same objective of controlling an agent to walk as fast as possible. Ant environment's state and action dimensions are 28 and 8 respectively. For walker, the dimensions are 22 and 6 respectively.

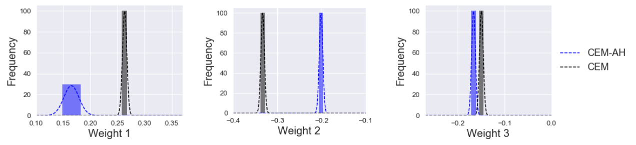
We compare CEM-AH with the guiding multivariate Gaussian CEM (Algorithm 1), state-of-the-art CEM-based approaches Qt-Opt [19], Qt-Opt+DDPG [36]. Other baseline RL techniques for both discrete and continuous problems such as REINFORCE (implemented using a policy network) [41], DDPG [22], DQN [27] are also selected to compare hyper-parameter sensitivity and stability. We run all the experiments for $50k$ episodes. For fair comparison, we use a 3-layer (1 input, 1 hidden, 1 output) or 2-layer (1 input, 1 output) neural network architecture, similar to the policy network architecture shown in Fig. 2b for all the approaches, whichever gave better results during hyper-parameter tuning (Sec. 5.2). For DDPG, Qt-Opt and Qt-Opt+DDPG, which are typically used for continuous action problems, a softmax activation is applied to the outputs in maze (discrete). We do not apply DQN on the continuous tasks (reacher, ant and walker). For CEM, N=100, ρ=90, for CEM-AH, M=100, K=10, $|\Phi_{buf}|$=10^8 and $|S_{real}|$=$5k$.

We measure the mean episodic rewards during training as well as the test performance (averaged over 10 episodes) of the learned policy after every training iteration. The purpose is to compare the actual quality of the learned policy and the rate of convergence, since most algorithms have different exploration and exploitation strategies. For example, REINFORCE and DQN use epsilon-greedy strategy for exploration during training, while they use absolute greedy policy during testing; DDPG and Qt-Opt+DDPG use a Q-network with output noise for exploration during training while a separate actor network is used during testing.

5.1 Performance Comparison of CEM-AH[4]



(a) 2k Episodes



(b) 50k Episodes

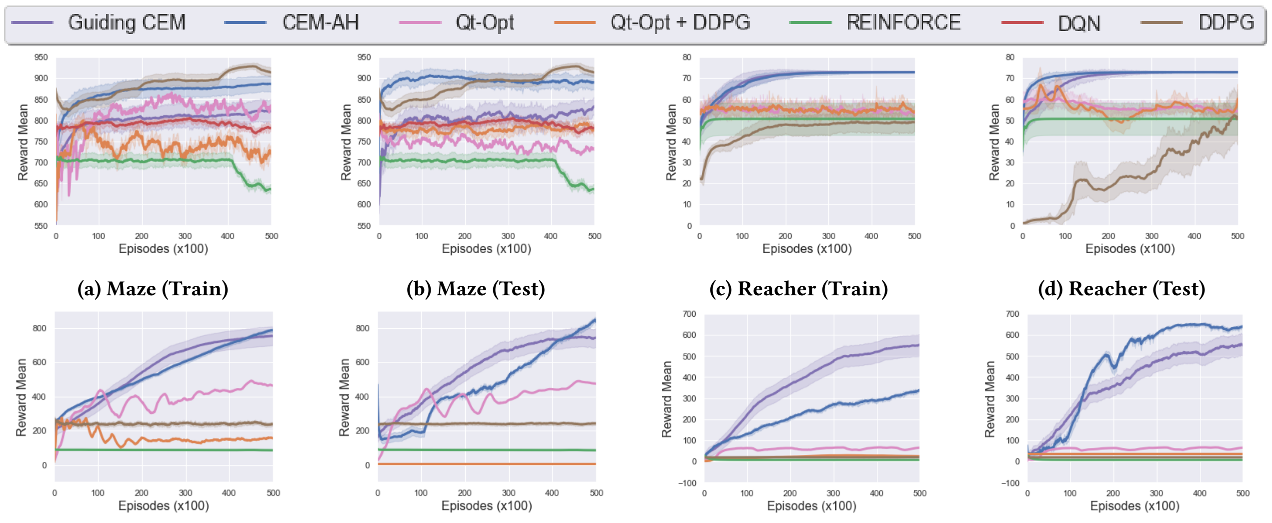
Figure 4: Weight distribution of CEM-AH vs CEM

Fig. 5 shows the mean episodic rewards during training and testing with the best performing set of hyper-parameters for each method, along with the standard deviation (shaded region) across 3 runs. Fig. 5a and 5b show the results for the maze problem. We see that CEM-AH outperforms CEM, especially during testing. CEM-AH allows richer policy representation, and along with the ensemble inference method (Sec. 4.2.2), it results in better performance. To analyze the policy distribution learnt by CEM and CEM-AH, we collect 100 samples from each distribution for the maze problem and show the histogram plot for 3 randomly selected policy weights in Fig. 4. For better visibility, we discretized the sampled weights through binning. The dashed lines represent the approximate distribution shape by fitting the sampled values using kernel density estimation. In Fig. 4a, we notice that CEM quickly converges to a narrow distribution within $2k$ episodes, while CEM-AH maintains a complex and well-spread distribution. Unlike CEM, which directly uses the elite policy parameters to update its distribution, CEM-AH uses state-action tuples to update its distribution (represented by the hypernetwork) by backpropagation via the sampled policy network. The use of such a generative network allows for more complex distribution to be learned. In Fig. 4b, after $50k$ training episodes, most sampled weights converge to a narrow distribution in CEM-AH. However, we can also observe other distributions such as mixtures of Dirac deltas for Weight 1 shown in Fig. 4b. This supports our claim to remove the uni-modal distribution assumption that is followed by most CEM-based approaches, retaining better parameter exploration especially during the early stages of training.

Fig. 5a and 5b also show that CEM-AH has a significantly more stable training process than Qt-Opt and Qt-Opt+DDPG, which were originally designed for handling continuous actions. Both

[3]This is an open source implementation of Roboschool and MuJoCo environments.

[4]The source code can be found at https://github.com/tangshiyuan/cem-ah



(a) Maze (Train) **(b) Maze (Test)** **(c) Reacher (Train)** **(d) Reacher (Test)**

(e) Ant (Train) **(f) Ant (Test)** **(g) Walker2D (Train)** **(h) Walker2D (Test)**

Figure 5: Performance Comparison on: (a-b) Maze; (c-d) Reacher; (e-f) Ant; (g-h) Walker2D in Pybullet-Gym environment

methods sample actions from a multivariate Gaussian distribution and perform updates based on the Q-values approximated using a Q-network. Their noisy training process can be attributed to the change in the network inputs for discrete actions. For discrete actions, we concatenate the action probabilities instead of continuous action values along with the state to be given as input to the Q-network. This stochastic nature of the policy for discrete actions makes Q-network training difficult. Qt-Opt+DDPG has a slightly more stable testing process as it uses another policy network for predicting the actions. We can observe the highly noisy training process for REINFORCE, DDPG and DQN. DDPG obtains better rewards during training, however during testing, DDPG achieves a higher reward of 931.31[5] only after 38k training episodes (28 hours) while CEM-AH converges faster and obtains a reward of 884.55 within 4k episodes (4.5 hours). After 50k episodes, CEM-AH obtains 888.66, DDPG 909.90 (drops slightly), REINFORCE 622.30, DQN 805.47, Qt-Opt 724.47 and Qt-Opt+DDPG 767.54.

Fig. 5c and 5d, we show the results on reacher. The training process in this dense reward setting is less noisy than the maze task across all methods. We again notice fast convergence characteristics of CEM-AH (in less than 18k episodes) with its test performance outperforming that of CEM during early iterations. The similar performance of CEM and CEM-AH after convergence is because in reacher, the policy search space is low-dimensional and less complex resulting in a similar optima. Qt-Opt and Qt-Opt+DDPG consistently perform lower and obtain a reward of 52.67 and 72.06 at the end of 50k episodes during testing. Similar to the maze environment, CEM-AH outperforms REINFORCE and DDPG. The performance of REINFORCE and DDPG differs across runs resulting in a larger standard deviation. Further, convergence in DDPG is very slow achieving 60.79 only at the end of 50k episodes while CEM-AH achieves 72.61 within 18k episodes. DDPG is also highly sensitive to hyper-parameters as shown by our results in Fig. 6f.

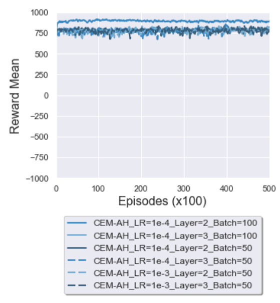
We show the results on ant in Fig. 5e and 5f, and walker in Fig. 5g and 5h. CEM and CEM-AH outperform other methods significantly. Both environments have a higher dimensional search space than reacher, resulting in slow convergence while the performance still improves after 50k episodes. Compared to reacher, both ant and walker control the walking movement of the agent and are more unstable. Hence, the performance gains between train and test (bagging through ensemble inference) are more obvious. For ant, the test rewards for CEM, CEM-AH, Qt-Opt, Qt-Opt+DDPG, REINFORCE, DDPG at the end of 50k episodes are 739.59, 845.51, 509.13, 8.49, 85.34 and 242.31[6] respectively. For walker, the test rewards are 550.78, 640.54, 65.31, 36.57, 7.10 and 19.52 respectively. Overall, CEM-based methods (including CEM-AH) achieve better and more reliable performance in both discrete and continuous tasks.

Table 1 shows the training and test time per episode for all methods on maze and reacher[7]. We use a workstation with dual core Intel Xeon Gold 6148 CPU @ 2.40GHz, and a Tesla V100 GPU. While CEM-AH takes a slightly longer computation time *per episode* (maze: 4.03s, reacher: 4.32s, ant: 6.70s, walker: 11.82s) than CEM, REINFORCE, DDPG and DQN, it achieves faster convergence as early as 4k episodes in maze and 18k in reacher. Training time can still be improved by parallelizing the sampling process in CEM-AH. The testing time (maze: 2.79, reacher: 0.47, ant: 2.02s, walker: 2.07s) remains comparable as the policy sampling process in CEM-AH is lightweight. Qt-Opt and Qt-Opt+DDPG take a much longer training time due to the expensive CEM sampling process during action selection. While the testing time remains long for Qt-Opt due to the same sampling process, it is much shorter for Qt-Opt+DDPG as it uses a learned policy network (the actor network) instead.

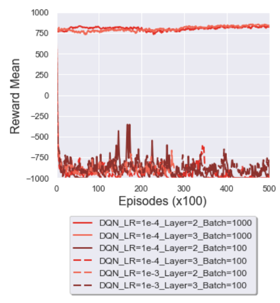
[5]The values reported are the mean episodic rewards obtained during testing.

[6]DDPG result is inline with the findings of Pybullet-Gym benchmark [4].

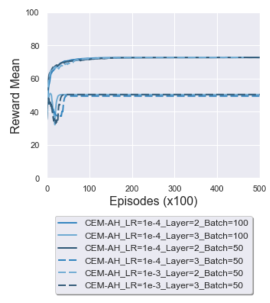
[7]Refer Appendix B for the time complexity on all environments.



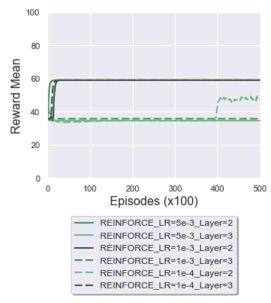
(a) CEM-AH (Maze)



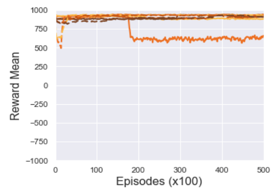
(b) DQN (Maze)



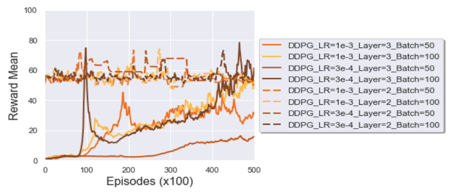
(c) CEM-AH (Reacher)



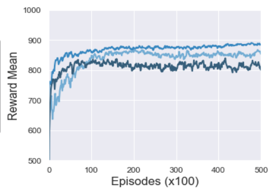
(d) REINFORCE (Reacher)



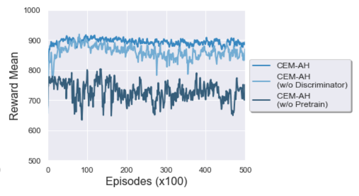
(e) DDPG (Maze)



(f) DDPG (Reacher)



(g) Ablation (Train)



(h) Ablation (Test)

Figure 6: Hyper-parameter sensitivity: (a, b, e) Maze; (c, d, f) Reacher; (g-h) Ablation Study for Maze

5.2 Hyper-parameters & Ablation Study

We perform hyper-parameter tuning in maze and reacher environments (as shown in Fig. 6) with respect to learning rate $\in \{0.001, 0.005, 0.0001\}$, batch size $\in \{50, 100, 1000\}$, and number of neural network layers $\in \{2, 3\}$. In Fig. 6a, 6b and 6e, we show the test performance for CEM-AH, DQN and DDPG on maze for different hyper-parameter configurations. CEM-AH performs consistently better than DQN. We notice in most cases DQN failed abruptly for sparse rewards scenario. For DDPG, although some hyper-parameters can obtain higher rewards than CEM-AH, convergence is very slow. In Fig. 6c, 6d and 6f, we show the test results for CEM-AH, REINFORCE and DDPG on reacher. We can see relatively stagnant or slow learning process for both REINFORCE and DDPG in undesired hyper-parameter configurations. DDPG learns only with a 3-layer network, however, its convergence speed varies significantly with respect to different configurations. In contrast, CEM-AH is able to converge consistently faster than the other methods. CEM-AH performs best when it uses a 2-layer network with batch size of 100 and a learning rate (using Adam optimizer) of 0.0001. The 2-layer CEM-AH configuration performs better than the 3-layer CEM-AH configuration because of the guiding CEM's performance, which is restricted by the sizes of state and action spaces.

Table 1: Train & Test Time Per Episode

	Method	Train (s)	Test (s)
Maze	CEM	0.84 ± 0.13	0.84 ± 0.30
	CEM-AH	4.03 ± 1.60	2.79 ± 0.48
	REINFORCE	3.85 ± 0.37	3.64 ± 0.20
	DQN	2.18 ± 0.14	1.95 ± 0.79
	DDPG	2.67 ± 0.72	2.55 ± 0.51
	Qt-Opt	19.04 ± 24.36	11.54 ± 4.46
	Qt-Opt+DDPG	24.62 ± 36.90	5.32 ± 2.93
Reacher	CEM	0.03 ± 0.01	0.03 ± 0.31
	CEM-AH	4.32 ± 0.95	0.47 ± 0.54
	REINFORCE	3.79 ± 0.22	0.35 ± 0.47
	DDPG	3.33 ± 0.11	0.30 ± 0.05
	Qt-Opt	6.14 ± 0.65	2.91 ± 0.73
	Qt-Opt+DDPG	6.47 ± 0.55	0.33 ± 0.01

However, we can observe that with the same network size, the performance of CEM-AH is consistent and insensitive to learning rate and batch size, and would converge to a similar performance.

We also performed ablation study for CEM-AH in Fig. 6g and 6h on maze, one without pre-training and another without the discriminator. CEM-AH obtains slightly better results than CEM-AH without discriminator, demonstrating the performance gain in using adversarial training. CEM-AH without pre-training performs worst and suffers from convergence issues during adversarial training.

6 CONCLUSION

We present CEM-AH, a CEM guided adversarially-trained hypernetwork to address the drawback of restrictive policy representation leading to sub-optimal policies while using a multivariate Gaussian CEM. Experiments demonstrate that CEM-AH not only inherits the fast convergence and stable training properties of CEM, but also outperforms CEM in learning a richer policy distribution by using a hypernetwork. CEM-AH outperforms other CEM-based RL methods Qt-Opt, Qt-Opt+DDPG and is less sensitive to hyper-parameters when compared to REINFORCE, DDPG and DQN. Future research could explore how CEM-AH can accommodate larger network sizes using methods to re-initialize the guiding CEM policy with a different parameterization technique and transfer weights between hypernetwork-generated policy and the guiding CEM. The generalizability of CEM-AH and its ensemble inference method can be explored by applying it to other transfer learning tasks.

ACKNOWLEDGMENTS

SY.T. acknowledges support from the Alibaba Group and the Alibaba-NTU Singapore Joint Research Institute. F.A.O. received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 758824 —INFLUENCE).

REFERENCES

[1] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight Uncertainty in Neural Network. In *International Conference on Machine Learning (ICML)*. 1613–1622.

[2] C Chalons, RO Fox, and M Massot. 2010. A multi-Gaussian quadrature method of moments for gas-particle flows in a LES framework. In *Proceedings of the Summer Program*. 347–358.

[3] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. 2018. Minimalistic Gridworld Environment for OpenAI Gym. https://github.com/maximecb/gym-minigrid.

[4] Jou ching Sung. 2018. Benchmark results for TD3 and DDPG using the PyBullet reinforcement learning environments. https://github.com/georgesung/TD3.

[5] Andre Costa, Owen Dafydd Jones, and Dirk Kroese. 2007. Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters* 35, 5 (2007), 573–580.

[6] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. 2005. A tutorial on the cross-entropy method. *Annals of operations research* 134, 1 (2005), 19–67.

[7] Lior Deutsch. 2018. Generating neural networks with neural networks. *arXiv preprint arXiv:1801.01952* (2018).

[8] Siegmund Duell and Steffen Udluft. 2013. Ensembles for Continuous Actions in Reinforcement Learning.. In *ESANN*.

[9] Benjamin Ellenberger. 2018. Pybullet gymperium, Open-source implementations of OpenAI Gym MuJoCo environments. https://github.com/benelot/pybullet-gym.

[10] DL Elliott, KC Santosh, and Charles Anderson. 2020. Gradient boosting in crowd ensembles for Q-learning using weight sharing. *International Journal of Machine Learning and Cybernetics* (2020), 1–13.

[11] Louis Faury, Clement Calauzenes, Olivier Fercoq, and Syrine Krichen. 2019. Improving Evolutionary Strategies with Generative Neural Networks. *arXiv preprint arXiv:1901.11271* (2019).

[12] Scott Fujimoto, Herke van Hoof, and David Meger. 2018. Addressing Function Approximation Error in Actor-Critic Methods. In *International Conference on Machine Learning (ICML)*. 1587–1596.

[13] Sebastian Geyer, Iason Papaioannou, and Daniel Straub. 2019. Cross entropy-based importance sampling using Gaussian densities revisited. *Structural Safety* 76 (2019), 15–27.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NIPS)*. 2672–2680.

[15] David Ha, Andrew M. Dai, and Quoc V. Le. 2017. HyperNetworks. In *International Conference on Learning Representations (ICLR)*.

[16] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *International Conference on Machine Learning (ICML)*. 1861–1870.

[17] Alexander Hans and Steffen Udluft. 2011. Ensemble Usage for More Reliable Policy Identification in Reinforcement Learning.. In *ESANN*.

[18] Christian Henning, Johannes von Oswald, João Sacramento, Simone Carlo Surace, Jean-Pascal Pfister, and Benjamin F Grewe. 2018. Approximating the predictive distribution via adversarially-trained hypernetworks. In *Bayesian Deep Learning Workshop, Advances in Neural Information Processing Systems (NeurIPS)*.

[19] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. 2018. Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation. In *Conference on Robot Learning (CoRL)*. 651–673.

[20] Nolan Kurtz and Junho Song. 2013. Cross-entropy-based adaptive importance sampling using Gaussian mixture. *Structural Safety* 42 (2013), 35–44.

[21] Maxim Lapan. 2018. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd.

[22] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning.. In *International Conference on Learning Representations (ICLR)*.

[23] Christos Louizos and Max Welling. 2017. Multiplicative normalizing flows for variational bayesian neural networks. In *International Conference on Machine Learning (ICML)*. 2218–2227.

[24] Shie Mannor, Reuven Y Rubinstein, and Yohai Gat. 2003. The cross entropy method for fast policy search. In *International Conference on Machine Learning (ICML)*. 512–519.

[25] Leonid Margolin. 2005. On the convergence of the cross-entropy method. *Annals of Operations Research* 134, 1 (2005), 201–214.

[26] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*. 1928–1937.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. In *Nature*, Vol. 518. 529–533.

[28] Nick Pawlowski, Martin Rajchl, and Ben Glocker. 2017. Implicit Weight Uncertainty in Neural Networks. In *Bayesian Deep Learning Workshop, Advances in Neural Information Processing Systems (NIPS)*.

[29] Aloïs Pourchot and Olivier Sigaud. 2018. CEM-RL: Combining evolutionary and gradient-based methods for policy search. *arXiv preprint arXiv:1810.01222* (2018).

[30] Kevin Roth, Aurelien Lucchi, Sebastian Nowozin, and Thomas Hofmann. 2017. Stabilizing training of generative adversarial networks through regularization. In *Advances in Neural Information Processing Systems (NIPS)*. 2018–2028.

[31] Reuven Y Rubinstein and Dirk P Kroese. 2013. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media.

[32] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).

[33] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International Conference on Machine Learning (ICML)*. 1889–1897.

[34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[35] Hans-Paul Schwefel. 1981. *Numerical optimization of computer models*. John Wiley & Sons, Inc.

[36] Riley Simmons-Edler, Ben Eisner, Eric Mitchell, Sebastian Seung, and Daniel Lee. 2019. Q-Learning for Continuous Actions with Cross-Entropy Guided Policies. In *RL4RealLife Workshop, International Conference on Machine Learning (ICML)*.

[37] István Szita and András Lörincz. 2006. Learning Tetris using the noisy cross-entropy method. *Neural computation* 18, 12 (2006), 2936–2941.

[38] Kenya Ukai, Takashi Matsubara, and Kuniaki Uehara. 2018. Hypernetwork-based implicit posterior estimation and model averaging of cnn. In *Asian Conference on Machine Learning*. 176–191.

[39] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *AAAI Conference on Artificial Intelligence*. 2094–2100.

[40] Johannes von Oswald, Christian Henning, João Sacramento, and Benjamin F. Grewe. 2020. Continual learning with hypernetworks. In *International Conference on Learning Representations (ICLR)*.

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