Upside Down Reinforcement Learning with Policy Generators

Jacopo Di Ventura [*1] **Dylan R. Ashley** [1,2,3,4] **Vincent Herrmann** [1,2,3]

Francesco Faccio [1,2,3,4] **Jürgen Schmidhuber** [1,2,3,4,5]

[1] Università della Svizzera italiana, Lugano, Switzerland
[2] The Swiss AI Lab IDSIA (USI-SUPSI), Lugano, Switzerland
[3] Scuola universitaria professionale della Svizzera italiana, Lugano, Switzerland
[4] Center of Excellence for Generative AI, KAUST, Thuwal, Saudi Arabia
[5] NNAISENSE, Lugano, Switzerland

Abstract

Upside Down Reinforcement Learning (UDRL) is a promising framework for solving reinforcement learning problems which focuses on learning command-conditioned policies. In this work, we extend UDRL to the task of learning a command-conditioned generator of deep neural network policies. We accomplish this using Hypernetworks—a variant of Fast Weight Programmers, which learn to decode input commands representing a desired expected return into command-specific weight matrices. Our method, dubbed Upside Down Reinforcement Learning with Policy Generators (UDRLPG), streamlines comparable techniques by removing the need for an evaluator or critic to update the weights of the generator. To counteract the increased variance in last returns caused by not having an evaluator, we decouple the sampling probability of the buffer from the absolute number of policies in it, which, together with a simple weighting strategy, improves the empirical convergence of the algorithm. Compared with existing algorithms, UDRLPG achieves competitive performance and high returns, sometimes outperforming more complex architectures. Our experiments show that a trained generator can generalize to create policies that achieve unseen returns zero-shot. The proposed method appears to be effective in mitigating some of the challenges associated with learning highly multimodal functions. Altogether, we believe that UDRLPG represents a promising step forward in achieving greater empirical sample efficiency in RL.

1 Introduction

Reinforcement Learning (RL) is a powerful framework for solving sequential decision-making problems. In RL, the standard approach to policy optimization typically involves training a policy network to maximize expected returns. Upside Down RL (Schmidhuber, 2019), or UDRL, bridges the gap between RL and supervised learning by transforming much of the RL problem into a supervised learning task. With UDRL, the algorithm is no longer directly learning to maximize the expected return, but is instead learning a mapping between commands (e.g., desired returns) and actions. The data for training the model can be collected either offline—as in Decision Transformers (Chen et al., 2021)—or online as the model learns to generalize to higher and higher returns. One notable extension

[*]Now at Leiden University. Correspondence to j.di.ventura@liacs.leidenuniv.nl

18th European Workshop on Reinforcement Learning (EWRL 2025).

of UDRL is GoGePo (Faccio et al., 2023), which extends UDRL from working in action space to working in parameter space. However, this approach relies on a generator-evaluator (actor-critic) architecture. Here, the generator learns to produce policies that follow a given command, and the evaluator network assesses the quality of the generated policies. This introduces a considerable amount of additional complexity, as the evaluator must be jointly optimized with the generator.

We propose UDRL with Policy Generators (UDRLPG): a simpler alternative to GoGePo that sidesteps the need for an evaluator. UDRLPG learns a single policy generator, capable of producing policies that can achieve any desired return, without relying on a generator-evaluator pair. Through hindsight learning, the model minimizes the error between policies it previously generated and new policies it produces, without a critic, thereby reducing the architectural complexity.

2 Background

Formally, the RL problem is often modeled as a Markov Decision Process (Puterman, 2014; Stratonovich, 1960), or MDP, which is a tuple $(S, A, P, R, \gamma, \mu_0)$, where at each timestep t, the agent observes a state $s_t \in S$, chooses an action $a_t \in A$, and receives a reward $r_t = R(s_t, a_t)$. The action leads to a new state according to the transition probability $P(s_{t+1}|s_t, a_t)$. Each episode begins in an initial state s_0 selected with probability μ_0. The policy $\pi_\theta : S \to \Delta(A)$ controls the agent, where $\theta \in \Theta$ are the policy parameters. The objective is to find π_θ that maximizes the expected return: $\pi_\theta = \arg\max_{\pi_\theta} J(\theta)$, where

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

Here, $p(\tau|\theta)$ is the distribution over trajectories induced by policy π with parameters θ.

Goal- and command-conditioned RL (Andrychowicz et al., 2017; Schaul et al., 2015; Schmidhuber, 1991; Schmidhuber & Wahnsiedler, 1993) agents differ from classic RL agents as they learn to maximize a goal- or command-conditioned expected return. UDRL and related approaches employ supervised learning to train command-conditioned RL agents by receiving command inputs that specify the desired outcome within a certain timeframe. However, in the episodic setting, it is often the case that there is no single sequence of actions or behaviors that satisfies a given command. For this reason, a unimodal maximum likelihood approach may not be able to capture the variability in the data.

UDRL takes the problem of learning to act within an environment closer to a supervised learning task, as the goal of the model now becomes learning a mapping from state and command to actions rather than learning a mapping from state-action pair to value (expected return) or maximizing return directly using policy search. The primary benefit of this formulation lies in its ability to convert a portion of the RL problem into a supervised task. This allows us to handle part of the complexity of reinforcement learning problems within the supervised learning domain, which is the main area where artificial neural networks are most successfully applied. In UDRL, the agent is trained using hindsight. It learns to predict which action it took, given the current state and command (g, h), where g is the actual return observed. UDRL also requires the definition of a command selection strategy. Typically, for online RL reward-maximization problems, the commands being issued should increase over time.

Building on the paradigm introduced by Fast Weight Programmers (FWPs), where one neural network learns to generate weight updates for another network, parameter-based methods in RL (Mania et al., 2018; Salimans et al., 2017b; Sehnke et al., 2008, 2010) sample policy parameters θ from a hyperpolicy distribution $\nu_\rho(\theta)$ (Faccio et al., 2023), transforming the RL problem from finding the parameters θ of a policy π such that the expected return obtained in the environment is maximized, to finding the hyperpolicy parameters ρ that maximizes the expected return

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

The objective can be made context-dependent as $J(\rho, c)$ by conditioning the hyperpolicy on context c,

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

3 Related Work

The development of UDRLPG builds upon several foundational concepts in RL and neural networks. FWPs (Schmidhuber, 1992b, 1993), introduced the concept of using one neural network to output weight updates for another target network—including implementations with deep and recurrent neural architectures (Schmidhuber, 1992a)—enabling dynamic context-dependent weights after training. This concept was later popularized under the name Hypernetworks. FWPs have found applications across various domains, including memory-based meta learning (Miconi et al., 2018; Schmidhuber, 1993) and RL (Gomez & Schmidhuber, 2005).

Parameter-based value functions (Faccio et al., 2021; Harb et al., 2020) abstract traditional value functions by learning to estimate expected returns conditioned on policy parameters over states or state-action pairs. This enables evaluation and adjustment in parameter space, providing a mapping between policy parameters and expected returns. The concept of inverting this mapping—determining parameters given an expected return—forms a core principle of UDRLPG. UDRLPG removes the need for the evaluator function $V_w : \Theta \to \mathbb{R}$ that GoGePo uses to to optimize a policy generator. The generator in GoGePo learns to minimize the difference between return commands and estimated returns of generated policies through parameter-based evaluation.

Evolution Strategies demonstrate the effectiveness of directly exploring policy parameter space, showing that parameter space optimization can overcome limitations of action-space methods like sparse rewards. Policy Gradients with Parameter-Based Exploration addresses high variance in policy gradient methods by replacing policies with probability distributions over parameters, enabling trajectory sampling from single parameter samples (Salimans et al., 2017a).

4 Method

UDRLPG, as shown in Figure 1, is a parameter-based method that directly optimizes over policy space to generate policies achieving desired returns. At its core, UDRLPG employs a FWP, in the form of a hypernetwork $G_\rho : \mathbb{R} \to \Theta$ functioning as a decoder, where $c \in \mathbb{R}$ represents the command (desired return) and ρ are the FWP parameters. For exploration purposes, we consider a non-deterministic FWP $g_\rho(\theta, c) = G_\rho(c) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. Here, σ is a hyperparameter that controls the extent of the perturbation, therefore directly controlling the exploration-exploitation balance. Higher σ values enable broader exploration and help escape local optima, while lower values favor exploitation of known high-performing policies. The introduction of noise is vital for the learning process as it allows the algorithm to explore a wider range of potential policies, lowering the chance of getting stuck in a local optimum.

The hypernetwork G_ρ is trained to minimize the error

$$\mathcal{L}_G(\rho) = \mathbb{E}_{c \in D}[(G_\rho(c) - \theta_c)^2], \tag{4}$$

where D represents the replay buffer and θ_c is a policy with expected return c. The replay buffer is initialized with random policies to ensure diverse starting conditions, enabling effective exploration during the early stages of training. As a UDRL method, UDRLPG performs the usual hindsight learning (Andrychowicz et al., 2017), which uses past experiences as examples of successfully following specific commands. Generated policies are stored in the replay buffer together with their observed return replacing the command (desired return), rather than the command (desired return).

The training process consists of an update and a rollout stage. The first stage is the update, where the hypernetwork undergoes typical iterative gradient updates using policies sampled from the replay buffer. During the rollout phase, new policies are generated using the updated weights, and, after noise is added for exploration, the policies are evaluated using observation normalization and added to the replay buffer. To address overrepresentation of particular return ranges during training, the buffer is organized into performance-based buckets containing policies within specific return ranges. This organization decouples sampling probability from the absolute number of policies in each performance category, allowing one to follow the desired weighting strategy independently of the returns of policies in the buffer.

Algorithm 1 UDRLPG

Initialize: $D \leftarrow$ empty replay buffer with performance-based buckets
Collect n random policies and add their (r, θ) pairs to D
$G_\rho \leftarrow$ differentiable policy generator (hypernetwork)
while not converged **do**
    for *update_repeats* **do**
        Sample minibatch $B = \{(c, \theta_c)\}$ from D
        Update generator by stochastic gradient descent: $\nabla_\rho \mathbb{E}_{\{(c,\theta_c)\}\in B}[(G_\rho(c) - \theta_c)^2]$
    end for
    for *rollout_repeats* **do**
        Select a desired return command c
        Sample policy parameters $\theta \sim G_\rho(\theta \mid c)$
        Sample noise $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$
        $\theta \leftarrow \theta + \epsilon$ ▷ Exploration
        Simulate policy π_θ for one episode to obtain return r
        Add $(r,\ \theta)$ to D ▷ Store observed return r as hindsight command c
    end for
end while

Figure 1: Pseudocode for Upside-Down Reinforcement Learning with Policy Generators. A full implementation of UDRLPG is publicly available at `https://github.com/JacopoD/udrlpg_`

5 Results and Discussion

We compare UDRLPG to two baseline algorithms: GoGePo and DDPG in the `InvertedPendulum-v4`, `Swimmer-v4`, and `Hopper-v4` environments from the OpenAI gym suite (Brockman et al., 2016). For each environment, we report the mean return and the variance of the last few returns. We also analyze the model's ability to produce policies across the return spectrum.

Results in Figure 2, show competitive performance against both baselines. In InvertedPendulum, UDRLPG converges to the same value but slower than both baselines. While it achieves the maximum possible reward of 1000, it exhibits higher variance in final returns compared to GoGePo, indicating less stability across runs. For Swimmer, UDRLPG reaches a mean final return of 300, underperforming against GoGePo which reaches 320. The method shows a steady improvement throughout training with no signs of plateauing, with higher variance in final performance compared to GoGePo. In Hopper, the hardest environment tested, UDRLPG matches the performance of GoGePo with a mean final return of 2070. UDRLPG appears to explore the parameter space more extensively than the baselines, resulting in higher return bounds. As shown in Figure 5, UDRLPG can produce policies across the return spectrum, resulting in strong identity curves, suggesting robust generalization over commands. Performance-based buckets and a fine-tuned weighting strategy for the replay buffer were crucial for stable training. This approach reduces learning stagnation and ensures a balanced representation of high and low return policies during training, leading to more consistent convergence toward higher return policies. Our ablation experiments, shown in Figure 4, provide empirical evidence for this claim.

UDRLPG inherits from UDRL potential challenges deriving from multimodality. The newly generated policy θ_{new}, obtained by conditioning the generator on a command c may be significantly different from the policies θ in the buffer, where $R(\theta) = c$. If $|D(c)| \geq 2$, the generator will try to fit θ_{mean}: the mean of the policies. However, there is no guarantee that $R(\theta_{mean}) = R(\theta)$. In this case, the learning process may start degrading. This issue does not arise frequently while training UDRLPG. One possible explanation is that, early on during training, the hypernetwork develops an understanding of the underlying structure or distribution of the policy weights based on the first policies in the buffer. Weight initialization biases the hypernetwork towards a specific arrangement of the neural network's weights (Chen et al., 1993). This arrangement includes the order of the neurons within hidden layers (neuron permutation) and the magnitudes of the weights (weight scaling). The

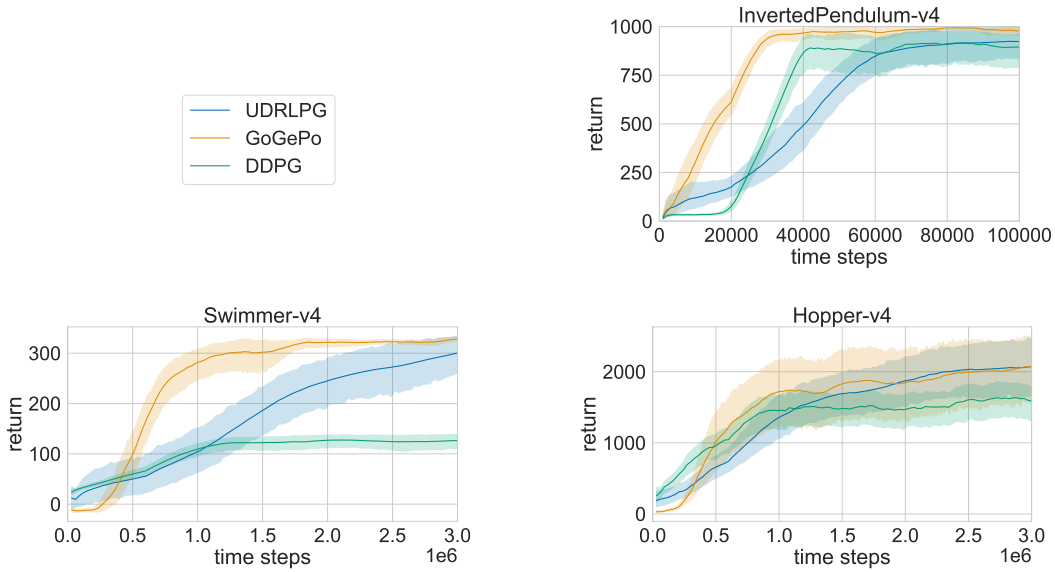

Figure 2: Performance of policies from UDRLPG, GoGePo, and DDPG during training in all environments. Lines show mean return and 95% bootstrapped confidence intervals from 20 independent runs.

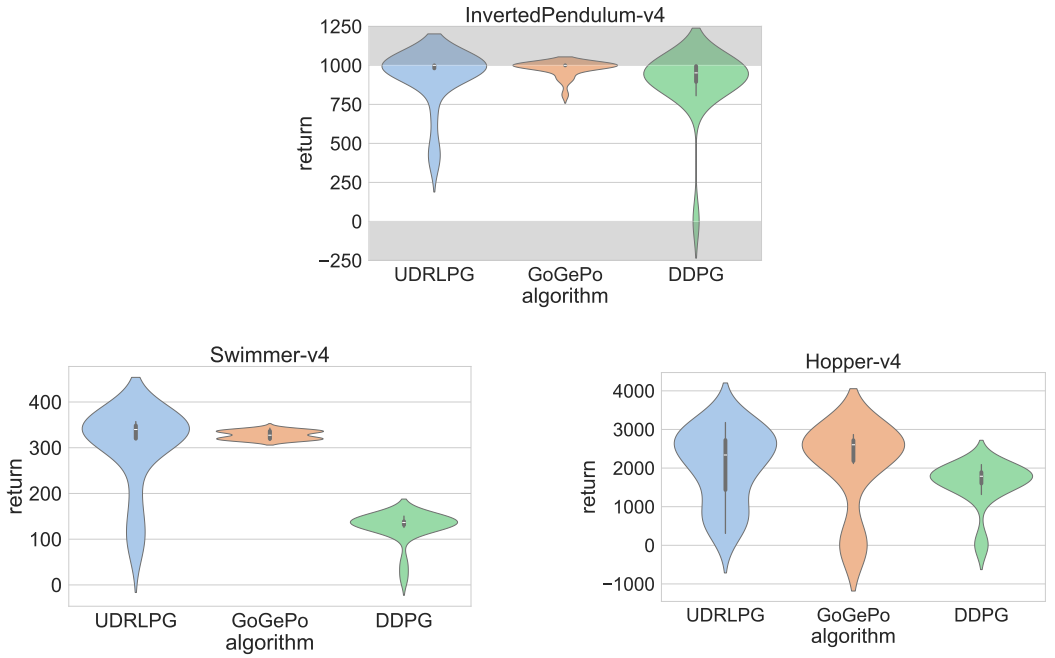

Figure 3: Variance of final returns. Lines show mean and 95% confidence intervals from 20 evaluation runs.

bias induced by the initialization constrains the search space in a part of the solution space where all weights follow the same configuration. The hypernetwork learns changes around the configuration, reducing the likelihood of generating policies with similar returns that are very different in weight space.

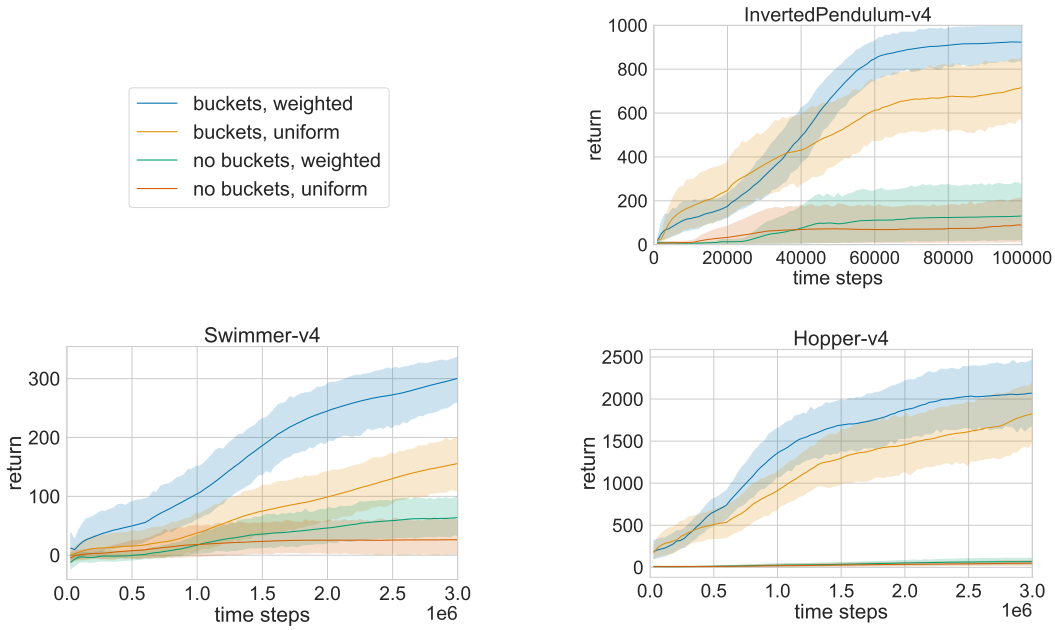

Figure 4: Performance comparison of UDRLPG policies across test all environments using four buffer strategies. The proposed strategy is buckets and weighted sampling, in blue. Curves show mean returns with 95% bootstrapped confidence intervals from 20 runs.

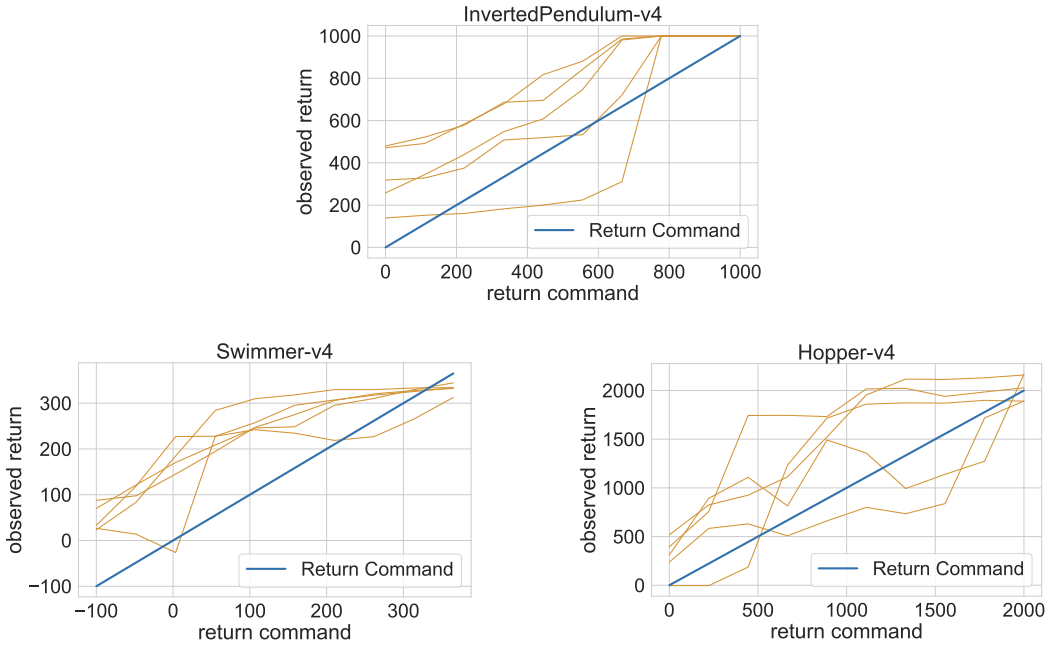

Figure 5: Mean returns over 10 episodes of policies from the generator as a function of the given command. Results are averaged over 5 independent runs.

6 Conclusion

This work introduced UDRLPG, an approach to RL focused on generating policy parameters conditioned on return commands. Compared with existing methods, UDRLPG removes the need for a separate evaluator in the architecture, thus simplifying the overall structure. Empirical results show that UDRLPG generalizes effectively across commands, is competitive with existing methods, and is able to explore the parameter space more extensively than some competing methods, resulting in higher return bounds. Additionally, we note that the hypernetwork's initialization bias confines the search to a specific region of the solution space where weights share a common configuration, effectively circumventing the challenge of multimodality. We identify some limitations of UDRLPG here. In some environments, convergence is slower and the variance in final returns across runs is higher than that of GoGePo. Compared to GoGePo, UDRLPG simplifies the learning process and provides further insight into goal-conditioned policy generation.

Acknowledgements

This work was supported by the European Research Council (ERC, Advanced Grant Number 742870) and the Center of Excellence for Generative AI at the King Abdullah University of Science and Technology (KAUST, Award Number 5940). We also want to thank both the NVIDIA Corporation for donating a DGX-1 as part of the Pioneers of AI Research Award and IBM for donating a Minsky machine.

References

Andrychowicz, M., Crow, D., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., & Zaremba, W. (2017). Hindsight experience replay. *Advances in Neural Information Processing Systems*, *30*, 5048–5058. https://proceedings.neurips.cc/paper_files/paper/2017/file/453fadbd8a1a3af50a9df4df899537b5-Paper.pdf

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *OpenAI gym*. arXiv. https://doi.org/10.48550/arXiv.1606.01540

Chen, A. M., Lu, H.-m., & Hecht-Nielsen, R. (1993). On the geometry of feedforward neural network error surfaces. *Neural computation*, *5*(6), 910–927.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., & Mordatch, I. (2021). Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, *34*, 15084–15097.

Faccio, F., Herrmann, V., Ramesh, A., Kirsch, L., & Schmidhuber, J. (2023). Goal-conditioned generators of deep policies. *Proceedings of the AAAI Conference on Artificial Intelligence*, *37*(6), 7503–7511.

Faccio, F., Kirsch, L., & Schmidhuber, J. (2021). Parameter-based value functions. *International Conference on Learning Representations*. https://openreview.net/forum?id=tV6oBfuyLTQ

Gomez, F. J., & Schmidhuber, J. (2005). Co-evolving recurrent neurons learn deep memory pomdps. *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, 491–498.

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

Mania, H., Guy, A., & Recht, B. (2018). Simple random search of static linear policies is competitive for reinforcement learning. *Advances in neural information processing systems*, *31*.

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

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

Salimans, T., Ho, J., Chen, X., Sidor, S., & Sutskever, I. (2017a). Evolution strategies as a scalable alternative to reinforcement learning. https://arxiv.org/abs/1703.03864

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

Schaul, T., Horgan, D., Gregor, K., & Silver, D. (2015). Universal value function approximators. *Proceedings of the 32nd International Conference on Machine Learning*, *37*, 1312–1320. https://proceedings.mlr.press/v37/schaul15.pdf

Schmidhuber, J. (1992a). Steps towards\self-referential. *learning. Technical Report CU-CS-627-92, Dept. of Comp. Sci., University of Colorado at Boulder*.

Schmidhuber, J. (2019). Reinforcement learning upside down: Don't predict rewards–just map them to actions. *arXiv preprint arXiv:1912.02875*.

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

Schmidhuber, J. (1992b). 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. *ICANN'93: Proceedings of the International Conference on Artificial Neural Networks Amsterdam, The Netherlands 13–16 September 1993 3*, 446–450.

Schmidhuber, J., & Wahnsiedler, R. (1993). Planning simple trajectories using neural subgoal generators.

Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., & Schmidhuber, J. (2008). Policy gradients with parameter-based exploration for control. In V. Kůrková, R. Neruda, & J. Koutník (Eds.), *Artificial neural networks - icann 2008* (pp. 387–396). Springer Berlin Heidelberg.

Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., & Schmidhuber, J. (2010). Parameter-exploring policy gradients. *Neural Networks*, *23*(4), 551–559. https://doi.org/10.1016/j.neunet.2009.12.004

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