Goal-Conditioned Generators of Deep Policies

Francesco Faccio[1,2*], **Vincent Herrmann**[1*], **Aditya Ramesh**[1], **Louis Kirsch**[1], **Jürgen Schmidhuber**[1,2,3]

[1]The Swiss AI Lab IDSIA/USI/SUPSI, Lugano, Ticino, Switzerland
[2]AI Initiative, KAUST, Thuwal, Saudi Arabia
[3]NNAISENSE, Lugano, Switzerland
{francesco, vincent.herrmann, aditya.ramesh, louis, juergen}@idsia.ch

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. Our code is public.

Introduction

General reinforcement learning (RL) is about training agents to execute action sequences that maximize cumulative rewards in possibly non-continuous, non-differentiable, partially observable environments (Kaelbling, Littman, and Moore 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 and Huber 1991; Schaul et al. 2015).

Many RL methods learn value functions (Sutton and Barto 2018) or estimate stochastic policy gradients (with possibly high variance) (Williams 1992; Sutton et al. 1999). Upside-down RL (UDRL) (Srivastava 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 Gaussian maximum likelihood approach may fail to capture the variability in the data.[1]

To overcome this limitation, we introduce *GoGePo*, 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, Dai, and Le 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 (Faccio, Kirsch, and Schmidhuber 2021). 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.

The paper is structured as follows: Section introduces the MDP frameworks for action-based and parameter-based methods; Section reviews the concept of Fast Weight Programmers; Section describes GoGePo including architectural choices; Section evaluates our method on continuous control tasks where it demonstrates competitive performance. Our analysis shows how a single learned policy generator can produce policies yielding any desired return seen during training. Finally, we discuss related and future work in Sections and . Our implementations are publicly available[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

*These authors contributed equally.

[1]Note that in stochastic environments with episodic resets, certain UDRL variants will fail to maximize the probability of satisfying their commands (Štrupl et al. 2022).

[2]https://github.com/IDSIA/GoGePo

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 $\tau \in \mathcal{T}$ 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 $p(\tau|\theta) = \mu_0(s_0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$ is the distribution over trajectories induced by 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, Guy, and Recht 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, Guy, and Recht 2018). This only requires differentiability and stochasticity of the hyperpolicy.

For deterministic hyperpolicy and stochastic policy, 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.

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 UDRL-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 $G_\rho : \mathbb{R}^{n_c} \longrightarrow \Theta$, 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 . Using the FWP to generate the weights of a policy, we can rewrite the RL objective, making it context-dependent:

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

Compared to Eq. 2, $J(\rho, c)$ induces a set of optimization problems that now are context-specific [3]. 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.

Deep Policy Generators (GoGePo)

Here we develop *GoGePo*, 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 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 $V_{\mathbf{w}} : \Theta \longrightarrow \mathbb{R}$ parametrized by $\mathbf{w}$ that estimates $J(\theta)$ using supervised learning (Faccio, Kirsch, and Schmidhuber 2021). 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.

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 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

[3] 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.

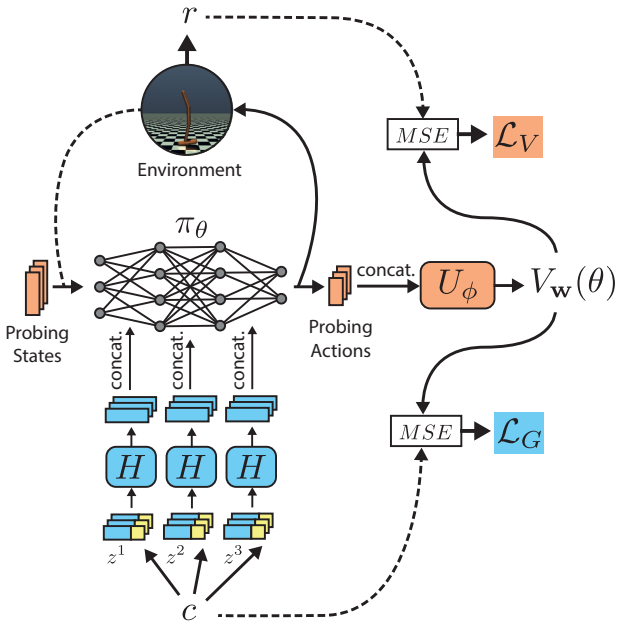

Figure 1: GoGePo generates policies using a Fast Weight Programmer (hypernetwork) conditioned on a desired return and evaluates the resulting policy using a parameter-based value function based on fingerprinting. This enables training using supervised learning.

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 $\mathcal{L}_G(\rho) = \mathbb{E}_{r\in B}[(r - V_{\mathbf{w}}(G_\rho(r)))^2]$.

By relying on an evaluator function V that maps policy parameters to expected return, we do not need to model multimodal behavior. The generator is trained to produce policies such that the scalar return command matches the evaluator's scalar return prediction. In practice, the generator finds for each return command c a point θ in the domain of the evaluator function such that $V(\theta) = c$. As a very simple example, assume $J(\theta) = \theta^2$, $\theta \in [-1, 1]$. If methods like UDRL observe $\theta_1 = 1/2$ with $r_1 = 1/4$ and $\theta_2 = -1/2$ with $r_2 = 1/4$, a Gaussian maximum likelihood approach will learn to map $r = 1/4$ to $\theta = 0$, but for $\theta = 0$ the return is 0. Our evaluator function learns to approximate $J(\theta)$, so our generator will learn to find a single θ such that $V(\theta) = 1/4$. The value of θ produced by the generator depends on the optimization process and on the shape of V. In other words, the multimodality issue is turned into having multiple optimal points in the optimization of the generator.

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 necessarily help finding better policies. On the other hand, asking for too much will produce policies that are out of distribution, given 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 command strategies can be found in Section .

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, Dai, and Le 2016) for the generator, and on parameter-based value functions (Faccio, Kirsch, and Schmidhuber 2021) using a fingerprinting mechanism (Harb et al. 2020; Faccio et al. 2022) for the evaluator. We discuss these two approaches in the next section.

HyperNetworks

The idea behind certain feed-forward FWPs called hypernetworks (Ha, Dai, and Le 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 for each l, i.e. $s_l = H_\xi(z_l)$. 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:

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

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 whose parameters θ 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_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

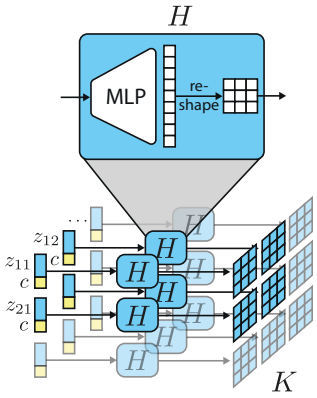

Figure 2: Generating a weight matrix K by concatenating slices that are generated from learned embeddings z and conditioning c using a shared network H.

Algorithm 1: GoGePo 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 $G_\rho \; s.t. \; V(G_\rho(r)) \approx r \forall r$

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

that turns a vector of size f^2 into an $f \times f$ 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:

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

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 $\rho = \{\xi, z^j_{mn} \forall m, n, j\}$.

Generator G_ρ 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^j_{mn}. The resulting vectors are the inputs to the network H. Figure 2 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).

Policy Fingerprinting

Recent work (Faccio et al. 2022) 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 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 reflect possible 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 such symmetries. In fact, they are invariant even with respect to the general policy architecture. This entails advantages not only for the value function $V_\mathbf{w}$, 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, when followed, actually yield changes of 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).

Experiments

We empirically evaluate GoGePo 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.

Results on Continuous Control RL Environments

We evaluate our method on continuous control tasks from the MuJoCo (Todorov, Erez, and Tassa 2012) suite. Augmented Random Search (ARS) (Mania, Guy, and Recht 2018), a competitive parameter-based method, serves as a strong baseline. We also compare our method to other popular algorithms for continual control tasks: Deep Deterministic Policy Gradients (DDPG) (Silver et al. 2014), Soft Actor Critic (SAC) (Haarnoja et al. 2018) and Twin Delayed Deep Deterministic Policy Gradients (TD3) (Fujimoto, Hoof, and Meger 2018). In addition, we include as a baseline UDRL (Srivastava et al. 2019) and confirm that UDRL is not sample efficient for continuous control in environments with episodic resets (Schmidhuber 2019), in line with previous experimental results.

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 and UDRL, we tune a set of hyperparameters separately for each environment (step size, population size, and noise for ARS; nonlinearity, learning rate and the "last few" parameter for UDRL). For DDPG, SAC and TD3, we use the established sets of default hyperparameters. Details can be found in Appendix A.

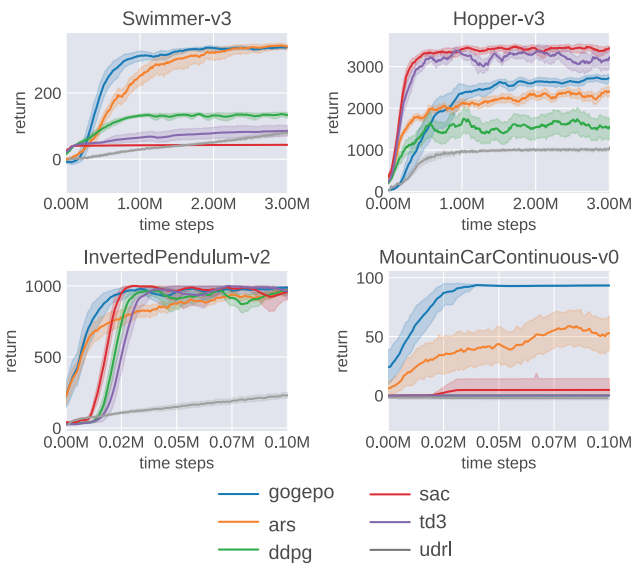

Figure 3: Performance of policies created with GoGePo (our method), ARS, DDPG, SAC, TD3 and UDRL over the course of training. Curves show the mean return and 95% bootstrapped confidence intervals from 20 runs as a function of total environment interactions.

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 B.3. This suggests that our method's success is not only due to random exploration in parameter space but also to generalization over commands: *it learns to understand and exploit the nature of performance improvements in a given environment.*

For our method and ARS, we use observation normalization (see (Mania, Guy, and Recht 2018; Faccio, Kirsch, and Schmidhuber 2021)). 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 Figure 7 in the Appendix we provide an ablation showing the importance of this component. Figure 3 shows our main experimental result (see also Table 1 in the Appendix).

Our Algorithm 1 performs very competitively in the tested environments with the exception of Hopper, where TD3 and SAC achieve higher expected return. In Swimmer and Hopper environments, our method learns faster than ARS, while eventually reaching the same asymptotic performance. In MountainCarContinuous, DDPG, SAC and TD3 are unable to explore the action space, and parameter-based methods quickly learn the optimal policy. Our method always outperforms UDRL.

Analyzing the Generator's Learning Process

The probing actions created by the fingerprinting mechanism of the value function $V_{\mathbf{w}}$ can be seen as a compact meaningful policy embedding useful to visualize policies for a specific environment. In Figure 4 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 4, 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 5 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

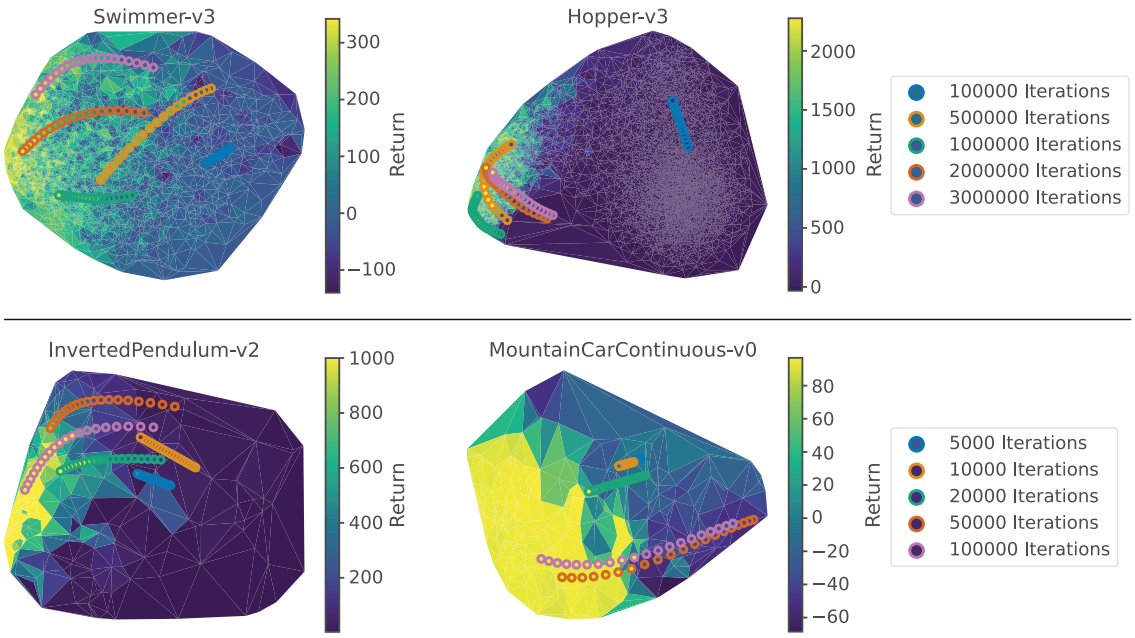

Figure 4: Policies generated by the generator during different stages of training. The background shows all policies executed during training (i.e., in the replay buffer), colored according to their returns. The 2D coordinates of the policies are determined by the PCA of their probing actions (obtained by the final critic $V_{\mathbf{w}}$). The chains of points show the policies created by the generator when given return commands ranging from the minimum (darker end of the chain) to the maximum (last point at the brighter end) possible return in the environment. Each chain represents a different stage of training, from almost untrained to fully trained. After training, the generator is able to produce policies across the whole performance spectrum.

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 5a and 5b), 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—see Figures 5c and 5d. 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 B.2.

Related Work

Policy Conditioned Value Functions Compared to standard value functions conditioned on a specific policy, policy-conditioned value functions generate values across several policies (Faccio, Kirsch, and Schmidhuber 2021; Harb et al. 2020; Faccio et al. 2022). This has been used to directly maximize the value using gradient ascent in the policy parameters. Here we use it to evaluate any policy generated by our policy generator. In contrast to previous work, this allows for generating policies of 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 "achieve a desired return" (Schmidhuber 2019; Srivastava et al. 2019). UDRL methods are related to hindsight RL where the commands correspond to desired goal states in the environment (Schmidhuber 1991; Kaelbling 1993; Andrychowicz et al. 2017; Rauber et al. 2018). In UDRL, just as in our method GoGePo, the required dataset of states, actions, and rewards is collected online during iterative improvements of the policy (Srivastava et al. 2019).

The conceptually highly related Decision Transformer (DT) (Janner, Li, and Levine 2021)) is designed for offline RL and thus requires a dataset of experiences from policies trained using other methods. A recent DT variant called "Online DT" (Chen et al. 2021) alternates between an offline pretraining phase, using data already collected, and an online finetuning phase, where hindsight learning is used. Online DTs and UDRL suffer from the same multi-modality issues when fitting data using unimodal distributions and a maximum likelihood approach. In addition to the multi-modality issue, UDRL and Goal-Conditioned Supervised Learning (GCSL) can diverge when the environments are stochastic and the task episodic (Štrupl et al. 2022).

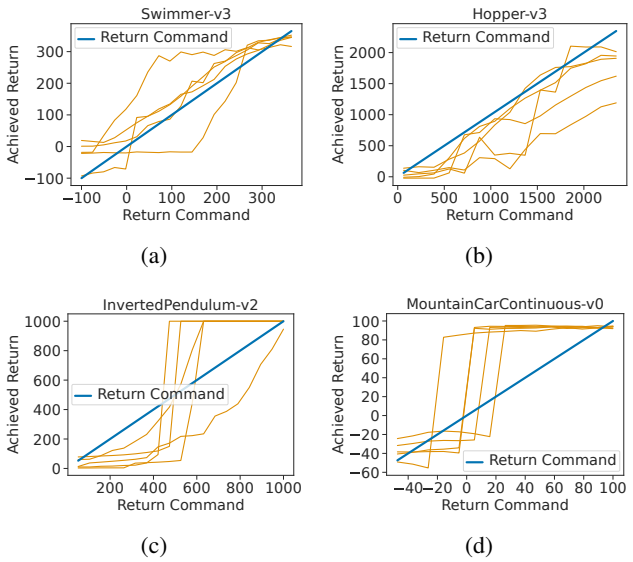

Figure 5: Achieved returns (mean of 10 episodes) of policies created by fully trained generators as a function of the given return command. A perfect generator would produce policies that lie on the diagonal identity line (if the environment permits such returns). For each environment, results of five independent runs are shown.

Our method learns online and does not rely on offline pretraining. It solves the multimodality issue of UDRL and outperforms it in terms of sample efficiency. We have not analyzed the convergence properties of our method. However, since we do not use hindsight learning, the counterexamples of Štrupl et al. (2022) do not apply. Instead of optimizing the policy to achieve a desired reward in action space, our method GoGePo evaluates the generated policies in command space. This is done by generating, conditioning on a command, a policy that is then evaluated using a parameter-based value function and trained to match the command to the evaluated return. This side-steps the issue with multi-modality in certain types of UDRL for episodic environments, where a command may be achieved through many different behaviors, and fitting the policy to varying actions may lead to sub-optimal policies.

Alternatively, the multimodality problem could be solved using a return-conditioned policy that directly outputs a multimodal action distribution, or is conditioned on random latent variables. To the best of our knowledge, this has not been tried for return-conditioned RL.

Fast Weight Programmers and HyperNetworks The idea of using a neural network (NN) to generate weight changes 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, Dai, and Le 2016). While in traditional NNs the weight matrix remains fixed after training, FWPs make these weights context-dependent. More generally, FWPs can be used as neural functions that involve 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, Irie, and Schmidhuber 2021; Schmidhuber 1992). FWPs are used in the context of memory-based meta learning (Schmidhuber 1993; Miconi, Stanley, and Clune 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 conditionally generate policies given a command.

Conclusion and Future Work

Our GoGePo 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, GoGePo is competitive with ARS and DDPG and outperforms UDRL on continuous control tasks. It also circumvents the multi-modality issue found in many approaches of the UDRL family. Further, our approach can be used to generate policies with any desired return.

There are two current limitations of our approach that we want to highlight: First, NNs created by an untrained generator might have weights that are far from typical initialization schemes. Exploration starting with such policies might be hard. We include further investigation of this in Appendix B.4. Second, our method is based on the episodic return signal. Extending it by considering also the state of the agent might help to increase sample efficiency. Future work will also 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.

Appendix

For the Appendix, see https://arxiv.org/abs/2207.01570.

Acknowledgements

We thank Mirek Strupl, Dylan Ashley, Róbert Csordás, Aleksandar Stanić and Anand Gopalakrishnan for their feedback. This work was supported by the ERC Advanced Grant (no: 742870), the Swiss National Science Foundation grant (200021_192356), 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

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. *Advances in neural information processing systems*, 34.

Faccio, F.; Kirsch, L.; and Schmidhuber, J. 2021. Parameter-Based Value Functions. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Faccio, F.; Ramesh, A.; Herrmann, V.; Harb, J.; and Schmidhuber, J. 2022. General policy evaluation and improvement by learning to identify few but crucial states. *arXiv preprint arXiv:2207.01566*.

Fujimoto, S.; Hoof, H.; and Meger, D. 2018. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, 1587–1596. PMLR.

Ghosh, D.; Gupta, A.; Reddy, A.; Fu, J.; Devin, C.; Eysenbach, B.; and Levine, S. 2019. Learning to Reach Goals via Iterated Supervised Learning. arXiv:1912.06088.

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, 491–498. New York, NY, USA: ACM. ISBN 1-59593-010-8.

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*.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, 1861–1870. PMLR.

Harb, J.; Schaul, T.; Precup, D.; and Bacon, P.-L. 2020. Policy Evaluation Networks. *arXiv preprint arXiv:2002.11833*.

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*, 1800–1809.

Miconi, T.; Stanley, K.; and Clune, J. 2018. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, 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.

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*.

Schaul, T.; Horgan, D.; Gregor, K.; and Silver, D. 2015. Universal Value Function Approximators. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, 1312–1320. JMLR.org.

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

Schmidhuber, J. 1990. An On-Line Algorithm for Dynamic Reinforcement Learning and Planning in Reactive Environments. In *Proc. IEEE/INNS International Joint Conference on Neural Networks, San Diego*, volume 2, 253–258.

Schmidhuber, J. 1991. Learning to generate sub-goals for action sequences. In *Artificial neural networks*, 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*, 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*.

Schmidhuber, J.; and Huber, R. 1991. Learning to Generate Artificial Fovea Trajectories for Target Detection. *International Journal of Neural Systems*, 2(1 & 2): 135–141. (Based on TR FKI-128-90, TUM, 1990).

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., eds., *Artificial Neural Networks - ICANN 2008*, 387–396. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-87536-9.

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, 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.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. USA: A Bradford Book. ISBN 0262039249, 9780262039246.

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, 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*, 5026–5033.

van Hasselt, H. 2012. Reinforcement Learning in Continuous State and Action Spaces. In Wiering, M.; and van Otterlo, M., eds., *Reinforcement Learning*, 207–251. Springer.

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*, 5–32. Springer.