Goal-Conditioned Generators of Deep Policies

Francesco Faccio*1 Vincent Herrmann*1 Aditya Ramesh1 Louis Kirsch1 Jürgen Schmidhuber1 2 3

Abstract

Goal-conditioned Reinforcement Learning (RL) aims at learning optimal policies, given goals encoded in special command inputs. Here we study goal-conditioned neural nets (NNs) that learn to generate deep NN 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 generalization across commands 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

General reinforcement learning (RL) is about training agents to execute action sequences that maximize cumulative rewards (Kaelbling et al., 1996; van Hasselt, 2012; Schmidhuber, 1990). Goal-conditioned RL agents can learn to solve many different tasks, where the present task is encoded by special command inputs (Schmidhuber & Huber, 1991; Schaul et al., 2015). Upside-down RL (UDRL) (Sri-vastava et al., 2019; Schmidhuber, 2019) and related methods (Ghosh et al., 2019), however, use supervised learning to train goal-conditioned RL agents. UDRL agents receive command inputs of the form “act in the environment and achieve a desired return within so much time” (Schmidhuber, 2019). Typically, hindsight learning (Andrychowicz et al., 2017; Rauber et al., 2018) is used to transform the RL problem into the problem of predicting actions, given reward commands. This is quite powerful. Consider a command-based agent interacting with an environment, given a random command c, and achieving return r. Its behavior would have been optimal if the command had been r. Hence the agent’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 satisfy 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.1

To overcome this limitation, we introduce PoliGen, a novel method for return-conditioned generation of 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, given a “desired return” command. 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 return-conditioned generator producing deep NN policies by matching the commands (desired returns) to the evaluated returns.

2. Background

We consider a Markov Decision Process (MDP) (Puterman, 2014) \( M = (\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu_0) \). 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 \( \mu_0 \). The behavior of the agent is expressed through its stochastic policy \( \pi_\theta : \mathcal{S} \rightarrow \Delta(\mathcal{A}) \), 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 \( \tau \in T \) as the sequence of state-
action pairs that an agent encounters during an episode in the MDP $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T, r_T)$, 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 $R(\tau) = \sum_{t=0}^{T} \gamma^t r(s_t, a_t)$, where $\gamma \in (0, 1)$ is the discount factor. The RL problem consists in finding the policy $\pi_\theta$ that maximizes the expected return obtained from the environment, i.e., $\pi_{\theta^*} = \arg\max_{\pi_\theta} J(\theta)$:

$$J(\theta) = \int_T p(\tau | \theta) R(\tau) \, d\tau,$$

where $p(\tau | \theta) = \mu_0(s_0) \prod_{t=0}^{T} \pi_\theta(a_t | s_t) P(s_{t+1} | s_t, a_t)$ is the distribution over trajectories induced by $\pi_\theta$ in the MDP.

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 $\nu(\rho)$, called the hyperpolicy, which is parametrized by $\rho$. Typically, the stochasticity of the hyperpolicy is sufficient for exploration, and deterministic policies are used. The RL problem translates into finding the hyperpolicy parameters $\rho$ maximizing expected return, i.e., $\nu_{\rho^*} = \arg\max_{\nu(\rho)} J(\rho)$:

$$J(\rho) = \int_\Theta \nu_\rho(\theta) \int_T p(\tau | \theta) R(\tau) \, d\tau \, d\theta.$$  (2)

This objective is maximized by taking the gradient of $J(\rho)$ with respect to the hyperpolicy parameters. It is intuitive to observe that, when the hyperpolicy becomes deterministic and the policy is stochastic, the dependency on $\rho$ is lost and the policy parameters $\theta$ can be directly maximized using Equation 1. Since the optimization problem is episodic, we can set the discount factor $\gamma$ to 1.

3. Fast Weight Programmers

Fast Weight Programmers (FWPs) (Schmidhuber, 1992; 1993) are NNs that generate changes of weights of another NN conditioned on some contextual input. In our UDR-like case, the context is the desired return to be obtained by a generated policy. The outputs of the FWP are the policy parameters $\theta \in \Theta$. Formally, our FWP is a function $G_\rho : \mathbb{R}^{n_\rho} \rightarrow \Theta$, where $c \in \mathbb{R}^{n_c}$ is the context-input and $\rho \in \mathbb{P}$ are the FWP parameters. Here, we consider a probabilistic FWP of the form $g_\rho(c) = G_\rho(c) + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ and $\sigma$ 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. Using the FWP to generate the weights of a policy, we can rewrite the RL objective, making it context-dependent:

$$J(\rho, c) = \int_\Theta g_\rho(c) \int_T p(\tau | \theta) R(\tau) \, d\tau \, d\theta.$$  (3)

Compared to Eq. 2, $J(\rho, c)$ induces a set of optimization problems that now are context-specific. Here, $J(\rho, c)$ is the expected return for generating a policy with a generator parametrized by $\rho$, 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 $\rho$ to equal $c$. FWP applications include the case where $g$ is deterministic, $\theta$ 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.

4. Deep Policy Generators (PoliGen)

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

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 $\mathcal{L}_G(\rho) = \mathbb{E}_{c \in \mathbb{D}, \theta \sim g_\rho(c)} [(J(\theta) - c)^2]$, where the context $c$ comes from some set of possible commands $\mathbb{D}$. This is because in supervised learning $J(\theta)$, the expected return, is a differentiable function of the policy parameters, unlike in general RL. Therefore, to make the objective differentiable, we learn an evaluator function $V_w : \Theta \rightarrow \mathbb{R}$ parametrized by $w$ that estimates $J(\theta)$ using supervised learning.
This function is a map from the policy parameters to the expected return. Once $V$ is learned, the objective $L_C(\rho)$ can be optimized end-to-end, like in the supervised learning scenario, to directly learn the generator’s parameters. Concretely, we minimize $L_C(\rho) = \mathbb{E}_{c \in D}[|V_w(G_\rho(c)) - c|^2]$ to learn the parameters $\rho$. Our method is described in Algorithm 1 and consists of three steps. First, in each iteration, a command $c$ is chosen following some strategy. Ideally, to ensure that the generated policies improve over time, the generator should be instructed to produce larger and larger returns. We discuss command strategies in the next section. The generator observes $c$ and produces policy $\pi_\theta$ which is run in the environment. The return and the policy $(r, \theta)$ 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 $L_V(w) = \mathbb{E}_{(r,\theta) \in B}[(r - V_w(\theta))^2]$. Third, we use the learned evaluator to directly minimize $L_C(\rho) = \mathbb{E}_{r \in B}[(r - V_w(G_\rho(r)))^2]$.

Scaling to deep policies Both generating and evaluating the weights of a deep feedforward MLP-based policy is difficult for large 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 parameter-based value functions (Faccio et al., 2020) using a fingerprinting mechanism (Harb et al., 2020) for the evaluator. We discuss these two approaches in Appendix B.

5. Experiments

We empirically evaluate PoliGen as follows: First, we show competitive performance on common continuous control problems. Then we use 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 hyperparameters 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 hyperparameters. Details can be found in Appendix B. We find that while always asking to generate a policy with return equal to the best return ever seen, there is a slight advantage when asking for more than that. In particular, we demonstrate that a simple strategy such as “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 C.4. This suggests that our method’s success is partially due to its ability to learn to explore performance improvements. 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 staying alive without any movement.

In tasks without 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. In Appendix C.2 we provide an ablation showing the importance of this component. Figure 2 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 C.1.

5.2. Analyzing the generator’s learning process

The probing actions created by the fingerprinting mechanism of the value function $V_w$ can be seen as a compact
Figure 3. 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 return in the environment. The color shows the achieved return of each policy. The positions are determined by the probing actions (obtained by the final critic $V^*$). The background shows the policies in the buffer, i.e., policies observed during training, and total reward. Probing actions are reduced to two dimensions by applying PCA to the buffer policies.

Figure 4. 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.

meaningful policy embedding useful to visualize policies for a specific environment. In Figure 3 we apply PCA to probing actions to show all policies in the buffer after training, as well as policies created by the generator at different stages of training when given the same range of return commands. Policies are colored in line with achieved return. 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 3, 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 4 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, 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 every 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. There the generator struggles to produce the desired identity 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 C.3.

6. Conclusion and Future Work

Our PoliGen is an RL framework for generating policies yielding given desired returns. Hypernetworks in conjunction with fingerprinting-based value functions can be used to train a Fast Weight Programmer through supervised learning to directly generate parameters of a policy that achieves a given return. By iteratively asking for higher returns than those observed so far, our algorithm trains the generator to produce highly performant policies from scratch. Empirically, PoliGen is competitive with ARS and DDPG on continuous control tasks, and able to generate policies with any desired return. Future work will consider context commands other than those asking for particular returns, as well as generators based on latent variable models (e.g., conditional variational autoencoders) allowing for capturing diverse sets of policies, to improve exploration of complex RL environments.
Acknowledgements

We thank Kazuki Irie, Mirek Strupl, Dylan Ashley, Robert Csordás, Aleksandar Stanić and Anand Gopalakrishnan for their feedback. This work was supported by the ERC Advanced Grant (no: 742870) and by the Swiss National Supercomputing Centre (CSCS, projects: s1090, s1154). We also thank NVIDIA Corporation for donating a DGX-1 as part of the Pioneers of AI Research Award and to IBM for donating a Minsky machine.

References


Algorithm 1 PoliGen with return commands

Input: Differentiable generator $G_\rho : \mathcal{R} \rightarrow \Theta$ with parameters $\rho$; differentiable evaluator $V_w : \Theta \rightarrow \mathcal{R}$ with parameters $w$; empty replay buffer $D$

Output: Learned $V_w \approx V(\theta) \forall \theta$, learned $G_\rho$ s.t. $V(G_\rho(r)) \approx r \forall r$

Initialize generator and critic weights $\rho, w$, 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 $\pi_\theta$
  Compute return $r = \sum_{k=1}^{T} r_k$
  Store $(r, \theta)$ in the replay buffer $D$
  for many steps do
    Sample a batch $B = \{(r, \theta)\}$ from $D$
    Update evaluator by stochastic gradient descent: $\nabla_w \mathbb{E}_{(r, \theta) \in B}[(r - V_w(\theta))^2]$  
  end for
  for many steps do
    Sample a batch $B = \{r\}$ from $D$
    Update generator by stochastic gradient descent: $\nabla_\rho \mathbb{E}_{r \in B}[(r - V_w(G_\rho(r)))^2]$  
  end for
  Set next return command $c$ using some strategy
until convergence

A. 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 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 & 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 & Schmidhuber, 2021; Irie et al., 2021; Kirsch et al., 2022), predicting parameters for varying architectures (Knyazev et al., 2021), and reinforcement learning (Gomez & Schmidhuber, 2005; Najarro & 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).
B. Implementation details

B.1. HyperNetwork

The idea behind certain feed-forward FWPs called hypernetworks (Ha et al., 2016) is to split the parameters of the generated network $\theta$ into smaller slices $s_l$. A shared NN $H$ with parameters $\xi$ 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-valued: $s_l = H_\xi(z_l, c)$. Then the weights are combined by concatenating all generated slices:

$$\theta = [s_1 \ s_2 \ s_3 \ldots].$$

The splitting of $\theta$ into slices and the choice of $H$ depend on the specific architecture of the generated policy. Here we are interested in generating MLP policies whose parameters $\theta$ consist of weight matrices $K_j$ with $j \in \{1, 2, \ldots, n_K\}$, where $n_K$ is the policy’s number of layers. We use an MLP $H_\xi$ to generate each slice of each weight matrix: the hypernetwork generator $G_\rho$ splits each weight matrix into slices $s_{mn}^j \in \mathbb{R}^{f \times f}$, 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 $z_{mn}^j \in \mathbb{R}^d$ is learned. Our network $H_\xi$ is an MLP, followed by a reshaping operation that turns a vector of size $f^2$ into an $f \times f$ matrix:

$$s_{mn}^j = H_\xi(z_{mn}^j, c).$$

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

$$K_j = \begin{bmatrix} s_{11}^j & s_{12}^j & \ldots \\ s_{21}^j & s_{22}^j & \ldots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

The full hypernetwork generator $G_\rho$ consists of the shared network $H_\xi$, as well as all embeddings $z_{mn}^j$. Its learnable parameters are $\rho = \{\xi, z_{mn}^j, \forall m, n, j\}$.

Generator $G_\rho$ is supposed to dynamically generate policy parameters, conditioned on the total return these policies should achieve. The conditioning input $c$ is simply this scalar return command. It is appended to each learned slice embedding $z_{mn}^j$. The resulting vectors are the inputs to the network $H$. Figure 5 shows a diagram of this process.

For the the slicing to work, the widths and heights 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 numbers of neurons. For the input and output layers, however, we are constrained by the dimensions of environmental observations and actions. To accommodate any number of input and output neurons, we use dedicated networks $H_i$ and $H_o$ for the input and output layers. 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).

B.2. Policy Fingerprinting

We use a policy fingerprinting mechanism (Harb et al., 2020) as an effective method to evaluate the performance of multiple NNs through a single function. Policy fingerprinting works by giving a set of learnable probing states as input to the policy $\pi_\rho$. The resulting outputs of the policy—called probing actions—are concatenated and given as input to an MLP $U$ that computes the prediction $V_w(\theta)$. Here the set of parameters $w$ of this evaluator consists of the MLP parameters $\phi$ and all the parameters of the probing states. When training $V_w$, the probing states learn to query the policy in meaningful situations, so that the policy’s success can be judged by its probing actions.

Figure 5. Generating a weight matrix $K$ by concatenating slices that are generated from learned embeddings $z$ and return conditioning $r$ using a shared network $H$. 
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 reflect possible permutations of hidden neurons and scalings of the weight matrices (Kůrková & Kainen, 1994).

The probing actions of the fingerprinting mechanism are invariant with respect to such symmetries. In fact, they are invariant even with respect to the general policy architecture. This entails advantages not only for the value function \( V_w \), but also for the generator: the gradients w.r.t. the generator’s weights \( \rho \) are obtained by backpropagating through \( V_w \). If \( V_w \) is fingerprinting-based, these gradients will point only in directions which, when followed, actually change the generated policy’s probing actions. Consequently, the generator will ignore potential policy weight changes that have no effect on the policy’s probing actions (which are proxies for the policy’s general behavior in the environment).

B.3. 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 relevant to at least two of the three methods. They stay fixed across environments.

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

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, no output activation function
- Size of learnable hypernetwork embeddings $z_{in}$: 8
- Size of slices $s_{in}$ produced by the hypernetwork: $16 \times 16$
- 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:

<table>
<thead>
<tr>
<th>ARS Hyperparameter</th>
<th>Swimmer</th>
<th>Hopper</th>
<th>Inverted-Pendulum</th>
<th>MountainCar-Continuous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Size</td>
<td>0.01</td>
<td>0.01</td>
<td>0.001</td>
<td>0.01</td>
</tr>
<tr>
<td>Number of Directions, Number of Elite Directions</td>
<td>(8, 4)</td>
<td>(8, 4)</td>
<td>(1, 1)</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>Exploration Noise Scale</td>
<td>0.05</td>
<td>0.05</td>
<td>0.025</td>
<td>0.05</td>
</tr>
</tbody>
</table>

**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 an NN with 2 hidden layers and 256 neurons per layer. Below we report the best hyperparameters found for UDRL.

<table>
<thead>
<tr>
<th>UDRL Hyperparameter</th>
<th>Swimmer</th>
<th>Hopper</th>
<th>Inverted-Pendulum</th>
<th>MountainCar-Continuous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonlinearity</td>
<td>ReLU</td>
<td>ReLU</td>
<td>tanh</td>
<td>ReLU</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>1e-3</td>
<td>1e-5</td>
<td>1e-3</td>
<td>1e-5</td>
</tr>
<tr>
<td>Last Few</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure 6. 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.4. 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 B.1. Analogously to Equations 5 and 6, the embeddings $z_{mn}$ are fed to a dedicated bias-generating network $H_\chi$ 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 5), 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).

B.5. 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.

C. Experimental details

C.1. Additional experimental results

In Figure 6 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.

C.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 B.3 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. For 10 episodes, Figures 2 and 6 evaluate each run every 10000 time steps for Swimmer and Hopper, every 1000 steps for InvertedPendulum and MountainCarContinuous. Table 1 shows the final return and standard deviation of each algorithm.
Obtaining suitable policies from the start Randomly initialized policy generators produce weights 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 tends to generate much larger weights, roughly uniform in every NN layer. 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 7 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 heuristic and does not match the standard PyTorch initialization for all environments. It might happen that a randomly initialized generator produces policies that are difficult to perturb. This exploration issue seems to cause some difficulties for InvertedDoublePendulum, highlighting a possible limitation of our method.

### C.3. Details on generated policy visualization

To create Figure 3, we perform Principal Component Analysis (PCA) on the probing actions of all policies in the buffer after training. The first two principal components indicate a policy’s position 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 $V_w$ (and the same PCA).

### C.4. Command strategies

In early experiments, 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 $\hat{J}(c', w') = \sum_{i=1}^N \beta_i(\theta) p(\theta_i | c'; w') r_i$, 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 & Guibas, 1995) estimator for $\beta_i$, which is known to have small variance (Papini et al., 2019). 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 for asking the generator for 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.

![Swimmer-v3](image1.png)  ![Hopper-v3](image2.png)

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.

D. Environment details

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