Deep Policy Generators

Anonymous Author(s)
Affiliation
Address
email

Abstract

Traditional Reinforcement Learning (RL) learns policies that maximize expected return. Here we study neural nets (NNs) that learn to generate policies in form of context-specific weight matrices, similar to Fast Weight Programmers and other methods from the 1990s. Using context commands of the form "generate a policy that achieves a desired expected return," our NN generators combine powerful exploration of parameter space with greedy command choices to iteratively find better and better policies. A form of weight-sharing HyperNetworks and policy embeddings scales our method to generate deep NNs. Experiments show how a single learned policy generator can produce policies that achieve any return seen during training. Finally, we evaluate our algorithm on a set of continuous control tasks where it exhibits competitive performance.

1 Introduction

Reinforcement learning (RL) (Kaelbling et al., 1996; Sutton and Barto, 2018) is a framework for training artificial agents to maximize rewards in possibly non-continuous, non-differentiable environments. Typical RL methods learn value functions (Sutton, 1988) or estimate stochastic policy gradients (with possibly high variance) (Williams, 1992; Sutton et al., 1999).

In contrast, Upside Down Reinforcement Learning (UDRL) (Srivastava et al., 2019; Schmidhuber, 2019) uses supervised learning to train RL agents conditioned on sequences of special command inputs of the form "act in the environment and achieve a desired return within so much time." Using hindsight learning, UDRL transforms the RL problem into the problem of predicting actions, given reward commands. This is quite powerful: when an UDRL agent interacts with an environment, given a random command c, and achieves return r, its behavior would have been optimal if the command had been r. Hence UDRL's parameters can be learned by maximizing the likelihood of the agent's behavior, given command r. Unfortunately, in the episodic setting, many behaviors may sastisfy the same command. Hence the function to be learned may be highly multimodal, and a simple maximum likelihood approach may fail to capture the variability in the data.

To overcome this limitation, we introduce *PoliGen*, a novel method to generate return-conditioned policies evaluated in parameter space. First, we use a Fast Weight Programmer (FWP) (Schmidhuber, 1992, 1993; Ha et al., 2016) to generate the parameters of a desired policy. Then, we evaluate the policy using a parameter-based value function (Harb et al., 2020; Faccio et al., 2020). This allows for end-to-end optimization of the generator producing return-conditioned policies by matching the commands (desired returns) to the observed returns.

The paper is structured as follows: Section 2 introduces the MDP frameworks for action-based and parameter-based methods; Section 3 reviews the concept of Fast Weight Programmers; Section 4 describes our method (PoliGen) including architectural choices; Section 5 evaluates our method on continuous control tasks where it demonstrates competitive performance. Our analysis shows how

a single learned policy generator can produce policies with any desired return seen during training. Finally, we discuss related and future work in Sections 6 and 7.

2 Background

We consider a Markov Decision Process (MDP) (Stratonovich, 1960; Puterman, 2014) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu_0)$. At each time step t, an artificial agent observes a state $s_t \in \mathcal{S}$, chooses an action $a_t \in \mathcal{A}$, obtains a reward $r_t = R(s_t, a_t)$ and transitions to a new state with probability $P(s_{t+1}|s_t, a_t)$. The initial state of the agent is chosen with probability μ_0. The behavior of the agent is expressed through its stochastic policy $\pi_\theta : \mathcal{S} \to \Delta(\mathcal{A})$, where $\theta \in \Theta$ are the policy parameters. If for each state s there is an action a such that $\pi_\theta(a|s) = 1$, we will call the policy deterministic. The agent interacts with the environment through episodes, starting from the initial states, and ending either when the agent reaches a set of particular states—these can be failing states or goal states—or when it hits a time horizon H. We define a trajectory $\tau \in \mathcal{T}$ as the sequence of state-action pairs that an agent encounters during an episode in the MDP $\tau = (s_{\tau,0}, a_{\tau,0}, s_{\tau,1}, a_{\tau,1}, \ldots, s_{\tau,T}, a_{\tau,T})$, where T denotes the time-step at the end of the episode ($T \leq H$). The return of a trajectory $R(\tau)$ is defined as the cumulative discounted sum of rewards over the trajectory $R(\tau) = \sum_{t=0}^{T} \gamma^t R(s_{\tau,t}, a_{\tau,t})$, where $\gamma \in (0, 1]$ is the discount factor.

The RL problem consists in finding the policy π_{θ^*} that maximizes the expected return obtained from the environment, i.e. $\pi_{\theta^*} = \arg\max_{\pi_\theta} J(\theta)$:

$$J(\theta) = \int_{\mathcal{T}} p(\tau|\theta) R(\tau) \, \mathrm{d}\tau, \tag{1}$$

where $p(\tau|\theta) = \mu_0(s0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$ is the distribution over trajectories induced by π_θ in the MDP. When the policy is stochastic and differentiable, by taking the gradient of $J(\theta)$ with respect to the policy parameters we obtain an algorithm called REINFORCE (Williams, 1992): $\nabla_\theta J(\theta) = \int_{\mathcal{T}} p(\tau|\theta) \nabla_\theta p(\tau|\theta) R(\tau) \, \mathrm{d}\tau$.

In parameter-based methods (Sehnke et al., 2010, 2008; Salimans et al., 2017; Mania et al., 2018), at the beginning of each episode, the weights of a policy are sampled from a distribution $\nu_\rho(\theta)$, called the hyperpolicy, which is parametrized by ρ. Typically, the stochasticity of the hyperpolicy is sufficient for exploration, and deterministic policies are used. The RL problem translates into finding the hyperpolicy parameters ρ maximizing expected return, i.e. $\nu_{\rho^*} = \arg\max_{\nu_\rho} J(\rho)$:

$$J(\rho) = \int_{\Theta} \nu_\rho(\theta) \int_{\mathcal{T}} p(\tau|\theta) R(\tau) \, \mathrm{d}\tau \, \mathrm{d}\theta. \tag{2}$$

This objective is maximized by taking the gradient of $J(\rho)$ with respect to the hyperpolicy parameters: $\nabla_\rho J(\rho) = \int_{\Theta} \int_{\mathcal{T}} \nu_\rho(\theta) \nabla_\rho \log \nu_\rho(\theta) p(\tau|\theta) R(\tau) \, \mathrm{d}\tau \, \mathrm{d}\theta$. This gradient can be either approximated through samples (Sehnke et al., 2010, 2008; Salimans et al., 2017) or estimated using finite difference methods (Mania et al., 2018). This only requires differentiability and stochasticity of the hyperpolicy.

It is intuitive to observe that, when the hyperpolicy becomes deterministic and the policy is stochastic, the dependency on ρ is lost and the policy parameters θ can be directly maximized using Equation 1. Since the optimization problem is episodic, we can set the discount factor γ to 1.

3 Fast Weight Programmers

Fast Weight Programmers (FWPs) (Schmidhuber, 1992, 1993) generate changes of weights of another network conditioned on some contextual input. In our case, the context corresponds to the desired return a generated policy should obtain. The outputs of the FWP are the policy parameters $\theta \in \Theta$. Formally, our FWP is a deterministic function $G_\rho : \mathbb{R}^{n_c} \longrightarrow \Theta$, where $c \in \mathbb{R}^{n_c}$ is the context-input and $\rho \in \mathrm{P}$ are the FWP parameters. Here, we consider a probabilistic FWP of the form $g_\rho(\theta|c) = G_\rho(c) + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ and σ is fixed. In this setting, the FWP conditioned on context c induces a probability distribution over the parameter space, similar to the one induced by the hyperpolicy in Section 2. If the FWP is used to generate the weights of a policy, we can rewrite the RL objective, making it context-dependent:

$$J(\rho, c) = \int_{\Theta} g_\rho(\theta|c) \int_{\mathcal{T}} p(\tau|\theta) R(\tau) \, \mathrm{d}\tau \, \mathrm{d}\theta. \tag{3}$$

Compared to Eq. 2, $J(\rho, c)$ induces a set of optimization problems that now are context-specific [1]. Here, $J(\rho, c)$ is the expected return for generating a policy with a generator parametrized by ρ, when observing context c. Instead of optimizing Eq. 2 using policy gradient methods, we are interested in learning a good policy through pure supervised learning by following a sequence of context-commands of the form "generate a policy that achieves a desired expected return." Under such commands, for any c, the objective $J(\rho, c)$ can be optimized with respect to ρ to equal c. FWPs offer a suitable framework for this setting, since the generator network can learn to create weights of the policy network so that it achieves what the given context requires.

4 Deep Policy Generators (PoliGen)

Here we develop *PoliGen*, our algorithm to generate policies that achieve any desired return. In the supervised learning scenario, it is straightforward to learn the parameters of the FWP that minimize the error $\mathcal{L}_G(\rho) = \mathbb{E}_{c \in D, \theta \sim g_\rho(\cdot|c)}[(J(\theta) - c)^2]$, where the context c comes from some set of possible commands D. This is because in supervised learning $J(\theta)$ the return is a differentiable function of the policy parameters. However, in RL, it is not the case. Therefore, we learn an evaluator function $V_{\mathbf{w}} : \Theta \longrightarrow \mathbb{R}$ parametrized by $\mathbf{w}$ that estimates $J(\theta)$ using supervised learning (Faccio et al., 2020). This function is a map from the policy parameters to the expected return. Once V is learned, the objective $\mathcal{L}_G(\rho)$ can be optimized end-to-end, like in the supervised learning scenario, to directly learn the generator's parameters. Concretely, we minimize $\mathcal{L}_G(\rho) = \mathbb{E}_{c \in D}[(V_{\mathbf{w}}(G_\rho(c)) - c)^2]$ to learn the parameters ρ.

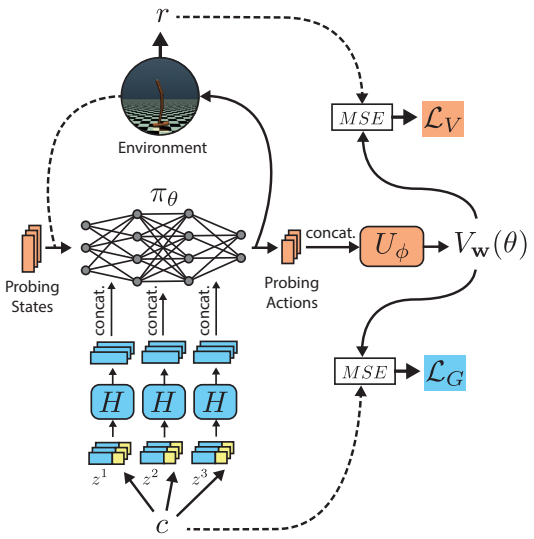

Figure 1: PoliGen generates policies conditioned on a desired return using a Fast Weight Programmer (HyperNetwork) and evaluates the resulting policy using a parameter-based value function based on fingerprinting. This enables training using supervised learning.

Our method is described in Algorithm 1 and consists of three steps. **First**, at each iteration a command c is chosen following some strategy. Ideally, to ensure that the generated policies are better and better, one should query the generator for larger returns. We discuss command strategies in the next paragraph. The generator observes c and produces policy π_θ which is run in the environment. The return and the policy (r, θ) are then stored in a replay buffer. **Second**, the evaluator function is trained to predict the return of the policies observed during training. This is achieved by minimizing MSE loss $\mathcal{L}_V(w) = \mathbb{E}_{(r,\theta) \in B}[(r - V_{\mathbf{w}}(\theta))^2]$. **Third**, we use the learned evaluator to directly minimize $\mathcal{L}_G(\rho) = \mathbb{E}_{r \in B}[(r - V_{\mathbf{w}}(G_\rho(r)))^2]$.

[1] Note the generality of Eq. 3. In supervised learning, common FWP applications include the case where g is deterministic, θ are the weights of an NN (possibly recurrent), $p(\tau|\theta)$ is the output of the NN given a batch of input data, $R(\tau)$ is the negative supervised loss.

Algorithm 1 PoliGen with return commands

Input: Differentiable generator $G_\rho : \mathcal{R} \to \Theta$ with parameters ρ; differentiable evaluator $V_\mathbf{w} : \Theta \to \mathcal{R}$ with parameters $\mathbf{w}$; empty replay buffer D

Output : Learned $V_\mathbf{w} \approx V(\theta) \forall \theta$, learned G_ρ $s.t.$ $V(G_\rho(r)) \approx r \forall r$

Initialize generator and critic weights $\rho, \mathbf{w}$, set initial return command $c = 0$
repeat:
  Sample policy parameters $\theta \sim g_\rho(\theta, c)$
  Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$ with policy π_θ
  Compute return $r = \sum_{k=1}^{T} r_k$
  Store (r, θ) in the replay buffer D
  for many steps **do**:
    Sample a batch $B = \{(r, \theta)\}$ from D
    Update evaluator by stochastic gradient descent: $\nabla_\mathbf{w} \mathbb{E}_{(r,\theta)\in B}[(r - V_\mathbf{w}(\theta))^2]$
  end for
  for many steps **do**:
    Sample a batch $B = \{r\}$ from D
    Update generator by stochastic gradient descent: $\nabla_\rho \mathbb{E}_{r\in B}[(r - V_\mathbf{w}(G_\rho(r)))^2]$
  end for
  Set next return command c using some strategy
until convergence

Choosing the command The strategy of choosing the command c before interacting with the environment is important. Intuitively, asking the generator to produce low return policies will not help finding better policies. On the other hand, asking for too much will produce policies that are out of distribution from the training data, and the generator cannot be trusted to produce such values. Hence it is reasonable to ask the generator to produce a return close to the highest one observed so far. More on the command strategy can be found in Section 5.1.

Scaling to deep policies Both generating and evaluating the weights of a deep NN policy using simple feedforward architectures like an MLP is very difficult for larger policies. The sheer number of policy weights, as well as their lack of easily recognizable structure, requires special solutions for generator and evaluator. To scale FWPs to deep policies, we rely on the relaxed weight-sharing of HyperNetworks (Ha et al., 2016) for the generator, and on a fingerprinting mechanism (Harb et al., 2020) for the evaluator. We discuss these two approaches in the next section.

4.1 HyperNetwork

The idea behind certain feed-forward FWPs called HyperNetworks (Ha et al., 2016) is to split the parameters of the generated network θ into smaller slices s_l. A shared NN H with parameters ξ receives as input a learned embedding z_l and outputs the slice s_l, i.e. $s_l = H_\xi(z_l)$ for each l. Following von Oswald et al. (2020), further context information can be given to H in form of an additional conditioning input c, which can be either either scalar or vector-sized: $s_l = H_\xi(z_l, c)$. Then the weights are combined by concatenating all generated slices:

$$\theta = [s_1 \quad s_2 \quad s_3 \quad \ldots]. \tag{4}$$

The splitting of θ into slices and the choice of H depend on the specific architecture of the generated policy. Here we are interested in generating MLP policies. The policy's parameters θ consist of weight matrices K^j, where j indicates the layer. We use an MLP H_ξ to generate each slice of each weight matrix: the HyperNetwork generator G_ρ splits each weight matrix into slices $s^j_{mn} \in \mathbb{R}^{f\times f}$, where j is the policy layer, and m, n are indexes of the slice in weight matrix of layer l. For each of these slices, a small embedding vector $z^j_{mn} \in \mathbb{R}^d$ is learned. Our network H_ξ is an MLP, followed by a reshaping operation that turns a vector of size f^2 into a $f \times f$ matrix:

$$s^j_{mn} = H_\xi(z^j_{mn}, c). \tag{5}$$

The slices are then concatenated over two dimensions to obtain the full weight matrices:

$$K^j = \begin{bmatrix} s^j_{11} & s^j_{12} & \dots \\ s^j_{21} & s^j_{22} & \\ \vdots & & \ddots \end{bmatrix}. \tag{6}$$

The full HyperNetwork generator G_ρ consists of the shared network H_ξ, as well as all embeddings z^j_{mn}. Its learnable parameters are $\rho = \{\xi, z^j_{mn} \forall m, n, j\}$.

Our aim for the generator G_ρ is to dynamically generate policy parameters, conditioned on the total return these policies should achieve. For this, our conditioning input c is simply this scalar return command. It is appended to each learned slice embedding z^j_{mn}. The resulting vectors are the inputs to the network H. Figure 2 shows a diagram this process.

For the the slicing to work, the width and height of the weight matrices have to be multiples of f. For the hidden layers of an MLP, this is easily achieved since we can freely choose the number neurons. For the input and output layers, however, we are constrained by the environment's observation and action dimension, respectively. To accommodate any number of input and output neurons, we use dedicated networks h_i and h_o for the input and output layer. The generated slices have the shape $f \times N_i$ for the input layer (N_i is the number of input neurons) and $N_o \times f$ for the output layer (N_o is the number of output neurons).

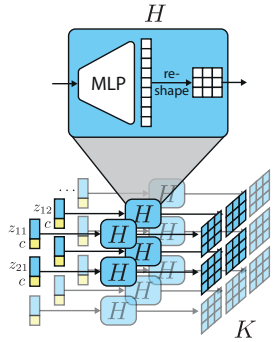

Figure 2: Generating a weight matrix K by concatenating slices that are generated from learned embeddings z and reward conditioning r using a shared network H.

4.2 Policy Fingerprinting

We use a policy fingerprinting mechanism (Harb et al., 2020) as an effective method to evaluate the performance of multiple NNs while using a single function. Policy fingerprinting works by giving a set of learnable probing states as input to the policy π_θ. The resulting outputs of the policy—called probing actions—are concatenated and given as input to an MLP U that computes the prediction $V_{\mathbf{w}}(\theta)$. Here the set of parameters $\mathbf{w}$ of this evaluator consists of the MLP parameters ϕ and all the parameters of the probing states. When training $V_{\mathbf{w}}$, the probing states learn to query the policy in meaningful situations, so that the policy's success can be judged by its probing actions. Fingerprinting is similar to a previous technique (Schmidhuber, 2015) where an NN learns to send queries (sequences of activation vectors) into another already trained NN, and learns to use the answers (sequences of activation vectors) to improve its own performance. Figure 1 shows a diagram of our method with a HyperNetwork generator and a fingerprinting value function.

The benefits of policy fingerprinting over directly observing policy weights become apparent as soon as we have at least one hidden layer in an MLP policy: the weights then have a large number of symmetries, i.e., many different weight configurations that are entirely equivalent in terms of the input-output mapping of the network. The main symmetries are across permutations of hidden neurons and scalings of the weight matrices (Kůrková and Kainen, 1994).

The probing actions of the fingerprinting mechanism are invariant with respect to these symmetries. They are, in fact, invariant even with respect to the general policy architecture. For our purposes, this brings advantages not only for the value function $V_{\mathbf{w}}$, but also for the generator: the gradients w.r.t. the generator's weights ρ are obtained by backpropagating through $V_{\mathbf{w}}$. If $V_{\mathbf{w}}$ is fingerprinting-based, these gradients will point only in directions which actually change the policy's probing actions. Consequently, the generator will waste no capacity trying to capture differences in the policy weights that have no effect on the policy's probing actions (which are proxies for the policy's general behavior in the environment).

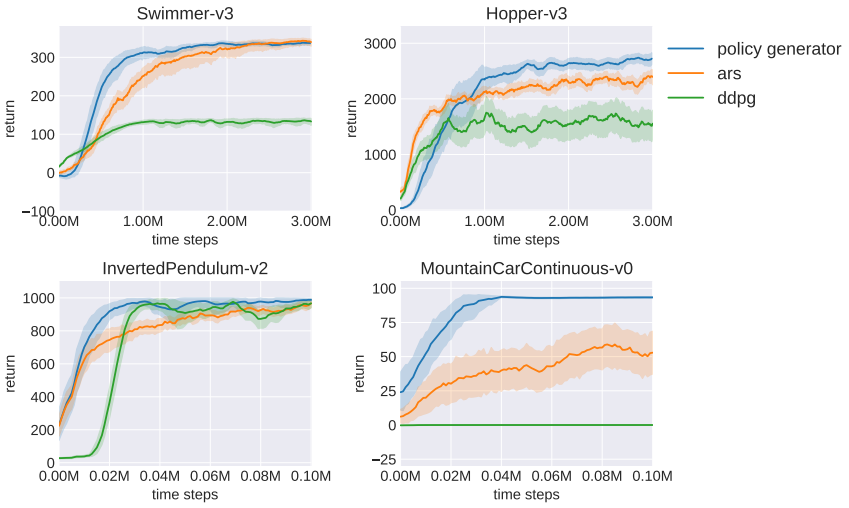

Figure 3: Performance of policies created with PoliGen (our method), ARS and DDPG over the course of training. Curves show the mean return and 95% bootstrapped confidence intervals from 20 runs as a function of total environment interactions.

5 Experiments

We empirically evaluate PoliGen as follows: First, we show competitive performance on common continuous control problems. Then we use the the learned fingerprinting mechanism to visualize the policies created by the generator over the course of training, and investigate its learning behavior.

5.1 Results on continuous control RL environments

We evaluate our method on continuous control tasks from the MuJoCo (Todorov et al., 2012) suite. Parameter-based Augmented Random Search (ARS) (Mania et al., 2018) serves as a strong baseline. We also compare our method to the Deep Deterministic Policy Gradient (DDPG) algorithm (Silver et al., 2014), another popular method for continuous control tasks.

In the experiments, all policies are MLPs with two hidden layers, each having 256 neurons. Our method uses the same set of hyper-parameters in all environments. For ARS, we tune step size, population size, and noise independently for each environment. For DDPG, we use the established set of default hyper-parameters. Details can be found in Appendix A.

We find that while always asking to generate a policy with return equal to the best return seen in the history, there is a slight advantage when asking for more than that. In particular, we demonstrate that a simple strategy such that "produce a policy whose return is 20 above the one of the best policy seen so far" can be very effective. We present an ablation showing that this strategy is slightly better than the strategy "produce a policy whose return equal to the one of the best policy seen so far" in Appendix B.4. This suggests that our method's improvements are due not only to the exploration in parameter space but also to the greediness of the command choice.

For our method and ARS, we use observation normalization (see Mania et al. (2018); Faccio et al. (2020)). Furthermore, following ARS, the survival bonus of +1 for every timestep is removed for the Hopper-v3 environment, since for parameter-based methods it leads to the local optimum of only staying alive without any movement.

In environments that do not have a fixed episode length, quickly failing bad policies from the early stages of training tend to dominate the replay buffer. To counteract this, we introduce a recency bias when sampling training batches from the buffer, assigning higher probability to newer policies. It is treated as an additional hyperparameter. Below we provide an ablation showing the importance of this component. Figure 3 shows our main experimental result.

Our Algorithm 1 performs competitively in the tested environments. In Swimmer and Hopper environments, our method learns faster than ARS, while eventually reaching the same asymptotic

performance. In MountainCarContinuous, DDPG is unable to explore the action space, and parameter-based methods quickly learn the optimal policy. For a comparison to UDRL with episodic resets, see Appendix B.1.

Obtaining suitable policies from the start When using a randomly initialized generator, we found that the weights of the policy generated were far from those of typical initialization schemes. In particular, the standard PyTorch (Paszke et al., 2019) initialization is uniform in $[-1/\sqrt{n}, 1/\sqrt{n}]$, where n is the number of neurons in the previous layer, resulting in a distribution uniform in $[-0.0625, 0.0625]$ in the second and last layers. Our network was approximately generating weights much larger and roughly uniform in every layer of the NN. We therefore scale our output such that it is close to the default initialization. Concretely, we multiply for each layer the output of the generator by $2/\sqrt{n}$, where n is the number of neurons in the previous layer. Here we provide an ablation showing that this choice is crucial. Figure 4 shows the importance of scaling the output of the generator in Swimmer and Hopper. We compare this with and without weighted sampling from the replay buffer. We observe that in Swimmer, output scaling is very important, while in Hopper, most of the performance gain is due to weighted sampling. This choice of output scaling is rather an heuristic and does not match the standard PyTorch initialization for all environments. Therefore it might happen that a randomly initialized generator produces policies that are difficult to perturb. We found some difficulties in training policies for the InvertedDoublePendulum due to this exploration issue. This highlights a possible limitation of our method.

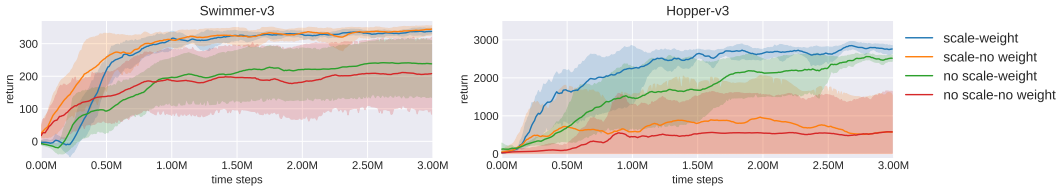

Figure 4: Comparison between our algorithm with/without weighted sampling from the replay buffer and output scaling. "No weight" denotes uniform sampling from the replay buffer. Average over 5 independent runs and 95% bootstrapped confidence intervals.

5.2 Analyzing the generator's learning process

The probing actions from the fingerprinting mechanism of the value function $V_{\mathbf{w}}$ can be seen as a compact meaningful policy embedding which can be used to visualize policies for a specific environment. We make use of this in Figure 5. It shows the return of all policies in the buffer after training, as well as the returns of policies created by the generator at different stages of training when given the same range of return commands.

The generator's objective can be seen as finding a trajectory through policy space, defined by the return commands, connecting the lowest with the highest return. In Figure 5, this corresponds to a trajectory going from a dark to a bright area. Indeed, we observe that the generator starts out being confined to the dark region (producing only bad policies) and over the course of training finds a trajectory leading from the darkest (low return) to the brightest (high return) regions.

Figure 6 shows the the returns achieved by policies that are created by a fully trained generator when given a range of return commands. This highlights a feature of the policy generator: while most RL algorithms generate only the best-performing policy, our generator is in principle able to produce by command policies across the whole performance spectrum. For the environments Swimmer and Hopper (Figures 6a and 6b), this works in a relatively reliable fashion. In Hopper the return used does not include survival bonus. A return of 2000 without survival bonus corresponds roughly to a return of 3000 with survival bonus.

It is worth noting, however, that in some environments it is hard or even impossible to achieve any given intermediate return. This might be the case, for example, if the optimal policy is much simpler than a slightly sub-optimal one, or if a large reward is given once a goal state is reached. We can observe this effect for the environments InvertedPendulum and MountainCar—see Figures 6c and 6d.

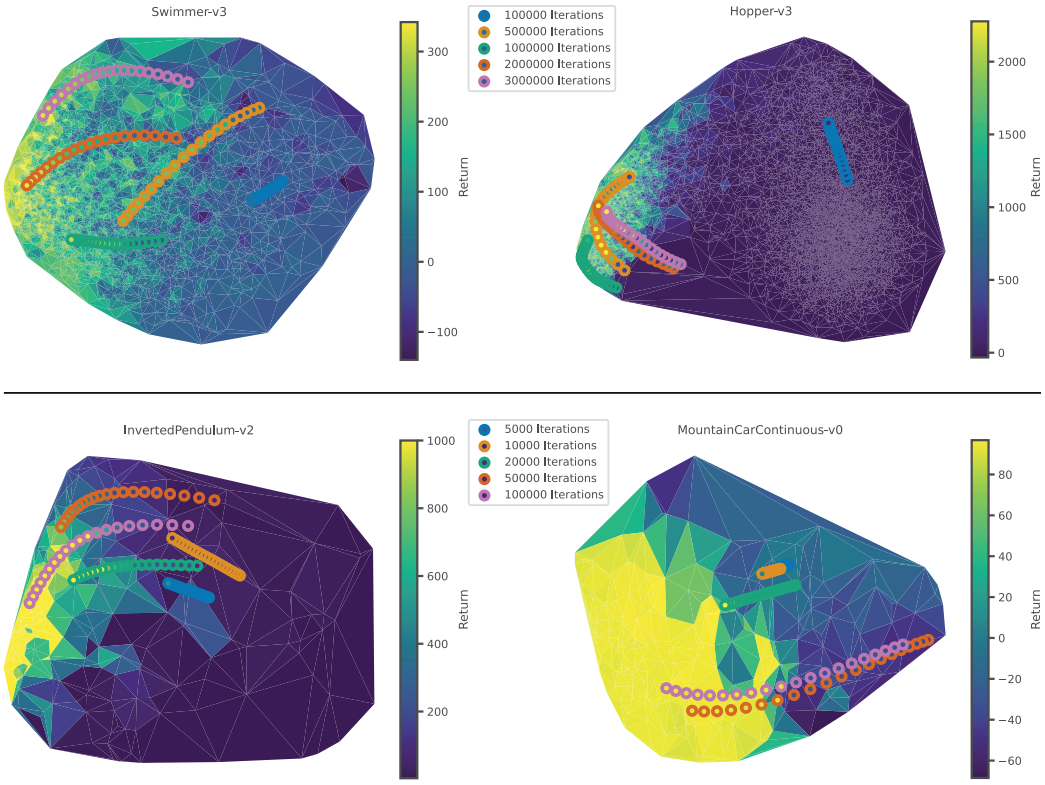

Figure 5: Policies generated by the generator during different stages of training. The generator is able to produce policies across the whole performance spectrum. To visualize this, each generator is given 20 return commands ranging from the minimum to the maximum possible reward in the environment. The color shows the achieved reward of each policy. The positions are determined by the probing actions (obtained by the final critic $V_{\mathbf{w}}$). The background shows the policies in the buffer, i.e. policies observed during training, and total reward. Probing actions are reduced to two dimensions using PCA of the buffer policies.

There the generator struggles to produce the desired identity line of return command and achieved return—instead we get something closer to a step function. However, this does not prevent our method from quickly finding optimal policies in these environments. More details in Appendix B.3.

6 Related Work

Policy conditioned value functions Compared to standard value functions that are conditioned on a specific policy, policy conditioned value functions can generate values across several policies (Faccio et al., 2020; Harb et al., 2020). This has been used to directly maximize the value using gradient ascent in the policy parameters. In the context of the present work, this can be used to evaluate any policy generated by our policy generator. In contrast to previous work, this allows for generating policies of an arbitrary quality in a zero-shot manner, without any gradient-based iterative training procedure.

Hindsight and Upside Down RL Upside Down RL (UDRL) transforms the RL problem into a supervised learning problem by conditioning the policy on commands such as the instruction to achieve a desired return (Schmidhuber, 2019; Srivastava et al., 2019). The required dataset of states, actions, and rewards can be collected online with iterative improvements to the policy (Srivastava et al., 2019), or offline (Janner et al., 2021; Chen et al., 2021). UDRL methods are related to hindsight RL when the instructed commands correspond to desired goals in the environment (Schmidhuber, 1991; Kaelbling, 1993; Andrychowicz et al., 2017; Rauber et al., 2018). Instead of optimizing the policy to achieve a desired reward in action space, our method PoliGen evaluates the generated

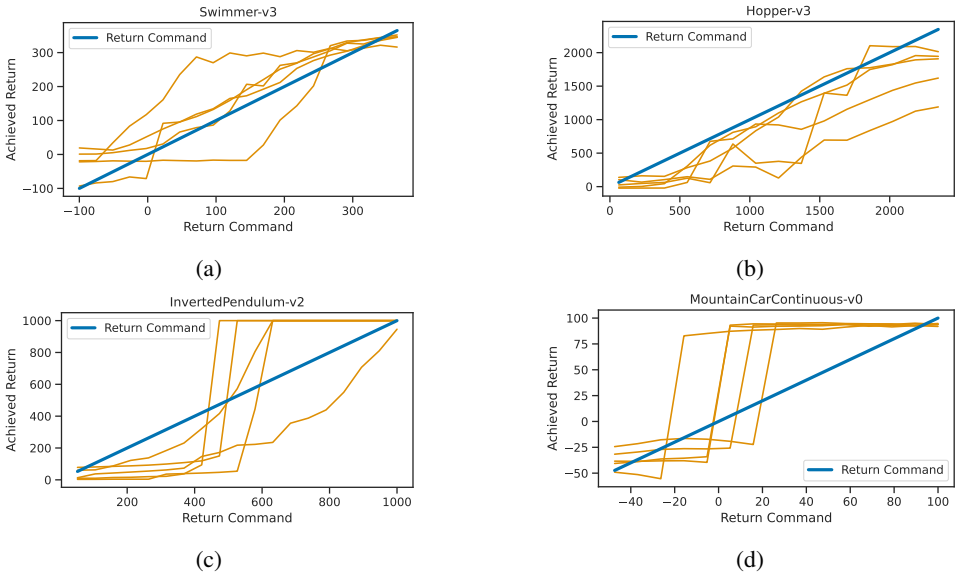

Figure 6: Achieved returns (mean of 10 episodes) of policies created by fully trained generators as a function of the given return command. A perfect generator would produce policies that lie on the diagonal identity line (if the environment permits such returns). For each environment, results of five independent runs are shown.

policies in command space. This is done by generating a command-conditioned policy that is then evaluated using a parameter-based value function and trained to match the command to the evaluated return.

Fast Weight Programmers and Hyper Networks The idea of using a neural network (NN) to generate weights for another NN dates back to Fast Weight Programmers (FWPs) (Schmidhuber, 1992, 1993), later scaled up to deeper neural networks under the name of HyperNetworks (Ha et al., 2016). While in traditional NNs the weight matrix remains fixed after training, FWPs make these weights context dependent. More generally, FWPs can be viewed as a neural function that involves multiplicative interactions and parameter sharing (Kirsch and Schmidhuber, 2021). When updated in recurrent fashion, FWPs can be used as memory mechanisms. Linear transformers are a type of FWP where information is stored through outer products of keys and values (Schlag et al., 2021). FWPs are used in the context of memory-based meta learning (Schmidhuber, 1993; Miconi et al., 2018; Gregor, 2020; Kirsch and Schmidhuber, 2021; Irie et al., 2021; Kirsch et al., 2022), predicting parameters for varying architectures (Knyazev et al., 2021), and reinforcement learning (Gomez and Schmidhuber, 2005; Najarro and Risi, 2020; Kirsch et al., 2022). In contrast to all of these approaches, ours uses FWPs to generate policies conditioned on a command (target return).

7 Conclusion and Future Work

This work introduced PoliGen, a framework for generating deep RL policies that achieve a given desired return. We showed that Hyper Networks in conjunction with fingerprinting-based value functions can be used to train a Fast Weight Programmer using supervised learning techniques that directly generates policy parameters with the desired return. By iteratively asking for higher returns than those that have been observed so far, our algorithm trains the generator to produce highly performant policies from scratch. We empirically demonstrate PoliGen's competitiveness to ARS and DDPG on continuous control tasks, and its ability to generate policies with any desired return.

A natural extension of this work will be the consideration of other context commands besides the desired return. Furthermore, latent variable models (e.g., conditional variational autoencoders) would allow the generator to capture a more diverse set of policies, which might benefit effective exploration in complex RL environments.

References

Achiam, J. (2018). Spinning Up in Deep Reinforcement Learning.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O., and Zaremba, W. (2017). Hindsight Experience Replay. In *NeurIPS*.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., and Mordatch, I. (2021). Decision Transformer: Reinforcement Learning via Sequence Modeling. *arXiv:2106.01345*.

Faccio, F., Kirsch, L., and Schmidhuber, J. (2020). Parameter-based value functions. *arXiv preprint arXiv:2006.09226*.

Gomez, F. J. and Schmidhuber, J. (2005). Co-evolving recurrent neurons learn deep memory pomdps. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 491–498, New York, NY, USA. ACM.

Gregor, K. (2020). Finding online neural update rules by learning to remember. *arXiv preprint arXiv:2003.03124*.

Ha, D., Dai, A., and Le, Q. V. (2016). HyperNetworks. In *International Conference on Learning Representations*.

Harb, J., Schaul, T., Precup, D., and Bacon, P.-L. (2020). Policy evaluation networks. *arXiv preprint arXiv:2002.11833*.

Hesterberg, T. C. (1988). *Advances in importance sampling*. PhD thesis, Stanford University.

Irie, K., Schlag, I., Csordás, R., and Schmidhuber, J. (2021). A modern self-referential weight matrix that learns to modify itself. In *Deep RL Workshop NeurIPS 2021*.

Janner, M., Li, Q., and Levine, S. (2021). Offline Reinforcement Learning as One Big Sequence Modeling Problem. In *NeurIPS*.

Kaelbling, L. P. (1993). Learning to Achieve Goals. In *IJCAI*.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

Kirsch, L., Flennerhag, S., van Hasselt, H., Friesen, A., Oh, J., and Chen, Y. (2022). Introducing Symmetries to Black Box Meta Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Kirsch, L. and Schmidhuber, J. (2021). Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34.

Knyazev, B., Drozdzal, M., Taylor, G. W., and Romero Soriano, A. (2021). Parameter prediction for unseen deep architectures. *Advances in Neural Information Processing Systems*, 34.

Kůrková, V. and Kainen, P. C. (1994). Functionally equivalent feedforward neural networks. *Neural Computation*, 6(3):543–558.

Mania, H., Guy, A., and Recht, B. (2018). Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1800–1809.

Miconi, T., Stanley, K., and Clune, J. (2018). Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568. PMLR.

Najarro, E. and Risi, S. (2020). Meta-learning through hebbian plasticity in random networks. *Advances in Neural Information Processing Systems*, 33:20719–20731.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Rauber, P., Ummadisingu, A., Mutz, F., and Schmidhuber, J. (2018). Hindsight policy gradients. In *International Conference on Learning Representations*.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Schlag, I., Irie, K., and Schmidhuber, J. (2021). Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR.

Schmidhuber, J. (1991). Learning to generate sub-goals for action sequences. In *Artificial neural networks*, pages 967–972.

Schmidhuber, J. (1992). Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139.

Schmidhuber, J. (1993). A 'self-referential'weight matrix. In *International Conference on Artificial Neural Networks*, pages 446–450. Springer.

Schmidhuber, J. (2015). On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *Preprint arXiv:1511.09249*.

Schmidhuber, J. (2019). Reinforcement Learning Upside Down: Don't Predict Rewards–Just Map Them to Actions. *arXiv:1912.02875*.

Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., and Schmidhuber, J. (2008). Policy gradients with parameter-based exploration for control. In Kůrková, V., Neruda, R., and Koutník, J., editors, *Artificial Neural Networks - ICANN 2008*, pages 387–396, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., and Schmidhuber, J. (2010). Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org.

Srivastava, R. K., Shyam, P., Mutz, F., Jaśkowski, W., and Schmidhuber, J. (2019). Training Agents Using Upside-down Reinforcement Learning. In *NeurIPS Deep RL Workshop*.

Stratonovich, R. (1960). Conditional Markov processes. *Theory of Probability And Its Applications*, 5(2):156–178.

Štrupl, M., Faccio, F., Ashley, D. R., Schmidhuber, J., and Srivastava, R. K. (2022). Upside-down reinforcement learning can diverge in stochastic environments with episodic resets. *arXiv preprint arXiv:2205.06595*.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, USA.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA. MIT Press.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033.

Veach, E. and Guibas, L. J. (1995). Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 419–428.

von Oswald, J., Henning, C., Sacramento, J., and Grewe, B. F. (2020). Continual learning with hypernetworks. In *8th International Conference on Learning Representations (ICLR 2020)(virtual)*. International Conference on Learning Representations.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer.

A Implementation details

A.1 Hyperparameters

Here we report the hyperparameters used for PoliGen and the baselines. For DDPG, we use the spinning-up RL implementation (Achiam, 2018), whose results are on par with the best reported results. For ARS, we use the implementation of the authors (Mania et al., 2018), adapted to Deep NN policies.

Shared hyperparameters The table below shows hyperparameters that are relevant to at least two of the three methods. They stay fixed across environments.

Hyperparameter	ARS	PoliGen	DDPG
Policy Architecture	MLP, 2 hidden layers, 256 neurons each, with bias		
Policy Nonlinearity	tanh		ReLU
Value Function Architecture		MLP, 2 hidden layers, 256 neurons each, with bias	
Value Function Nonlinearity		ReLU	
Initialization MLPs		PyTorch default (for value function)	PyTorch default (for actor & critic)
Batch Size		16	128
Optimizer		Adam	
Learning Rate Actor/Generator		2e-6	1e-3
Learning Rate Value Function		5e-3	1e-3
Exploration Noise Scale	tuned (see below)	0.1 in parameter space	0.1 in action space
Update Frequency Actor/Generator	every batch	every episode	every 50 time steps
Update Frequency Value Function		every episode	every 50 time steps
Number of Actor/Generator Updates		20	50
Number of Value Function Updates		5	50
Replay Buffer Size		10k	100k
Discount Factor	1		0.99
Survival Reward Adjustment	True (for Hopper)		False
Observation Normalization	True		False
Environmental interactions	100k for InvertedPendulum and MountCarContinuous, 3M for all other environments		

Hyperparameters for specific algorithms Fixed across environments:

PoliGen:

- Architecture of the networks H in the generator: MLP with bias, two hidden layers of size 256, ReLU nonlinearity and no output activation function
- Size of learnable HyperNetwork embeddings z^j_{mn}: 8
- Size of slices s^j_{mn} produced by the HyperNetwork: 16×16
- Number of probing states: 200
- Initialization of probing states: Uniformly random in $[0, 1)$
- Priority sampling from replay buffer: True, with weights $1/x^{1.1}$, where x is the number of episodes since the data was stored in the buffer

DDPG:

- Start-steps (random actions): 10000 time steps
- Update after (no training): 1000 time steps
- Polyak parameter: 0.995

Tuned hyperparameters For ARS, we tune the following hyperparameters for each environment separately using grid search:

- Step size for ARS: tuned with values in $\{1e-2, 1e-3, 1e-4\}$
- Number of directions and elite directions for ARS: tuned with values in $\{[1,1],[8,4],[8,8],[32,4],[32,16],[64,8],[64,32]\}$, where the first element denotes the number of directions and the second element the number of elite directions
- Noise for exploration in ARS: tuned with values in $\{0.1, 0.05, 0.025\}$

Here we report the best hyperparameters found for each environment:

ARS Hyperparameter	Swimmer	Hopper	Inverted-Pendulum	MountainCar-Continuous
Step Size	0.01	0.01	0.001	0.01
Number of Directions, Number of Elite Directions	(8, 4)	(8, 4)	(1, 1)	(1, 1)
Exploration Noise Scale	0.05	0.05	0.025	0.05

UDRL For UDRL we use a previous implementation (Srivastava et al., 2019) for discrete control environments, and implemented additional classes to use it in continuous control tasks with episodic resets (although the original UDRL report (Schmidhuber, 2019) focused on continuous control in single-life settings without resets). We use the previous hyperparameters (Srivastava et al., 2019) and tune learning rate (in $\{1e-3, 1e-4, 1e-5\}$), activation (ReLU, tanh), and their "last_few" parameter (1, 10, 100), which is used to select the command for exploration. For Swimmer, we are not able to reproduce the performance with the original reported hyperparameters. Like for the other algorithms, we use a NN with 2 hidden layers and 256 neurons per layer. Below we report the best hyperparameters found for UDRL.

UDRL Hyperparameter	Swimmer	Hopper	Inverted-Pendulum	MountainCar-Continuous
Nonlinearity	ReLU	ReLU	tanh	ReLU
Learning Rate	1e-3	1e-5	1e-3	1e-5
Last Few	10	10	1	1

A.2 Generator implementation details

Generating bias vectors Here we describe how to generate the bias vectors of the policies, which is not explicitly mentioned in section 4.1. Analogously to Equations 5 and 6, the embeddings z^j_{mn} are fed to a dedicated bias-generating network H_χ that produces slices of the shape $f \times 1$, and those slices are concatenated. Since we have a two-dimensional grid of learned embeddings z (see Figure 2), we take the mean across the input dimensions of the concatenated slices so that we end up with a bias vector (and not a matrix).

A.3 GPU usage / compute

We use cloud computing resources for our experiments. Our nodes have an Intel Xeon 12 core CPU and an NVIDIA Tesla P100 GPU with 16GB of memory. We were able to run four PoliGen experiments on one node in parallel. Our estimate of computation time for the main results is 40 node hours.

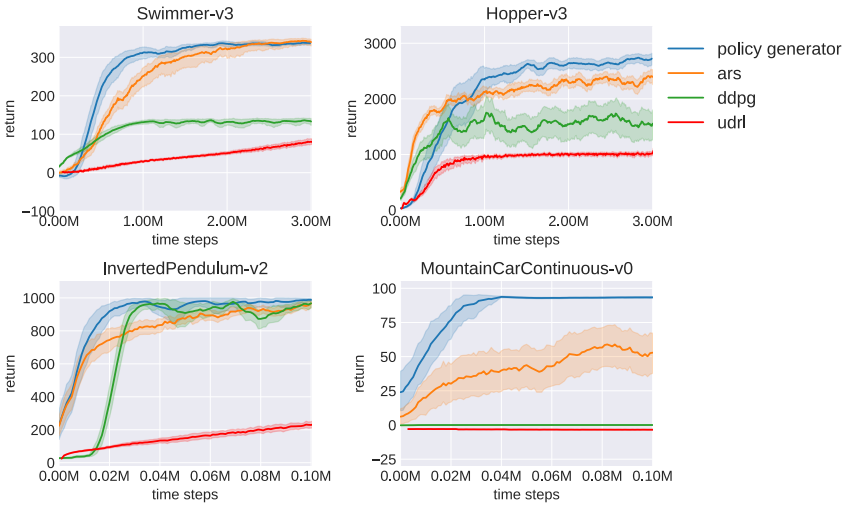

Figure 7: Performance of policies created with PoliGen (our method), ARS, DDPG and UDRL over the course of training. Curves show the mean return and 95% bootstrapped confidence intervals from 20 runs as a function of total environmental interactions.

B Experimental details

B.1 Additional experimental results

In Figure 7 we provide additional results to compare our method to UDRL with episodic resets. We confirm that UDRL is not sample efficient for continuous control in environments with episodic resets (Schmidhuber, 2019), in line with previous experimental results. We argue that the multimodality issue discussed in the introduction is the main issue with UDRL. Note that UDRL is also known to diverge in stochastic environments with episodic resets (Štrupl et al., 2022).

B.2 Main experiments on MuJoCo

For ARS and UDRL, the best hyperparameters for each environment are determined by running the algorithm with each hyperparameter configuration across 5 random seeds. The best configurations are those reported in section A.1 We use them for the final 20 evaluation runs shown in our main results. For DDPG and PoliGen, we use the same hyperparameters for all environments during the 20 evaluation runs.

For the results shown in Figures 3 and 7, we evaluate each run every 10000 time steps (in the case of Swimmer and Hopper, every 1000 for InvertedPendulum and MountainCarContinuous) for 10 episodes. Table 1 shows the final return and standard deviation of each algorithm.

Table 1: Final return (average over final 20 evaluations)

Environment	PoliGen	ARS	DDPG	UDRL
Swimmer-v3	$\mathbf{334 \pm 16}$	$\mathbf{342 \pm 21}$	129 ± 25	78 ± 17
MountainCarContinuous-v0	$\mathbf{93 \pm 1}$	55 ± 33	-1 ± 0.01	-3 ± 0.3
Hopper-v3	$\mathbf{2589 \pm 300}$	2340 ± 199	1634 ± 1036	1010 ± 78
InvertedPendulum-v2	$\mathbf{980 \pm 40}$	$\mathbf{936 \pm 42}$	$\mathbf{960 \pm 175}$	219 ± 299

B.3 Details on generated policy visualization

To create Figure 5, we perform Principal Component Analysis (PCA) on the probing actions of all policies in the buffer after training. The first two principal components indicate the position of a policy in our visualization. Using Delaunay triangulation, we assign an area to every policy and color it according to its achieved return. We then take the generator at different stages of training

(of the same run). Each of these generators is given a set of 20 commands, evenly spaced across the range of possible returns ($[-100, 365]$ for Swimmer, $[-100, 3000]$ for Hopper, $[0, 1000]$ for InvertedPendulum and $[-100, 100]$ for MountainCarContinuous). The resulting policies are plotted using probing actions on the probing states of the fully trained value function $V_{\mathbf{w}}$ and the same PCA.

B.4 Command strategies

We tried an alternative approach using Importance Sampling (Hesterberg, 1988) estimators. Given a mixture of weights $\beta_i(\theta)$, we considered estimators of the form $\hat{J}(c', w') = \sum_{i=1}^{N} \beta_i(\theta_i) \frac{p(\theta_i|c';w')}{p(\theta_i|c_i;w_i)} r_i$, which provides an unbiased estimate of the performance of a policy produced by a generator with parameters w' and command c', using past data derived from old generators with different commands. Maximizing $\hat{J}(c', w')$ with respect to the command c' should yield commands encouraging the generator to produce highly performant policies. We tested this using the Balance Heuristic (Veach and Guibas, 1995) estimator for β_k, which is known to have small variance. However, in our experiments we observed that generators using such command strategies did not significantly outperform the simple strategy mentioned earlier.

Ablation command Figure 8 shows that when choosing the command for exploration there is a slight advantage in asking to generate a policy whose return exceeds the best return so far by 20. However, just asking for the maximum return (drive parameter = 0) is also competitive.

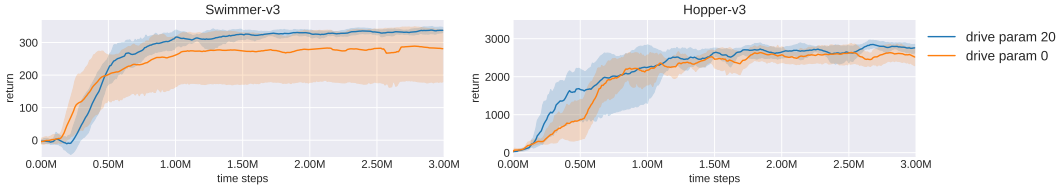

Figure 8: Comparison of variants of our algorithm with/without drive parameter for command exploration. Average over 5 independent runs and 95% bootstrapped confidence intervals.

C Environment details

MuJoCo (Todorov et al., 2012) is licensed under Apache 2.0.