# Diffusion Policy Policy Optimization

**Anonymous Author(s)**
Affiliation
Address
`email`

**Abstract:** We introduce *Diffusion Policy Policy Optimization*, **DPPO**, an algorithmic framework including best practices for fine-tuning diffusion-based policies (e.g. Diffusion Policy [1]) in continuous control and robot learning tasks using the policy gradient (PG) method from reinforcement learning (RL). PG methods are ubiquitous in training RL policies with other policy parameterizations; nevertheless, they had been conjectured to be less efficient for diffusion-based policies. Surprisingly, we show that **DPPO** achieves the strongest overall performance and efficiency for fine-tuning in common benchmarks compared to other RL methods for diffusion-based policies and also compared to PG fine-tuning of other policy parameterizations. We further demonstrate the strengths of **DPPO** in a range of realistic settings, including simulated robotic tasks with pixel observations, and via zero-shot deployment of simulation-trained policies on robot hardware. Website with videos: **diffusionppoanon.github.io**.

**Keywords:** Reinforcement learning, diffusion policy

## 1 Introduction

Behavior cloning with expert data [2] is rapidly emerging as dominant paradigm for pre-training *robot policies* [3, 4, 5, 6, 7], but their performance can be suboptimal [8] due to expert data being suboptimal or expert data exhibiting limited coverage of possible environment conditions. As robot policies entail interaction with their environment, reinforcement learning (RL) [9] is a natural candidate for further optimizing their performance beyond the limits of demonstration data. However, RL fine-tuning can be nuanced for pre-trained policies parameterized as diffusion models [10], which have emerged as a leading parameterization for action policies [1, 11, 12].

**Contribution 1** *(**DPPO**)*. We introduce *Diffusion Policy Policy Optimization* (**DPPO**), a generic framework as well as a set of carefully chosen design decisions for fine-tuning a diffusion-based robot learning policy via popular policy gradient methods [13, 14] in reinforcement learning.

The literature has already studied improving/fine-tuning diffusion-based policies using RL [15, 16, 17]. Yet policg gradient (PG) methods have been believed to be inefficient in training Diffusion Policy for continuous control tasks [15, 18]. On the contrary, we show that for a Diffusion Policy pre-trained from expert demonstrations, our methodology for *fine-tuning* via PG updates yields robust, high-performing policies with favorable training behavior.

**Contribution 2** *(Demonstration of **DPPO**'s Performance).* We show that for *fine-tuning* a pre-trained Diffusion Policy, **DPPO** yields consistent and marked improvements in training stability and often final policy performance in comparison to those based on off-policy Q-learning [16, 17, 18, 15] and weighted regression [19, 20, 21], other demo-augmented RL methods [22, 23, 24], as well as common policy parameterizations such as Gaussian and Gaussian Mixture models.

Through ablations, we further show that our design decisions overcome the speculated limitation of PG methods for fine-tuning Diffusion Policy. Finally, to justify the broad utility of **DPPO**, we verify its efficacy across both simulated and real environments, and in situations when either ground-truth states or pixels are given to the policy as input.

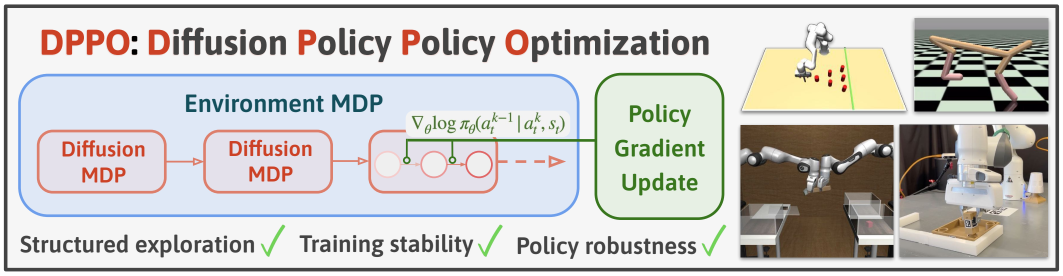


Figure 1: We introduce **DPPO**, *Diffusion Policy Policy Optimization*, that fine-tunes pre-trained Diffusion Policy using policy gradient. **DPPO** affords structured exploration and training stability during policy fine-tuning, and the fine-tuned policy exhibits strong robustness and generalization.

## 2 Preliminaries

**Markov Decision Process.** We consider a *Markov Decision Process* (MDP) $\mathcal{M}_{\text{ENV}} := (\mathcal{S}, \mathcal{A}, P_0, P, R)$ with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution $P_0$, transition probabilities $P$, and reward $R$. At each timestep $t$, the agent (e.g., robot) observes the state $s_t \in \mathcal{S}$, takes an action $a_t \sim \pi(a_t \mid s_t) \in \mathcal{A}$, transitions to the next state $s_{t+1}$ according to $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$ while receiving the reward $R(s_t, a_t)$. We aim to train a policy to optimize the cumulative reward, discounted by a function $\gamma(\cdot)$, $\mathcal{J}(\pi_\theta) = \mathbb{E}^{\pi_\theta, P_0}[\sum_{t\geq 0} \gamma(t) R(s_t, a_t)]$.

**Diffusion models.** A denoising diffusion probabilistic model (DDPM) [25, 10, 26] represents a data distribution $p(\cdot) = p(x^0)$ as the reverse process of a forward noising process $q(x^k | x^{k-1})$ that iteratively adds Gaussian noise to the data. The reverse process is parameterized by $\varepsilon_\theta(x_k, k)$, predicting the added noise $\varepsilon$ that converts $x_0$ to $x_k$ [10]. Sampling starts with $x^K \sim \mathcal{N}(0, \mathrm{I})$ and iteratively generates the denoised sample: $x^{k-1} \sim p_\theta(x^{k-1} | x^k) := \mathcal{N}(x^{k-1}; \mu_k(x^k, \varepsilon_\theta(x^k, k)), \sigma_k^2 \mathrm{I})$. $\sigma_k^2$ is a variance term that abides by a fixed schedule from $k = 1, \ldots, K$.

**Diffusion models as policies.** *Diffusion Policy* (DP; see Chi et al. [1]) is a policy $\pi_\theta$ parameterized by a DDPM which takes in $s$ as a conditioning argument, and parameterizes $p_\theta(a^{k-1} \mid a^k, s)$. DPs can be trained via behavior cloning by fitting the conditional noise prediction $\varepsilon_\theta(a^k, s, k)$ to predict the added noise. Notice that unlike more standard policy parameterizations such as unimodal Gaussian policies, DPs do not maintain an explicit likelihood of $p_\theta(a^0 \mid s)$.

## 3 **DPPO**: Diffusion Policy Policy Optimization

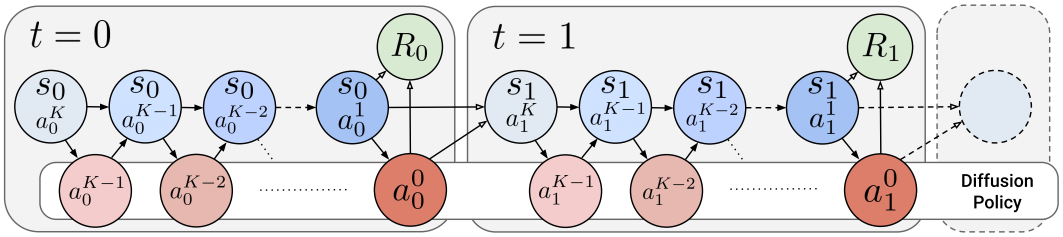


Figure 2: We treat the denoising process as an MDP, and the whole environment episode can be considered as a chain of such MDPs. Now the entire chain ("Diffusion Policy MDP", $\mathcal{M}_{\text{DP}}$) involves a Gaussian likelihood at each (denoising) step and thus can be optimized with policy gradient.

As observed in [27] and [15], a denoising process can be represented as a multi-step MDP in which policy likelihood of each denoising step can be obtained directly. We extend this formalism by embedding the Diffusion MDP into the environmental MDP, obtaining a larger "Diffusion Policy MDP" denoted $\mathcal{M}_{\text{DP}}$, visualized in Fig. 2. The Diffusion MDP $\mathcal{M}_{\text{DP}}$ uses indices $\bar{t}(t,k) = tK + (K - k - 1)$ corresponding to $(t, k)$, which increases in $t$ but (to keep the indexing conventions of diffusion) *decreases* lexicographically with $K - 1 \geq k \geq 0$. The states, actions and rewards are

$$\bar{s}_{\bar{t}(t,k)} = (s_t, a_t^{k+1}), \quad \bar{a}_{\bar{t}(t,k)} = a_t^k, \quad \bar{R}_{\bar{t}(t,k)}(\bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}) = \begin{cases} 0 & k > 0 \\ R(s_t, a_t^0) & k = 0 \end{cases},$$

where the bar-action at $\bar{t}(t,k)$ is the action $a_t^k$ after one denoising step. Reward is only given at times corresponding to when $a_t^0$ is taken. The initial state distribution is $\bar{P}^0 = P_0 \otimes \mathcal{N}(0, \mathbf{I})$, corresponding to $s_0 \sim P_0$ is the initial distribution from the environmental MDP and $a_0^K \sim \mathcal{N}(0, \mathbf{I})$ independently. Finally, the transitions are

$$\bar{P}(\bar{s}_{\bar{t}+1} \mid \bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}) = \begin{cases} (s_t, a_t^k) \ \sim \delta_{(s_t, a_t^k)} & \bar{t} = \bar{t}(t,k), k > 0 \\ (s_{t+1}, a_{t+1}^K) \sim P(s_{t+1} \mid s_t, a_t^0) \otimes \mathcal{N}(0, \mathbf{I}) & \bar{t} = \bar{t}(t,k), k = 0 \end{cases}.$$

That is, the transition moves the denoised action $a_t^k$ at step $\bar{t}(t,k)$ *into the next state* when $k > 0$, or otherwise progresses the environment MDP dynamics with $k = 0$. The policy in $\mathcal{M}_{\mathrm{DP}}$ is

$$\bar{\pi}_\theta(\bar{a}_{\bar{t}(t,k)} \mid \bar{s}_{\bar{t}(t,k)}) = \pi_\theta(a_t^k \mid a_t^{k+1}, s_t) = \mathcal{N}(a_t^k; \mu(a_t^{k+1}, \varepsilon_\theta(a_t^{k+1}, k+1, s_t)), \sigma_{k+1}^2 \mathrm{I}). \quad (3.1)$$

Fortunately, (3.1) is a *Gaussian likelihood*, which can be evaluated analytically and is amenable to the policy gradient updates (see also [15] for an alternative derivation):

$$\nabla_\theta \bar{\mathcal{J}}(\bar{\pi}_\theta) = \mathbb{E}^{\bar{\pi}_\theta, \bar{P}, \bar{P}^0}[\sum_{\bar{t} \geq 0} \nabla_\theta \log \bar{\pi}_\theta(\bar{a}_{\bar{t}} \mid \bar{s}_{\bar{t}}) \bar{r}(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}})], \quad \bar{r}(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}) := \sum_{\tau \geq \bar{t}} \gamma(\tau) \bar{R}(\bar{s}_\tau, \bar{a}_\tau). \quad (3.2)$$

Evaluating the above involves sampling through the denoising process, which is the usual "forward pass" that samples actions in Diffusion Policy.

### 3.1 Instantiating **DPPO** with Proximal Policy Optimization

**Definition 3.1** (Generalized PPO, clipping variant)**.** Consider a general MDP. Given an advantage estimator $\hat{A}(s,a)$, the PPO update [14] is the sample approximation to

$$\nabla_\theta \, \mathbb{E}^{(s_t, a_t) \sim \pi_{\theta_{\text{old}}}} \min \left( \hat{A}^{\pi_{\theta_{\text{old}}}}(s_t, a_t) \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \ \hat{A}^{\pi_{\theta_{\text{old}}}}(s_t, a_t) \operatorname{clip}\left( \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, 1-\varepsilon, 1+\varepsilon \right) \right),$$

where $\varepsilon$, the clipping ratio, controls the maximum magnitude of the policy updated. We instantiate PPO in our diffusion MDP with $(s, a, t) \leftarrow (\bar{s}, \bar{a}, \bar{t})$. Our advantage estimator respects the two-level nature of the MDP: let $\gamma_{\text{ENV}} \in (0,1)$ be the environment discount and $\gamma_{\text{DENOISE}} \in (0,1)$ be the denoising discount. Consider the environment-discounted return:

$$\bar{r}(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}) := \sum_{t' \geq t} \gamma_{\text{ENV}}^t \bar{r}(\bar{s}_{\bar{t}(t',0)}, \bar{a}_{\bar{t}(t',0)}), \quad \bar{t} = \bar{t}(t,k),$$

since $\bar{R}(\bar{t}) = 0$ at $k > 0$. This fact also obviates the need of estimating the value at $k > 1$ and allows us to use the following denoising-discounted advantage estimator:

$$\hat{A}^{\pi_{\theta_{\text{old}}}}(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}) := \gamma_{\text{DENOISE}}^k \left( \bar{r}(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}) - \hat{V}^{\bar{\pi}_{\theta_{\text{old}}}}(\bar{s}_{\bar{t}(t,0)}) \right)$$

Lastly, we choose the value estimator to *only depend* on the "$s$" component of $\bar{s}$: $\hat{V}^{\bar{\pi}_{\theta_{\text{old}}}}(\bar{s}_{\bar{t}(t,0)}) := \tilde{V}^{\bar{\pi}_{\theta_{\text{old}}}}(s_t)$, which we find leads to more efficient and stable training compared to also estimating the value of applying the denoised action $a_t^{k=1}$ (part of $\bar{s}_{\bar{t}(t,0)}$) as shown in Appendix D.3.

## 4 Performance Evaluation of **DPPO**

We study the performance of **DPPO** in popular RL and robotics benchmarking environments including OpenAI GYM, ROBOMIMIC, and FURNITURE-BENCH. Due to the limited space, we defer descriptions of the benchmarks, baselines, and experimental details to Appendix G.

### 4.1 Comparison to diffusion-based RL algorithms

We compare **DPPO** to an extensive list of RL methods for fine-tuning diffusion models in Fig. 3. We evaluate on the three OpenAI GYM tasks and the four ROBOMIMIC tasks with **state** input. Overall, **DPPO** performs consistently, exhibits great training stability, and enjoys strong fine-tuning performance across tasks. In the GYM tasks (top row), **IDQL** and **DAWR** exhibit competitive performance, while the other methods perform worse and train less stably. **DPPO** is the strongest performer in the ROBOMIMIC tasks (bottom row), especially in the challenging `Transport` tasks. Surprisingly, **DRWR** is a strong baseline in {`Lift`, `Can`, `Square`} but underperforms in `Transport`, while all other baselines fare worse still. We postulate that the other baselines, using off-policy updates, suffers from training instability in sparse-reward ROBOMIMIC tasks given continuous action space plus large action chunk sizes (see furtuer studies in Appendix D.3).

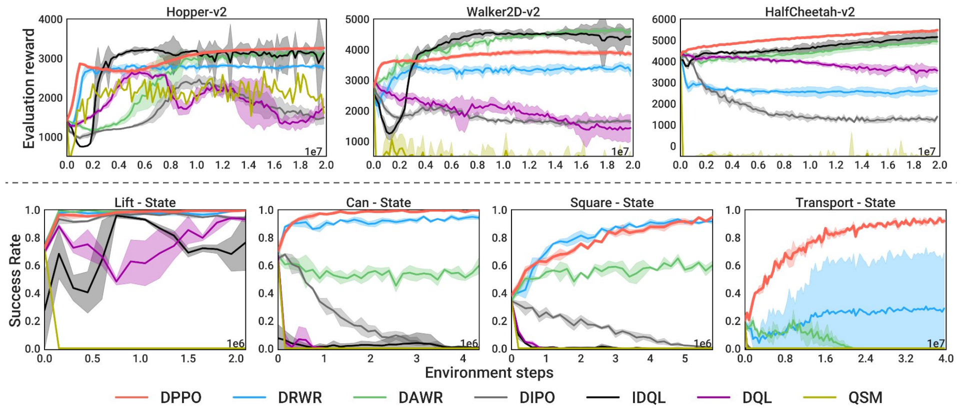


Figure 3: **Comparing to other diffusion-based RL algorithms.** Top row: GYM tasks [28] averaged over five seeds. Bottow row: ROBOMIMIC tasks [29], averaged over three seeds.

## 4.2 Comparison to other policy parameterizations

We compare **DPPO** with popular RL policy parameterizations: unimodal Gaussian with diagonal covariance [13] and Gaussian Mixture Model (GMM), using either MLPs or Transformers [30], and also fine-tuned with the PPO objective. We compare these to **DPPO**-MLP and **DPPO**-UNet, which use either MLP or UNet as the network backbone. We evaluate on the four tasks from ROBOMIMIC (`Lift`, `Can`, `Square`, `Transport`) with both **state** and **pixel** input.

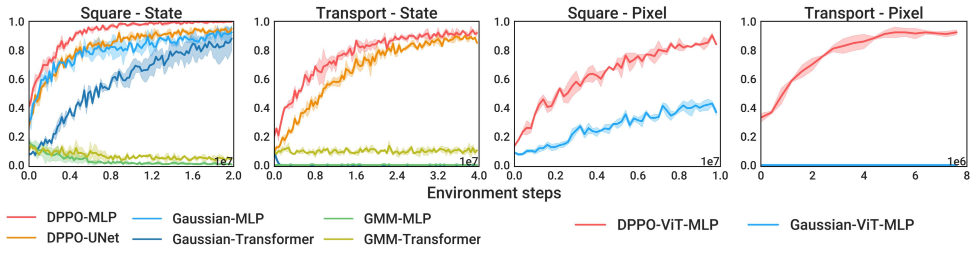


Figure 4: **Comparing to other policy parameterizations** in the more challenging `Square` and `Transport` tasks from ROBOMIMIC. Results are averaged over three seeds.

Fig. 4 display results for the more challenging `Square` and `Transport` — we defer the results in `Lift` and `Can` to Fig. 15. With **state** input, **DPPO** outperforms Gaussian and GMM policies, with faster convergence to $\sim$100% success rate in `Lift` and `Can`, and greater final performance on `Square` and the challenging `Transport`, where it reaches $> 90\%$. With **pixel** inputs, we use a Vision-Transformer-based (ViT) image encoder introduced in Hu et al. [24] and an MLP head and compare the resulting variants **DPPO**-ViT-MLP and Gaussian-ViT-MLP (we omit GMM due to poor performance in state-based training). While the two are comparable on `Lift` and `Can`, **DPPO** trains more quickly and to higher accuracy on `Square`, and *drastically outperforms* on `Transport`, whereas Gaussian does not improve from its 0% pre-trained success rate.

## 4.3 Evaluation on Furniture-Bench, and sim-to-real transfer

Here we evaluate **DPPO** on the long-horizon manipulation tasks from FURNITURE-BENCH [31]. We compare **DPPO** to Gaussian-MLP, the overall most effective baseline from Section 4.2. Overall, **DPPO** exhibits strong training stability and improves policy performance in all six settings. **DPPO** also transfers well to physical hardware zero-shot. Please see Appendix D.1 for detailed results.

## References

[1] C. Chi, Z. Xu, S. Feng, E. Cousineau, Y. Du, B. Burchfiel, R. Tedrake, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 2024.

[2] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1988.

[3] P. Florence, L. Manuelli, and R. Tedrake. Self-supervised correspondence in visuomotor policy learning. *IEEE Robotics and Automation Letters*, 2019.

[4] P. Florence, C. Lynch, A. Zeng, O. A. Ramirez, A. Wahid, L. Downs, A. Wong, J. Lee, I. Mordatch, and J. Tompson. Implicit behavioral cloning. In *Conference on Robot Learning*. PMLR, 2022.

[5] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.

[6] S. Lee, Y. Wang, H. Etukuru, H. J. Kim, N. M. M. Shafiullah, and L. Pinto. Behavior generation with latent actions. *arXiv preprint arXiv:2403.03181*, 2024.

[7] Z. Fu, T. Z. Zhao, and C. Finn. Mobile aloha: Learning bimanual mobile manipulation with low-cost whole-body teleoperation. *arXiv preprint arXiv:2401.02117*, 2024.

[8] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, J. Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 2018.

[9] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[10] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 2020.

[11] M. Reuss, M. Li, X. Jia, and R. Lioutikov. Goal-conditioned imitation learning using score-based diffusion policies. *arXiv preprint arXiv:2304.02532*, 2023.

[12] T. Pearce, T. Rashid, A. Kanervisto, D. Bignell, M. Sun, R. Georgescu, S. V. Macua, S. Z. Tan, I. Momennejad, K. Hofmann, et al. Imitating human behaviour with diffusion models. *arXiv preprint arXiv:2301.10677*, 2023.

[13] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 1999.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[15] M. Psenka, A. Escontrela, P. Abbeel, and Y. Ma. Learning a diffusion model policy from rewards via q-score matching. *arXiv preprint arXiv:2312.11752*, 2023.

[16] Z. Wang, J. J. Hunt, and M. Zhou. Diffusion policies as an expressive policy class for offline reinforcement learning. *arXiv preprint arXiv:2208.06193*, 2022.

[17] P. Hansen-Estruch, I. Kostrikov, M. Janner, J. G. Kuba, and S. Levine. Idql: Implicit q-learning as an actor-critic method with diffusion policies. *arXiv preprint arXiv:2304.10573*, 2023.

[18] L. Yang, Z. Huang, F. Lei, Y. Zhong, Y. Yang, C. Fang, S. Wen, B. Zhou, and Z. Lin. Policy representation via diffusion probability model for reinforcement learning. *arXiv preprint arXiv:2305.13122*, 2023.

[19] X. B. Peng, A. Kumar, G. Zhang, and S. Levine. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*, 2019.

[20] J. Peters and S. Schaal. Reinforcement learning by reward-weighted regression for operational space control. In *Proceedings of the 24th international conference on Machine learning*, 2007.

[21] B. Kang, X. Ma, C. Du, T. Pang, and S. Yan. Efficient diffusion policies for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 2024.

[22] P. J. Ball, L. Smith, I. Kostrikov, and S. Levine. Efficient online reinforcement learning with offline data. In *International Conference on Machine Learning*, pages 1577–1594. PMLR, 2023.

[23] M. Nakamoto, S. Zhai, A. Singh, M. Sobol Mark, Y. Ma, C. Finn, A. Kumar, and S. Levine. Cal-ql: Calibrated offline rl pre-training for efficient online fine-tuning. *Advances in Neural Information Processing Systems*, 36, 2024.

[24] H. Hu, S. Mirchandani, and D. Sadigh. Imitation bootstrapped reinforcement learning. *arXiv preprint arXiv:2311.02198*, 2023.

[25] A. Q. Nichol and P. Dhariwal. Improved denoising diffusion probabilistic models. In *International conference on machine learning*, 2021.

[26] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, 2015.

[27] K. Black, M. Janner, Y. Du, I. Kostrikov, and S. Levine. Training diffusion models with reinforcement learning. *arXiv preprint arXiv:2305.13301*, 2023.

[28] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[29] A. Mandlekar, D. Xu, J. Wong, S. Nasiriany, C. Wang, R. Kulkarni, L. Fei-Fei, S. Savarese, Y. Zhu, and R. Martín-Martín. What matters in learning from offline human demonstrations for robot manipulation. In *arXiv preprint arXiv:2108.03298*, 2021.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 2017.

[31] M. Heo, Y. Lee, D. Lee, and J. J. Lim. Furniturebench: Reproducible real-world benchmark for long-horizon complex manipulation. *arXiv preprint arXiv:2305.12821*, 2023.

[32] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017.

[33] H. Zhu, A. Gupta, A. Rajeswaran, S. Levine, and V. Kumar. Dexterous manipulation with deep reinforcement learning: Efficient, general, and low-cost. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3651–3657. IEEE, 2019.

[34] M. Torne, A. Simeonov, Z. Li, A. Chan, T. Chen, A. Gupta, and P. Agrawal. Reconciling reality through simulation: A real-to-sim-to-real approach for robust manipulation. *arXiv preprint arXiv:2403.03949*, 2024.

[35] M. Alakuijala, G. Dulac-Arnold, J. Mairal, J. Ponce, and C. Schmid. Residual reinforcement learning from demonstrations. *arXiv preprint arXiv:2106.08050*, 2021.

[36] S. Haldar, J. Pari, A. Rai, and L. Pinto. Teach a robot to fish: Versatile imitation from one minute of demonstrations. *arXiv preprint arXiv:2303.01497*, 2023.

[37] L. Ankile, A. Simeonov, I. Shenfeld, M. Torne, and P. Agrawal. From imitation to refinement–residual rl for precise visual assembly. *arXiv preprint arXiv:2407.16677*, 2024.

[38] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, et al. Deep q-learning from demonstrations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[39] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

[40] A. Nair, A. Gupta, M. Dalal, and S. Levine. Awac: Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359*, 2020.

[41] J. Luo, Z. Hu, C. Xu, Y. L. Tan, J. Berg, A. Sharma, S. Schaal, C. Finn, A. Gupta, and S. Levine. Serl: A software suite for sample-efficient robotic reinforcement learning. *arXiv preprint arXiv:2401.16013*, 2024.

[42] Z. Zhu, H. Zhao, H. He, Y. Zhong, S. Zhang, Y. Yu, and W. Zhang. Diffusion models for reinforcement learning: A survey. *arXiv preprint arXiv:2311.01223*, 2023.

[43] M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine. Planning with diffusion for flexible behavior synthesis. *arXiv preprint arXiv:2205.09991*, 2022.

[44] A. Ajay, Y. Du, A. Gupta, J. B. Tenenbaum, T. S. Jaakkola, and P. Agrawal. Is conditional generative modeling all you need for decision making? In *The Eleventh International Conference on Learning Representations*, 2023.

[45] W. Goo and S. Niekum. Know your boundaries: The necessity of explicit behavioral cloning in offline rl. *arXiv preprint arXiv:2206.00695*, 2022.

[46] H. Chen, C. Lu, C. Ying, H. Su, and J. Zhu. Offline reinforcement learning via high-fidelity generative behavior modeling. *arXiv preprint arXiv:2209.14548*, 2022.

[47] M. Rigter, J. Yamada, and I. Posner. World models via policy-guided trajectory diffusion. *arXiv preprint arXiv:2312.08533*, 2023.

[48] J. Song, C. Meng, and S. Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020.

[49] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention (MICCAI)*, 2015.

[50] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

[51] X. Jia, D. Blessing, X. Jiang, M. Reuss, A. Donat, R. Lioutikov, and G. Neumann. Towards diverse behaviors: A benchmark for imitation learning with human demonstrations. *arXiv preprint arXiv:2402.14606*, 2024.

[52] A. Block, A. Jadbabaie, D. Pfrommer, M. Simchowitz, and R. Tedrake. Provable guarantees for generative behavior cloning: Bridging low-level stability and high-level behavior. *Advances in Neural Information Processing Systems*, 2024.

[53] B. Chen, D. M. Monso, Y. Du, M. Simchowitz, R. Tedrake, and V. Sitzmann. Diffusion forcing: Next-token prediction meets full-sequence diffusion. *arXiv preprint arXiv:2407.01392*, 2024.

[54] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, 2012.

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

[56] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. AraÃšjo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 2022.

[57] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

[58] Y. Lin, A. S. Wang, G. Sutanto, A. Rai, and F. Meier. Polymetis. https://facebookresearch.github.io/fairo/polymetis/, 2021.

[59] J. Wang and E. Olson. Apriltag 2: Efficient and robust fiducial detection. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

## Contents

## A Extended Related Work

### A.1 RL training of robot policies with offline data

Here, we discuss related work in training robot policies using RL augmented with offline data to help RL better explore online in sparse reward settings.

One simple form is to use offline data to pre-train the policy, typically using behavior cloning, and then fine-tune the policy online. This is the approach that **DPPO** takes. Often, a regularization loss is applied to constrain the fine-tuned policy to stay close to the base policy, leading to natural fine-tuned behavior and often better learning [32, 33, 34]. **DPPO** does not apply regularization at fine-tuning as we find the on-manifold exploration helps **DPPO** maintain natural behavior after fine-tuning Section 4.3. Another popular approach is to learn a *residual* policy with RL on top of the frozen base policy [35, 36]. A closer work to ours is Ankile et al. [37], which trains a one-step residual non-diffusion policy with on-policy RL on top of a pre-trained chunked diffusion policy. This approach has the benefit of being fully closed-loop but lacks the structured on-manifold exploration of **DPPO**. Another hybrid approach is from Hu et al. [24], which uses pre-trained and fine-tuned policies to sample online experiences.

Another popular line of work, instead of training a base policy using offline data, directly adds the data in the replay buffer for online, off-policy learning in a single stage [38, 39, 40]. One recent approach from Ball et al. [22], **RLPD**, further improves sample efficiency from previous off-policy methods incorporating, e.g., critic ensembling. Luo et al. [41] demonstrates **RLPD** solving real-world manipulation tasks (although generally less challenging than ones solved by **DPPO**). Other approaches including **Cal-QL** build on offline RL to learn from offline data and then switch to online RL [23, 17]. **IBRL** from Hu et al. [24] pre-trains the base policy and samples offline data in fine-tuning.

### A.2 Diffusion-based RL methods

This section discusses related methods that directly train or improve diffusion-based policies with RL methods. The baselines to which we compare in Section 4.1 are discussed below as well, and are highlighted in their corresponding colors. We also refer the readers to Zhu et al. [42] for an extensive survey on diffusion models for RL.

Most previous works have focused on the **offline** setting with a static dataset. One line of work focuses on state trajectory planning and *guiding* the denoising sampling process such that the sampled actions satisfy some desired objectives. Janner et al. [43] applies classifier guidance that generates trajectories with higher predicted rewards. Ajay et al. [44] introduces classifier-free guidance that avoids learning the value of noisy states. There is another line of work that uses diffusion models as an action policy (instead of state planner) and generally applies Q-learning. **DQL** [16] introduces Diffusion Q-Learning that learns a state-action critic for the final denoised actions and backpropagates the gradient from the critic through the entire Diffusion Policy (actor) denoising chain, akin to the usual Q-learning. **IDQL** [17], or Implicit Diffusion Q-learning, proposes learning the critic to select the actions at inference time for either training or evaluation while fitting the actor to all sampled actions. Kang et al. [21] instead proposes using the critic to re-weight the sampled actions for updating the actor itself, similar to weighted regression baselines **DAWR** and **DRWR** introduced in our work. Goo and Niekum [45] similarly extracts the policy in the spirit of AWR [19]. Chen et al. [46] trains the critic using value iteration instead based on samples from the actor.

We note that methods like DQL and IDQL can also be applied in the **online** setting. A small amount of work also focuses entirely on the online setting. DIPO [18] differs from DQL and related work in that it uses the critic to update the sampled actions ("action gradient") instead of the actor — the actor is then fitted with updated actions from the replay buffer. QSM, or Q-Score Matching [15], suggests that optimizing the likelihood of the entire chain of denoised actions can be inefficient (contrary to our findings in the fine-tuning setting) and instead proposes learning the optimal policy by iteratively aligning the gradient of the actor (i.e., score) with the action gradient of the critic. Rigter et al. [47] proposes learning a diffusion dynamic model to generate synthetic trajectories for online training of a non-diffusion RL policy.

## B Best Practices for DPPO

**Fine-tune only the last few denoising steps.** Diffusion Policy often uses up to $K = 100$ denoising steps with DDPM to better capture the complex data distribution of expert demonstrations. With DPPO, we can choose to fine-tune only a subset of the denoising steps instead, e.g., the last $K'$ steps. We find this speeds up DPPO training and reduces GPU memory usage without sacrificing the asymptotic performance. Instead of fine-tuning the pre-trained model weights $\theta$, we make a copy $\theta_{\text{FT}}$ — $\theta$ is frozen and used for the early denoising steps, while $\theta_{\text{FT}}$ is used for the last $K'$ steps and updated with DPPO.

**Fine-tune DDIM.** Instead of fine-tuning all $K$ or the last few steps of the DDPM, one can also apply Denoising Diffusion Implicit Model (DDIM) [48] during fine-tuning, which greatly reduces the number of sampling steps $K^{\text{DDIM}} \ll K$, e.g., as few as 5 steps, and thus potentially improves DPPO efficiency as fewer steps are fine-tuned.

$$x^{k-1} \sim p_\theta^{\text{DDIM}}(x^{k-1}|x^k) := \mathcal{N}(x^{k-1}; \mu^{\text{DDIM}}(x^k, \varepsilon_\theta(x^k, k)), \eta\sigma_k^2 \mathrm{I}), \quad k = K^{\text{DDIM}}, ..., 0. \quad \text{(B.1)}$$

Although DDIM is typically used as a deterministic sampler by setting $\eta = 0$ in (B.1), we can use $\eta > 0$ for fine-tuning that provides exploration noise and avoids calculating Gaussian likelihood with a Dirac distribution. In practice, we set $\eta = 1$ for training (equivalent to applying DDPM [48]) and then $\eta = 0$ for evaluation. *We reserve DDIM sampling for our pixel-based experiments and long-horizon furniture assembly tasks, where the efficiency improvements are much desired.*

**Diffusion noise scheduling.** We use the cosine schedule for $\sigma_k$ introduced in [25], which was originally annealed to a small value on the order of $1E-4$ at $k = 0$. In DPPO, the value of $\sigma_k$ also translates to the exploration noise that is crucial to training efficiency. Empirically, we find that clipping $\sigma_k$ to a higher minimum value (denoted $\sigma_{\text{min}}^{\text{exp}}$, e.g., $0.01 - 0.1$) when sampling actions helps exploration (see sensitivity analysis in Appendix D.3). Additionally we clip $\sigma_k$ to be at least 0.1 (denoted $\sigma_{\text{min}}^{\text{prob}}$) when evaluating the Gaussian likelihood $\log \bar{\pi}_\theta(\bar{a}_{\bar{t}}|\bar{s}_{\bar{t}})$, which improves training stability by avoiding large magnitude.

**Network architecture.** We study both Multi-layer Perceptron (MLP) and UNet [49] as the policy heads in Diffusion Policy. An MLP offers simpler setup and we find it generally fine-tunes more stably with DPPO. Moreover, since the UNet applies convolution to the denoised action, we can pre-train and fine-tune with different action chunk size $T_a$ (the number of environment timesteps that the policy predicts future actions with), e.g., 16 and 8. We find that DPPO benefits from pre-training with larger $T_a$ (better prediction) and fine-tuning with smaller $T_a$ (more amenable to policy gradient)[1].

[1]With fully-connected layers in MLP, empirically we find that using different chunk sizes for pre-training and fine-tuning with MLP leads to training instability.

## C Summary of All Baselines

**Comparison to Other Diffusion RL Methods**

| *Method Name* | *Summary* |
|---|---|
| **DPPO** (ours) | Competitive on GYM; much **stronger** on ROBOMIMIC; the only one to solve `Transport`. |
| **DAWR** (ours) | Competitive on GYM; much weaker on ROBOMIMIC. |
| **DRWR** (ours) | Weaker on GYM; competitive on all of ROBOMIMIC but `Transport` |
| **IDQL** [17] | Competitive on GYM, much weaker on ROBOMIMIC. |
| **DQL** [16] | Much weaker on GYM and ROBOMIMIC. |
| **QSM** [15] | Much weaker on GYM and ROBOMIMIC. |
| **DIPO** [18] | Much weaker on GYM and ROBOMIMIC. |

**Comparison to Other Demonstration-Augmented RL Methods**

| *Method Name* | *Summary* |
|---|---|
| **DPPO** (ours) | Much **stronger** on ROBOMIMIC; underperforms **RLPD** and **Cal-QL** on `HalfCheetah-v2`. |
| **RLPD** [22] | Very efficient on `HalfCheetah-v2`; zero reward on ROBOMIMIC. |
| **Cal-QL** [23] | More efficient than **DPPO** but less efficient than **RLPD** on `HalfCheetah-v2`; zero reward on ROBOMIMIC. |
| **IBRL** [24] | Weaker than **DPPO** on ROBOMIMIC |

**Comparison to Other Policy Parameterization/Architecture**

| ROBOMIMIC State | *Summary* |
|---|---|
| **DPPO**-MLP (ours) | Performs **best overall**; attains max reward on `Square`. |
| **DPPO**-UNet (ours) | Slightly underperforms **DPPO**-MLP; second best. |
| Gaussian-MLP | Competitive on `Square`; zero reward on `Transport`. |
| Gaussian-Transformer | Weaker on `Square`; zero reward on `Transport`. |
| GMM-MLP | Low reward on `Square`; zero reward on `Transport`. |
| GMM-Transformer | Low reward on `Square` and `Transport`. |
| ROBOMIMIC Pixel | |
| **DPPO**-ViT-MLP (ours) | Performs **better overall**; attains strong reward on `Square` and `Transport`. |
| Gaussian-ViT-MLP | Much weaker on `Square`; zero reward on `Transport`. |
| FURNITURE-BENCH | |
| **DPPO**-UNet (ours) | Performs **better overall** except slightly weaker on `Lamp` (`Low` randomness) and tied on `One-leg Low`. |
| Gaussian-MLP | Slightly stronger on `Lamp` with `Low` randomness and tied on `One-leg Low`; much weaker otherwise. |
| Sim-to-Real | |
| **DPPO**-UNet (ours) | Tied with Gaussian-MLP in sim; **much stronger** in transfer to real. |
| Gaussian-MLP | Strong in sim; zero success in real |
| Gaussian w/ BC Loss | Markedly weaker in sim; markedly weaker than **DPPO** (but non-zero reward) in real. |

## D Additional experimental results

### D.1 Evaluation on Furniture-Bench, and sim-to-real transfer

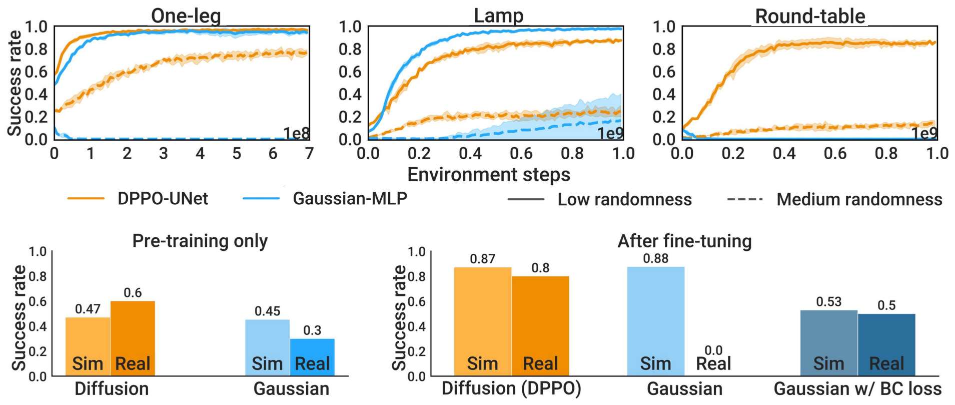


Figure 5: (Top) **DPPO** vs. Gaussian-MLP baseline in **simulated** FURNITURE-BENCH **tasks**. Results are averaged over three seeds. (Bottom) **Sim-to-real transfer results in** `One-leg`.

Here we evaluate **DPPO** on the long-horizon manipulation tasks from FURNITURE-BENCH [31]. We compare **DPPO** to Gaussian-MLP, the overall most effective baseline from Section 4.2. Fig. 5 (top row) shows the evaluation success rate over fine-tuning iterations. **DPPO** exhibits strong training stability and improves policy performance in all six settings. **DPPO** also transfers well to physical hardware zero-shot. Please see Appendix D.1 for detailed results. Gaussian-MLP collapses to zero success rate in all three tasks with `Med` randomness (except for one seed in `Lamp`) and `Round-table` with `Low` randomness.

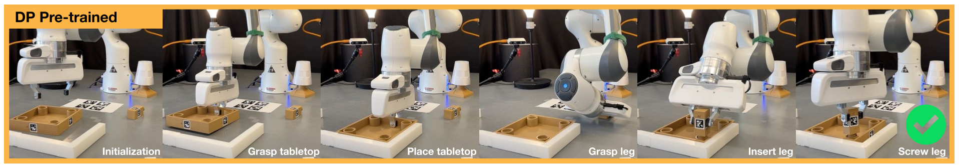
Note that we are only using 50 human demonstrations for pre-training; we expect **DPPO** can leverage additional human data (better state space coverage) to further improve in `Med`, which is corroborated by ablation studies in Appendix D.4.

**Sim-to-real transfer.** We evaluate **DPPO** and Gaussian policies trained in the simulated `One-leg` task on physical hardware zero-shot (i.e., **no real data fine-tuning / co-training**) over 20 trials. Please see additional simulation training and hardware details in Appendix G.8. Fig. 5 (bottom row) shows simulated and hardware success rates after pre-training and fine-tuning. Notably, **DPPO** improves the real-world performance to 80% (16 out of 20 trials). Though the Gaussian policy achieves a high success rate in simulation after fine-tuning (88%), it fails entirely on hardware (0%). Supplemental video suggests it exhibits volatile and jittery behavior. For stronger comparison, we also fine-tune the Gaussian policy with an auxiliary behavior-cloning loss [34] such that the fine-tuned policy is encouraged to stay close to the base policy. However, this limits fine-tuning and only leads to a 53% success rate in simulation and 50% in reality.

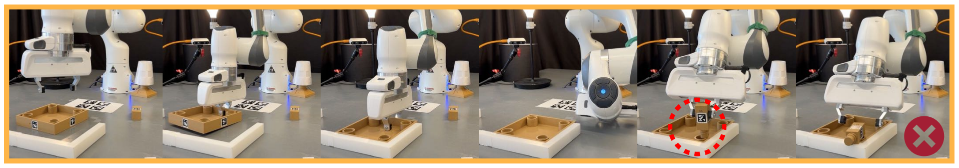
Qualitatively, we find fine-tuned policies to be more robust and exhibit more corrective behaviors than pre-trained-only policies, especially during the insertion stage of the task; Fig. 6 shows representative rollouts on hardware. Overall, these results demonstrate the strong sim-to-real capabilities of **DPPO**; Appendix F provides a conjectural mechanism for why this may be the case.

### D.2 Comparing to other demo-augmented RL methods

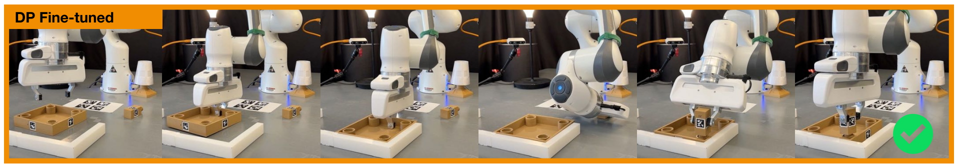
We also find **DPPO** leads to superior final performance in manipulation tasks compared to other RL methods leveraging offline data, including **RLPD** [22], **Cal-QL** [23], and **IBRL** [24]. The full results are shown in Fig. 7 below. We use action chunk size $T_a = 1$ following the setup from these methods (**DPPO** may benefit from longer chunk size, albeit). The three baselines all achieve high reward in `HalfCheetah-v2` with much higher sample efficiency thanks to performing off-policy updates. However, in sparse-reward ROBOMIMIC tasks including `Can` and `Square`, **DPPO**



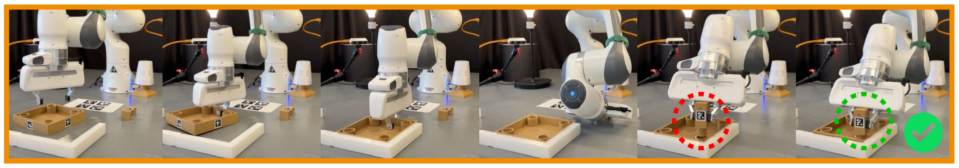
**(A)** Pre-trained Diffusion policy performs successful rollout



**(B)** Policy pushes peg down without proper alignment with the hole before releasing the peg, making it topple over



**(C)** Fine-tuned DPPO policy performs successful rollout



**(D)** Initial peg alignment is off, the policy corrects placement until it is properly inserted in the hole before letting go

Figure 6: **Qualitative comparison of pre-trained vs. fine-tuned DPPO policies in real evaluation. (A)** Successful rollout with the pre-trained policy. **(B)** Failed rollout with the pre-trained policy due to imprecise insertion. **(C)** Successful rollout with the fine-tuned policy. **(D)** Successful rollout with the fine-tuned policy that requires corrective behavior.

outperforms all three significantly and achieves ∼100% final success rates. **RLPD** and **Cal-QL** fail to achieve any success (0%) during evaluation, while **IBRL** saturates at lower success levels.

Our results with **RLPD** in `Can` and `Square` corroborates those from Hu et al. [24]. **IBRL** is shown to achieve high success rates ($> 90\%$) in both tasks in Hu et al. [24]; we hypothesize here it underperforms possibly due to (1) the noisier expert data (Multi-Human dataset from ROBOMIMIC) affects gradient update with mixed batches of online and offline data, and (2) our setup not using any history observation unlike Hu et al. [24] stacking three observations.

Lastly, we note that although **DPPO** uses more environment steps, it runs significantly faster than the baselines as it leverages sampling from highly parallelized environments (40 in `HalfCheetah-v2` and 50 in `Can` and `Square`), while off-policy methods may fail to fully leverage such parallelized setup as the policy is updated less often and the performance may be affected.

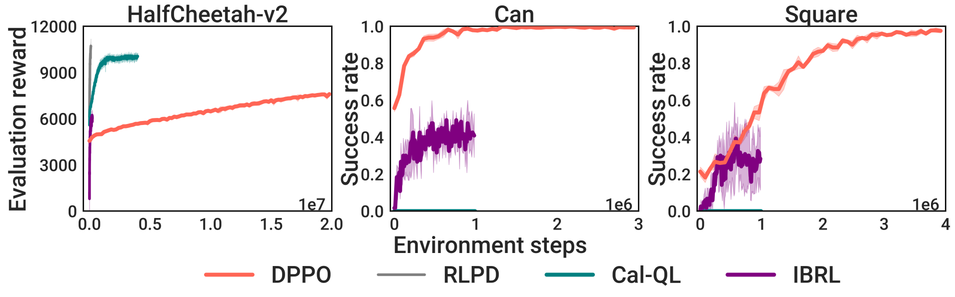


Figure 7: **Comparing to other demo-augmented RL methods.** Results are averaged over five seeds in `HalfCheetah-v2` and three seeds in `Can` and `Square`.

### D.3 Ablation studies on design decisions in DPPO

**1. Choice of advantage estimator.** In Section 3.1 we demonstrate how to efficiently estimate the advantage used in PPO updates by learning $\tilde{V}(s_t)$ that only depends on the environment state; the advantage used in DPPO is formally

$$\hat{A} = \gamma_{\text{DENOISE}}^{k}(\bar{r}(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}) - \tilde{V}(s_t)).$$

We now compare this choice with learning the value of the full state $\bar{s}_{\bar{t}(t,0)}$ that includes environment state $s_t$ *and* denoised action $a_t^{k=1}$. We additionally compare with the state-action Q-function estimator used in Psenka et al. [15][2], $\tilde{Q}(s_t, a_t^{k=0})$, that does not directly use the rollout reward $\bar{r}$ in the advantage.

Fig. 8 shows the fine-tuning results in `Hopper-v2` and `HalfCheetah-v2` from GYM, and `Can` and `Square` from ROBOMIMIC. On the simpler `Hopper-v2`, we observe that the two baselines, both estimating the value of some action, achieves higher reward during fine-tuning than DPPO's choice. However, in the more challenging tasks, the environment-state-only advantage used in DPPO consistently leads to the most improved performance. We believe estimating the accurate value of applying a continuous and high-dimensional action can be challenging, and this is exacerbated by the high stochasticity of diffusion-based policies and the action chunk size. The results here corroborate the findings in Section 4.1 that off-policy Q-learning methods can perform well in `Hopper-v2` and `Walker2D-v2`, but exhibit training instability in manipulation tasks from ROBOMIMIC.

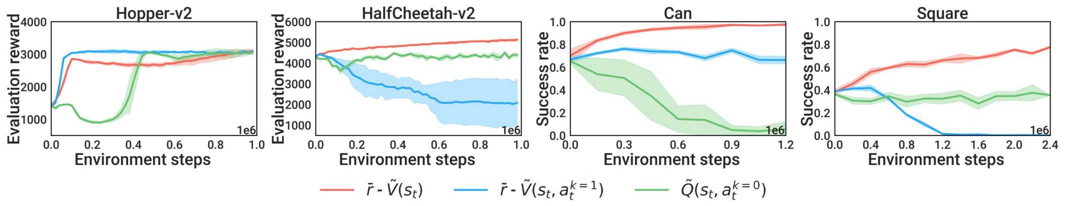


Figure 8: **Choice of advantage estimator.** Results are averaged over five seeds in `Hopper-v2` and `HalfCheetah-v2` and three seeds in `Can` and `Square`.

**Denoising discount factor.** We further examine how $\gamma_{\text{DENOISE}}$ in the DPPO advantage estimator affects fine-tuning. Using a smaller value (i.e., more discount) has the effect of downweighting the contribution of earlier denoising steps in the policy gradient. Fig. 9 shows the fine-tuning results in the same four tasks with varying $\gamma_{\text{DENOISE}} \in [0.5, 0.8, 0.9, 1]$. We find in `Hopper-v2` and `HalfCheetah-v2` $\gamma_{\text{DENOISE}} = 0.8$ leads to better efficiency while smaller $\gamma_{\text{DENOISE}} = 0.5$ slows training. The value does not affect training noticeably in `Can`. In `Square` the smaller $\gamma_{\text{DENOISE}} = 0.5$ works slightly better. Overall in manipulation tasks, DPPO training seems relatively robust to this choice.

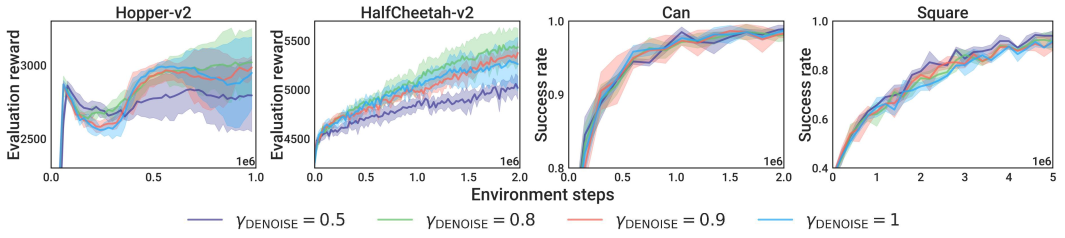


Figure 9: **Choice of denoising discount factor.** Results are averaged over five seeds in `Hopper-v2` and `HalfCheetah-v2` and three seeds in `Can` and `Square`.

[2]Psenka et al. [15] applies off-policy training with double Q-learning (according to its open-source implementation) and policy gradient over the denoising steps. Note that this is a baseline in Psenka et al. [15] that is conjectured to be inefficient. We follow the same except for applying on-policy PPO updates.

**2. Choice of diffusion noise schedule.** We find it helpful to clip the diffusion noise $\sigma_k$ to a higher minimum value $\sigma_{\text{min}}^{\text{exp}}$ to ensure sufficient exploration. In Figure 10, we perform analysis on varying $\sigma_{\text{min}}^{\text{exp}} \in \{.001, .01, .1, .2\}$ (keeping $\sigma_{\text{min}}^{\text{prob}} = .1$ to evaluate likelihoods). Although in `Can` the choice of $\sigma_{\text{min}}^{\text{exp}}$ does not affect the fine-tuning performance, in `Square` a higher $\sigma_{\text{min}}^{\text{exp}} = 0.1$ is required to prevent the policy from collapsing. We conjecture that this is due to limited exploration causing policy over-optimizing the collected samples that exhibit limited state-action coverage. We also visualize the trajectories at the beginning of fine-tuning in `Avoid` task from D3IL. With higher $\sigma_{\text{min}}^{\text{exp}}$, the trajectories still remain near the two modes of the pre-training data but exhibit a higher coverage in the state space — we believe this additional coverage leads to better exploration. Anecdotally, we find terminating the denoising process early can also provide exploration noise and lead to comparable results, but it requires a more involved implementation around the denoising MDP.

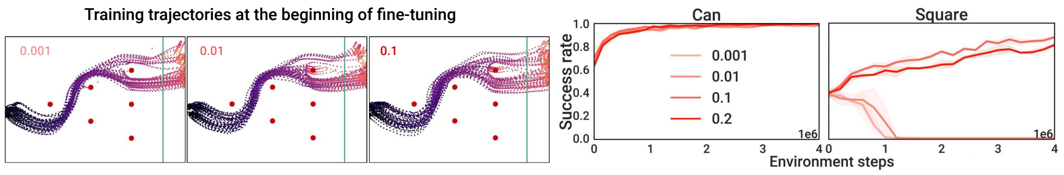


Figure 10: **Choice of minimum diffusion noise.** Results are averaged over three seeds. Note in Left, with higher minimum noise level, the sampled trajectories exhibit wider coverage at the two modes but still maintain the overall structure.

**3. Choice of the number of fine-tuned denoising steps.** We examine how the number of fine-tuned denoising steps in **DPPO**, $K'$, affects the fine-tune performance and wall-clock time in Fig. 11. We show the curves of individual runs (three for each $K'$) instead of the average as their wall-clock times (X-axis) are not perfectly aligned. Generally, fine-tuning too few denoising steps (e.g., 3) can lead to subpar asymptotic performance and slower convergence especially in `Can`. Fine-tuning 10 steps leads to the overall best efficiency. Similar results are also shown in Fig. 14 with `Avoid` task. Lastly, we note that the GPU memory usage scales linearly with $K'$.

We note that the findings here mostly correlate with those from varying the denoising discount factor, $\gamma_{\text{DENOISE}}$. Discounting the earlier denoising steps in the policy gradient can be considered as a soft version of hard limiting the number of fine-tuned denoising steps. Depending on the amount of fine-tuning needed from the pre-trained action distribution, one can flexibly adjust $\gamma_{\text{DENOISE}}$ and $K'$ to achieve the best efficiency.

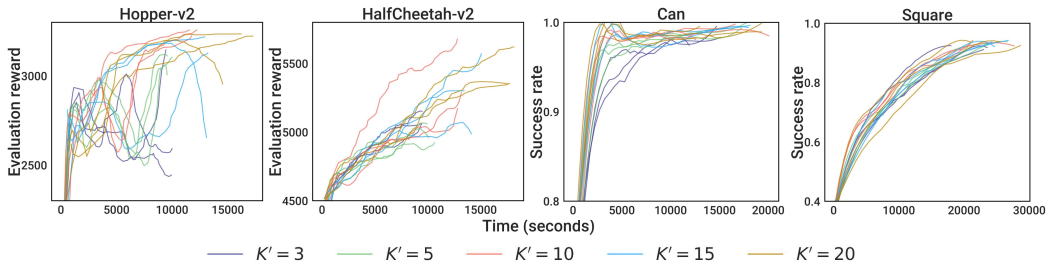


Figure 11: **Choice of number of fine-tuned denoising steps, $K'$.** Individual runs are shown. The curves are smoothed using a Savitzky–Golay filter.

## D.4 Effect of expert data

We investigate the effect of the amount of pre-training expert data on fine-tuning performance. In Fig. 12 we compare **DPPO** and Gaussian in `Hopper-v2`, `Square`, and `One-leg` task from FURNITURE-BENCH, using varying numbers of expert data (episodes) denoted in the figure. Overall, we find **DPPO** can better leverage the pre-training data and fine-tune to high success rates. Notably, **DPPO** obtains non-trivial performance (60% success rate) on `One-leg` from only 10 episode of demonstrations.

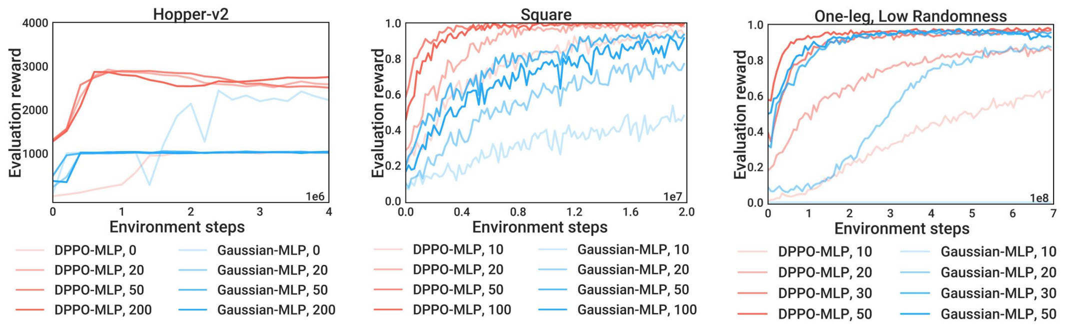


Figure 12: **Varying the number of expert demonstrations.** The numbers in the legends indicates the number of episodes used in pre-training.

**Training from scratch.** In Fig. 13 we compare **DPPO** (10 denoising steps) and Gaussian *trained from scratch* (no pre-training on expert data) in the three OpenAI GYM tasks. As using larger action chunk sizes $T_a$ leads to poor from-scratch training shown in Fig. 12, we focus on single-action chunks $T_a = 1$ as is typical in RL benchmarking. Though we find Gaussian trains faster than **DPPO** (expected since **DPPO** solves an MDP with longer effective horizon), **DPPO** still attains reasonable final performance. However, due to the multi-step (10) denoising sampling, **DPPO** takes about $6\times$ wall-clock time compared to Gaussian. We hope that future work will explore how to design the training curriculum of denoising steps for the best balance of training performance and wall-clock efficiency.

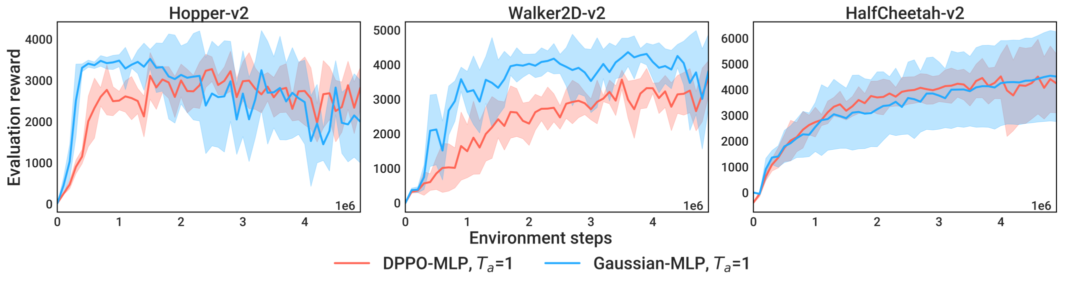


Figure 13: **No expert data / pre-training** with GYM tasks. Results are averaged over five seeds.

## D.5 Comparing to other policy parameterizations in `Avoid`

Figure 14 depicts the performance of various parameterizations of **DPPO** (with differing numbers of fine-tuned denoising steps, $K'$) to Gaussian and GMM baselines. We study the `Avoid` task from D3IL, after pre-training with the data from M1, M2, M3 as described in Appendix F. We find that, for $K' \in \{15, 20\}$, **DPPO** attains the highest performance of all methods and trains the quickest in terms of environment steps; on M1, M2, it appears to attain the greatest terminal performance as well. $K' = 10$ appears slightly better than, but roughly comparable to, the Gaussian baseline, with GMM and $K' < 10$ performing less strongly.

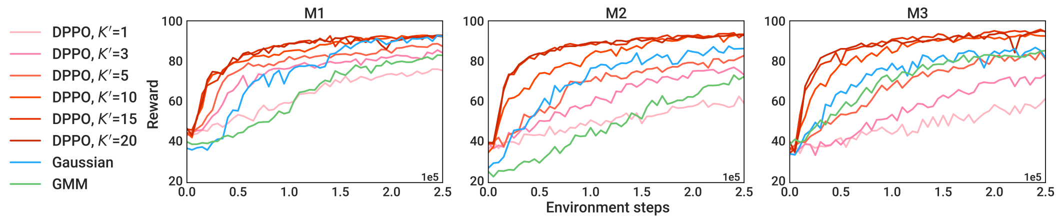


Figure 14: Fine-tuning performance (averaged over five seeds, standard deviation not shown) after pre-training with M1, M2, and M3 in **`Avoid` task from D3IL**. **DPPO** ($K = 20$), Gaussian, and GMM policies are compared. We also sweep the number of fine-tuned denoising steps $K'$ in **DPPO**.

### D.6 Comparing to other policy parameterizations in the easier tasks from ROBOMIMIC

Figure 15 compares the performance of **DPPO** to Gaussian and GMM baslines, across a variety of architectures, and with **state** and **pixel** inputs, in `Lift` and `Can` environments in the ROBOMIMIC suite. Compared to the `Square` and `Transport` (results shown in Section 4), these environments are considered to be "easier", and this is reflected in the greater performance of **DPPO** and Gaussian baselines (GMM still exhibits subpar performance). Nonetheless, **DPPO** still achieves similar or even better sample efficiency compared to Gaussian baseline.

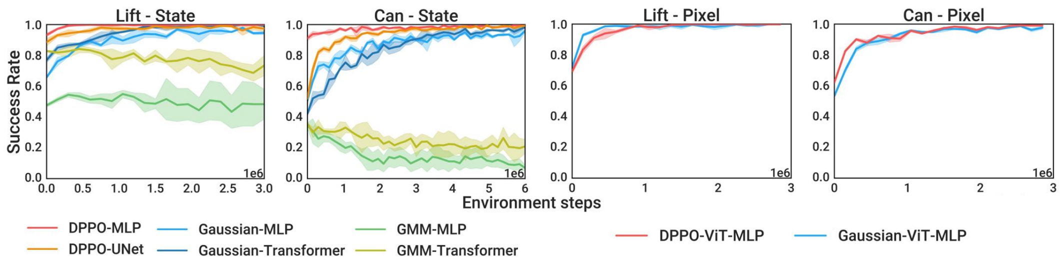


Figure 15: **Comparing to other policy parameterizations** in the easier `Lift` and `Can` tasks from ROBOMIMIC, with **state** (left) or **pixel** (right) observation. Results are averaged over three seeds.

### D.7 Comparing to policy gradient using exact likelihood of Diffusion Policy

Here we experiment another novel method (which, to our knowledge, has not been explicitly studied in any previous work) for performing policy gradient with diffusion-based policies. Although diffusion model does not directly model the action likelihood, $p_\theta(a_0|s)$, there have been ways to *estimate* the value, e.g., by solving the probability flow ODE that implements DDPM [50]. We refer the readers to Appendix. D in Song et al. [50] for a comprehensive exposition. We follow the official open-source code from Song et al.[3], and implement policy gradient (single-level MDP) that uses the exact action likelihood $\pi_\theta(a_t|s_t)$.

Fig. 16 shows the comparison between **DPPO** and diffusion policy gradient using exact likelihood estimate. Exact policy gradient improves the base policy in `Hopper-v2` but does not outperform **DPPO**. It also requires more runtime and GPU memory as it backpropagates through the ODE. In the more challenging `Can` its success rate drops to zero. Moreover, policy gradient with exact likelihood does not offer the flexibility of fine-tuning fewer-than-$K$ denoising steps or discounting the early denoising steps that **DPPO** offers, which have shown in Appendix D.3 to often improve fine-tuning efficiency.

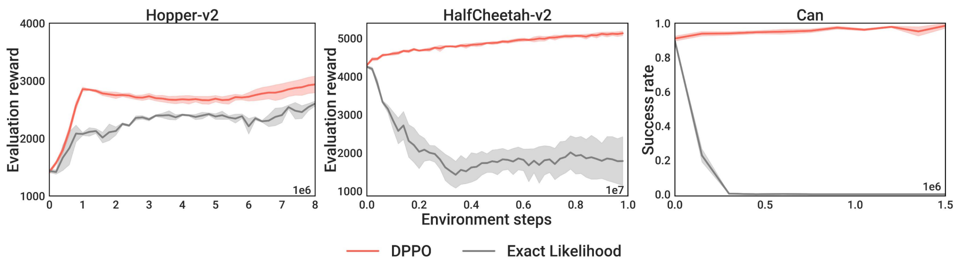


Figure 16: **Comparing to diffusion policy gradient with exact action likelihood.** Results are averaged over five seeds in `Hopper-v2` and `HalfCheetah-v2`, and three seeds in `Can`.

## E Reporting of Wall-Clock Times

**Comparing to other diffusion-based RL algorithms Section 4.1.** Table 1 and Table 2 shows the the wall-clock time used in each OpenAI GYM task and ROBOMIMIC task. In GYM tasks, on average

[3]https://github.com/yang-song/score_sde_pytorch

**DPPO** trains 41%, 37%, and 12% faster than **DAWR**, **DIPO**, and **DQL**, respectively, which all require a significant amount of gradient updates per sample to train stably. **QSM**, **DRWR**, and **IDQL** trains 43%, 33%, and 7% faster than **DPPO**, respectively. ROBOMIMIC tasks are more expensive to simulate, especially with `Transport` task, and thus the wall-clock difference is smaller among the different methods. All methods use comparable time except for **DIPO** that uses slightly more on average.

| Method | Task | | |
|---|---|---|---|
| | `Hopper-v2` | `Walker2D-v2` | `HalfCheetah-v2` |
| **DRWR** | 11.3 | 12.7 | 10.4 |
| **DAWR** | 30.4 | 30.7 | 27.1 |
| **DIPO** | 27.8 | 27.9 | 26.0 |
| **IDQL** | 16.3 | 16.1 | 15.5 |
| **DQL** | 20.5 | 20.5 | 17.6 |
| **QSM** | 9.6 | 9.9 | 9,7 |
| **DPPO** | 16.6 | 18.3 | 16.8 |

Table 1: **Wall-clock time** in seconds for a single training iteration in **OpenAI GYM tasks** when comparing diffusion-based RL algorithms. Each iteration involves 500 environment timesteps in each of the 40 parallelized environments running on 40 CPU threads and a NVIDIA RTX 2080 GPU (20000 steps total).

| Method | Task | | | |
|---|---|---|---|---|
| | `Lift` | `Can` | `Square` | `Transport` |
| **DRWR** | 32.5 | 39.5 | 59.8 | 346.1 |
| **DAWR** | 38.6 | 46.0 | 70.5 | 354.3 |
| **DIPO** | 43.9 | 51.6 | 73.3 | 359.7 |
| **IDQL** | 33.8 | 41.7 | 63.7 | 349.9 |
| **DQL** | 36.9 | 44.4 | 68.5 | 353.5 |
| **QSM** | 31.8 | 44.5 | 68.7 | 322.5 |
| **DPPO** | 35.2 | 42.0 | 65.6 | 350.3 |

Table 2: **Wall-clock time** in seconds for a single training iteration in **ROBOMIMIC tasks with state input** when comparing diffusion-based RL algorithms. Each iteration involves 4 episodes (1200 environment timesteps for `Lift` and `Can`, 1600 for `Square`, and 3200 for `Transport`) from each of the 50 parallelized environments running on 50 CPU threads and a NVIDIA L40 GPU (60000, 80000, 160000 steps).

**Comparing to other policy parameterizations and architecture Section 4.2 and Section 4.3.** Table 3 and Table 4 shows the wall-clock time used in fine-tuning in each ROBOMIMIC task with state or pixel input, respectively. Gaussian and GMM use similar times and Transformer is slightly more expensive than MLP. On average with state input, **DPPO**-MLP trains 24%, 21%, 24%, and 22% slower than baselines due to the more expensive diffusion sampling. **DPPO**-UNet requires more time with the extensive use of convolutional and normalization layers and trains on average 49% slower than **DPPO**-MLP. On average with pixel input, **DPPO**-ViT-MLP trains 14% slower than Gaussian-ViT-MLP — the difference is smaller than the state input case as the rendering in simulation can be expensive. Table 5 shows the wall-clock time used in FURNITURE-BENCH tasks. **DPPO**-UNet trains 20% slower than Gaussian-MLP on average.

| Method | Task | | | |
|---|---|---|---|---|
| | Lift | Can | Square | Transport |
| Gaussian-MLP | 27.7 | 35.7 | 56.2 | 255.6 |
| Gaussian-Transformer | 29.8 | 37.1 | 57.8 | 266.1 |
| GMM-MLP | 28.0 | 36.2 | 55.2 | 254.5 |
| GMM-Transformer | 29.5 | 37.4 | 58.1 | 260.2 |
| DPPO-MLP | 35.6 | 43.3 | 65.0 | 350.5 |
| DPPO-UNet | 83.6 | 92.7 | 130.4 | 431.1 |

Table 3: **Wall-clock time** in seconds for a single training iteration in **ROBOMIMIC tasks with state input** when comparing policy parameterizations. Each iteration involves 4 episodes (1200 environment timesteps for `Lift` and `Can`, 1600 for `Square`, and 3200 for `Transport`) from each of the 50 parallelized environments running on 50 CPU threads and a NVIDIA L40 GPU (60000, 80000, 160000 steps).

| Method | Task | | | |
|---|---|---|---|---|
| | Lift | Can | Square | Transport |
| Gaussian-ViT-MLP | 153.6 | 173.1 | 277.0 | 770.0 |
| DPPO-ViT-MLP | 194.9 | 202.5 | 328.5 | 871.3 |

Table 4: **Wall-clock time** in seconds for a single training iteration in **ROBOMIMIC tasks with pixel input** when comparing policy parameterizations. Each iteration involves 4 episodes (1200 environment timesteps for `Lift` and `Can`, 1600 for `Square`, and 3200 for `Transport`) from each of the 50 parallelized environments running on 50 CPU threads and a NVIDIA L40 GPU (60000, 80000, 160000 steps).

| Method | Task | | |
|---|---|---|---|
| | One-leg | Lamp | Round-table |
| Gaussian-MLP | 101.8 | 202.8 | 168.7 |
| DPPO-UNet | 148.4 | 258.2 | 188.6 |

Table 5: **Wall-clock time** in seconds for a single training iteration in **FURNITURE-BENCH tasks** when comparing policy parameterizations. Each iteration involves 1 episodes (700 environment timesteps for `One-leg`, and 1000 for `Lamp` and `Round-table`) from each of the 1000 parallelized environments running on a NVIDIA L40 GPU (700000, 1000000, 1000000 steps).

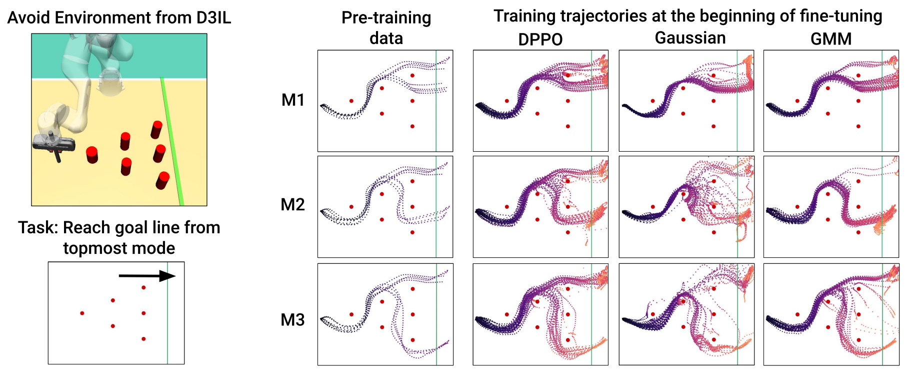
## F Understanding the performance of DPPO

We study the factors contributing to **DPPO**'s improvements in performance over the popular Gaussian and GMM methods introduced in Section 4.2. We use the `Avoid` environment from D3IL benchmark [51], where a robot arm needs to reach the other side of the table while avoiding an array of obstacles (Fig. 17, top-left). The action space is the 2D target location of the end-effector. D3IL provides expert demonstrations that covers different possible paths to the goal line — we consider three subsets of the demonstrations, M1, M2, and M3 in Fig. 17, each with two distinct modes; with only two modes in each setting, Gaussian (with exploration noise)[4] and GMM can fit the expert data distribution reasonably well, allowing fair comparisons in fine-tuning.

We pre-train MLP-based Diffusion, Gaussian, and GMM policies ($T_a = 4$ unless noted) with the demonstrations. For fine-tuning, we assign (sparse) reward when the robot reaches the goal line from the topmost mode. Gaussian and GMM policies are also fine-tuned with the PPO objective.

**Benefit 1: Structured, on-manifold exploration.** Fig. 17 (right) shows the sampled trajectories (with exploration noise) from **DPPO**, Gaussian, and GMM during the first iteration of fine-tuning. **DPPO** explores in wide coverage **around the expert data manifold**, whereas Gaussian generates less structured exploration noise (especially in M2) and GMM exhibits narrower coverage. More-

[4]Without noise, Gaussian policy is fully deterministic and cannot capture the two modes.



Avoid Environment from D3IL
Task: Reach goal line from topmost mode
Pre-training data
Training trajectories at the beginning of fine-tuning
DPPO
Gaussian
GMM
M1
M2
M3

Figure 17: (Left) We use the `Avoid` environment from Jia et al. [51] to visualize the **DPPO**'s exploration tendencies. The task is to reach the green goal line from the topmost mode. (Right) **Structured exploration.** We show sampled trajectories at the *first iteration of fine-tuning* for DPPO, Gaussian, and GMM after pre-training on three sets of expert demonstrations, M1, M2, and M3.

over, the combination of diffusion parameterization with the denoising of *action chunks* means that policy stochasticity in **DPPO** is **structured in both action dimension and time horizon**.

**Benefit 2: Training stability from multi-step denoising process.** In Fig. 18 (left), we run fine-tuning after pre-training with M2 and *attempt to de-stabilize fine-tuning* by gradually adding noise to the action during the fine-tuning process (see Appendix G.9 for details). We find that Gaussian and GMM's performance both collapse, while with **DPPO**, the performance is robust to the noise if at least four denoising steps are used. This property also allows **DPPO** to apply significant noise to the sampled actions, simulating an imperfect low-level controller to facilitate sim-to-real transfer in Section 4.3. In Fig. 18 (right), we also find **DPPO** enjoys greater training stability when fine-tuning long action chunks, e.g., up to $T_a = 16$, while Gaussian and GMM can fail to improve at all.

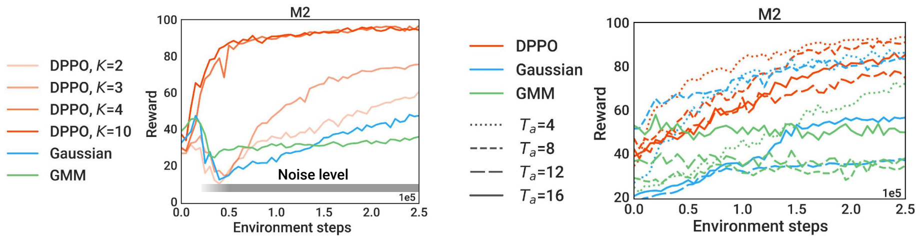


Figure 18: **Training stability.** Fine-tuning performance (averaged over five seeds, standard deviation not shown) after pre-training with M2. (Left) Noise is injected into the applied actions after a few training iterations. (Right) The action chunk size $T_a$ is varied.

Fig. 19 visualizes how **DPPO** affects the multi-step denoising process. Over fine-tuning iterations, the action distribution gradually converges through the denoising steps — the iterative refinement is largely preserved, as opposed to, e.g., "collapsing" to the optimal actions at the first fine-tuned denoising step or the final one. We postulate this contributes to the training stability of **DPPO**.

**Benefit 3: Robust and generalizable fine-tuned policy.** **DPPO** also generates final policies robust to perturbations in dynamics and the initial state distribution. In Fig. 20, we again add noise to the actions sampled from the fine-tuned policy (no noise applied during training) and find that **DPPO** policy exhibits strong robustness to the noise compared to the Gaussian policy. **DPPO** policy also converges to the (near-)optimal path from a larger distribution of initial states. This finding echoes theoretical guarantees that Diffusion Policy, capable of representing complex multi-modal data distribution, can effectively deconvolve noise from noisy states [52], a property used in Chen et al. [53] to stabilize long-horizon video generation.

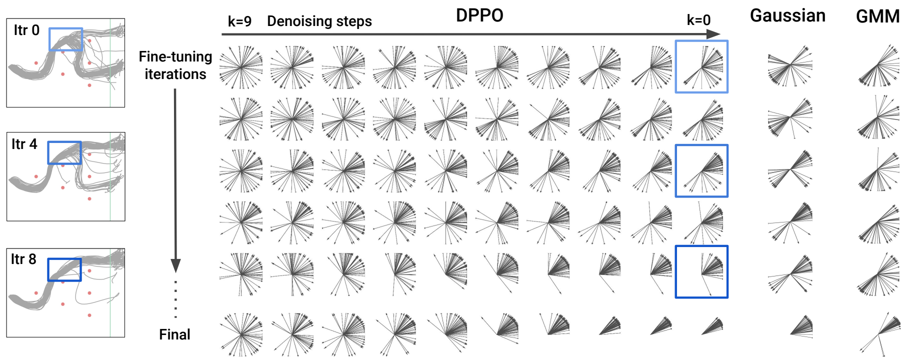


Figure 19: **Preserving the iterative refinement.** The 2D actions from 50 trajectories at the branching point *through fine-tuning* iterations after pre-training with M2. For **DPPO**, we also visualize the action distribution through the final denoising steps at each fine-tuning iteration.

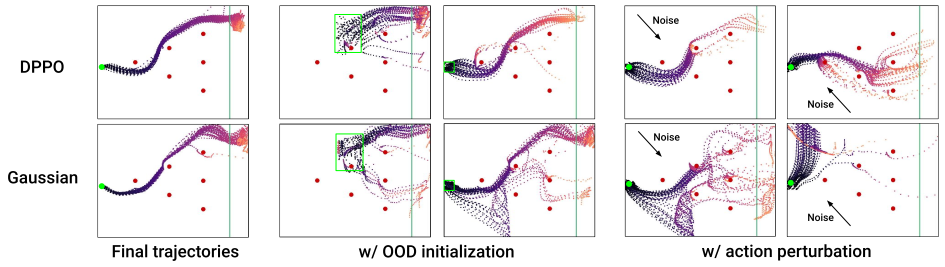


Figure 20: **Policy robustness** *after fine-tuning*. Green dot / box indicates the initial state region.

## G Additional Experimental Details

### G.1 Details of policy architectures used in all experiments

**MLP.** For most of the experiments, we use a Multi-layer Perceptron (MLP) with two-layer residual connection as the policy head. For diffusion-based policies, we also use a small MLP encoder for the state input and another small MLP with sinusoidal positional encoding for the denoising timestep input. Their output features are then concatenated before being fed into the MLP head. Diffusion Policy, proposed by Chi et al. [1], does not use MLP as the diffusion architecture, but we find it delivers comparable (or even better) pre-training performance compared to UNet.

**Transformer.** For comparing to other policy parameterizations in Section 4.2, we also consider Transformer as the policy architecture for the Gaussian and GMM baselines. We consider decoder only. No dropout is used. A learned positional embedding for the action chunk is the sequence into the decoder.

**UNet.** For comparing to other policy parameterizations in Section 4.2, we also consider UNet [49] as a possible architecture for DP. We follow the implementation from Chi et al. [1] that uses sinusoidal positional encoding for the denoising timestep input, except for using a larger MLP encoder for the observation input in each convolutional block. We find this modification helpful in more challenging tasks.

**ViT.** For pixel-based experiments in Section 4.2 we use Vision-Transformer(ViT)-based image encoder introduced by Hu et al. [24] before an MLP head. Proprioception input is appended to each channel of the image patches. We also follow [24] and use a learned spatial embedding for the ViT output to greatly reduce the number of features, which are then fed into the downstream MLP head.

### G.2 Additional details of GYM tasks and training in Section 4.1

**Pre-training.** The observations and actions are normalized to $[0, 1]$ using min/max statistics from the pre-training dataset. For all three tasks the policy is trained for 3000 epochs with batch size 128, learning rate of 1e-3 decayed to 1e-4 with a cosine schedule, and weight decay of 1e-6. Exponential Moving Average (EMA) is applied with a decay rate of 0.995.

**Fine-tuning.** All methods from Section 4.1 use the same pre-trained policy. Fine-tuning is done using online experiences sampled from 40 parallelized MuJoCo environments [54]. Reward curves shown in Fig. 3 are evaluated by running fine-tuned policies with $\sigma^{\text{exp}}_{\text{min}} = 0.001$ (i.e., without extra noise) for 40 episodes. Each episode terminates if the default conditions are met or the episode reaches 1000 timesteps. Detailed hyperparameters are listed in Table 7 and Table 8.

| Task | | Obs dim - State | Obs dim - Pixel | Act dim | $T$ | Sparse reward ? |
|---|---|---|---|---|---|---|
| GYM | `Hopper-v2` | 11 | - | 3 | 1000 | No |
| | `Walker2D-v2` | 17 | - | 6 | 1000 | No |
| | `HalfCheetah-v2` | 17 | - | 6 | 1000 | No |
| ROBOMIMIC, state input | `Lift` | 19 | - | 7 | 300 | Yes |
| | `Can` | 23 | - | 7 | 300 | Yes |
| | `Square` | 23 | - | 7 | 400 | Yes |
| | `Transport` | 59 | - | 14 | 800 | Yes |
| ROBOMIMIC, pixel input | `Lift` | 9 | 96×96 | 7 | 300 | Yes |
| | `Can` | 9 | 96×96 | 7 | 300 | Yes |
| | `Square` | 9 | 96×96 | 7 | 400 | Yes |
| | `Transport` | 18 | 2×96×96 | 14 | 800 | Yes |
| FURNITURE-BENCH | `One-leg` | 58 | - | 10 | 700 | Yes |
| | `Lamp` | 44 | - | 10 | 1000 | Yes |
| | `Round-table` | 44 | - | 10 | 1000 | Yes |
| D3IL | `Avoid` | 4 | - | 2 | 100 | Yes |

Table 6: **Comparison of the different tasks considered.** "Obs dim - State": dimension of the state observation input. "Obs dim - State": dimension of the pixel observation input. "Act dim - State": dimension of the action space. $T$: maximum number of steps in an episode. "Sparse reward ?": whether sparse reward is used in training instead of dense reward.

### G.3 Descriptions of diffusion-based RL algorithm baselines in Section 4.1

**DRWR:** This is a **customized** reward-weighted regression (RWR) algorithm [20] that fine-tunes a pre-trained DP with a supervised objective with higher weights on actions that lead to higher reward-to-go $r$.

The reward is scaled with $\beta$ and the exponentiated weight is clipped at $w_{\max}$. The policy is updated with experiences collected with the current policy (no buffer for data from previous iteration) and a replay ratio of $N_\theta$. No critic is learned.

$$\mathcal{L}_\theta = \mathbb{E}^{\bar{\pi}_\theta, \varepsilon_t}\Big[\min(e^{\beta r_t}, w_{\max}) \|\varepsilon_t - \varepsilon_\theta(a_t^0, s_t, k)\|^2\Big].$$

**DAWR:** This is a **customized** advantage-weighted regression (AWR) algorithm [19] that builds on **DRWR** but uses TD-bootstrapped [9] advantage estimation instead of the higher-variance reward-to-go for better training stability and efficiency. **DAWR** (and **DRWR**) can be seen as approximately optimizing (3.2) with a Kullback–Leibler (KL) divergence constraint on the policy [19, 27].

The advantage is scaled with $\beta$ and the exponentiated weight is clipped at $w_{\max}$. Unlike **DRWR**, we follow [19] and trains the actor in an off-policy manner: recent experiences are saved in a replay buffer $\mathcal{D}$, and the actor is updated with a replay ratio of $N_\theta$.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}, \varepsilon_t}\big[\min(e^{\beta \hat{A}_\phi(s_t, a_t^0)}, w_{\max}) \|\varepsilon_t - \varepsilon_\theta(a_t^0, s_t, k)\|^2\big].$$

The critic is updated less frequently (we find diffusion models need many gradient updates to fit the actions) with a replay ratio of $N_\phi$.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|\hat{A}_\phi(s_t, a_t^0) - A(s_t, a_t^0)\|^2\right],$$

where $A$ is calculated using TD($\lambda$), with $\lambda$ as $\lambda_{\text{DAWR}}$ and the discount factor $\gamma_{\text{ENV}}$.

**`DIPO` [18]:** This baseline applies "action gradient" that uses a learned state-action Q function to update the actions saved in the replay buffer, and then has DP fitting on them without weighting.

Similar to **`DAWR`**, recent experiences are saved in a replay buffer $\mathcal{D}$. The actions ($k = 0$) in the buffer are updated for $M_{\text{DIPO}}$ iterations with learning rate $\alpha_{\text{DIPO}}$.

$$a_t^{m+1,k=0} = a_t^{m,k=0} + \alpha_{\text{DIPO}} \nabla_\phi \hat{Q}_\phi(s_t, a_t^{m,k=0}),\ m = 0, \ldots, M_{\text{DIPO}} - 1.$$

The actor is then updated with a replay ratio of $N_\theta$.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[\|\varepsilon_t - \varepsilon_\theta(a_t^{M_{\text{DIPO}},k=0}, s_t, k)\|^2\right].$$

The critic is trained to minimize the Bellman residual with a replay ratio of $N_\phi$. Double Q-learning is also applied.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|(R_t + \gamma_{\text{ENV}} \hat{Q}_\phi(s_{t+1}, \bar{\pi}_\theta(a_{t+1}^{k=0}|s_{t+1})) - \hat{Q}_\phi(s_t, a_t^{m=0,k=0})\|^2\right]$$

**`IDQL` [17]:** This baseline learns a state-action Q function and state V function to choose among the sampled actions from DP. DP fits on new samples without weighting.

Again recent experiences are saved in a replay buffer $\mathcal{D}$. The state value function is updated to match the expected Q value with an expectile loss, with a replay ratio of $N_\psi$.

$$\mathcal{L}_\psi = \mathbb{E}^{\mathcal{D}}\left[|\tau_{\text{IDQL}} - \mathbb{1}(\hat{Q}_\phi(s_t, a_t^0) < \hat{V}_\psi^2(s_t))|\right].$$

The value function is used to update the Q function with a replay ratio of $N_\phi$.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|(R_t + \gamma_{\text{ENV}} \hat{V}_\psi(s_{t+1}) - \hat{Q}_\phi(s_t, a_t^0)\|^2\right].$$

The actor fits all sampled experiences without weighting, with a replay ratio of $N_\theta$.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[\|\varepsilon_t - \varepsilon_\theta(a_t^0, s_t, k)\|^2\right].$$

At inference time, $M_{\text{IDQL}}$ actions are sampled from the actor. For training, Boltzmann exploration is applied based on the difference between $Q$ value of the sampled actions and and the $V$ value at the current state. For evaluation, the greedy action under $Q$ is chosen.

**`DQL` [16]:** This baseline learns a state-action Q function and backpropagates the gradient from the critic through the entire actor (with multiple denoising steps), akin to the usual Q-learning.

Again recent experiences are saved in a replay buffer $\mathcal{D}$. The actor is then updated using both a supervised loss and the value loss with a replay ratio of $N_\theta$.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[\|\varepsilon_t - \varepsilon_\theta(a_t^0, s_t, k)\|^2 - \alpha_{\text{DQL}} \hat{Q}_\phi(s_t, \bar{\pi}_\theta(a_t^0|s_t))\right],$$

where $\alpha_{\text{DQL}}$ is a weighting coefficient. The critic is trained to minimize the Bellman residual with a replay ratio of $N_\phi$. Double Q-learning is also applied.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|(R_t + \gamma_{\text{ENV}} \hat{Q}_\phi(s_{t+1}, \bar{\pi}_\theta(a_{t+1}^0|s_{t+1})) - \hat{Q}_\phi(s_t, a_t^0)\|^2\right]$$

**`QSM` [15]:** This baselines learns a state-action Q function, and then updates the actor by aligning the score of the diffusion actor with the gradient of the Q function.

Again recent experiences are saved in a replay buffer $\mathcal{D}$. The critic is trained to minimize the Bellman residual with a replay ratio of $N_\phi$. Double Q-learning is also applied.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|(R_t + \gamma_{\text{ENV}} \hat{Q}_\phi(s_{t+1}, \bar{\pi}_\theta(a_{t+1}^0|s_{t+1})) - \hat{Q}_\phi(s_t, a_t^0)\|^2\right].$$

The actor is updated as follows with a replay ratio of $N_\theta$.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[\|\alpha_{\text{QSM}} \nabla_a \hat{Q}_\phi(s_t, a_t) - (-\varepsilon_\theta(a_t^0, s_t, k))\|^2\right],$$

where $\alpha_{\text{QSM}}$ scales the gradient. The negative sign before $\varepsilon_\theta$ is from taking the gradient of the mean $\mu$ in the denoising process.

### G.4 Descriptions of RL fine-tuning algorithm baselines in Appendix D.2

In this subsection, we detail the baselines **RLPD**, **Cal-QL**, and **IBRL**. All policies $\pi_\theta$ are parameterized as unimodal Gaussian with an action chunk size of 1.

**RLPD [22]:** This baseline is based on Soft Actor Critic (SAC, Haarnoja et al. [55]) — it learns an entropy-regularized state-action Q function, and then updates the actor by maximizing the Q function w.r.t. the action.

A replay buffer $\mathcal{D}$ is initialized with offline data, and online samples are added to $\mathcal{D}$. Each gradient update uses a batch of mixed 50/50 offline and online data. An ensemble of $N_{\text{critic}}$ critics is used, and at each gradient step two critics are randomly chosen. The critics are trained to minimize the Bellman residual with replay ratio $N_\phi$:

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\big[\|(R_t + \gamma_{\text{ENV}}\hat{Q}_{\phi'}(s_{t+1}, \pi_\theta(a_{t+1}|s_{t+1})) - \hat{Q}_\phi(s_t, a_t)\|^2\big].$$

The target critic parameter $\phi'$ is updated with delay. The actor minimizes the following loss with a replay ratio of 1:

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\big[-\hat{Q}_\phi(s_t, a_t) + \alpha_{\text{ent}} \log \pi_\theta(a_t|s_t)\big],$$

where $\alpha_{\text{ent}}$ is the entropy coefficient (automatically tuned as in SAC starting at 1).

**Cal-QL [23]:** This baseline trains the policy $\mu$ and the action-value function $Q^\mu$ in an offline phase and then an online phase. During offline phase only offline data is sampled for gradient update, while during online phase mixed 50/50 offline and online data are sampled. The critic is trained to minimize the following loss (Bellman residual and calibrated Q-learning):

$$\begin{aligned}\mathcal{L}_\phi =& \mathbb{E}^{\mathcal{D}}\big[\|(R_t + \gamma_{\text{ENV}}\hat{Q}_{\phi'}(s_{t+1}, \pi_\theta(a_{t+1}|s_{t+1}))) - \hat{Q}_\phi(s_t, a_t)\|^2\big] \\ &+ \beta_{\text{cql}}(\mathbb{E}^{\mathcal{D}}\big[\max(Q_\phi(s_t, a_t), V(s_t))\big] - \mathbb{E}^{\mathcal{D}}\big[Q_\phi(s_t, a_t)\big]),\end{aligned}$$

where $\beta_{\text{cql}}$ is a weighting coefficient between Bellman residual and calibration Q-learning and $V(s_t)$ is estimated using Monte-Carlo returns. The target critic parameter $\phi'$ is updated with delay. The actor minimizes the following loss:

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\big[-\hat{Q}_\phi(s_t, a_t) + \alpha_{\text{ent}} \log \pi_\theta(a_t|s_t)\big],$$

where $\alpha_{\text{ent}}$ is the entropy coefficient (automatically tuned as in SAC starting at 1).

**IBRL [24]:** This baseline first pre-trains a policy $\mu_\psi$ using behavior cloning, and for fine-tuning it trains a RL policy $\pi_\theta$ initialized as $\mu_\psi$. During fine-tuning recent experiences are saved in a replay buffer $\mathcal{D}$. An ensemble of $N_{\text{critic}}$ critics is used, and at each gradient step two critics are randomly chosen. The critics are trained to minimize the Bellman residual with replay ratio $N_\phi$:

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\big[\|(R_t + \gamma_{\text{ENV}} \max_{a' \in \{a^{IL}, a^{RL}\}} \hat{Q}_{\phi'}(s_{t+1}, a') - \hat{Q}_\phi(s_t, a_t)\|^2\big]$$

where $a^{IL} = \mu_\psi(s_{t+1})$ (no noise) and $a^{RL} \sim \pi_{\theta'}(s_{t+1})$, and $\pi_{\theta'}$ is the target actor. The target critic parameter $\phi'$ is updated with delay. The actor minimizes the following loss with a replay ratio of 1:

$$\mathcal{L}_\theta = -\mathbb{E}^{\mathcal{D}}\big[\hat{Q}_\phi(s_t, a_t)\big].$$

The target actor parameter $\theta'$ is also updated with delay.

### G.5 Additional details of DPPO implementation in all tasks

Similar to all baselines in Appendix G.3, we denote $N_\theta$ and $N_\phi$ the replay ratio for the actor (Diffusion Policy) and the critic (state value function) in **DPPO**; in practice we always set $N_\theta = N_\phi$ in **DPPO**, with the combined loss $\mathcal{L} = \mathcal{L}_\theta + \mathcal{L}_\phi$. Similar to usual PPO implementations [56], the batch updates in an iteration terminate when the KL divergence between $\pi_\theta$ and $\pi_{\theta_{\text{old}}}$ reaches 1.

We also find the PPO clipping ratio, $\varepsilon$, can affect the training stability significantly in **DPPO** (as well as in Gaussian and GMM policies) especially in sparse-reward manipulation tasks. In practice we find that, a good indicator of the amount of clipping leading to optimal training efficiency, is to aim for a clipping fraction (fraction of individual samples being clipped in a batch) of 10% to 20%. For each method in different tasks, we vary $\varepsilon$ in $\{.1, .01, .001\}$ and choose the highest value that satisfies the clipping fraction target. Empirically we also find that, using a higher $\varepsilon$ for earlier denoising steps in **DPPO** further improves training stability in manipulation tasks. Denote $\varepsilon_k$ the clipping value at denoising step $k$, and in practice we set $\varepsilon_{k=(K-1)} = 0.1\varepsilon_{k=0}$, and it follows an exponential schedule among intermediate $k$.

### G.6 Additional details of Robomimic tasks and training in Section 4.2

**Tasks.** We consider four tasks from the Robomimic benchmark [29]: (1) `Lift`: lifting a cube from the table, (2) `Can`: picking up a Coke can and placing it at a target bin, (3) `Square`: picking up a square nut and place it on a rod, and (4) `Transport`: two robot arms removing a bin cover, picking and placing a cube, and then transferring a hammer from one container to another one.

**Pre-training.** Robomimic provides the Multi-Human (MH) dataset with noisy human demonstrations for each task, which we use to pre-train the policies. The observations and actions are normalized to $[0, 1]$ using min/max statistics from the pre-training dataset. No history observation (pixel, proprioception, or ground-truth object states) is used. All policies are trained with batch size 128, learning rate 1e-4 decayed to 1e-5 with a cosine schedule, and weight decay 1e-6. Diffusion-based policies are trained with 8000 epochs, while Gaussian and GMM policies are trained with 5000 epochs — we find diffusion models require more gradient updates to fit the data well.

**Fine-tuning.** Diffusion-based, Gaussian, and GMM pre-trained policies are then fine-tuned using online experiences sampled from 50 parallelized MuJoCo environments [54]. Success rate curves shown in Fig. 3, Fig. 4, and Fig. 15 are evaluated by running fine-tuned policies with $\sigma^{\text{exp}}_{\text{min}} = 0.001$ (i.e., without extra noise) for 50 episodes. Episodes terminates only when they reach maximum episode lengths (shown in Table 6). Detailed hyperparameters are listed in Table 10.

**Pixel training.** We use the wrist camera view in `Lift` and `Can`, the third-person camera view in `Square`, and the two robot shoulder camera views in `Transport`. Random-shift data augmentation is applied to the camera images during both pre-training and fine-tuning. Gradient accumulation is used in fine-tuning so that the same batch size (as in state-input training) can fit on the GPU. Detailed hyperparameters are listed in Table 11.

### G.7 Descriptions of policy parameterization baselines in Section 4.2

**Gaussian.** We consider unimodal Gaussian with diagonal covariance, the most commonly used policy parameterization in RL. The standard deviation for each action dimension, $\sigma_{\text{Gau}}$, is fixed during pre-training; we also tried to learn $\sigma_{\text{Gau}}$ from the dataset but we find the training very unstable. During fine-tuning $\sigma_{\text{Gau}}$ is learned starting from the same fixed value and also clipped between $0.01$ and $0.2$. Additionally we clip the sampled action to be within 3 standard deviation from the mean. As discusses in Appendix G.5, we choose the PPO clipping ratio $\varepsilon$ based on the empirical clipping fraction in each task. This setup is also used in the Furniture-Bench experiments. We note that we spend significant amount of efforts tuning the Gaussian baseline, and our results with it are some of the best known ones in RL training for long-horizon manipulation tasks (exceeding our initial expectations), e.g., reaching $\sim$100% success rate in `Lamp` with `Low` randomness.

**GMM.** We also consider Gaussian Mixture Model as the policy parameterization. We denote $M_{\text{GMM}}$ the number of mixtures. The standard deviation for each action dimension in each mixture, $\sigma_{\text{GMM}}$, is also fixed during pre-training. Again during fine-tuning $\sigma_{\text{GMM}}$ is learned starting from the same fixed value and also clipped between $0.01$ and $0.2$.

### G.8 Additional details of FURNITURE-BENCH tasks and training in Section 4.3

**Tasks.** We consider three tasks from the FURNITURE-BENCH benchmark [31]: (1) `One-leg`: assemble one leg of a table by placing the tabletop in the fixture corner, grasping and inserting the table leg, and screwing in the leg, (2) `Lamp`: place the lamp base in the fixture corner, grasp, insert, and screw in the light bulb, and finally place the lamp shade, (3) `Round-table`: place a round tabletop in the fixture corner, insert and screw in the table leg, and then insert and screw in the table base. See Fig. 21 for the visualized rollouts in simulation.

**Pre-training.** The pre-training dataset is collected in the simulated environments using a SpaceMouse[5], a 6 DoF input device. The simulator runs at 10Hz. At every timestep, we read off the state of the SpaceMouse as $\delta \mathbf{a} = [\Delta x, \Delta y, \Delta z, \Delta\text{roll}, \Delta\text{pitch}, \Delta\text{yaw}]$, which is converted to a quaternion before passed to the environment step and stored as the action alongside the current observation in the trajectory. If $|\Delta \mathbf{a}_i| < \varepsilon \; \forall i$ for some small $\varepsilon = 0.05$ defining the threshold for a no-op, we do not record any action nor pass it to the environment. Discarding no-ops is important for allowing the policies to learn from demonstrations effectively. When the desired number of demonstrations has been collected (typically 50), we process the actions to convert the delta actions stored from the SpaceMouse into absolute pose actions by applying the delta action to the current EE pose at each timestep.

The observations and actions are normalized to $[-1, 1]$ using min/max statistics from the pre-training dataset. No history observation (proprioception or ground-truth object states) is used, i.e., only the current observation is passed to the policy. All policies are trained with batch size 256, learning rate 1e-4 decayed to 1e-5 with a cosine schedule, and weight decay 1e-6. Diffusion-based policies are trained with 8000 epochs, while Gaussian policies are trained with 3000 epochs. Gaussian policies can easily overfit the pre-trained dataset, while diffusion-based policies are more resilient. Gaussian policies also require a very large MLP ($\sim$10 million parameters) to fit the data well.

**Fine-tuning.** Diffusion-based and Gaussian pre-trained policies are then fine-tuned using online experiences sampled from 1000 parallelized IsaacGym environments [57]. Success rate curves shown in Fig. 5 are evaluated by running fine-tuned policies with $\sigma^{\text{exp}}_{\text{min}} = 0.001$ (i.e., without extra noise) for 1000 episodes. Episodes terminate only when they reach maximum episode length (shown in Table 6). Detailed hyperparameters are listed in Table 12. We find a smaller amount of exploration noise (we set $\sigma^{\text{exp}}_{\text{min}}$ and $\sigma_{\text{Gau}}$ to be 0.04) is necessary for the pre-trained policy achieving nonzero success rates at the beginning of fine-tuning.

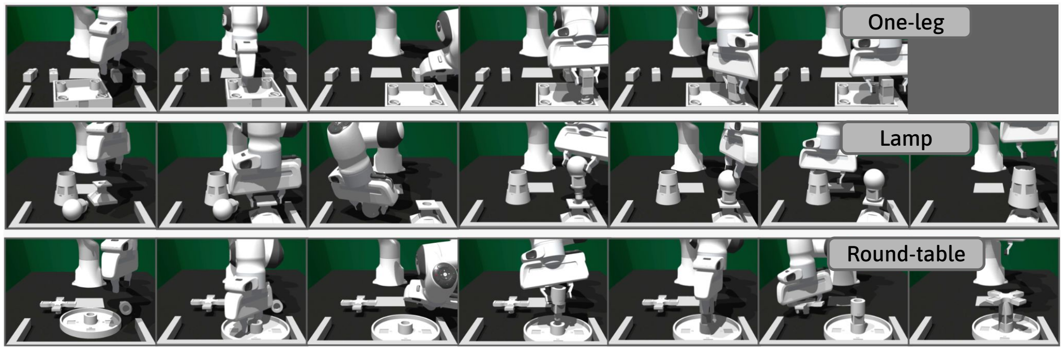


Figure 21: Representative rollouts from simulated FURNITURE-BENCH tasks.

**Hardware setup - robot control.** The physical robot used is a Franka Emika Panda arm. The policies output a sequence of desired end-effector poses in the robot base frame to control the robot. These poses are converted into joint position targets through differential inverse kinematics. We

[5] https://3dconnexion.com/us/product/spacemouse-wireless/

calculate the desired end-effector velocity as the difference between the desired and current poses divided by the delta time $dt = 1/10$. We then convert this to desired joint velocities using the Jacobian and compute the desired joint positions with a first-order integration over the current joint positions and desired velocity. The resulting joint position targets are passed to a low-level joint impedance controller provided by Polymetis [58], running at 1kHz.

**Hardware setup - state estimation.** To deploy state-based policies on real hardware, we utilize AprilTags [59] for part pose estimation. The FURNITURE-BENCH [31] task suite provides AprilTags for each part and code for estimating part poses from tag detections. The process involves several steps: (1) detecting tags in the camera frame, (2) mapping tag detections to the robot frame for policy compatibility, (3) utilizing known offsets between tags and object centers in the simulator, and (4) calibrating the camera pose using an AprilTag at a known position relative to the robot base. Despite general accuracy, detections can be noisy, especially during movement or partial occlusion, which the `One-leg` task features. Since the task requires high precision, we find the following to help make the estimation reliable enough:

- **Camera coverage:** We find detection quality sensitive to distance and angle between the camera and tag. This issue is likely due to the RealSense D435 camera having mediocre image quality and clarity and the relatively small tags. To remedy this, we opt to use 4 cameras roughly evenly spread out around the scene to ensure that at least one camera has a solid view of a tag on all the parts (i.e., as close as possible with a straight-on view). To find the best camera positions, we start with having a camera in each of the cardinal directions around the scene. Then, we adjust the pose of each to get it as close as possible to the objects while still covering the necessary workspace and capturing the base tag for calibration. Moving the robot arm around the scene to avoid the worst occlusion is also helpful.
- **Lighting:** Even with better camera coverage and placement, detection quality depends on having crisp images. We find proper lighting helpful to improve image quality. In particular, the scene should be well and evenly lit around the scene without causing reflections in either the tag or table.
- **Filtering:** Bad detections can sometimes cause the resulting pose estimate to deviate significantly from the true pose, i.e., jumping several centimeters from one frame to the next. This usually only happens on isolated frames, and thus before "accepting" a given detection, we check if the new position and orientation are within 5 cm and 20 degrees of the previously accepted pose. In addition, we apply low-pass filtering on the detection using a simple exponential average (with $\alpha = 0.25$) to smooth out the high-frequency noise.
- **Averaging:** The objects have multiple tags that can be detected from multiple cameras. After performing the filtering step, we average all pose estimates for the same object across different tags and cameras, which also helps smooth out noise. This alone, however, does not fully cancel the case when a single detection has a large jump, as this can severely skew the average, still necessitating a filtering step. Having multiple cameras benefits this step, too, as it provides more detections to average over.
- **Caching part pose in hand:** A particularly difficult phase of the task to achieve good detections is when the robot transports the table leg from the initial position to the tabletop for insertion. The main problems are that the movement can blur the images, and the grasping can cause occlusions. Therefore, we found it helpful to assume that once the part was grasped by the robot, it would not move in the grasp until the gripper opened. With this, we can "cache" the pose of the part relative to the end-effector once the object is fully grasped and use this instead of relying on detections during the movement.
- **Normalization pitfalls and clipping:** We generally use min-max normalization of the state observations to ensure observations are in $[-1, 1]$. The tabletop part moves very little in the $z$-direction demonstration data, meaning the resulting normalization limits (the minimum and maximum value of the data) can be very close, $x_{\text{max}} - x_{\text{min}} \approx 0$. With these tight limits, the noise in the real-world detection can be amplified greatly as $x_{\text{norm}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$. Therefore, ensure

that normalization ranges are reasonable. As an extra safeguard, clipping the data to $[-1, 1]$ can also help.

- **Only estimate necessary states:** Despite the `One-leg` task having 5 parts, only 2 are manipulated. Only estimating the pose of those parts can eliminate a lot of noise. In particular, the pose of the 3 legs that are not used and the obstacle (the U-shaped fixture) can be set to an arbitrary value from the dataset.
- **Visualization for debugging:** We use the visualization tool MeshCat[6] extensively for debugging of state estimation. The tool allows for easy visualizations of poses of all relevant objects in the scene, like the robot end-effector and parts, which makes sanity-checking the implementation far easier than looking at raw numbers.

**Hardware evaluation.** We perform 20 trials for each method. We adopt a single-blind model selection process: at the beginning of each trial, we first randomize the initial state. Then, we randomly select a method and roll it out, but the experimenter does not observe which model is used. We record the success and failure of each trial and then aggregate statistics for each model after all trials are completed.

**Domain randomization for sim-to-real transfer.** To facilitate the sim-to-real transfer, we apply additional domain randomization to the simulation training. We record the range of observation noises in hardware without any robot motion and then apply the same amount of noise to state observations in simulation. We find the state estimation in hardware particularly sensitive to the object heights. Also, we apply random noise (zero mean with 0.03 standard deviation) to the sampled action from **DPPO** to simulate the imperfect low-level controller; we find adding such noise to the Gaussian policy leads to zero task success rate while **DPPO** is robust to it (also see discussion in Appendix F).

**BC regularization loss used for Gaussian baseline.** Since the fine-tuned Gaussian policy exhibits very jittery behavior and leads to zero success rate in real evaluation, we further experiment with adding a behavior cloning (BC) regularization loss in fine-tuning with the Gaussian baseline. The combined loss follows

$$\mathcal{L}_{\theta,+\text{BC}} = \mathcal{L}_\theta - \alpha_{\text{BC}} \mathbb{E}^{\pi_{\theta_{\text{old}}}} \left[ \sum_{k=0}^{K-1} \log \pi_{\theta_{\text{pre-trained}}}(a_t^k | a_t^{k+1}, s_t) \right],$$

where $\pi_{\theta_{\text{pre-trained}}}$ is the frozen BC-only policy. The extra term encourages the newly sampled actions from the fine-tuned policy to remain high-likelihood under the BC-only policy. We set $\alpha_{\text{BC}} = 0.1$. However, although this regularization reduces the sim-to-real gap, it also significantly limits fine-tuning, leading to the fine-tuning policy saturating at 53% success rate shown in Fig. 5.

### G.9 Additional details of `Avoid` task from D3IL and training in Appendix F

**Pre-training.** We split the original dataset from D3IL based on the three settings, M1, M2, and M3; in each setting, observations and actions are normalized to $[0, 1]$ using min/max statistics. All policies are trained with batch size 16 (due to the small dataset size), learning rate 1e-4 decayed to 1e-5 with a cosine schedule, and weight decay 1e-6. Diffusion-based policies are trained with about 15000 epochs, while Gaussian and GMM policies are trained with about 10000 epochs; we manually examine the trajectories from different pre-trained checkpoints and pick ones that visually match the expert data the best.

**Fine-tuning.** Diffusion-based, Gaussian, and GMM pre-trained policies are then fine-tuned using online experiences sampled from 50 parallelized MuJoCo environments [54]. Reward curves shown in Fig. 18 and Fig. 14 are evaluated by running fine-tuned policies with the same amount of exploration noise used in training for 50 episodes; we choose to use the training (instead of evaluation)

[6] https://github.com/meshcat-dev/meshcat

setup since Gaussian policies exhibit multi-modality only with training noise. Episodes terminate only when they reach 100 steps.

**Added action noise during fine-tuning.** In Fig. 18 left, we demonstrate that **DPPO** exhibits stronger training stability when noise is added to the sampled actions during fine-tuning. The noise starts at the 5th iteration. It is sampled from a uniform distribution with the lower limit ramping up to 0.1 and the upper limit ramping up to 0.2 linearly in 5 iterations. The limits are kept the same from the 10th iteration to the end of fine-tuning.

## G.10 Listed training hyperparameters

| | | Task(s) | | | |
|---|---|---|---|---|---|
| Method | Parameter | Gym | `Lift, Can` | `Square` | `Transport` |
| Common | $\gamma_{\text{ENV}}$ | 0.99 | 0.999 | 0.999 | 0.999 |
| | $\sigma^{\text{exp}}_{\text{min}}$ | 0.1 | 0.1 | 0.1 | 0.08 |
| | $\sigma^{\text{prob}}_{\text{min}}$ | 0.1 | | | |
| | $T_a$ | 4 | 4 | 4 | 8 |
| | $K$ | 20 | | | |
| | Actor learning rate | 1e-4 | 1e-5 | 1e-5 | 1e-5 (decayed to 1e-6) |
| | Critic learning rate (if applies) | 1e-3 | | | |
| | Actor MLP dims | [512, 512, 512] | [512, 512, 512] | [1024, 1024, 1024] | [1024, 1024, 1024] |
| | Critic MLP dims (if applies) | [256, 256, 256] | | | |
| `DRWR` | $\beta$ | 10 | | | |
| | $w_{\max}$ | 100 | | | |
| | $N_\theta$ | 16 | | | |
| | Batch size | 1000 | | | |
| `DAWR` | $\beta$ | 10 | | | |
| | $w_{\max}$ | 100 | | | |
| | $\lambda_{\text{DAWR}}$ | 0.95 | | | |
| | $N_\theta$ | 64 | | | |
| | $N_\phi$ | 16 | 4 | 4 | 4 |
| | Buffer size | 200000 | 150000 | 150000 | 150000 |
| | Batch size | 256 | | | |
| `DIPO` | $\alpha_{\text{DIPO}}$ | 1e-4 | | | |
| | $M_{\text{DIPO}}$ | 10 | | | |
| | $N_\theta$ | 64 | | | |
| | Buffer size | 400000 | | | |
| | Batch size | 5000 | | | |
| `IDQL` | $M_{\text{IDQL}}$ | 20 | 10 | 10 | 10 |
| | $N_\theta$ | 16 | | | |
| | $N_\phi$ | 16 | | | |
| | Buffer size | 200000 | 150000 | 150000 | 150000 |
| | Batch size | 256 | 512 | 512 | 512 |
| `DQL` | $\alpha_{\text{DQL}}$ | 1 | | | |
| | $N_\theta$ | 64 | | | |
| | $N_\phi$ | 64 | | | |
| | Buffer eize | 400000 | | | |
| | Batch size | 5000 | | | |

Table 7: Fine-tuning hyperparameters for OpenAI Gym and Robomimic tasks when **comparing diffusion-based RL methods**. We list hyperparameters shared by all methods first, and then method-specific ones.

| | | Task(s) (cont'd) | | | |
|---|---|---|---|---|---|
| Method (cont'd) | Parameter (cont'd) | Gym | `Lift, Can` | `Square` | `Transport` |
| `QSM` | $\alpha_{\text{QSM}}$ | 50 | | | |
| | $N_\theta$ | 32 | | | |
| | $N_\phi$ | 32 | | | |
| | Buffer size | 200000 | 150000 | 150000 | 150000 |
| | Batch size | 5000 | | | |
| `DPPO` | $\gamma_{\text{DENOISE}}$ | 0.99 | | | |
| | GAE $\lambda$ | 0.95 | | | |
| | $N_\theta$ | 5 | 10 | 10 | 8 |
| | $N_\phi$ | 5 | 10 | 10 | 8 |
| | $\varepsilon$ | 0.01 | | | |
| | Batch size | 5000 | 7500 | 10000 | 10000 |
| | $K'$ | 10 | | | |

Table 8: Continuation of Table 7.

|  |  | Task(s) |  |  |
|---|---|---|---|---|
| Method | Parameter | `HalfCheetah-v2` | `Can` | `Square` |
| Common | $\gamma_{\text{ENV}}$ | 0.99 | 0.999 | 0.999 |
|  | $T_a$ |  | 1 |  |
| **RLPD** | $N_\phi$ | 20 | 3 | 3 |
|  | $N_{\text{critic}}$ | 10 | 5 | 5 |
|  | Batch size |  | 256 |  |
| **Cal-QL** | $\beta_{\text{cql}}$ |  | 5 |  |
|  | Batch size |  | 256 |  |
| **IBRL** | $N_\phi$ | 5 | 3 | 3 |
|  | $N_{\text{critic}}$ |  | 5 |  |
|  | Batch size |  | 256 |  |
| **DPPO** | $\sigma^{\text{exp}}_{\text{min}}$ |  | 0.1 |  |
|  | $\sigma^{\text{prob}}_{\text{min}}$ |  | 0.1 |  |
|  | $\gamma_{\text{DENOISE}}$ |  | 0.99 |  |
|  | GAE $\lambda$ |  | 0.95 |  |
|  | $N_\theta$ | 5 | 10 | 10 |
|  | $N_\phi$ | 5 | 10 | 10 |
|  | $\varepsilon$ |  | 0.01 |  |
|  | Batch size | 5000 | 7500 | 10000 |
|  | $K$ |  | 20 |  |
|  | $K'$ |  | 10 |  |

Table 9: Fine-tuning hyperparameters for `HalfCheetah-v2`, `Can`, and `Square` when **comparing demo-augmented RL methods**. We list hyperparameters shared by all methods first, and then method-specific ones.

|  |  | Task |  |  |
|---|---|---|---|---|
| Method | Parameter | `Lift`, `Can` | `Square` | `Transport` |
| Common | $\gamma_{\text{ENV}}$ |  | 0.999 |  |
|  | $T_a$ | 4 | 4 | 8 |
|  | Actor learning rate | 1e-4 | 1e-5 | 1e-5 (decayed to 1e-6) |
|  | Critic learning rate |  | 1e-3 |  |
|  | GAE $\lambda$ |  | 0.95 |  |
|  | $N_\theta$ | 10 | 10 | 8 |
|  | $N_\phi$ | 10 | 10 | 8 |
|  | $\varepsilon$ |  | 0.01 (annealed in **DPPO**) |  |
|  | Batch size | 7500 | 10000 | 10000 |
| Gaussian, Common | $\sigma_{\text{Gau}}$ | 0.1 | 0.1 | 0.08 |
| Gaussian-MLP | Model size | 552K | 2.15M | 1.93M |
| Gaussian-Transformer | Model size | 675K | 1.86M | 1.87M |
| GMM, Common | $M_{\text{GMM}}$ |  | 5 |  |
|  | $\sigma_{\text{GMM}}$ | 0.1 | 0.1 | 0.08 |
| GMM-MLP | Model size | 1.15M | 4.40M | 4.90M |
| GMM-Transformer | Model size | 680K | 1.87M | 1.89M |
| **DPPO**, Common | $\gamma_{\text{DENOISE}}$ |  | 0.99 |  |
|  | $\sigma^{\text{exp}}_{\text{min}}$ | 0.1 | 0.1 | 0.08 |
|  | $\sigma^{\text{prob}}_{\text{min}}$ | 0.1 | 0.1 | 0.1 |
|  | $K$ |  | 20 |  |
|  | $K'$ |  | 10 |  |
| **DPPO**-MLP | Model size | 576K | 2.31M | 2.43M |
| **DPPO**-UNet | Model size | 652K | 1.62M | 1.68M |

Table 10: Fine-tuning hyperparameters for ROBOMIMIC tasks with **state** input when **comparing policy parameterizations**. We list hyperparameters shared by all methods first, and then method-specific ones. Since the different policy parameterizations use different neural network architecture, we list the total model size here instead of the details such as MLP dimensions.

| Method | Parameter | Task | | |
|---|---|---|---|---|
| | | `Lift,Can` | `Square` | `Transport` |
| Common | $\gamma_{\text{ENV}}$ | 0.999 | | |
| | $T_a$ | 4 | 4 | 8 |
| | Actor learning rate | 1e-4 | 1e-5 | 1e-5 (decayed to 1e-6) |
| | Critic learning rate | 1e-3 | | |
| | GAE $\lambda$ | 0.95 | | |
| | $N_\theta$ | 10 | 10 | 8 |
| | $N_\phi$ | 10 | 10 | 8 |
| | $\varepsilon$ | 0.01 (annealed in **DPPO**) | | |
| | Batch size | 7500 | 10000 | 10000 |
| Gaussian-ViT-MLP | Model size | 1.03M | 1.03M | 1.93M |
| | $\sigma_{\text{Gau}}$ | 0.1 | 0.1 | 0.08 |
| **DPPO**-ViT-MLP | Model size | 1.06M | 1.06M | 2.05M |
| | $\gamma_{\text{DENOISE}}$ | 0.9 | | |
| | $\sigma^{\text{exp}}_{\text{min}}$ | 0.1 | 0.1 | 0.08 |
| | $\sigma^{\text{prob}}_{\text{min}}$ | 0.10 | | |
| | $K$ | 100 | | |
| | $K'$ | 5 (DDIM) | | |

Table 11: Fine-tuning hyperparameters for ROBOMIMIC tasks with **pixel** input when **comparing policy parameterizations**. We list hyperparameters shared by all methods first, and then method-specific ones. Since the different policy parameterizations use different neural network architecture, we list the total model size here instead of the details such as MLP dimensions.

| Method | Parameter | Task | | |
|---|---|---|---|---|
| | | `One-leg` | `Lamp` | `Round-table` |
| Common | $\gamma_{\text{ENV}}$ | 0.999 | | |
| | $T_a$ | 8 | | |
| | Actor learning rate | 1e-5 (decayed to 1e-6) | | |
| | Critic learning rate | 1e-3 | | |
| | GAE $\lambda$ | 0.95 | | |
| | $N_\theta$ | 5 | | |
| | $N_\phi$ | 5 | | |
| | $\varepsilon$ | 0.001 | | |
| | Batch size | 8800 | | |
| Gaussian-MLP | Model size | 10.64M | 10.62M | 10.62M |
| | $\sigma_{\text{Gau}}$ | 0.04 | | |
| **DPPO**-UNet | Model size | 6.86M | 6.81M | 6.81M |
| | $\gamma_{\text{DENOISE}}$ | 0.9 | | |
| | $\sigma^{\text{exp}}_{\text{min}}$ | 0.04 | | |
| | $\sigma^{\text{prob}}_{\text{min}}$ | 0.1 | | |
| | $K$ | 100 | | |
| | $K'$ | 5 (DDIM) | | |

Table 12: Fine-tuning hyperparameters for FURNITURE-BENCH tasks when **comparing policy parameterizations**. We list hyperparameters shared by all methods first, and then method-specific ones.