Evolved Policy Gradients

Rein Houthooft [1] **Richard Y. Chen** [1] **Phillip Isola** [1 2 3] **Bradly C. Stadie** [2] **Filip Wolski** [1]
Jonathan Ho [1 2] **Pieter Abbeel** [1 2]

Abstract

We propose a metalearning approach for learning gradient-based reinforcement learning (RL) algorithms. The idea is to evolve a differentiable loss function, such that an agent, which optimizes its policy to minimize this loss, will achieve high rewards. The loss is parametrized via temporal convolutions over the agent's experience. Because this loss is highly flexible in its ability to take into account the agent's history, it enables fast task learning. Empirical results show that our evolved policy gradient algorithm achieves faster learning on several randomized environments compared to an off-the-shelf policy gradient method.

1. Introduction

When a human learns to solve a new control task, such as playing the violin, they immediately have a feel for what to try. At first, they may try a quick, rough stroke, and, producing a screech, will intuitively know this was the wrong thing to do. Just by listening to the sounds they produce, they will have a sense of whether or not they are making progress toward the goal. Effectively, humans have access to very well shaped internal reward functions, derived from prior experience on other motor tasks, or perhaps from listening to and playing other musical instruments (36; 49).

In contrast, most current reinforcement learning (RL) agents approach each new task de novo. Initially, they have no notion of what actions to try out, nor which outcomes are desirable. Instead, they rely entirely on external reward signals to guide their initial behavior. Coming from such a blank slate, it is no surprise that RL agents take far longer than humans to learn simple skills (21).

Our aim in this paper is to devise agents that have a prior notion of what constitutes making progress on a novel task. Rather than encoding this knowledge explicitly through memorized behaviors, we encode it implicitly through a learned loss function. The end goal is agents that can use this loss function to learn quickly on a novel task.

[1]OpenAI [2]UC Berkeley [3]MIT. Correspondence to: Rein Houthooft <rein.houthooft@openai.com>.

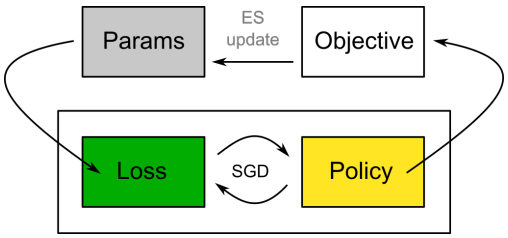


Figure 1: High-level overview of our approach. The method consists of an inner and outer optimization loop. The inner loop (boxed) optimizes the agent's policy against a loss provided by the outer loop, using gradient descent. The outer loop optimizes the parameters of the loss function, such that the optimized inner-loop policy achieves high performance on an arbitrary task, such as solving a control task of interest. The evolved loss L can be understood as a parametrization of policy gradients' surrogate loss, lending the name "evolved policy gradients".

This approach can be seen as a form of metalearning, in which we learn a learning algorithm (12). Rather than mining rules that generalize across data points, as in traditional machine learning, metalearning concerns itself with devising algorithms that generalize across tasks, by infusing prior knowledge of the task distribution.

Our method consists of two optimization loops. In the inner loop, an agent learns to solve a task, sampled from a particular distribution over a family of tasks. The agent learns to solve this task by minimizing a loss function provided by the outer loop. In the outer loop, the parameters of the loss function are adjusted so as to maximize the final returns achieved after inner loop learning. Figure 1 provides a high-level overview of this approach.

Although the inner loop can be optimized with stochastic gradient descent (SGD), optimizing the outer loop presents substantial difficulty. Each evaluation of the outer objective requires training a complete inner-loop agent, and this objective cannot be written as an explicit function of the loss parameters we are optimizing over. Due to the lack of easily exploitable structure in this optimization problem, we turn to evolution strategies (ES) (35; 46; 16; 37) as a

blackbox optimizer. The evolved loss L can be viewed as a parametrization of policy gradients' surrogate loss (43; 44), lending the name "evolved policy gradients" (EPG).

In addition to encoding prior knowledge, the learned loss offers several advantages compared to current RL methods. Since RL methods optimize for short-term returns instead of accounting for the complete learning process, they may get stuck in local minima and fail to explore the full search space. Prior works add auxiliary reward terms that emphasize exploration (8; 19; 32; 56; 6; 33) and entropy loss terms (31; 42; 15; 26). These terms are often traded off using a separate hyperparameter that is not only task-dependent, but also dependent on which part of the state space the agent is visiting. As such, it is unclear how to include these terms into the RL algorithm in a principled way.

Using ES to evolve the loss function allows us to optimize the true objective, namely the final trained policy performance, rather than short-term returns. Our method improves on standard RL algorithms by allowing the loss function to be adaptive to the environment and agent history, leading to faster learning and the potential for learning without external rewards. EPG can in theory be combined with policy initialization metalearning algorithms, such as MAML (11), since EPG imposes no restriction on the policy it optimizes.

There has been a flurry of recent work on metalearning policies, e.g., (10; 11), and it is worth asking why metalearn the loss as opposed to directly metalearning the policy? Our motivation is that we expect loss functions to be the kind of object that may generalize very well across substantially different tasks. This is certainly true of hand-engineered loss functions: a well-designed RL loss function, such as that in (45), can be very generically applicable, finding use in problems ranging from playing Atari games to controlling robots (45). In Section 4, we find evidence that a loss learned by EPG can train an agent to solve a task *outside the distribution* of tasks on which EPG was trained. This generalization behavior differs qualitatively from a baseline method that directly metalearns a policy, providing initial indication of the generalization potential of loss learning.

Our contributions include the following:

- Formulating a metalearning approach that learns a differentiable loss function for RL agents, called EPG.
- Optimizing the parameters of this loss function via ES, overcoming the challenge that final returns are not explicit functions of the loss parameters.
- Designing a loss architecture that takes into account agent history via temporal convolutions.
- Demonstrating that EPG produces learned loss which can train agents faster than an off-the-shelf policy gradient method.
- Showing that EPG's learned loss can generalize to *out of distribution* test time tasks, exhibiting qualitatively different behaviors from other popular metalearning algorithms.

We set forth the notation in Section 2. Section 3 explains the main algorithm and Section 4 shows its results on several randomized continuous control environments. In Section 5, we compare our methods with the most related ideas in literature. We conclude this paper with a discussion in Section 6. An implementation of EPG is available at

http://github.com/openai/EPG.

2. Notation and Background

We model reinforcement learning (54) as a Markov decision process (MDP), defined as the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, p_0, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ are the state and action space. The transition dynamic $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}_+$ determines the distribution of the next state s_{t+1} given the current state s_t and the action a_t. $R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is the reward function and $\gamma \in (0, 1)$ is a discount factor. p_0 is the distribution of the initial state s_0. An agent's policy $\pi : \mathcal{S} \mapsto \mathcal{A}$ generates an action after observing a state.

An episode $\tau \sim \mathcal{M}$ with horizon H is a sequence $(s_0, a_0, r_0, \ldots, s_H, a_H, r_H)$ of state, action, and reward at each timestep t. The discounted episodic return of τ is defined as $R_\tau = \sum_{t=0}^{H} \gamma^t r_t$, which depends on the initial state distribution p_0, the agent's policy π, and the transition distribution T. The expected episodic return given agent's policy π is $\mathbb{E}_\pi[R_\tau]$. The optimal policy π^* maximizes the expected episodic return

$$\pi^* = \arg\max_\pi \mathbb{E}_{\tau \sim \mathcal{M}, \pi}[R_\tau].$$

In high-dimensional reinforcement learning settings, the policy π is often parametrized using a deep neural network $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$. The goal is to solve for $\boldsymbol{\theta}^*$ that attains the highest expected episodic return

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \mathcal{M}, \pi_{\boldsymbol{\theta}}}[R_\tau]. \tag{1}$$

This objective can be optimized via policy gradient methods (60; 55) by stepping in the direction of $\mathbb{E}[R_\tau \nabla \log \pi(\tau)]$. This gradient can be transformed into a surrogate loss function (43; 44)

$$L_{\text{pg}} = \mathbb{E}[R_\tau \log \pi(\tau)] = \mathbb{E}\left[R_\tau \sum_{t=0}^{H} \log \pi(a_t|s_t)\right], \tag{2}$$

such that the gradient of L_{pg} equals the policy gradient. Through variance reduction techniques including actor-critic

algorithms (20), the loss function L_{pg} is often changed into

$$L_{\text{ac}} = \mathbb{E}\left[\sum_{t=0}^{H} A(s_t, a_t) \log \pi(a_t|s_t)\right], \qquad (3)$$

that is, the log-probability of taking action a_t at state s_t is multiplied by an advantage function $A(s_t, a_t)$ (4).

However, this procedure remains limited since it relies on a particular form of discounting the returns, and taking a fixed gradient step with respect to the policy. Our approach learns a loss rather than using a hand-defined function such as L_{ac}. Thus, it may be able to discover more effective surrogates for making fast progress toward the ultimate objective of maximizing final returns.

3. Methodology

Our metalearning approach aims to learn a loss function $L_{\boldsymbol{\phi}}$ that outperforms the usual policy gradient loss. This loss function consists of temporal convolutions over the agent's recent history. In addition to internalizing environment rewards, this loss could, in principle, have several other positive effects. For example, by examining the agent's history, the loss could incentivize desirable extended behaviors, such as exploration. Further, the loss could perform a form of system identification, inferring environment parameters and adapting how it guides the agent as a function of these parameters (e.g., by adjusting the effective learning rate of the agent).

The loss function parameters $\boldsymbol{\phi}$ are evolved through ES and the loss trains agent's policy $\pi_{\boldsymbol{\theta}}$ in an on-policy fashion via gradient descent.

3.1. Metalearning Objective

In our metalearning setup, we assume access to a distribution $p(\mathcal{M})$ over MDPs. Given a sampled MDP $\mathcal{M}$, the inner loop optimization problem is to minimize the loss $L_{\boldsymbol{\phi}}$ with respect to the agent's policy $\pi_{\boldsymbol{\theta}}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\tau\sim\mathcal{M},\pi_{\boldsymbol{\theta}}}[L_{\boldsymbol{\phi}}(\pi_{\boldsymbol{\theta}}, \tau)]. \qquad (4)$$

Note that this is similar to the usual RL objectives (Eqs. (1) (2) (3)), except that we are optimizing a learned loss $L_{\boldsymbol{\phi}}$ rather than directly optimizing the expected episodic return $\mathbb{E}_{\pi_{\boldsymbol{\theta}}}[R_\tau]$ or other surrogate losses. The outer loop objective is to learn $L_{\boldsymbol{\phi}}$ such that an agent policy $\pi_{\boldsymbol{\theta}^*}$ trained with the loss function achieves high expected returns in the MDP distribution:

$$\boldsymbol{\phi}^* = \arg\max_{\boldsymbol{\phi}} \mathbb{E}_{\mathcal{M}\sim p(\mathcal{M})}\mathbb{E}_{\tau\sim\mathcal{M},\pi_{\boldsymbol{\theta}^*}}[R_\tau]. \qquad (5)$$

Algorithm 1: Evolved Policy Gradients (EPG)

[Outer Loop] for epoch $e = 1, \ldots, E$ **do**
  Sample $\boldsymbol{\epsilon}_v \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$ and calculate the loss parameter $\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_v$ for $v = 1, \ldots, V$
  Each worker $w = 1, \ldots, W$ gets assigned noise vector $\lceil wV/W \rceil$ as $\boldsymbol{\epsilon}_w$
  for each worker $w = 1, \ldots, W$ **do**
    Sample MDP $\mathcal{M}_w \sim p(\mathcal{M})$
    Initialize buffer with N zero tuples
    Initialize policy parameter $\boldsymbol{\theta}$ randomly
    [Inner Loop] for step $t = 1, \ldots, U$ **do**
      Sample initial state $s_t \sim p_0$ if $\mathcal{M}_w$ needs to be reset
      Sample action $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
      Take action a_t in $\mathcal{M}_w$ and receive r_t, s_{t+1}, and termination flag d_t
      Add tuple (s_t, a_t, r_t, d_t) to buffer
      if $t \bmod M = 0$ **then**
        With loss parameter $\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_w$, calculate losses L_i for steps $i = t - M, \ldots, t$ using buffer tuples $i - N, \ldots, i$
        Sample minibatches mb from last M steps shuffled, compute $L_{\mathrm{mb}} = \sum_{j\in\mathrm{mb}} L_j$, and update the policy parameter $\boldsymbol{\theta}$ and memory parameter (Eq. (6))
    In $\mathcal{M}_w$, using the trained policy $\pi_{\boldsymbol{\theta}}$, sample several trajectories and compute the mean return R_w
  Update the loss parameter $\boldsymbol{\phi}$ (Eq. (7))
Output: Loss $L_{\boldsymbol{\phi}}$ that trains π from scratch according to the inner loop scheme, on MDPs from $p(\mathcal{M})$

3.2. Algorithm

The final episodic return R_τ of a trained policy $pi_{\boldsymbol{\theta}^*}$ cannot be represented as an explicit function of the loss function $L_{\boldsymbol{\phi}}$. Thus we cannot use gradient-based methods to directly solve Eq. (5). Our approach, summarized in Algorithm 1, relies on evolution strategies (ES) to optimize the loss function in the outer loop.

As described by Salimans et al. (37), ES computes the gradient of a function $F(\boldsymbol{\phi})$ according to

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

Similar formulations also appear in prior works including (52; 47; 27). In our case, $F(\boldsymbol{\phi}) = \mathbb{E}_{\mathcal{M}\sim p(\mathcal{M})}\mathbb{E}_{\tau\sim\mathcal{M},\pi_{\boldsymbol{\theta}^*}}[R_\tau]$ (Eq. (5)). Note that the dependence on $\boldsymbol{\phi}$ comes through $\boldsymbol{\theta}^*$ (Eq. (4)).

Step by step, the algorithm works as follows. At the start

Algorithm 2: EPG test-time training

[Input]: learned loss function L_ϕ from EPG, MDP $\mathcal{M}$
Initialize buffer with N zero tuples
Initialize policy parameter $\boldsymbol{\theta}$ randomly
for step $t = 1, \ldots, U$ **do**
  Sample initial state $s_t \sim p_0$ if $\mathcal{M}$ needs to be reset
  Sample action $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
  Take action a_t in $\mathcal{M}$, receive r_t, s_{t+1}, and termination flag d_t
  Add tuple (s_t, a_t, r_t, d_t) to buffer
  if $t \bmod M = 0$ **then**
    Calculate losses L_i for steps $i = t - M, \ldots, t$ using buffer tuples $i - N, \ldots, i$
    Sample minibatches mb from last M steps shuffled, compute $L_{\text{mb}} = \sum_{j \in \text{mb}} L_j$, and update the policy parameter $\boldsymbol{\theta}$ and memory parameter (Eq. (6))
[Output]: A trained policy $\pi_{\boldsymbol{\theta}}$ on MDP $\mathcal{M}$

of each epoch in the outer loop, for W inner-loop workers, we generate V standard multivariate normal vectors $\boldsymbol{\epsilon}_v \in \mathcal{N}(0, \mathrm{I})$ with the same dimension as the loss function parameter $\boldsymbol{\phi}$, assigned to V sets of W/V workers. As such, for the w-th worker, the outer loop assigns the $\lceil wV/W \rceil$-th perturbed loss function

$$L_w = L_{\boldsymbol{\phi}+\sigma\boldsymbol{\epsilon}_v} \text{ where } v = \lceil wV/W \rceil$$

with perturbed parameters $\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_v$ and σ as the standard deviation.

Given a loss function L_w, $w \in \{1, \ldots, W\}$, from the outer loop, each inner-loop worker w samples a random MDP from the task distribution, $\mathcal{M}_w \sim p(\mathcal{M})$. The worker then trains a policy $\pi_{\boldsymbol{\theta}}$ in $\mathcal{M}_w$ over U steps of experience. Whenever a termination signal is reached, the environment resets with state s_0 sampled from the initial state distribution $p_0(\mathcal{M}_w)$. Every M steps the policy is updated through SGD using minibatches sampled from the steps $t - M, \ldots, t$ and the loss function L_w:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \delta_{\text{in}} \cdot \nabla_{\boldsymbol{\theta}} L_w\big(\pi_{\boldsymbol{\theta}}, \tau_{t-M,\ldots,t}\big). \tag{6}$$

At the end of the inner-loop training, each worker returns the final return R_w[1] to the outer loop. The outer-loop aggregates the final returns $\{R_w\}_{w=1}^W$ from all workers and updates the loss function parameter $\boldsymbol{\phi}$ as follows:

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \delta_{\text{out}} \cdot \frac{1}{V\sigma} \sum_{v=1}^{V} F(\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_v)\boldsymbol{\epsilon}_v, \tag{7}$$

where

$$F(\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_v) = \frac{R_{(v-1)*W/V+1} + \cdots + R_{v*W/V}}{W/V}.$$

[1]More specifically, the average return over 3 sampled trajectories using the final policy for worker w.

As a result, each perturbed loss function L_v is evaluated on W/V randomly sampled MDPs from the task distribution using the final returns. This achieves variance reduction by preventing the outer-loop ES update from promoting loss functions that are assigned to MDPs that consistently generate higher returns. Note that the actual implementation calculates each loss function's relative rank for the ES update. Algorithm 1 outputs a learned loss function L_ϕ after E epochs of ES updates.

At test time, we evaluate the learned loss function L_ϕ produced by Algorithm 1 on a test MDP $\mathcal{M}$ by training a policy from scratch. The test-time training schedule is the same as the inner loop of Algorithm 1 and we summarize it in Algorithm 2.

3.3. Architecture

The agent is parametrized using an MLP policy with observation space $\mathcal{S}$ and action space $\mathcal{A}$. The loss has a memory unit to assist learning in the inner loop. This memory unit is a single-layer neural network to which an invariable input

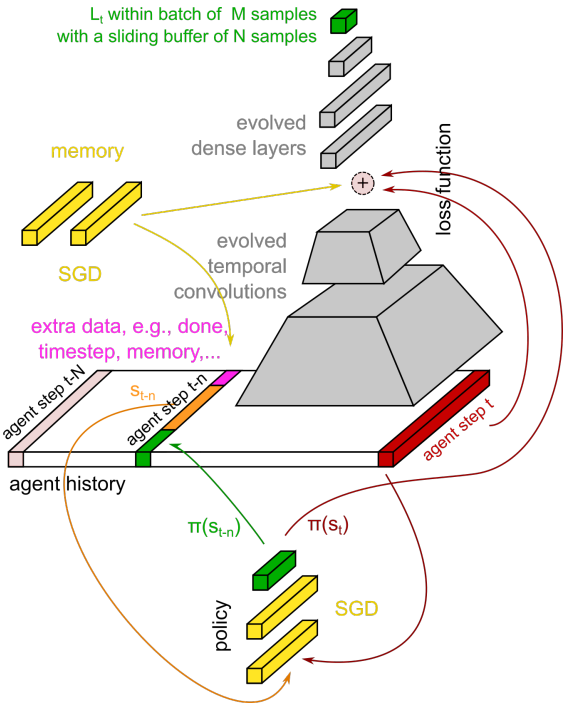


Figure 2: Architecture of a loss computed for timestep t within a batch of M sequential samples (from $t - M$ to t), using temporal convolutions over a buffer of size N (from $t - N$ to t), with $M \leq N$: dense net on the bottom is the policy $\pi(s)$, taking as input the observations (orange), while outputting action probabilities (green). The green block on the top represents the loss output. Gray blocks are evolved, yellow blocks are updated through SGD.

vector of ones is fed. Since this network has a constant input vector, we can view its weights as a very simple form of memory to which the loss can write via emitting the right gradient signals. An experience buffer stores the agent's N most recent experience steps, in the form of a list of tuples (s_t, a_t, r_t, d_t), with d_t the trajectory termination flag.

The loss function L_{ϕ} consists of temporal convolutional layers which generate a context vector f_{context}, and dense layers, which output the loss. The architecture is depicted in Figure 2.

At step t, the dense layers outputs the loss L_t by taking a batch of M sequential samples

$$\{s_i, a_i, d_i, \text{mem}, f_{\text{context}}, \pi_{\boldsymbol{\theta}}(\cdot|s_i)\}_{i=t-M}^{t}, \tag{8}$$

where $M < N$ and we augment each transition with the memory output mem, a context vector f_{context} generated from the loss's temporal convolutional layers, and the policy distribution $\pi_{\boldsymbol{\theta}}(\cdot|s_i)$. In continuous action space, $\pi_{\boldsymbol{\theta}}$ is a Gaussian policy, i.e., $\pi_{\boldsymbol{\theta}}(\cdot|s_i) = \mathcal{N}(\cdot; \mu(s_i; \boldsymbol{\theta}_0), \Sigma)$, with $\mu(s_i; \boldsymbol{\theta}_0)$ the MLP output and Σ a learnable parameter vector. The policy parameter vector is defined as $\boldsymbol{\theta} = [\boldsymbol{\theta}_0, \Sigma]$. In discrete action spaces, $\pi_{\boldsymbol{\theta}}$ represents a multinomial distribution over the discrete actions.

To generate the context vector, we first augment each transition in the buffer with the output of the memory unit mem and the policy distribution $\pi_{\boldsymbol{\theta}}(\cdot|s_i)$ to obtain a set

$$\{s_i, a_i, d_i, \text{mem}, \pi_{\boldsymbol{\theta}}(\cdot|s_i)\}_{i=t-N}^{t}. \tag{9}$$

We stack these items sequentially into a matrix and the temporal convolutional layers take it as input and output the context vector f_{context}. The memory unit's parameters are updated via gradient descent at each inner-loop update (Eq. (6)).

Note that both the temporal convolution layers and the dense layers do not observe the environment rewards directly. However, in cases where the reward cannot be fully inferred from the environment, such as the DirectionalHopper environment we will examine in Section 4.2, we augment the inputs Eqs. (8) and (9) to the loss function with rewards. In fact, any information that can be obtained from the environment can be added as an input to the loss function, e.g., exploration signals, the current timestep number, etc.

In practice, to bootstrap the learning process, we add to L_{ϕ} a guidance policy gradient surrogate loss signal L_{pg}, such as the REINFORCE (60) or PPO (45) surrogate loss function, making the total loss

$$\hat{L}_{\phi} = (1 - \alpha) L_{\phi} + \alpha L_{\text{pg}}, \tag{10}$$

and anneal α from 1 to 0 over a finite number of outer-loop epochs. As such, learning is first derived mostly from the well-structured L_{pg}, while over time L_{ϕ} takes over and drives learning completely after α has been annealed to 0.

4. Experiments

We apply our method to several randomized continuous control MuJoCo environments (5; 34; 9), namely RandomHopper and RandomWalker (with randomized gravity, friction, body mass, and link thickness), RandomReacher (with randomized link lengths), DirectionalHopper and DirectionalHalfCheetah (with randomized forward/backward reward function), GoalAnt (reward function based on the randomized target location), and Fetch (randomized target location). We describe these environments in detail in Appendix A. These environments are chosen because they require the agent to identify a randomly sampled environment at test time via exploratory behavior. Examples of the randomized Hopper environments are shown in Figure 3, the Fetch environment in Figure 4, and the GoalAnt environment in Figure 14. The plots in this section show the mean value of 20 test-time training curves as a solid line, while the shaded area represents the interquartile range. The dotted lines plot 5 randomly sampled curves.

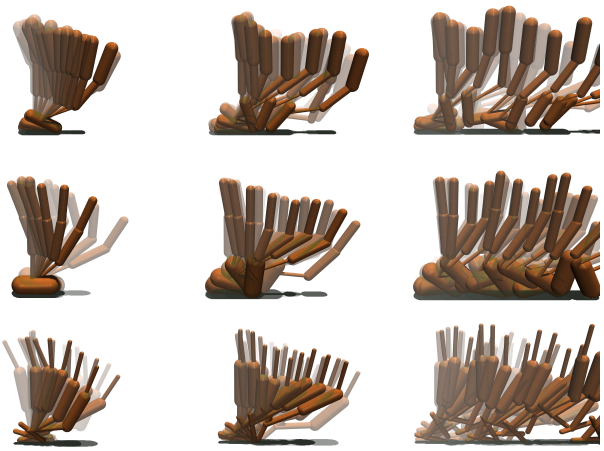


Figure 3: Example of learning to hop forward from a randomly initialized policy in RandomHopper environments with randomized morphology and physics parameters. Each row is a different environment randomization, while from left to right, trajectories are recorded as learning progresses.

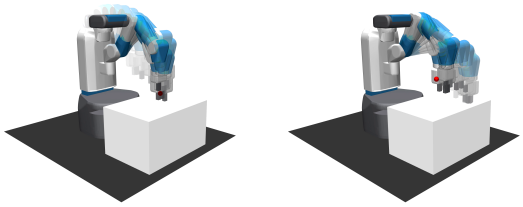


Figure 4: Examples of learning to reach random targets in the Fetch environment

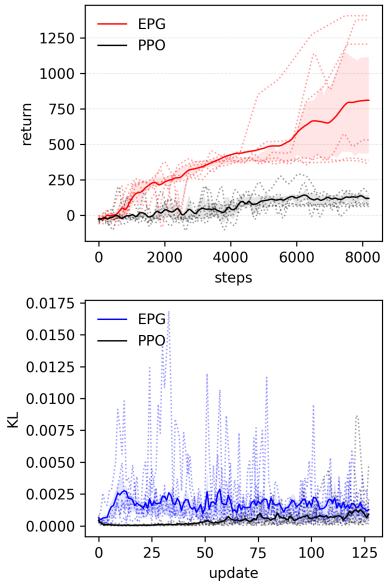


Figure 5: RandomHopper test-time training over 128 (policy updates) $\times 64$ (update frequency) $= 8196$ timesteps: PPO vs no-reward EPG

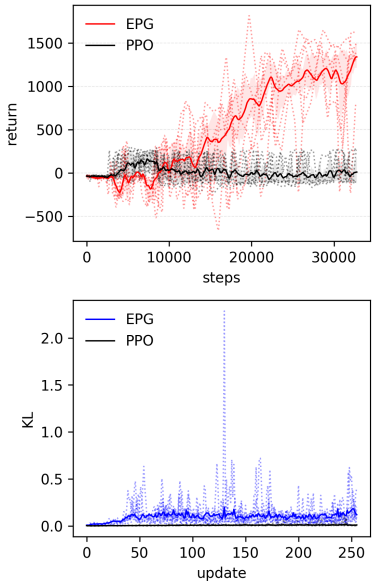


Figure 6: RandomWalker test-time training over 256 (policy updates) $\times 128$ (update frequency) = 32768 timesteps: PPO vs no-reward EPG

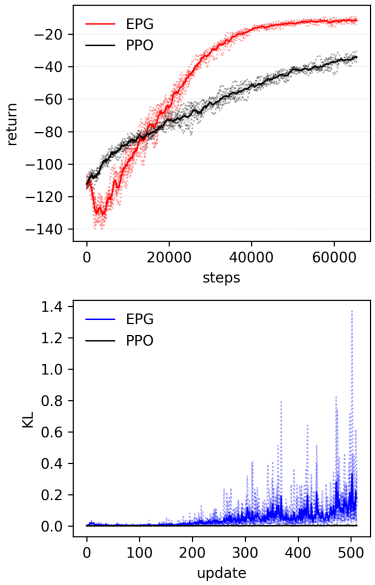
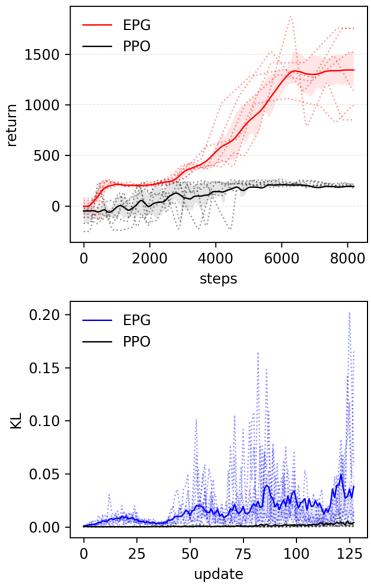


Figure 7: RandomReacher test-time training over 512 (policy updates) $\times 128$ (update frequency) $= 65536$ timesteps: PG vs no-reward EPG.



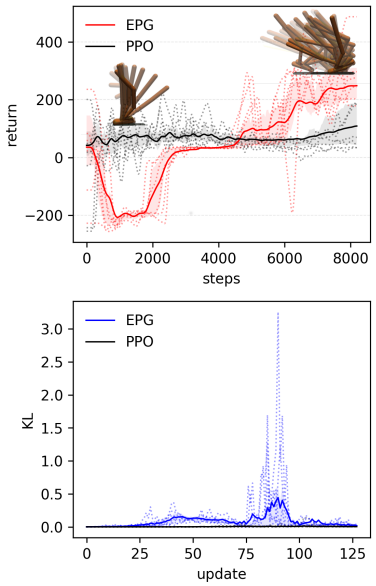


Figure 8: DirectionalHopper environment: each Hopper environment randomly decides whether to reward forward or backward hopping. The agent needs to identify whether to jump forward or backwards: PPO vs EPG. Here we can clearly see exploratory behavior, indicated by the negative spikes in the reward curve, the loss forces the policy to try out backwards behavior. Each subplot column corresponds to a different randomization of the environment.

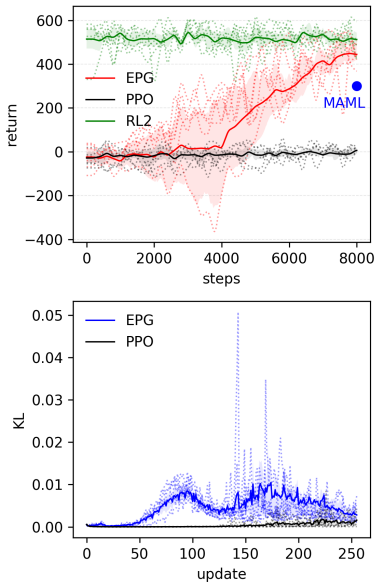


Figure 9: Comparison with MAML (single gradient step after metalearning a policy initialization) on the DirectionalHalfCheetah environment from Finn et al. (11) (Fig. 5)

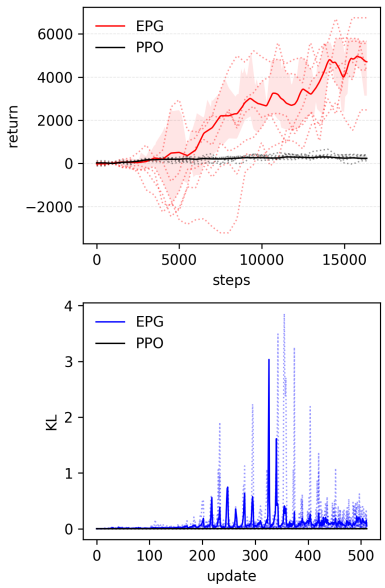


Figure 10: GoalAnt test-time training over 512 (policy updates) $\times 32$ (update frequency) $= 16384$ timesteps: PPO vs EPG

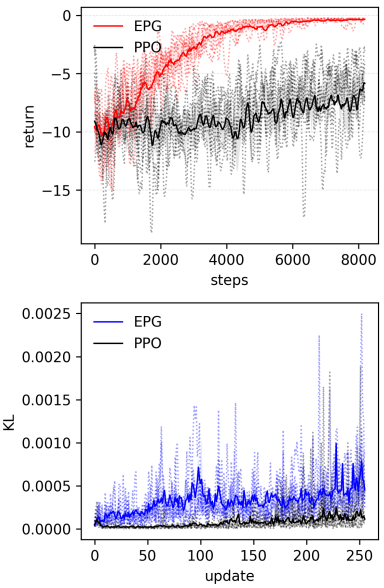


Figure 11: Fetch reaching environment learning over 256 (policy updates) $\times 32$ (update frequency) $= 8192$ timesteps: PPO vs no-reward EPG

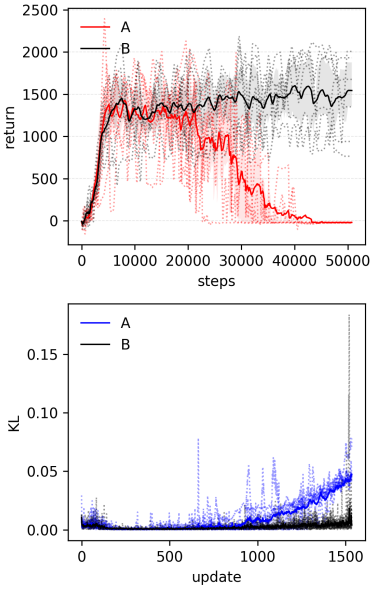


Figure 12: Transferring EPG (meta-learned using 128 policy updates on RandomHopper) to 1536 updates at test time: random policy initialization (A), initialization by sampled previous policies (B)

Implementation details In our experiments, the temporal convolutional layers of the loss function has 3 layers. The first layer has a kernel size of 8, stride of 7, and outputs 10 channels. The second layer has a kernel of 4, stride of 2, and outputs 10 channels. The third layer is fully-connected with 32 output units. Leaky ReLU activation is applied to each convolutional layer. The fully connected component takes the trajectory features from the convolutional component concatenated with state, action, reward, termination signal, and policy output as input. It has 1 hidden layer with 16 hidden units and leaky ReLU activation, followed by an output layer. The buffer size is $N \in \{512, 1024\}$. The agent's MLP policy has 2 hidden layers of 64 units with $\tanh$ activation. The memory unit is a 32-unit single layer with $\tanh$ activation.

We use $W = 256$ inner-loop workers in Algorithm 1, combined with $V = 64$ ES noise vectors. The loss function is evolved over 5000 epochs, with α, as in Eq. (10), annealed linearly from 1 to 0 over the first 500 epochs. The off-the-shelf PG algorithm (PPO) was moderately tuned to perform well on these tasks, however, it is important to keep in mind that these methods inherently have trouble optimizing when the number of samples drawn for each policy update batch is low. EPG's inner loop update frequency is set to $M \in \{32, 64, 128\}$ and the inner loop length is $U \in \{64 * M, 128 * M, 256 * M, 512 * M\}$. At every EPG inner loop update, the policy and memory parameters are updated by the learned loss function using shuffled minibatches of size 32 within each set of M most recent transition steps in the replay buffer, going over each step exactly once. We tabulate the hyperparameters for each randomized environment in Table 1 in Appendix C.

Normalization according to a running mean and standard deviation were applied to the observations, actions, and rewards for each EPG inner loop worker independently (Algorithm 1) and for test-time training (Algorithm 2). Adam (18) is used for the EPG inner loop optimization and test-time training with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, while the outer loop ES gradients are modified by Adam with $\beta_1 = 0$ and $\beta_2 = 0.999$ (which means momentum has been turned off) before updating the loss function. Furthermore, L2-regularization over the loss function parameters with coefficient 0.001 is added to outer loop objective. The inner loop step size is fixed to 10^{-3}, while the outer loop step size is annealed linearly from 10^{-2} to 10^{-3} over the first 2000 epochs.

4.1. Performance

We compare test-time training performance using the EPG loss function, Algorithm 2, against an off-the-shelf policy gradient method, PPO (45). Figures 5, 6, 7, and 11 show learning curves for these two methods on the RandomHop-

per, RandomWalker, RandomReacher, and Fetch environments respectively at test time. The top plot shows the episodic return w.r.t. the number of environment steps taken so far. The bottom plot shows how much the policy changes at every update by plotting the KL-divergence between the policy distributions before and after every update, w.r.t. the number of updates so far.

In all of these environments, the PPO agent learns by observing reward signals whereas at test time, the agent optimizing our learned loss does not observe rewards (note that at test time, α in Eq. (10) equals 0). Observing rewards is not needed in EPG, since any piece of information the agent encounters forms an input to the EPG loss function. As long as the agent can identify which task to solve within the distribution, it does not matter whether this identification is done through observations, or rewards. Keep in mind, however, that the rewards were used in the ES objective function during the EPG evolution phase.

In all experiments, EPG agent learns more quickly and obtains higher returns compared to the PPO agent, as expected, since the EPG loss function is able to tailor itself to the environment distribution it is metatrained on. This indicates that our method generates an objective that is more effective at training agents, within these task distributions, than an off-the-shelf on-policy policy gradient method. This is true even though the learned loss does not observe rewards at test time. This demonstrates the potential to use EPG when rewards are only available at training time, for example, if a system were trained in simulation but deployed in the real world where reward signals are hard to measure.

The correlation between the gradients of our learned loss and the PPO objective is around $\rho = 0.5$ (Spearman's rank correlation coefficient) for the environments tested. This indicates that the gradients produced by the learned loss differs sufficiently from the PPO objective.

Figures 8, 9, and 10 show experiments in which a signaling flag is required to identify the environment. Generally, this is done through a reward function or an observation flag, which is why EPG takes the reward as input in case the state space is partially-observed. Similar to the previous experiments, EPG significantly outperforms PPO on the task distribution it is metatrained on. Specifically, in Figure 9, we compare EPG with both MAML (data from (11)) and RL^2 (10). This experiment shows that, at least in this experimental setup, starting from a random policy initialization can bring as much benefit as learning a good policy initialization (MAML). In Section 4.2, we will investigate what the effect of evolving the policy initialization together with the loss parameters is. When comparing EPG to RL^2 (a method that learns a recurrent policy that does not reset the internal state upon trajectory resets), we see that RL^2 solves the DirectionalHalfCheetah task almost instantly through system identification. By learning both the algorithm and the policy initialization simultaneously, it is able to significantly outperform both MAML and EPG. However, this comes at the cost of generalization power, as we will discuss in Section 4.3.

4.2. Analysis

In this section, we first analyze whether EPG produces a loss function that encourages exploration and adaptive policy updates during test-time training. Next, we evaluate the effect of evolving the policy initialization.

Learning exploratory behavior Without additional exploratory incentives, PG methods lead to suboptimal policies. To understand whether EPG is able to train agents that explore, we test our method and PPO on the DirectionalHopper and GoalAnt environments. In DirectionalHopper, each sampled Hopper environment either rewards the agent for forward or backward hopping. Note that without observing the reward, the agent cannot infer whether the Hopper environment desires forward or backward hopping. Thus we augment the environment reward to the input batches of the loss function in this setting.

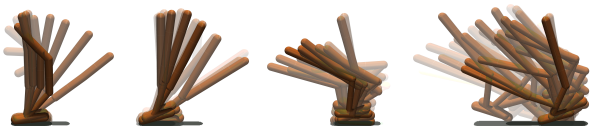


Figure 13: Example of learning to hop backward from a randomly initialized policy in a DirectionalHopper environment. From left to right, trajectories are recorded as learning progresses.

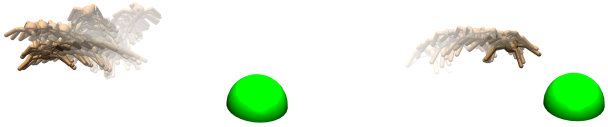
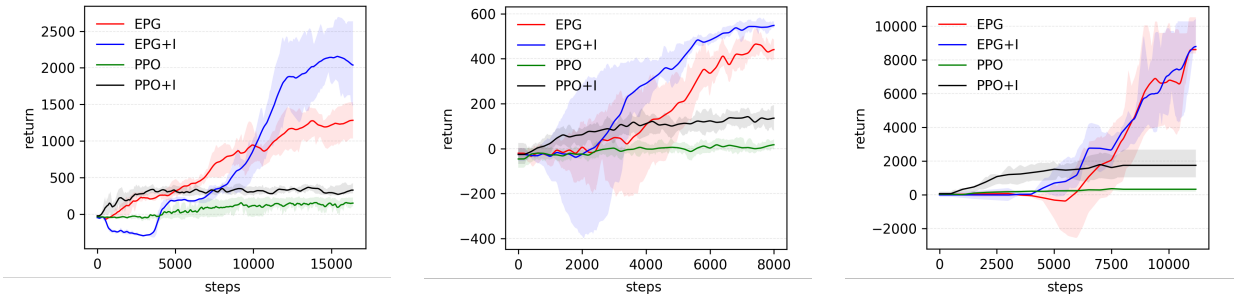


Figure 14: Examples of learning to reach random targets in the GoalAnt environment. The ant first learns to walk in various directions before converging on the target location.

Figure 8 shows learning curves of both PPO agents and agents trained with the learned loss in the DirectionalHopper environment. The learning curves give indication that the learned loss is able to train agents that exhibit exploratory behavior. We see that in most instances, PPO agents stagnate in learning, while agents trained with our learned loss manage to explore both forward and backward hopping and eventually hop in the correct direction. Figure 8 (right) demonstrates the qualitative behavior of our agent during learning and Figure 13 visualizes the exploratory behavior. We see that the hopper first explores one hopping direction before learning to hop backwards.



(a) RandomWalker (b) DirectionalHalfCheetah (c) GoalAnt

Figure 15: Effect of evolving the policy initialization (+I) on various randomized environments. test-time training curves with evolved policy initialization start at the same return value as those without evolved initialization. This is consistent with MAML trained on a wide task distributions (Figure 5 of (12)).

The GoalAnt environment randomizes the location of the goal. We augment the goal location to the input batches of the loss function. Figure 14 demonstrates the exploratory behavior of a learning ant trained by EPG. We see that the ant first learns to walk and explore various directions, before finally converging on the correct goal location. This is better visualized in Figure 16, in which four different learning procedures are depicted. The ant first explores in various directions, including the opposite direction of the target location. However, it quickly figures out in which quadrant to explore, before it fully learns to correct direction to walk.

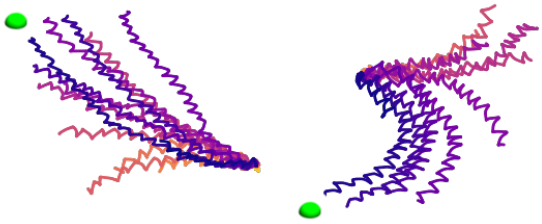


Figure 16: Trajectories sampled from test-time training on two sampled GoalAnt environments: the Ant learns how to explore various directions before going to the correct target. Lighter colors represent initial trajectories, darker colors are later trajectories, according to the agent's learning process.

Learning adaptive policy updates PG methods such as REINFORCE (60) suffer from unstable learning, such that a large learning step size leads to policy crashing during learning. To encourage smooth policy updates, methods such as TRPO (44) and PPO (45) were proposed to limit the distributional change from each policy update, through a hyperparameter constraining the KL-divergence between the policy distributions before and after each update. We demonstrate that EPG produces learned loss that adaptively scales the graduate updates.

Figure 17 shows the KL-divergence between policies from one update to the next during the course of training in RandomHopper, using a randomly initialized loss (left) versus a learned loss produced by Algorithm 1 (right). With a learned loss function, the policy updates tend to shift the policy distribution less on each step, but sometimes produce sudden changes, indicated by the spikes. These spikes are highly noticeable in Figure 23 of Appendix B, in which we plot individual test-time training curves for several randomized environments. The loss function has evolved in such a way to adapt its gradient magnitude to the current agent state: for example in the DirectionalHalfCheetah experiments, the agent first ramps up its velocity in one direction (visible by a increasing KL-divergence) until it realizes whether it is going in the right/wrong direction. Then it either further ramps up the velocity through stronger gradients, or emits a turning signal via a strong gradient spike (e.g., visible by the spikes in Figure 23 (a) in column three).

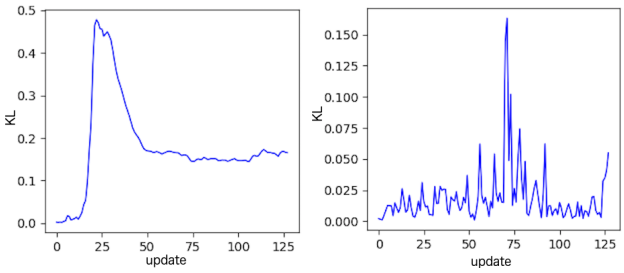
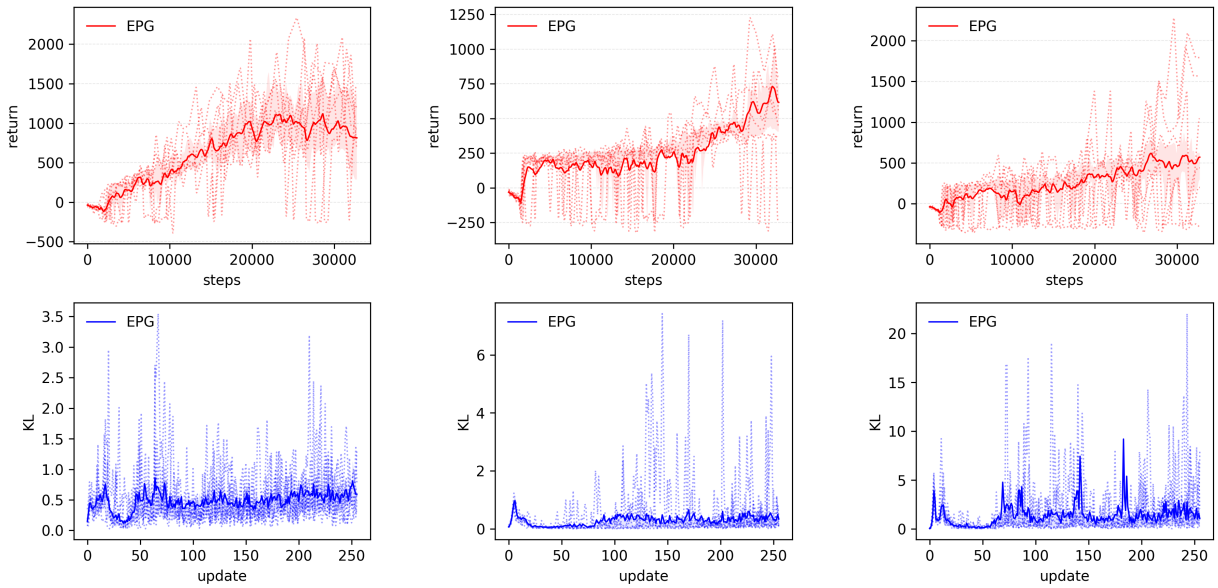


Figure 17: EPG on the RandomHopper environment: the KL-divergence between the policy before and after an update at the first epoch (left) vs the final epoch (right), w.r.t the number of updates so far, for a single inner loop run. These curves are generated with $\alpha = 0$ in Eq. (10).

In other experiments, such as Figures 8, 9, and 10, we see a similar pattern. Based on the agent's learning history, the gradient magnitudes are scaled accordingly. Often the gradient will be small initially, as it gets increasingly larger the more environment information it has encountered.



(a) 2 layers of 256 tanh units (b) 2 layers of 64 ReLU units (c) 4 layers of 64 tanh units

Figure 18: Transferring EPG (metalearned using 2-layer 64 tanh-unit policies on RandomWalker as in Figure 6) to policies of unseen configurations at test time

Effect of evolving policy initialization Prior works such as MAML (12) metalearn the policy initialization over a task distribution. While our proposed method, EPG, evolves the loss function parameters, we can also augment Algorithm 1 with simultaneously evolving the policy initialization in the ES outer loop. We investigate the benefits of evolving the policy initialization on top of EPG and PPO on our randomized environments. Figure 15 shows the comparison between EPG, EPG with evolved policy initialization (EPG+I), PPO, and PPO with evolved policy initialization (PPO+I). Evolving the policy initialization seems to help the most when the environments require little exploration, such as RandomWalker. However, the initialization plays a far less important role in DirectionalHalfCheetah and especially the GoalAnt environment. Hence the smaller performance difference between EPG and EPG+I.

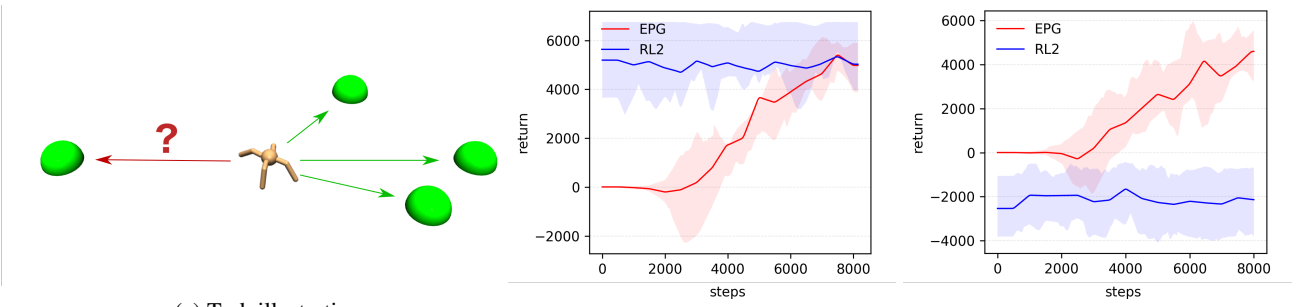
4.3. Generalization

Key components of Algorithm 1 include inner-loop training horizon U, the agent's policy architecture π_θ, and the task distribution $p(\mathcal{M})$. In this section, we investigate the test-time generalization properties of EPG: generalization to longer training horizons, to different policy architectures, and to out-of-distribution tasks.

Longer training horizons We evaluate the effect of transferring to longer agent training periods at test time on the RandomHopper environment by increasing the test-time training steps U in Algorithm 2 beyond the inner-loop training steps U of Algorithm 1. Figure 12 (A) shows that the learning curve declines and eventually crashes past the train-time horizon, which demonstrates that Algorithm 1 has limited generalization beyond EPG's inner-loop training steps. However, we overcome this limitation by initializing each inner-loop policy with randomly sampled policies that have been obtained by inner-loop training in past epochs. Figure 12 (B) illustrates continued learning past the train-time horizon, validating that this modification effectively makes the learned loss function robust to longer training length at test time.

Different policy architectures We evaluate EPG's transfer to different policy architectures by varying the number of hidden layers, the activation function, and hidden units of the agent's policy at test time (Algorithm 2), while keeping the agent's policy fixed at 2-layer with 64 tanh units during training time (Algorithm 1) on the RandomWalker environment. The test-time training curves on varied policy architectures are shown in Figure 18. Although compared to the learning curve Figure 6 with the same train-time and test-time policy architecture, the transfer performance is inferior. We still see that EPG produces a learned loss function that generalizes to policies other than it was trained on, achieving non-trivial walking behavior.

Out-of-distribution task learning We evaluate generalization to out-of-distribution task learning on the GoalAnt



(a) Task illustration

(b) Right direction (as metatrained)

(c) Left direction (generalization)

Figure 19: Generalization in the GoalAnt experiment: the ant has only been metatrained to reach target on the positive x-axis (its right side). Can it generalize to targets on the negative x-axis (its left side)?

environment. During metatraining, goals are randomly sampled on the positive x-axis and at test time, we sample goals from the negative x-axis. Figure 19 (a) illustrates this generalization task. We compare the performance of EPG against RL^2 (10).

First, we evaluate both methods' performance when the test-time task is sampled from the training-time task distribution. Figure 19 (b) shows the test-time training curve of both RL^2 and EPG when the test-time goals are sampled from the positive x-axis. As expected, RL^2 solves this task extremely fast, since it couples both the learning algorithm and the policy. EPG performs very well on this task as well, learning a randomly initialized policy from scratch in 8192 steps, with final performance matching that of RL^2.

Next, we look at the generalization setting with test-time goals sampled from the negative x-axis. Figure 19 (b) displays the test-time training curves of both methods. RL^2 seems to have completely overfit to the task distribution, it does not have the ability to learn a more general learning algorithm. In contrast, EPG learns a loss function that trains agents to reach goals sampled from negative x-axis, never seen during metatraining. This demonstrates rudimentary generalization properties, as we expected from learning a loss function that is decoupled from the policy. Figure 16 also shows trajectories sampled during the learning process for this exact experimental setup.

5. Relation to Existing Literature

The concept of learning an algorithm for learning is quite general, and hence there exists a large body of somewhat disconnected literature on the topic.

To begin with, there exist several relevant and recent publications in the metalearning literature (11; 10; 59). In (11), an algorithm named MAML is introduced. MAML treats the metalearning problem as in initialization problem. More specifically, MAML attempts to find a policy initialization from which only a minimal number of policy gradient steps are required to solve new tasks. This is accomplished by performing gradient descent on the original policy parameters with respect to the post policy update rewards. In Section 4.1 of Finn et al. (13), learning the MAML loss via gradient descent is proposed. Their loss has a more restricted formulation and relies on loss differentiability with respect to the objective function.

In a work concurrent with ours, Yu et al. (62) extended the model from (13) to incorporate a more elaborate learned loss function. The proposed loss function involves temporal convolutions over trajectories of experience, similar to the method proposed in this work. However, unlike our work, (62) primarily considers the problem of behavioral cloning. Typically, this means their method will require demonstrations, in contrast to our method which does not. Further, their outer objective does not require sequential reasoning and must be differentiable and their inner loop is a single SGD step. We have no such restrictions. Our outer objective is long horizon and non-differentiable and consequently our inner loop can run over tens of thousands of timesteps.

Another recent metalearning algorithm is RL^2 (10) (and related methods such as (59) and (25)). RL^2 is essentially a recurrent policy learning over a task distribution. The policy receives flags from the environment marking the end of episodes. Using these flags and simultaneously ingesting data for several different tasks, it learns how to compute gradient updates through its internal logic. RL^2 is limited by its decision to couple the policy and learning algorithm (using recurrency for both), whereas we decouple these components. Due to RL^2's policy-gradient-based optimization procedure, we see that it does not directly optimize final policy performance nor exhibit exploration. Hence, extensions have been proposed such as E-RL^2 (53) in which the rewards of episodes sampled early in the learning process are deliberately set to zero to drive exploratory behavior.

Further research on meta reinforcement learning comprises a vast selection. The literature's vastness is further compli-

cated by the fact that the research appears under many different headings. Specifically, there exist relevant literature on: life-long learning, learning to learn, continual learning, and multi-task learning. For example, (41; 40) consider self modifying learning machines (genetic programs). If we consider a genetic program that itself modifies the learned genetic program, we can subsequently derive a meta-GP approach (See (53), for further discussion on how this method relates to the more recent meta learning literature discussed above). The method described above is sufficiently general that it encompass most modern metalearning approaches. For a further review of other meta learning approaches, see the review articles (48; 57; 58) and citation graph they generate.

There are several other avenues of related work that tackle slightly different problems. For instance, several methods that attempt to learn a reward function to drive learning. See (7) (which suggests learning from human feedback) and the field of Inverse Reinforcement Learning (28) (which recovers the reward from demonstrations). Both of these fields relate to our ideas on loss function learning. Similarly, (29; 30) apply population-based evolutionary algorithms to reward function learning in gridworld environments. This algorithm is encompassed by the algorithms we present in this paper. However, it is typically much easier since learning just the reward function is in many cases a trivial task (e.g., in learning to walk, mapping the observation of distance to a reward function). See also (50) and (1) for additional evolutionary perspectives on reward learning. Other reward learning methods include the work of Guo et al. (14; 49), which focus on learning reward bonuses and the work of Sorg et al. (51), which focuses on learning reward functions through gradient descent. These bonuses are typically designed to augment but not replace the learned reward and have not easily shown to generalize across broad task distributions. Reward bonuses are closely linked to the idea of curiosity, in which an agent attempts to learn an internal reward signal to drive future exploration. Schmidhuber (39) was perhaps the first to examine the problem of intrinsic motivation in a metalearning context. The proposed algorithms make use of dynamic programming to explicitly partition experience into checkpoints. Further, there is usually little focus on metalearning the curiosity signal across several different tasks. Finally, the work of (17; 61; 2; 22; 24) studies metalearning over the optimization process in which metalearner makes explicit updates to a parametrized model in supervised settings.

Also worth mentioning is that approaches such as UVFA (38) and HER (3) which learn a universal goal-directed value function somewhat resemble EPG in the sense that their critic could be interpreted as a sort of loss function that is learned according to a specific set of rules. Furthermore, in DDPG (23), the critic can be interpreted in a similar way since it also makes use of back-propagation through a learned function into a policy network.

6. Discussion

In this paper, we introduced a new metalearning approach capable of learning a differentiable loss function over thousands of sequential environmental actions. Crucially, this learned loss is both highly adaptive (allowing for quicker learning of new tasks) and highly instructive (sometimes eliminating the need for environmental rewards at test time).

In certain cases, the adaptability of our learned loss is appreciated. For example, consider the DirectionalHopper experiments from Section 4. Here, the rewards at test time are impossible to infer from observations of the environment alone. Therefore, they cannot be completely internalized. However, if we do get to observe a reward signal on these environments, then we have shown that our algorithm *does* improve learning speed.

Meanwhile, in most other cases, our loss' instructive nature – which allows it to operate at test time without environmental rewards– is interesting and desirable. This instructive nature can be understood as the loss function's internalization of the reward structures it has previously encountered under the training task distribution. We see this internalization as a step toward learning intrinsic motivation. A good intrinsically motivated agent would successfully infer useful actions in new situations by using heuristics it developed over its entire lifetime. This ability is likely required to achieve truly intelligent agents (39).

There exist many interesting directions for future work. Our method demands sequential learning. That is to say, one must first perform outer loop update i before learning about update $i+1$. This can bottleneck the metalearning cycle and create large computational demands. Indeed, the number of sequential steps for each inner-loop worker in our algorithm is $E \times U$, using notation from Algorithm 1. In practice, this value may be very high, for example, each inner-loop worker takes approximately 196 million steps to train the loss function used in the RandomReacher experiments (Figure 7). Finding ways to parallelize parts of this process, or increase sample efficiency, could greatly improve the practical applicability of our algorithm. Improvements in computational efficiency would also allow the investigation of more challenging tasks. Nevertheless, we feel the success on the environments we tested is non-trivial and provides a proof of concept of our method's power. It is our hope that perhaps one day, reinforcement learning agents will live spectacularly long lives with highly refined internal rewards.

Acknowledgments

We thank Igor Mordatch, Ilya Sutskever, John Schulman, and Karthik Narasimhan for helpful comments and conversations. We thank Maruan Al-Shedivat for assisting with the random MuJoCo environments.

References

[1] David Ackley and Michael Littman. Interactions between learning and evolution. *Artificial life II*, 10:487–509, 1991.

[2] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv preprint arXiv:1606.04474*, 2016.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[4] Leemon C Baird III. Advantage updating. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[6] Richard Y Chen, John Schulman, Pieter Abbeel, and Szymon Sidor. UCB exploration via Q-ensembles. *arXiv preprint arXiv:1706.01502*, 2017.

[7] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, pages 4302–4310, 2017.

[8] Richard Dearden, Nir Friedman, and David Andre. Model based bayesian exploration. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 150–159. Morgan Kaufmann Publishers Inc., 1999.

[9] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[10] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[11] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[12] Chelsea Finn and Sergey Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *arXiv preprint arXiv:1710.11622*, 2017.

[13] Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. *arXiv preprint arXiv:1709.04905*, 2017.

[14] Xiaoxiao Guo, Satinder Singh, Richard Lewis, and Honglak Lee. Deep learning for reward design to improve monte carlo tree search in atari games. *arXiv preprint arXiv:1604.07095*, 2016.

[15] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. *arXiv preprint arXiv:1702.08165*, 2017.

[16] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[17] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] J Zico Kolter and Andrew Y Ng. Near-bayesian exploration in polynomial time. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 513–520. ACM, 2009.

[20] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, pages 1008–1014, 2000.

[21] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40, 2017.

[22] Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

[23] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[24] Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Learning unsupervised learning rules. *arXiv preprint arXiv:1804.00222*, 2018.

[25] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. Meta-learning with temporal convolutions. *arXiv preprint arXiv:1707.03141*, 2017.

[26] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2772–2782, 2017.

[27] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

[28] Andrew Y Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *in Proc. 17th International Conf. on Machine Learning*. Citeseer, 2000.

[29] Scott Niekum, Andrew G Barto, and Lee Spector. Genetic programming for reward function search. *IEEE Transactions on Autonomous Mental Development*, 2(2):83–90, 2010.

[30] Scott Niekum, Lee Spector, and Andrew Barto. Evolution of reward functions for reinforcement learning. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 177–178. ACM, 2011.

[31] Brendan O'Donoghue, Remi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. Pgq: Combining policy gradient and q-learning. *arXiv preprint arXiv:1611.01626*, 2016.

[32] Georg Ostrovski, Marc G Bellemare, Aaron van den Oord, and Rémi Munos. Count-based exploration with neural density models. *arXiv preprint arXiv:1703.01310*, 2017.

[33] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, volume 2017, 2017.

[34] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

[35] I. Rechenberg and M. Eigen. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. 1973.

[36] Richard M Ryan and Edward L Deci. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary educational psychology*, 25(1):54–67, 2000.

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

[38] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320, 2015.

[39] Juergen Schmidhuber. Exploring the predictable. In *Advances in evolutionary computing*, pages 579–612. Springer, 2003.

[40] Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook. *Diploma thesis, TUM*, 1987.

[41] Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pages 199–226. Springer, 2007.

[42] John Schulman, Pieter Abbeel, and Xi Chen. Equivalence between policy gradients and soft q-learning. *arXiv preprint arXiv:1704.06440*, 2017.

[43] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

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

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

[46] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: mit einer vergleichenden Einführung in die Hill-Climbing-und Zufallsstrategie*. Birkhäuser, 1977.

[47] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

[48] Daniel L Silver, Qiang Yang, and Lianghao Li. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium: Lifelong Machine Learning*, volume 13, page 05, 2013.

[49] Satinder Singh, Richard L Lewis, and Andrew G Barto. Where do rewards come from.

[50] Satinder Singh, Richard L Lewis, Andrew G Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82, 2010.

[51] Jonathan Sorg, Richard L Lewis, and Satinder P Singh. Reward design via online gradient ascent. In *Advances in Neural Information Processing Systems*, pages 2190–2198, 2010.

[52] James C Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

[53] B. C. Stadie, G. Yang, R. Houthooft, X. Chen, Y. Duan, W. Yuhuai, P. Abbeel, and I. Sutskever. Some considerations on learning to explore via meta-reinforcement learning. In *International Conference on Learning Representations (ICLR), Workshop Track*, 2018.

[54] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[55] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[56] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #Exploration: A study of count-based exploration for deep reinforcement learning. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[57] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[58] Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in Neural Information Processing Systems*, pages 640–646, 1996.

[59] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[60] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[61] A Steven Younger, Sepp Hochreiter, and Peter R Conwell. Meta-learning with backpropagation. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3. IEEE, 2001.

[62] Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot imitation from observing humans via domain-adaptive meta-learning. *arXiv preprint arXiv:1802.01557*, 2018.

A. Environment Description

We describe the randomized environments used in our experiments in the following:

- *RandomHopper* and *RandomWalker*: randomized gravity, friction, body mass, and link thickness at metatraining time, using a forward-velocity reward. At test-time, the reward is not fed as an input to EPG.
- *RandomReacher*: randomized link lengths, using the negative distance as a reward at metatraining time. At test-time, the reward is not fed as an input to EPG, however, the target location is fed as an input observation.
- *DirectionalHopper* and *DirectionalHalfCheetah*[2]: randomized velocity reward function.
- *GoalAnt*: ant environment with randomized target location. The velocity to the target is fed in as a reward. The target location is not observed.
- *Fetch*: randomized target location, the reward function is the negative distance to the target. The reward function is not an input to the EPG loss function, but the target location is.

B. Additional Experiments

Learning without environment resets We show that it is straightforward to evolve a loss that is able to perform

[2]Environment sourced from http://github.com/cbfinn/maml_rl.

well on no-reset learning, such that the agent is never reset to a fixed starting location and configuration after each episode. Figure 20 shows the average return w.r.t. the epoch on the GoalAnt environment without reset. The ant continues learning from the location and configuration after each episode finishes and is reset to the starting point only when the target is reached. Qualitative inspection of the learned behavior shows that the agent learns how to reach the target multiple times during its lifetime.

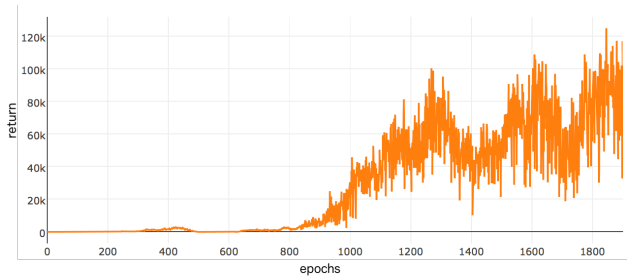


Figure 20: The average return w.r.t. the epoch on the GoalAnt environment with no reset.

EPG loss input sensitivity In the reward-free case (e.g., RandomHopper, RandomWalker, RandomReacher, and Fetch), the EPG loss function takes four kinds of inputs: observations, actions, termination signals, and policy outputs, and evaluates entire buffer with N transition steps. Which types of input and which time points in the buffer matter the most? In Figure 21, we plot the sensitivity of the learned loss function to each of these kinds of inputs by computing $||\frac{\partial L_{t=25}}{\partial x_t}||_2$ for different kinds of input x_t at different time points t in the input buffer. This analysis demonstrates that the loss is especially sensitive to experience at the current time step where it is being evaluated, but also depends on the entire temporal context in the input buffer. This suggests that the temporal convolutions are indeed making use of the agent's history (and future experience) to score the behavior.

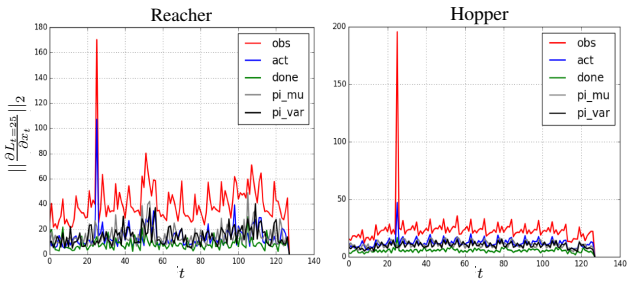
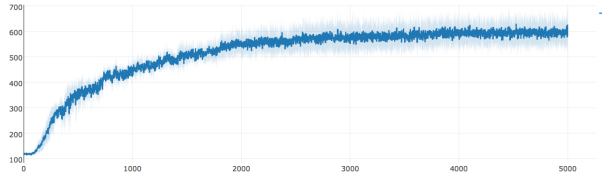
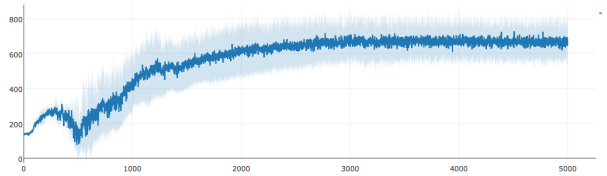


Figure 21: Loss input sensitivity: gradient magnitude of $L_{t=25}$ w.r.t. its inputs at different time steps within the input buffer. Notice not only the strong dependence on current time point ($t = 25$), but also the dependence on the entire buffer window.

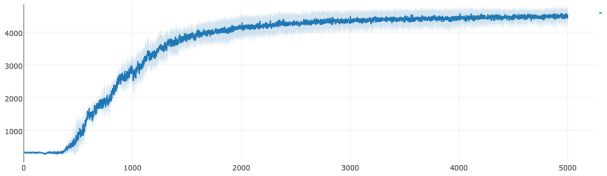
Training performance w.r.t. evolution epoch Figure 22 shows the metatraining-time performance (calculated based on the noise-perturbed loss functions) w.r.t. the number ES epochs so far, averaged across 256 different inner-loop workers for various random seeds, on several of our environments. This experiment highlights the stability of finding well-performing loss function via evolution.



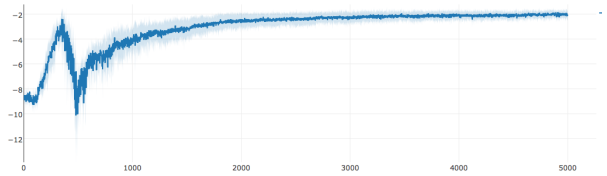
(a) RandomHopper



(b) DirectionalHopper



(c) GoalAnt



(d) Fetch

Figure 22: Final returns averaged across 256 inner-loop workers w.r.t. the number outer-loop ES epochs so far in EPG training (Algorithm 1). We run EPG training on each environment across 5 different random seeds and plot the mean and standard deviation as a solid line and a shaded area respectively.

Individual test-time training curves Figures 5, 9, and 10 show the test-time training trajectories of the EPG agent on RandomHopper, DirectionalHalfCheetah, and GoalAnt. A detailed plot of how individual learners behave in each environment is shown in Figure 23. Looking at both the

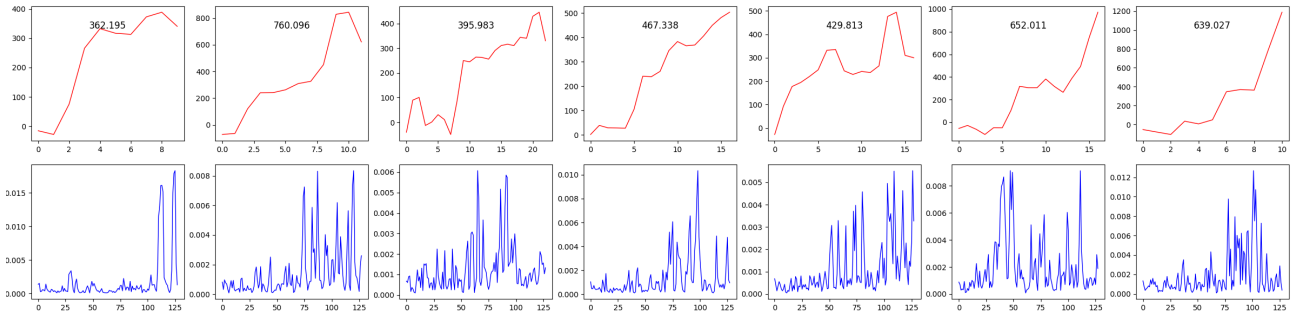
return and KL plots for the DirectionalHalfCheetah and GoalAnt environments, we see that the agent ramps up its velocity, after which it either finds out it's going in the right direction or not. If it is going in the wrong direction, it provides a counter signal, turns, and then ramps up its velocity in the appropriate direction, increasing its return. This demonstrates the exploratory behavior that occurs in these environments. In the RandomHopper case, only a slight period of system identification exists, after which the velocity of the hopper is quickly ramped up (visible by the increasing KL divergences).

C. Experiment Hyperparameters

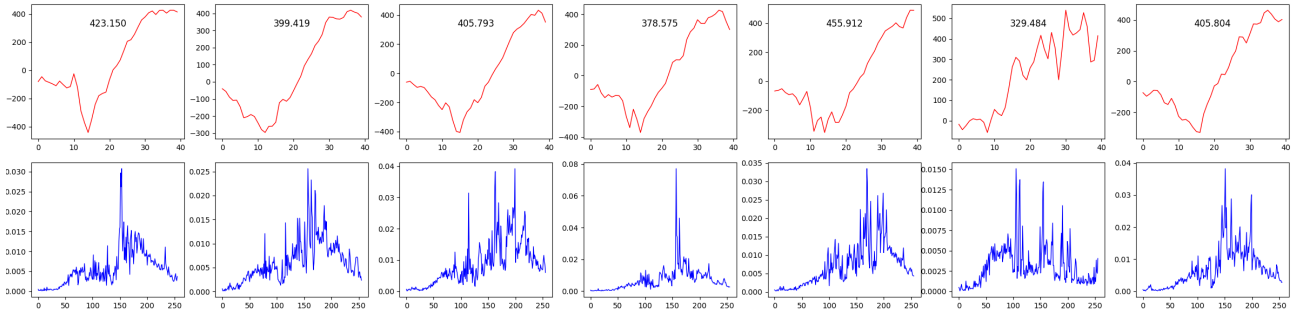
The experiment hyperparameters are listed in Table 1.

Environment	workers W	noise vectors V	update frequency M	updates	inner loop length
RandomHopper	256	64	64	128	8196
RandomWalker	256	64	128	256	32768
RandomReacher	256	64	128	512	65536
DirectionalHopper	256	64	64	128	8196
DirectionalHalfCheetah	256	64	32	256	8196
GoalAnt	256	64	32	512	16384
Fetch	256	64	32	256	8192

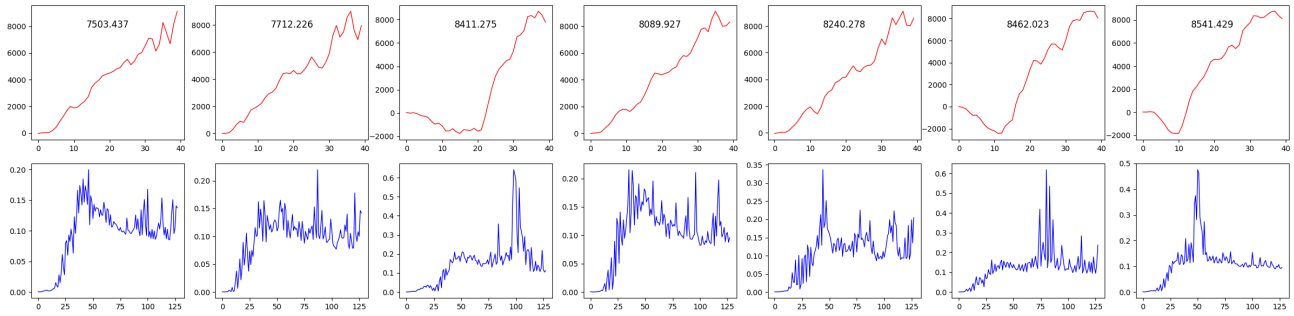
Table 1: EPG hyperparameters for different environments



(a) Different runs of the learning agent in Figure 5 (RandomHopper)



(b) Different runs of the learning agent in Figure 9 (DirectionalHalfCheetah)



(c) Different runs of the learning agent in Figure 10 (GoalAnt, but limited to forward/backward goals)

Figure 23: More test-time training curves in randomized environments. Each column represents a different sampled environment. The red curves plots the return w.r.t. the number of sampled trajectories during training, while the blue curves represent the KL divergence of the policy updates w.r.t. the number of policy updates. The number shown in each first row plot represents the final return averaged over the final 3 trajectories. The return curve (red) x-axis represent the number of trajectories sampled so far, while the KL-divergence (blue) x-axis represents the number of updates performed so far.