Report of Paper Diffusion Policy Policy Optimization

XINYU DU

The report should include:

(1) A literature review of potential methods for solving the problem.
(2) Propose your choice of methods and the reason for the choice.
(3) Present the answers of the question in clear and understandable language.
(4) Testing plans and testing results for checking your implementation is correct and your results are valid.

1 A literature review of potential methods for solving the problem.

Descriptions of Diffusion-based RL Algorithm Baselines

DRWR[22]. This is a customized reward-weighted regression (RWR) algorithm Peters and Schaal (2007) 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 β 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_θ. No critic is learned.

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

DAWR[19]. This is a customized advantage-weighted regression (AWR) algorithm Peng et al. (2019) that builds on DRWR but uses TD-bootstrapped Sutton and Barto (2018) 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 (4.2) with a Kullback-Leibler (KL) divergence constraint on the policy Peng et al. (2019); Black et al. (2023)[2].

The advantage is scaled with β and the exponentiated weight is clipped at $w_{\max}$. Unlike DRWR, the original paper follows (Peng et al., 2019) 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_θ.

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

The critic is updated less frequently (the authors of DPPO finds diffusion models need many gradient updates to fit the actions) with a replay ratio of N_ϕ.

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

where A is calculated using TD(λ), with λ as λ_{DAWR} and the discount factor γ_{ENV} .

DIPO[32] . DIPO 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_{DIPO} iterations with learning rate α_{DIPO} .

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

The actor is then updated with a replay ratio of N_θ.

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

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

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

IDQL[9]. IDQL 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_ψ.

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

The value function is used to update the Q function with a replay ratio of N_ϕ.

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

The actor fits all sampled experiences without weighting, with a replay ratio of N_θ.

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

At inference time, M_{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[31]. 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_θ.

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

where α_{DQL} is a weighting coefficient. The critic is trained to minimize the Bellman residual with a replay ratio of N_ϕ. Double Q-learning is also applied.

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

QSM[23]. 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_ϕ. Double Q-learning is also applied.

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

The actor is updated as follows with a replay ratio of N_θ.

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

where α_{QSM} scales the gradient. The negative sign before ε_θ is from taking the gradient of the mean μ in the denoising process.

The author of DPPO claims that he corrected one error of QSM in his implementation. But limited by time, I followed the DPPO implementation of QSM and didn't verify if he is correct.

Descriptions of RL Fine-tuning Algorithm Baselines

In this subsection, I detail the baselines RLPD, Cal-QL, and IBRL. All policies π_θ are parameterized as unimodal Gaussian.

RLPD[1]. This baseline is based on Soft Actor Critic (SAC, Haarnoja et al. (2018)[7]) - 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_{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_ϕ :

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

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

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

where α_{ent} is the entropy coefficient (automatically tuned as in SAC starting at 1).

Cal-QL[17]. This baseline trains the policy μ and the action-value function Q^μ in an offline phase and then an online phase. During the offline phase only offline data is sampled for gradient update, while during the 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):

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

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

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

where α_{ent} is the entropy coefficient (automatically tuned as in SAC starting at 1).

IBRL[12]. This baseline first pre-trains a policy μ_ψ using behavior cloning, and for fine-tuning it trains a RL policy π_θ initialized as μ_ψ. During fine-tuning recent experiences are saved in a replay buffer $\mathcal{D}$. An ensemble of N_{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_ϕ :

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

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

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

The target actor parameter θ' is also updated with delay.

Other Related Works.

DACER[30]. The DACER algorithm follows the structure of the Soft Actor-Critic (SAC) framework, which includes both an actor and a critic. The critic learns a state-action value function, similar to Q-learning, by minimizing the Bellman residual. The critic's loss function is computed as follows:

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

where $\mathcal{D}$ represents the experience replay buffer, and $\hat{Q}\phi$ is the state-action value function learned by the critic. The replay ratio, denoted Mϕ, controls the proportion of updates from the replay buffer.

On the actor side, DACER adopts a diffusion-based approach for action generation, which is iteratively refined via a denoising process. The actor's loss function is updated by aligning the gradient of the Q-function with the diffusion score:

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

where α_{DACER} scales the gradient, and ε_θ represents the noise term in the denoising process. The negative sign before ε_θ comes from the gradient of the mean μ in the diffusion process, effectively regularizing exploration.

Additionally, in DACER, the entropy is estimated using a Gaussian Mixture Model (GMM), and the resulting entropy is used to update the scaling factor α_{DACER}. This process helps to adjust the strength of the gradient, further aiding in exploration and exploitation balance. The GMM is employed to approximate the entropy of the policy, allowing the algorithm to adaptively modify the exploration strategy based on the current state of the learned distribution.

SDAC[16]. The algorithm follows the actor-critic framework, where the critic estimates the state-action value function and the actor is updated to maximize expected rewards. A key aspect of the method is the incorporation of diffusion models for action generation and entropy-based regularization to enhance learning stability.

The critic learns the state-action value function by minimizing the Bellman residual, similar to standard actor-critic methods. The critic's loss function is:

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

where $\mathcal{D}$ is the experience replay buffer, R_t is the reward at time t, and γ_{ENV} is the environment's discount factor.

The actor is updated to maximize the expected reward based on the current policy. The loss function for the actor is computed by aligning the Q-function gradient with the policy's score. This is represented by:

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

where α_{SPECIFIC} is a scaling factor that adjusts the strength of the gradient, and ε_θ represents the noise term in the process. The negative sign before ε_θ comes from the gradient of the mean μ in the exploration process.

A key component of the algorithm is the regularization process. This is used to enhance exploration and regularize the policy, allowing the actor to balance exploration and exploitation effectively. The update for (the exploration component) is given by:

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

where f_θ represents the regularization term, which may be related to a specific exploration strategy (e.g., an entropy term, noise injection, or another mechanism). This ensures that the policy is sufficiently diversified and avoids premature convergence to suboptimal solutions.

Train Once, Get a Family[29]. The algorithm proposed in this paper focuses on offline-to-online reinforcement learning (RL), where an agent first learns from an offline dataset and then fine-tunes its policy using online interactions with the environment. The core idea is to leverage offline data to initialize the policy and then gradually adapt it during online learning while avoiding overfitting to the offline dataset.

The offline phase involves training the agent on a fixed dataset, $\mathcal{D}_{\text{offline}}$, using a standard RL method. The value function and policy are learned by minimizing the Bellman residual and policy loss:

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

where $\hat{\mathrm{V}}_\phi(s)$ represents the value function, and R_t is the reward at time step t. The offline training is performed using the dataset $\mathcal{D}_{\text{offline}}$, ensuring that the model learns from historical data without interacting with the environment.

Once pretraining is completed, the policy is fine-tuned using online interactions. The online update is designed to correct for any potential bias introduced during the offline phase, with the policy and value function updated as follows:

$$\mathcal{L}_{\text{online}} = \mathbb{E}^{\mathcal{D}_{\text{online}}} \left[\left(R_t + \gamma_{\text{ENV}} \hat{V}_\phi \left(s_{t+1} \right) - \hat{V}_\phi \left(s_t \right) \right)^2 + \lambda \left\| \nabla_\theta \hat{\pi}_\theta \left(a_t \mid s_t \right) \right\|^2 \right]$$

where $\mathcal{D}_{\text{online}}$ represents the online experience buffer, $\hat{\pi}_\theta$ is the policy, and λ is a regularization term that controls the policy update strength to avoid large deviations from the offline-trained policy.

To mitigate the overfitting risk from offline data, the algorithm uses an exploration regularization term. This term encourages exploration during the online phase and prevents the model from overly relying on the offline policy. The regularization is integrated into the actor update:

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

where α is a scaling factor, and ε_θ is a noise term that drives exploration by adjusting the action distribution.

PEX[33]. The algorithm focuses on enhancing the transition from offline to online RL by expanding the policy learned from the offline phase to better adapt to online environments. The approach introduces a policy expansion mechanism, where the policy is refined to account for online experiences while avoiding overfitting to the offline dataset.

In the offline phase, the agent learns a policy $\hat{\pi}_\theta$ by training on a fixed offline dataset $\mathcal{DK}_{\text{offline}}$ using standard RL methods. The objective is to learn a value function and policy that approximate the optimal action-value function. The loss for the offline training is:

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

where $\hat{Q}_\phi$ is the Q-function, and R_t represents the reward. The offline phase ensures the policy is trained using a large batch of historical data.

After offline pretraining, the policy undergoes fine-tuning in the online phase, where the agent interacts with the environment to improve the policy further. The online fine-tuning objective involves minimizing the following loss:

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

where $\mathcal{D}_{\text{online}}$ is the new online experience, and λ is a regularization term to control the magnitude of policy updates and prevent overfitting.

A key feature of this approach is policy expansion. This involves expanding the learned policy during online fine-tuning by introducing a parameterized noise term, ϵ_θ, to increase the exploration capabilities of the agent. The exploration is enforced via the following regularization term:

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

where α is a scaling factor, and ε_θ represents a noise term added to the actions during online learning to encourage exploration and prevent the policy from being overly reliant on the offline dataset.

Policy Architecture Used In All Experiments

MLP. For most of the experiments, the authors of DPPO uses a Multi-layer Perceptron (MLP) with two-layer residual connection as the policy head. For diffusion-based policies, they 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. (2024b)[3], does not use MLP as the diffusion architecture, but they find it delivers comparable (or even better) pre-training performance compared to UNet.

Transformer[28]. For comparing to other policy parameterizations, the original paper also considers Transformer as the policy architecture for the Gaussian and GMM baselines. It considers decoder only. No dropout is used. A learned positional embedding for the action chunk is the sequence into the decoder.

UNet[24]. For comparing to other policy parameterizations, the original paper also considers UNet (Ronneberger et al., 2015) as a possible architecture for DP. It follows the implementation from Chi et al. (2024b) 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 and finds this modification helpful in more challenging tasks.

ViT[12]. For pixel-based experiments, the original paper uses Vision-Transformer(ViT)-based image encoder introduced by Hu et al. (2023) before an MLP head. Proprioception input is appended to each channel of the image patches. It also follow (Hu et al., 2023) 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.

2 Propose your choice of methods and the reason for the choice.

For this project, I chose to implement and modify diffusion-based reinforcement learning algorithms, specifically Diffusion Policy Policy Optimization (DPPO) and Diffusion Actor-Critic with Entropy Regulator (DACER), within the TensorFlow framework. I developed a comprehensive library to bridge the gap between PyTorch and TensorFlow. This library facilitates the seamless conversion of commonly used functions and network modules from PyTorch to TensorFlow, ensuring an efficient and smooth transition when the users want to replicate and deploy models at scale. I tested the individual functions and modules rigorously to ensure that the behavior and output of the converted functions remain consistent with their PyTorch counterparts.

In Part 1, I focused on replicating the results of the "Diffusion Policy Policy Optimization" (DPPO) algorithm presented in the original paper. Using the comprehensive library I developed for converting PyTorch code to TensorFlow, I implemented all the functionalities of the proposed Diffusion Policy Policy Optimization (DPPO) and performed cross-verification on the outputs. Under consistent input conditions, the model outputs of the PyTorch version (source code) and the TensorFlow version (re-implemented code) are completely identical.

As for the experiments in the paper, I first conducted a comprehensive comparison of DPPO with other reinforcement learning (RL) methods and other policy parameterizations. To achieve this, I utilized the code provided by the DPPO authors on GitHub and converted their implementations into TensorFlow using my library. This enabled a fair and consistent evaluation of DPPO against

other RL methods and policy parameterizations. Furthermore, I replicated the experiments of the Avoid environment from the D3IL benchmark in the same manner as the DPPO paper and performed a detailed comparison of DPPO, Gaussian, and GMM policies within this specific task.

In Part 2, I have attempted several versions of soft actor critic (SAC) type algorithms on the special Markov Decision Process(MDP) of DPPO, as discussed in Section 4.10. In the given paper of Diffusion Actor-Critic with Entropy Regulator(DACER)[30] , they address the challenge of entropy estimation in diffusion models by employing a Gaussian Mixture Model (GMM). Inspired by your given idea, I estimate the entropy of the actor by a Gaussian Mixture Model (GMM) to develop our algorithm Diffusion Policy Optimization with Entropy Regulator(DPOER), which better trades off the exploration and exploitation of DPOER. But different from the sampling approach, which adopts 200 times of the original data to estimate the entropy, we estimate the entropy directly from the original data points. DPOER is a Soft Actor-Critic(SAC) type algorithm tailored to the special Diffusion Policy MDP.

Moving forward, a potential direction for future research could focus on simplifying the design (further discussion in 4.10, and achieving high performance while using less time and computation. Recently, SDAC [16] was proposed to combine diffusion model and SAC efficiently. It utilizes the perspective of energy-based models. But it seems that they do not utilize the denoising process of the diffusion model as sequential actions to optimize, which is the crucial reason why DPPO works pretty well in the long-horizon tasks. Limited by their current experiments in the Gym dataset only, I am uncertain if SDAC could work pretty well in the long-horizon task like Transport and Square in the Robomimic task. In these tasks, DPPO and DPOER significantly outperform the other baselines.

3 Testing plans and testing results for checking your implementation is correct and your results are valid.

Model and Agent Level: To ensure our implementation aligns perfectly well with the original DPPO implememtation. We use numpy to generate the random number in both sides and convert these genetated random number arrays into the tensors in PyTorch and TensorFlow. Then we test if the output of each function is the same on both side. For the DPPO core methods. The actor generates the same output as the original code. This ensures that the Tensorflow implementation is correct and the results are valid. The slight difference comes from the different initialization of the Critic part. Even I implemented a Kaiming Uniform Initialization in the Tensorflow. I didn't control the random number of the critic to be the same for both PyTorch and Tensorflow.

The general testing functions are across our implementations. The user could find the debug function to output the implementation details. For the individual testing of each function we put the in the following atomic level testing subsection.

Function Level: In the function level, I wrote functions and tested each function modules one by one. Because the original code was written in PyTorch and transferred to Tensorflow. I write a Conversion Wrapper Library and test the TensorFlow code in comparison to their PyTorch counterparts. The correspondence of the code in the PyTorch and the code in the TensorFlow is listed in the table 1 , table 2 and table3.

These tests of the wrapper make sure the operation is consistent with the original PyTorch code. Furthermore, this library not only facilitates the implementation of the main codebase in TensorFlow and the testing of them. When I try to transfer the Robomimic task environment from the PyTorch to the Tensorflow, I also put these library file in the Robomimic libraries to help the conversion from the PyTorch operations to Tensorflow.

3.1 PyTorch-to-TensorFlow Conversion Wrapper Library

I write a PyTorch-to-TensorFlow Conversion Wrapper Library to help the testing. The motivation behind creating this library stems from the need to address the challenges researchers and developers face when transitioning between PyTorch and TensorFlow. While PyTorch is widely used for research and experimentation due to its flexibility and ease of use, TensorFlow excels in large-scale deployment, offering optimized performance and production-ready capabilities. However, the process of transferring codes from PyTorch to TensorFlow often involves significant manual adjustments, which can be time-consuming and error-prone.

To bridge this gap, I developed this library to simplify the code conversion process between the two frameworks. The primary goal is to reduce the manual effort required to adapt PyTorch models for TensorFlow deployment, ensuring that users can leverage the strengths of both frameworks with minimal friction. By automating most of the conversion work, this library provides a straightforward solution for researchers and developers who need to transition models from one framework to another without sacrificing performance or functionality.

This library was designed with efficiency in mind, allowing users to focus on model development and research, rather than the tedious task of framework adaptation. It enables a smoother workflow from prototyping in PyTorch to deployment in TensorFlow, facilitating a seamless and productive experience across both environments.

The Strength of this framework:

- Pretty fast to replicate any code written in PyTorch with TensorFlow agent
- Accurate and easy to test. I could test each single modules separately. If the test function of each function and module is complete, it guarantees the results are the same as the original PyTorch one.
- Easy to find accurate documentation. This is a wrapper to implement the PyTorch functions in TensorFlow. So I could directly search the documentation of the original PyTorch modules. The Pytest is conducted to guarantee the output of the wrapper function in TensorFlow is the same as the output of the original PyTorch functions in PyTorch when these two outputs are both transferred to a Numpy Array.
- Enjoy the merit of efficiency and deployment of TensorFlow.

The Weakness of this framework:

- Difficult for people to understand if they don't know one of TensorFlow and PyTorch.

The table1, table2, table3 are simple documentations to introduce the correspondence of the modules and functions. With this correspondence, the user could search the torch documentation for a detailed introduction of each functions and modules. The test files are inside the folder ./learning_code_tf/test/.

Table 1. Functions Table (Sorted by TestFile).

PyTorch	TensorFlow	TestFile
torch.abs()	torch_abs()	test_abs.py
torch.optim.Adam	torch_optim_Adam	test_Adam.py
torch.optim.AdamW	torch_optim_AdamW	test_AdamW.py
torch.argmax()	torch_argmax()	test_argmax.py
torch.arange()	torch_arange()	test_arange.py
torch.atanh()	torch_atanh()	test_atanh.py
torch.cat()	torch_cat()	test_cat.py

PyTorch	TensorFlow	TestFile
torch.clamp()	torch_clamp()	test_clamp.py
torch.Tensor.clamp_()	torch_tensor_clamp_()	test_clamp_.py
torch.clip()	torch_clip()	test_clip.py
Categorical()	CategoricalDistribution()	test_Categorical.py
torch.clamp_()	torch_tensor_clamp_()	test_clamp_.py
torch.clamp()	torch_clamp()	test_clamp.py
torch.clip()	torch_clip()	test_clip.py
torch.Tensor.clone()	torch_tensor_clone()	test_clone.py
torch.cumprod()	torch_cumprod()	test_cumprod.py
torch.Tensor.detach()	torch_tensor_detach()	test_detach.py
torch.dot()	torch_dot()	test_dot.py
torch.exp()	torch_exp()	test_exp.py
torch.Tensor.expand()	torch_tensor_expand()	test_expand.py
torch.Tensor.expand_as()	torch_tensor_expand_as()	test_expand_as.py
torch.flatten()	torch_flatten()	test_flatten.py
torch.flip()	torch_flip()	test_flip.py
torch.Tensor.float()	torch_tensor_float()	test_float()
torch.from_numpy()	torch_from_numpy()	test_from_numpy.py
torch.full()	torch_full()	test_full.py
torch.full_like()	torch_full_like()	test_full_like.py
torch.func.functional_call()	torch_func_functional_call()	test_func_functional_call.py
torch.gather()	torch_gather()	test_gather.py
torch.hstack()	torch_hstack()	test_hstack.py
torch.Tensor.item()	torch_tensor_item()	test_item.py
torch.linspace()	torch_linspace()	test_linspace.py
torch.log()	torch_log()	test_log.py
torch.logsumexp()	torch_logsumexp()	test_logsumexp.py
torch.Tensor.long()	torch_tensor_long()	test_long.py
torch.Tensor.masked_fill()	torch_tensor_masked_fill()	test_masked_fill.py
torch.max()	torch_max()	test_max.py
torch.meshgrid()	torch_meshgrid()	test_meshgrid.py
torch.min()	torch_min()	test_min.py
torch.nn.functional.mse_loss()	torch_mse_loss()	test_mse_loss.py
torch.multinomial()	torch_multinomial()	test_multinomial.py
torch.nanmean()	torch_nanmean()	test_nanmean.py
torch.nn.Conv1d()	nn_Conv1d()	test_nn_Conv1d.py
torch.nn.Conv2d()	nn_Conv2d()	test_nn_Conv2d.py
torch.nn.ConvTranspose1d()	nn_ConvTranspose1d()	test_nn_ConvTranspose1d()
torch.nn.functional.pad()	nn_functional_pad()	test_nn_functional_pad.py
torch.nn.GroupNorm()	nn_GroupNorm()	test_nn_GroupNorm.py
torch.nn.init.normal_()	torch_nn_init_normal_()	test_nn_init_normal.py
torch.nn.init.ones_()	torch_nn_init_ones_()	test_nn_init_ones.py
torch.nn.init.xavier_normal_()	torch_nn_init_xavier_normal_()	test_nn_init_xavier_normal_.py
torch.nn.init.zeros_()	torch_nn_init_zeros_()	test_nn_init_zeros.py
torch.ones_like()	torch_ones_like()	test_ones_like.py

PyTorch	TensorFlow	TestFile
torch.permute()	torch_tensor_permute()	test_permute.py
torch.prod()	torch_prod()	test_prod.py
torch.quantile()	torch_quantile()	test_quantile.py
torch.rand()	torch_rand()	test_rand.py
torch.randint()	torch_randint()	test_randint.py
torch.randn()	torch_randn()	test_randn.py
torch.randn_like()	torch_randn_like()	test_randn_like.py
torch.randperm()	torch_randperm()	test_randperm.py
torch.arange()	torch_arange()	test_range.py
torch.register_buffer()	torch_register_buffer()	test_register_buffer.py
torch.repeat_interleave()	torch_repeat_interleave()	test_repeat_interleave.py
torch.Tensor.repeat()	torch_tensor_repeat()	test_repeat.py
torch.Tensor.requires_grad_()	torch_tensor_requires_grad_()	test_requires_grad.py
torch.reshape()	torch_reshape()	test_reshape.py
torch.round()	torch_round()	test_round.py
torch.Tensor.reshape()	torch.convert_to_tensor().reshape()	test_shape.py
torch.softmax()	torch_softmax()	test_softmax.py
torch.split()	torch_split()	test_split.py
torch.sqrt()	torch_sqrt()	test_sqrt.py
torch.square()	torch_square()	test_square.py
torch.squeeze()	torch_squeeze()	test_squeeze.py
stack_module_state()	torch_func_stact_module_state()	test_stack_module_state.py
torch.stack()	torch_stack()	test_stack.py
torch.std()	torch_std()	test_std.py
torch.sum()	torch_sum()	test_sum.py
torch.tanh()	torch_tanh()	test_tanh.py
torch.no_grad()	torch_no_grad()	test_torch_no_grad.py
torch.ones()	torch_ones()	test_torch_ones.py
torch.zeros()	torch_zeros()	test_torch_zeros.py
torch.Tensor.transpose()	torch_tensor_transpose()	test_transpose.py
torch.triu()	torch_triu()	test_triu.py
torch.unsqueeze()	torch_unsqueeze()	test_unsqueeze.py
torch.unravel_index()	torch_unravel_index()	test_unravel_index.py
torch.Tensor.view()	torch_tensor_view()	test_view.py
torch.vmap()	torch_vmap()	test_vmap.py
torch.where()	torch_where()	test_where.py
torch.zeros_like()	torch_zeros_like()	test_zeros_like.py

Table 2. Modules, Optimizers, and Gradients Table.

PyTorch	TensorFlow	TestFile
torch.nn.Tanh	nn_Tanh	test_nn_Tanh.py
torch.nn.Identity	nn_Identity	test_nn_Identity.py
torch.nn.Softplus	nn_Softplus	test_nn_Softplus.py
torch.nn.Mish	nn_Mish	test_nn_Mish.py
torch.nn.ELU	nn_ELU	test_nn_ELU.py
torch.nn.GELU	nn_GELU	test_nn_GELU.py
torch.nn.ReLU	nn_ReLU	test_nn_ReLU.py
torch.nn.Dropout	nn_Dropout	test_Dropout.py
torch.nn.Linear	nn_Linear	test_nn_Linear.py
torch.nn.Parameter	nn_Parameter	test_nn_Parameter.py
torch.nn.Sequential	nn_Sequential	test_nn_Sequential.py
torch.nn.ModuleList	nn_ModuleList	test_ModuleList.py
torch.nn.Embedding	nn_Embedding	test_embedding.py
torch.nn.LayerNorm	nn_LayerNorm	test_LayerNorm.py
torch.nn.MultiheadAttention	nn_MultiheadAttention	test_MultiheadAttention.py

Table 3. Custom and Modified Functions Table.

PyTorch	TensorFlow	TestFile
CosineAnnealingWarmupRestarts	CosineAWR	test_cosineAWR.py
torch.distributions.normal.Normal	Normal	test_Normal.py
torch.distributions.independent.Independent	Independent	test_independent.py
torch.distributions.Categorical	Categorical	test_Categorical.py
torch.distributions.MixtureSameFamily	MixtureSameFamily	test_MixtureSameFamily.py

4 Present the answers of the question in clear and understandable language.

4.1 Part1 Q1

Can you replicate their results with Tensorflow Agent?

I have implemented the replication of this paper in TensorFlow, including the main algorithm as well as the additional algorithms from the following papers (baselines).

As for the replication results of experiments, I have successfully replicated the experiments "Figure 4: Comparing to other diffusion-based RL algorithms" and "Figure 7: Structured exploration in the Avoid environment from Jia et al.[14]", as well as partially replicated the experiment "Figure 5: Comparing to other policy parameterizations with state (left) or pixel (right) observation." However, due to randomization and the absence of some configuration files provided by the authors, there are certain discrepancies in the presentation of our results. I will delve deeper into these differences in the subsequent sections. The following is a comparison of the replicated results with the results presented in the paper.

Figure 4: Comparing to other diffusion-based RL algorithms.

DPPO is compared with RL methods for fine-tuning diffusion-based policies, including DRWR, DAWR, DIPO, IDQL, DQL, and QSM. Evaluations cover three GYM and four ROBOMIMIC tasks.

DPPO demonstrates strong fine-tuning performance, consistent training stability, and superior results across tasks. In GYM tasks, IDQL and DIPO perform competitively, while others struggle with stability. In ROBOMIMIC tasks, DPPO is the strongest overall, particularly excelling in Transport. To our knowledge, DPPO is the first RL algorithm to solve Transport from either state or pixel input to high ($> 50\%$) success rates.

Thanks to our rigorous verification process through step-by-step comparison, the replication results for DPPO are entirely consistent with those reported in the original paper. For other diffusion-based RL algorithms, the overall trends and their relative performance align well with the original findings. However, discrepancies are observed at specific data points. This can be attributed to the fact that the results in the original paper were based on the average of five runs with different random seeds, whereas due to computational and time constraints, I were only able to perform a single run for each method. Some methods exhibit lower stability, as reflected in the variance shown in the original paper's figures, and this instability contributes to the observed differences in certain data points.

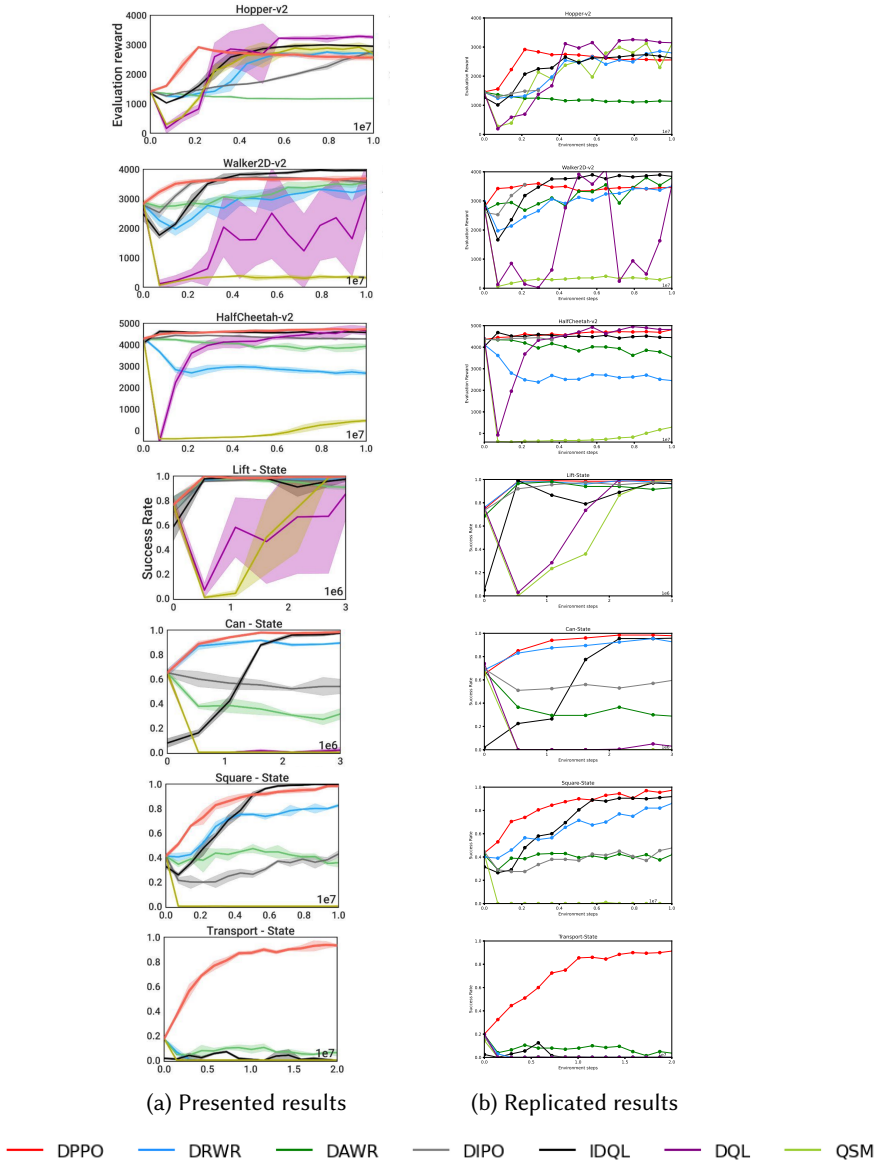
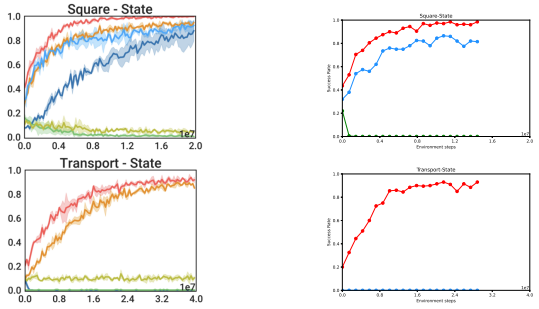


Fig. 1. Comparison between replicated results and Figure 4 (comparing to other diffusion-based RL algorithms)

Figure 5: Comparing to other policy parameterizations

Compare DPPO with popular RL policy parameterizations: unimodal Gaussian with diagonal covariance (Sutton et al., 1999) and Gaussian Mixture Model (GMM (Bishop and Nasrabadi, 2006)), using either MLPs or Transformers (Vaswani et al., 2017), and also fine-tuned with the PPO objective. There are compared to DPPO-MLP and DPPO-UNet, which use either MLP or UNet as the network backbone.

I evaluate on the four tasks from Robomimic (Lift, Can, Square, Transport) with both state and pixel input. With state input, DPPO pre-trains with 20 denoising steps and then fine-tunes the last 10. With pixel input, DPPO pre-trains with 100 denoising steps and then fine-tunes 5 DDIM steps.



(a) Presented results (b) Replicated results

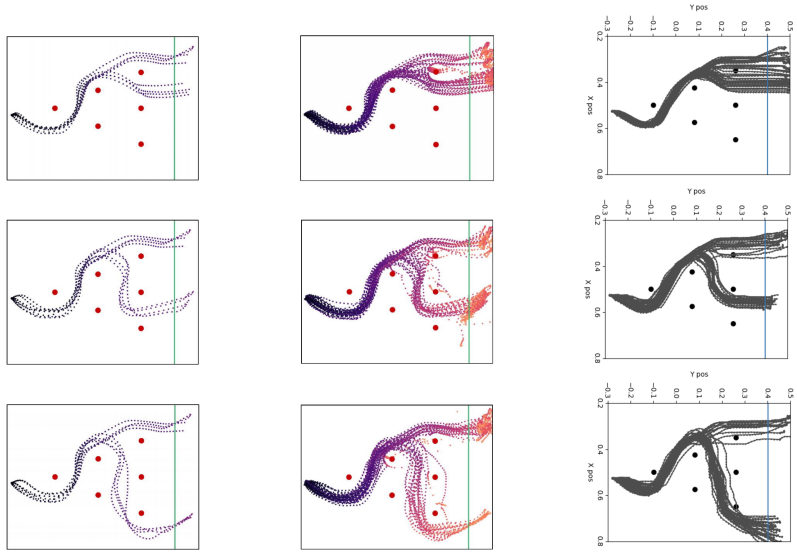
Fig. 2. Comparison between replicated results and Figure 5 (comparing to other policy parameterizations)

Figure 7: Structured exploration on the Avoid environment from[14]

Use the Avoid environment from D3IL benchmark (Jia et al., 2024), where a robot arm needs to reach the other side of the table while avoiding an array of obstacles. The action space is the 2D target location of the end-effector. D3IL provides a set of expert demonstrations that covers different possible paths to the goal line - I consider three subsets of the demonstrations, M1, M2, and M3, each with two distinct modes.

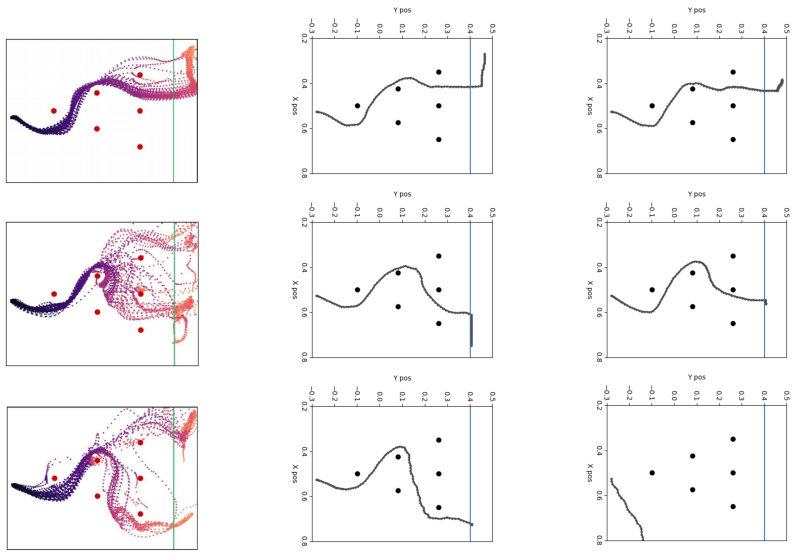
I pre-train MLP-based Diffusion, Gaussian, and GMM policies ($T_a = 4$ unless noted) with the demonstrations, and show sampled trajectories at the first iteration of fine-tuning for DPPO, Gaussian, and GMM after pre-training on three sets of expert demonstrations, M1, M2, and M3. For fine-tuning, I assign (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.

Since their configuration .yaml file is absent in the ./cfg folders, with the exception of DPPO at M1, I created a version of the missing 8 files by myself, which include DPPO at M2 and M3, Gaussian at M1, M2, and M3, and GMM at M1, M2, and M3. Furthermore, I use the pytorch Kaiming uniform in tensorflow to initialize the weights of the network in the beginning.



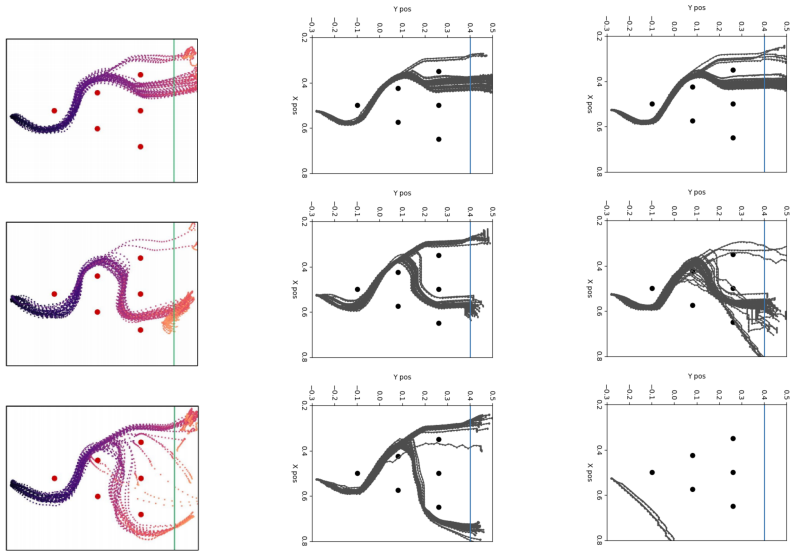
(a) Pre-training data (b) Presented results (c) Replicated results

Fig. 3. Comparison between replicated results and Figure 7 (DPPO)



(a) Presented results (b) Results by given codes (c) Replicated results

Fig. 4. Comparison between replicated results and Figure 7 (Gaussian)



(a) Presented results (b) Results by given codes (c) Replicated results

Fig. 5. Comparison between replicated results and Figure 7 (GMM)

DPPO achieves broader, more structured exploration around the expert data manifold, while Gaussian policies produce unstructured noise (notably in M2), and GMMs show limited coverage.

This advantage arises from DPPO's iterative denoising process, which introduces noise across multiple steps rather than adding it only to the final action, as in Gaussian policies. Each step expands exploration while guiding actions toward the expert manifold[21]. Additionally, diffusion parameterization and chunk-wise denoising ensure that DPPO's stochasticity is structured across both action dimensions and time, enhancing exploration efficiency.

4.2 Part1 Q2

From your experience of replication of their code, what are the parts that are unclear in the paper?

(1) The optimizer used in the method is not explicitly detailed in the paper. However, it appears to be based on the work by [15], which introduces the SGDR (Stochastic Gradient Descent with Restarts) optimizer. This is crucial for understanding how learning rate schedules are managed during training.

(2) In the diffusion_ppo.py and gaussian_ppo.py files, the PPO diffusion process should align with Eqn. 2 in [27]. This equation defines the reward mechanism for maximizing the probability of the teacher policy's action under the current policy. Actions are sampled along trajectories induced by the current policy, but the detailed interaction between these components is not clearly explained in the paper.

(3) In the unet.py file, FiLM modulation [20] is employed to predict per-channel scale and bias. While the use of FiLM is mentioned, its specific implementation details and interaction with the neural network layers are not fully elaborated in the paper.

(4) The reward_scaling.py file employs a reward scaling technique [6] to balance the actor and critic losses. In this process, rewards are normalized by dividing through by the standard deviation of a rolling discounted sum of rewards, without subtracting and re-adding the mean.

(5) In the diffusion.py file, DDIM [26] and DDPM [18] are utilized for the denoising process, which is crucial for generating actions.
(6) The paper does not clearly explain the matching of action chains before and after denoising. Additionally, it lacks details on how the denoising process across all environments and time steps is "mixed together" and shuffled before being sampled in batches. The operations on the chains are described with limited detail.

4.3 Part1 Q3

Can you please write a note explaining in more details the algorithm and the parts that are missing in the paper?

Answer: By following your instruction and examining the code implementation, I have gained a deeper understanding of the DPPO model and algorithm, as follows:

Diffusion Policy Policy Optimization (DPPO) integrates diffusion models with policy gradient methods. DPPO pre-trains the base diffusion policy in offline mode and fine-tunes them in online mode. The pre-training process involves with a forward diffusion process, where noise is gradually added to the original data. The base diffusion policy model learns to predict the added noise and reconstruct the original data. A behavior cloning loss is applied to train the reconstruction. Different policy parameterizations like multilayer perceptron(MLP), UNet, Transformer are attempted as the structure of the base policy. The author choose the MLP as the final backbone choice of DPPO.

In the pre-training phase with forward diffusion process, DPPO applies DDPM[18]. DDPM applies a cosine schedule function to generate the β_t parameters for each step in a forward diffusion process. It smoothly increases the noise intensity, avoiding a sudden increase. Suppose we have K forward steps in the diffusion model, the parameter $\alpha_k = 1 - \beta_t$ controls the strength of noise at each time step in the diffusion process. We also use this α_k parameter later in the de-noising process.

In the fine-tuning phase, the core idea of DPPO is to treat every de-noising steps equivalently as a sequential time-series model. Similarly, suppose we have K denoising steps, from $K-1$ to 0, the noise is denoised from the initial x_K to the final x_0. Different from other approaches, which only utilize the generated x_0 at the last step of the denoising process, DPPO utilize the whole denoising process. DPPO take every contiguous denoising steps x_{k+1} and x_k, where k is the current denoising steps. Through GAE, the advantage is first evaluated by the rewards generated at x_0^{t+1} and x_0^t, where t is the environmental steps. Then it discounted the advantages gain from x_0^t to x_k^t. This discounting scheme implies that the closer a given pair is to the end of the de-noising process(the smaller k becomes), the greater the advantage becomes. Because the agent only take actions with each given x_0^t, one environment step t correspond to K de-noising steps.

Algorithm1 is the pseudo code of DPPO given in the updated version of the paper.

Action Generation with Diffusion Model. In the fine-tuning process, Diffusion Model generates actions by iteratively denoising an initial noise sample through a reverse diffusion process. The goal is to progressively remove noise from the action prediction and refine it over multiple denoising steps. Given an action x_k, The model first predicts the noise ϵ_k at denoising steps k by the pre-trained actor network.

Algorithm 1: DPPO

Pre-train diffusion policy π_θ with offline dataset $\mathcal{D}_{\text{off}}$ using BC loss $\mathcal{L}_{\text{BC}}(\theta)$ Eq. (C.1).
Initialize value function V_ϕ.
for *iteration* $= 1, 2, \ldots$ **do**
  Initialize rollout buffer $\mathcal{D}_{\text{itr}}$.
  $\pi_{\theta_{\text{old}}} = \pi_\theta$.
  for *environment* $= 1, 2, \ldots, N$ *in parallel* **do**
    Initialize state $\bar{s}_{\bar{t}(0,K)} = \left(s_0, a_0^K\right)$ in $\mathcal{M}_{\text{DP}}$.
    for *environment step* t $= 1, \ldots, T$, *denoising step* $k = K-1, \ldots, 0$ **do**
      Sample the next denoised action $\bar{a}_{\bar{t}(t,k)} = a_t^k \sim \pi_{\theta_{\text{old}}}$.
      if $k = 0$ **then**
        then Run a_t^0 in the environment and observe $\bar{R}_{\bar{t}(t,0)}$ and $\bar{s}_{\bar{t}(t+1,K)}$.
      else
        Set $\bar{R}_{\bar{t}(t,k)} = 0$ and $\bar{s}_{\bar{t}(t,k-1)} = \left(s_t, a_t^k\right)$.
      Add $\left(k, \bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}, \bar{R}_{\bar{t}(t,k)}\right)$ to $\mathcal{D}_{\text{itr}}$.
  Compute advantage estimates $A^{\pi_{\theta_{\text{old}}}}\left(s_{\bar{t}(t,k=0)}, a_{\bar{t}(t,k=0)}\right)$ for $\mathcal{D}_{\text{itr}}$ using GAE Eq. (C.2).
  for *update* $= 1, 2, \ldots,$ *num_update* ▷ *Based on replay ratio* N_θ **do**
    for *minibatch* $= 1, 2, \ldots, B$ **do**
      Sample $\left(k, \bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}, \bar{R}_{\bar{t}(t,k)}\right)$ and $A^{\pi_{\theta_{\text{d}}}}\left(s_{\bar{t}(t,k)}, a_{\bar{t}(t,k)}\right)$ from $\mathcal{D}_{\text{itr}}$.
      Compute denoising-discounted advantage $\hat{A}_{\bar{t}(t,k)} = \gamma_{\text{DENOISE}}^k A^{\pi_{\theta_{\text{dat}}}}\left(s_{\bar{t}(t,0)}, a_{\bar{t}(t,0)}\right)$.
      Optimize π_θ using policy gradient loss $\mathcal{L}_\theta$ Eq. (C.3).
      Optimize V_ϕ using value loss $\mathcal{L}_\phi$ Eq. (C.4).
  return converged policy π_θ.

Similar to DDPM[18], we define the following equations:

$$
\begin{aligned}
\alpha_k &:= 1 - \beta_k \\
\bar{\alpha}_k &:= \prod_{s=0}^{k} \alpha_s \\
\tilde{\beta}_k &:= \frac{1 - \bar{\alpha}_{k-1}}{1 - \bar{\alpha}_k} \beta_k \\
q\left(x_k \mid x_0\right) &= \mathcal{N}\left(x_k; \sqrt{\bar{\alpha}_k} x_0, \left(1 - \bar{\alpha}_k\right) \mathbf{I}\right) \\
q\left(x_{k-1} \mid x_k, x_0\right) &= \mathcal{N}\left(x_{k-1}; \tilde{\mu}\left(x_k, x_0\right), \tilde{\beta}_k \mathbf{I}\right)
\end{aligned}
\tag{1}
$$

Then the DDPM(or DDIM) used in the denoising process estimates the original x_0 as $x_0^{'}$ by the following equation to guide the calculation of x_{k-1}.

For DDIM, we use:

$$
x_0^{'} = \frac{x_k - \sqrt{1 - \alpha_k} \cdot \epsilon_k}{\sqrt{\alpha_k}}
$$

For DDPM, we use:

$$
x_0^{'} = \sqrt{1/\alpha_k} \cdot x_k - \sqrt{(1/\alpha_k) - 1} \cdot \epsilon_k
$$

The model also computes the mean μ_k and variance σ_k^2 of the action distribution at the denoising step k to sample the action for the denoising step $k-1$, where μ_k represents the mean of x_k:

DDIM:

$$u_k = \sqrt{\alpha_{k-1}} x_0 + \sqrt{1-\alpha_{k-1}} \cdot \epsilon_k$$

DDPM:

$$\mu_k = \tilde{\beta}_k \frac{\sqrt{\bar{\alpha}_{k-1}}}{(1-\bar{\alpha}_k)} x_0 + \sqrt{\alpha_k} \frac{1-\bar{\alpha}_{k-1}}{1-\bar{\alpha}_k} x_k$$

$$\sigma_k^2 = \tilde{\beta}_k$$

Action Chain Matching, Random Shuffling, and Batch Sampling. In the forward() function, the p_mean_var() method is called to initialize noise and iteratively denoise it, ultimately generating the actions. Once the actions are generated, the agent takes these actions in the environment, receiving new states and rewards. This process follows the standard reinforcement learning paradigm, where the agent interacts with the environment, collects feedback (rewards), and updates its state.

Specially, in the DPPO algorithm, for each sample generated through the denoising process (where the diffusion model generates actions step by step), the action chain (from x_K to x_0) is stored in chains_trajs. The action chain is then split into chains_prev and chains_next, which represent the chains from x_K to x_1 and from x_{K-1} to x_0, respectively. These chains_prev and chains_next are matched pairwise, with the matching being based on the time step sequence. The chain at the current time step (chains_prev) will be used together with the chain from the next time step (chains_next) to compute the loss and update the model parameters. After matching, these chains are flattened, and then randomly shuffled.

During this shuffling process, all the time steps from all environments are "mixed together", as the shape of the flattened indices is self.n_steps × self.n_envs × self.model.ft_denoising_steps. These shuffled indices are then split into mini-batches of size self.batch_size, allowing for batchwise sampling. The log probabilities and losses are computed for each batch to optimize the model. During each loss computation, the model uses the current state of both chains (chains_prev and chains_next) to estimate the advantage and target values, which subsequently guide the model in updating its parameters.

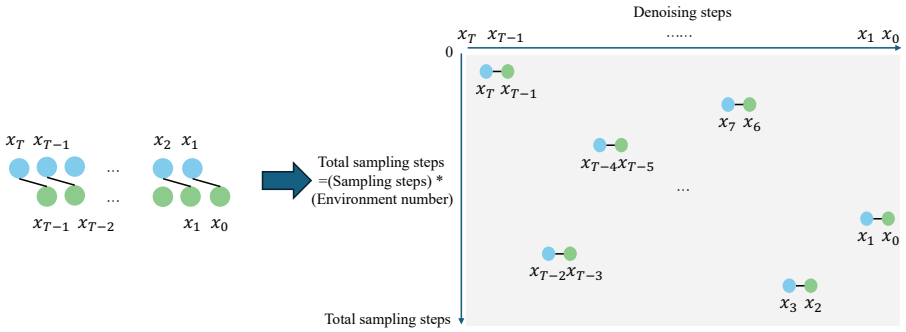


Fig. 6. Sequential pair-wise actions in the denoising process of DPPO.

Generalized Advantage Estimation (GAE). After the diffusion model generates actions, the environment provides feedback in the form of new states and rewards. Generalized Advantage Estimation (GAE) plays an essential role in this process. Its formulation can be broken down into

the following key components:

$$\begin{aligned} \text{Temporal Difference (TD) Error:} \quad & \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \\ \text{Recursive GAE Formula:} \quad & A_t = \delta_t + \gamma\lambda A_{t+1} \\ \text{Return Computation:} \quad & R_t = A_t + V(s_t) \end{aligned}$$

where δ_t represents the TD error at time t, r_t is the reward at time t, $V(s_t)$ is the value of state s_t, and γ is the discount factor. A_t represents the advantage at time t, and R_t is the return.

In DPPO, GAE-λ approximates the advantage function using the series

$$\hat{A}^{\lambda}_{\bar{t}(t,k=0)} = \sum_{l=0}^{\infty} (\gamma\lambda)^t \bar{\delta}_{\bar{t}(t+l,k=0)}, \text{ where } \bar{\delta}_{\bar{t}(t,k)} = \bar{R}_{\bar{t}(t,k)} + \gamma_{\text{ENV}} V_\phi\left(\bar{s}_{\bar{t}(t+1,k)}\right) - V_\phi\left(\bar{s}_{\bar{t}(t,k)}\right). \tag{2}$$

Notably, GAE-λ interpolates between a one-step temporal difference

$$\left(\hat{A}^{\lambda=0}_{\bar{t}(t,k)} = \bar{R}_{\bar{t}(t,k)} + \gamma_{\text{Env}} V_\phi\left(\bar{s}_{\bar{t}(t+1,k)}\right) - V_\phi\left(\bar{s}_{\bar{f}(t,k)}\right)\right)$$

and the Monte Carlo return of the episode relative to the baseline

$$\left(\hat{A}^{\lambda=1}_{\bar{t}(t,k)} = \sum_{l=0}^{T-t} \gamma^t_{\text{ENV}} \bar{R}_{\bar{t}(t+l,k)} - V_\phi\left(\bar{s}_{\bar{t}(t,k)}\right)\right).$$

In Eq. 2, advantages are calculated only for $k = 0$ (the final denoising step), with no need to compute them for intermediate denoising steps. We only need to apply denoising discounting to the calculated advantages so they can be applied to each denoising step k.

Proximal Policy Optimization (PPO). As for the strategy optimization process, DPPO adopts a method similar to the one introduced in the recent robotics paper [27]. During RL fine-tuning, it updates the policy ϕ_θ using the clipped objective:

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}_{\text{itr}}} \min\left(\hat{A}^{\bar{\pi}_{\theta_{\text{old}}}}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right) \frac{\bar{\pi}_\theta\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right)}{\bar{\pi}_{\theta_{\text{old}}}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right)}, \ \hat{A}^{\pi_{\text{old}}}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right) \text{clip}\left(\frac{\bar{\pi}_\theta\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right)}{\bar{\pi}_{\theta_{\text{old}}}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right)}, 1-\varepsilon, 1+\varepsilon\right)\right)$$

If we choose to fine-tune only the last K' denoising steps, then we sample only those from $\mathcal{D}_{\text{itr}}$. Finally, we train the value function to predict the future discounted sum of rewards (i.e., discounted returns):

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}_{\text{itu}}}\left[\left\|\sum_{l=0}^{T-t} \gamma^l_{\text{ENV}} \bar{R}_{\bar{t}(t+l,k)} - V_\phi\left(s_t\right)\right\|^2\right]$$

Similar to the baselines DRWR and DAWR, N_θ and N_ϕ represent the replay ratios for the actor (diffusion policy) and the value critic in DPPO. In practice, N_θ is always set equal to N_ϕ. Similar to usual PPO implementations (Huang et al., 2022), the batch updates in an iteration terminate when the KL divergence between π_θ and $\pi_{\theta_{\text{old}}}$ reaches 1.

Loss function. Inside the files of diffusion_ppo.py, gaussian_ppo.py, the loss function in the PPODiffusion class, derived from PPO (Proximal Policy Optimization) for Diffusion Models, is designed to update both the policy (actor) and value function (critic) while maintaining stability during training. It combines several components:

$$L = L^{\text{PPO}} + \lambda_{\text{entropy}} L^{\text{entropy}} + \lambda_{\text{value}} L^{\text{value}} + \lambda_{\text{BC}} L^{\text{BC}}$$

where L^{PPO} is the clipped policy gradient loss; L^{value} is the clipped value function loss; L^{entropy} is the entropy regularization term; L^{BC} is the behavior cloning loss (teacher-student distillation); λ values control the relative contribution of each term.

(1) PPO Objective Function(further discussion in the part of Proximal Policy Optimization (PPO)): PPO employs a clipped policy gradient loss to stabilize training:

$$L^{\text{PPO}} = \mathbb{E}\left[\min\left(r_t A_t, \text{clip}\left(r_t, 1-\epsilon, 1+\epsilon\right) A_t\right)\right],$$

where r_t is the probability ratio at timestep t; A_t is the advantage function at timestep t, quantifying the relative desirability of an action at state s compared to the average action; ϵ is a small constant that limits the policy update to maintain training stability.

(2) Value Function Loss: To optimize the value function $V(s)$, PPO applies a clipped squared error loss:

$$L^{\text{value}} = \frac{1}{2}\mathbb{E}\left[\max\left(\left(V_\theta(s) - R\right)^2, \left(V_{\text{clipped}}\left(s\right) - R\right)^2\right)\right],$$

where $V_\theta(s)$ is the value function estimated by the current policy, indicating the expected future reward from state s; $V_{\text{clipped}}(s)$ is the clipped value function, restricting updates to avoid drastic changes in value estimation; R represents the actual cumulative reward from a state onward.

(3) Entropy Regularization:

$$L^{\text{entropy}} = -\mathbb{E}[\eta],$$

where η indicats the level of randomness or uncertainty in action selection. High entropy encourages exploration.

(4) Behavior Cloning (BC) Loss[27]: Diffusion Policy leverages Teacher-Student Distillation, where the student policy learns from the teacher policy:

$$L^{\text{BC}} = -\mathbb{E}\left[\log \pi_\theta\left(a_{\text{teacher}} \mid s\right)\right],$$

where π_θ represents the probability of selecting action a given state s; a_{teacher} is the action chosen by the teacher policy. s: The current state.
Behavior cloning loss encourages the student policy to replicate the teacher's actions by maximizing the log-probability of the teacher's actions under the student policy.

KL Divergence and Stability Control. DPPO also computes the KL divergence to measure the difference between the current and old policies during training. The details of its method are given in the blog: http://joschu.net/blog/kl-approx.html.

The KL divergence between p and q is formally defined as:

$$KL[q,p] = \sum_x q(x) \log \frac{q(x)}{p(x)} = E_{x\sim q}\left[\log \frac{q(x)}{p(x)}\right]$$

A naive approximation approach, old_approx_kl, simply computed the expectation of the negative log probability ratio, i.e.,

$$D_{\text{KL}} \approx \mathbb{E}\left[-\log r\right]$$

where $r = \frac{p(x)}{q(x)}$ is the probability ratio.

While computationally efficient, this estimator tends to underestimate KL divergence, particularly when the policy update is large. This is because the log-ratio term does not fully capture the higher-order deviations in probability distributions, which can lead to uncontrolled policy shifts during training.

A better alternative, approx_kl, inspired by a second-order Taylor expansion, is:

$$D_{\text{KL}} \approx \mathbb{E}[(r-1) - \log r]$$

Since the Taylor series expansion of $\log r$ around $r = 1$ is $\log r = (r-1) - \frac{(r-1)^2}{2} + O\left((r-1)^3\right)$, the function $(r-1) - \log r$ retains the second-order term, making it a better approximation compared to $-\log r$.

This formulation provides a more reliable signal for policy updates, reducing the risk of excessively large steps that could destabilize training. This improved estimation is particularly beneficial in Proximal Policy Optimization (PPO), where maintaining a well-calibrated KL divergence is crucial for balancing exploration and policy stability.

Further datails:

Teacher-student distillation[10]. Diffusion Policy utilizes Teacher-Student Distillation, as outlined above in the Behavior Cloning Loss of Loss-function, with an interactive DAgger labeling framework.

In this setup, the privileged information policy $\pi^*_{\text{sim}}(a \mid s)$ acts as the teacher, while the perceptual policy $\pi^*_{\text{real}}(a \mid o)$ serves as the student. Given the inherent domain shift between simulation and real-world environments, I enhance the distillation process by co-training the objective with a combination of real-world expert demonstrations $\mathcal{D}_{\text{real}}$ and simulated data generated from $\pi^*_{\text{sim}}(a \mid s)$ following the DAgger objective. This results in the following co-training objective for policy learning:

$$\max_{\theta} \alpha \sum_{(s_i, o_i, a_i) \sim \tau_{\pi_\theta}} \frac{\pi_\theta\left(\pi_{\text{teacher}}(s_i) \mid o_i\right)}{\sum_{a_e} \pi_\theta\left(a_c \mid o_i\right)} + \beta \sum_{(o_i, a_i) \in \mathcal{D}_{\text{ral}}} \frac{\pi_\theta\left(a_i \mid o_i\right)}{\sum_{a_e} \pi_\theta\left(a_c \mid o_i\right)}$$

The first term corresponds to DAgger-based distillation in simulation, leveraging teacher supervision to refine the student's policy. The second term integrates real-world expert data, allowing the student policy to benefit from a limited but high-quality real-world dataset. This hybrid distillation approach helps mitigate the perceptual domain gap between simulated and real-world scenes, enhancing the policy's generalization ability. As demonstrated empirically (Section III-D), this significantly improves the policy's real-world success rate compared to relying solely on simulated training data.

Feature-wise Linear Modulation (FiLM) modulation[20]. In unet.py, FiLM (Feature-wise Linear Modulation) adjusts the representation of input features by predicting the scale and bias for each channel, enhancing the influence of conditional information and improving the model's performance.

FiLM is a mechanism that adaptively influences the output of a neural network by applying an affine transformation to its intermediate features based on external inputs. Specifically, FiLM learns functions f and h that generate per-feature scaling and bias parameters, $\gamma_{i,c}$ and $\beta_{i,c}$, as functions of the input $\boldsymbol{x}_i$:

$$\gamma_{i,c} = f_c\left(\boldsymbol{x}_i\right), \quad \beta_{i,c} = h_c\left(\boldsymbol{x}_i\right)$$

These parameters modulate the activations $\boldsymbol{F}_{i,c}$ of a neural network via a feature-wise affine transformation:

$$\text{FiLM}\left(\boldsymbol{F}_{i,c} \mid \gamma_{i,c}, \beta_{i,c}\right) = \gamma_{i,c}\boldsymbol{F}_{i,c} + \beta_{i,c}$$

Here, f and h can be arbitrary functions, such as neural networks. FiLM-based modulation can be conditioned either on the same input processed by the target network or on an external input, making it particularly effective for multi-modal and conditional tasks. Unlike concatenation-based conditioning, which increases input dimensionality by appending the condition vector, FiLM directly modulates intermediate activations, offering a more parameter-efficient and spatiallyaware approach.

In unet.py, FiLM is employed to adaptively modify feature maps within 1D convolutional residual blocks based on external conditions, such as state observations. A separate conditioning module

processes these external inputs and predicts per-channel scaling and bias parameters. When both scaling and bias are applied, the feature representation from the first convolutional layer is modulated as follows:

$$\text{out} = \gamma \cdot \text{out} + \beta$$

If only bias is applied, the feature map is simply shifted accordingly. This conditioning mechanism enables dynamic adaptation of feature representations at each layer, allowing the model to effectively integrate external information and enhance its learning capacity.

Reward scaling[6]. To balance actor and critic losses, in reward_scaling.py, DPPO implements a reward computation method where rewards are divided by the standard deviation of a rolling discounted sum of the rewards (without subtracting and re-adding the mean). This method, instead of directly using raw rewards from the environment, is one of the optimizations adopted in PPO but not in TRPO, helping to balance actor and critic losses.

Algorithm 2: PPO Scaling Optimization

Procedure *Initialize-Scaling()*
  $R_0 \leftarrow 0$;
  RS = RunningStatistics() ; // New running stats class that tracks mean, standard deviation
Procedure *Scale-Observation(r_t)*
  $R_t \leftarrow \gamma R_{t-1} + r_t$; // γ is the reward discount
  Add(RS, R_t);
  return r_t/Standard-Deviation(RS) ; // Returns scaled reward

This method offers several key advantages, particularly in balancing Actor and Critic losses. In reinforcement learning, the Actor optimizes policy updates based on rewards, while the Critic estimates value functions to guide these updates. If reward scales fluctuate significantly, the Critic's value estimates may become unstable, leading to erratic policy updates by the Actor. By normalizing rewards with a rolling discounted sum, this method ensures that the Critic receives more consistent target values, preventing overestimation or underestimation. This, in turn, stabilizes policy updates by the Actor, maintaining smooth training dynamics. Additionally, it keeps gradient magnitudes consistent, improving convergence and sample efficiency across different tasks.

Other optimizations used in PPO but not used in TRPO[6].

- Value function clipping: Schulman et al. (2017)[25] originally suggest fitting the value network via regression to target values:

  $$L^V = \left(V_{\theta_t} - V_{\text{targ}}\right)^2$$

  but the standard implementation instead fits the value network with a PPO-like objective:

  $$L^V = \max\left[\left(V_{\theta_t} - V_{targ}\right)^2, \left(\text{clip}\left(V_{\theta_t}, V_{\theta_{t-1}} - \varepsilon, V_{\theta_{t-1}} + \varepsilon\right) - V_{targ}\right)^2\right]$$

  where V_θ is clipped around the previous value estimates (and ε is fixed to the same value as the value used to clip probability ratios in the PPO loss function (cf. Eq. (2) in Section 4).
- Orthogonal initialization and layer scaling: Instead of using the default weight initialization scheme for the policy and value networks, the implementation uses an orthogonal initialization scheme with scaling that varies from layer to layer.
- Adam learning rate annealing: Depending on the task, the implementation sometimes anneals the learning rate of Adam (Kingma & Ba, 2014) (an already adaptive method) for optimization.

- Reward Clipping: The implementation also clips the rewards within a preset range (usually $[-5, 5]$ or $[-10, 10]$).
- Observation Normalization: In a similar manner to the rewards, the raw states are also not fed into the optimizer. Instead, the states are first normalized to mean-zero, variance-one vectors.
- Observation Clipping: Analagously to rewards, the observations are also clipped within a range, usually $[-10, 10]$.
- Hyperbolic tan activations: As observed by Henderson et al. (2017), implementations of policy gradient algorithms also use hyperbolic tangent function activations between layers in the policy and value networks.
- Global Gradient Clipping: After computing the gradient with respect to the policy and the value networks, the implementation clips the gradients such the "global ℓ_2 norm" (i.e. the norm of the concatenated gradients of all parameters) does not exceed 0.5.

4.4 Part1 Q4

Have you found any mistakes or errors?

(1) The learning rate scheduler seems to be adopted in a way slightly different from the original one.
(2) The definition of γ_{ENV}^{t} seems to be incorrect. It should be $\gamma_{\text{ENV}}^{t'-t}$.
(3) In gmm.py, the authors directly utilize component_distribution = D.Normal (loc=means, scale=scales) to generate the component normal distributions for the GMM, without considering the possibility of a multivariate normal distribution with off-diagonal correlations. It would be more appropriate to consider using torch.distributions.multivariate_normal.MultivariateNormal(loc=m covariance_matrix=covariance_matrix) in this context to account for the potential correlations between dimensions.

4.5 Part1 Q5

If you were the reviewer of this paper, would you accept or reject this paper for a major conference?
Answer: If I was the reviewer of this paper. I will give it an acceptance if the overall writing is improved to highlight the motivations behind their ideas and their own contributions of this paper. I tend to have more tolerance to new topics and concepts. This paper is relatively novel in their topics(diffusion + reinforcement learning) and solid in their experiments. That's the main reason I tend to give it an acceptance.

4.6 Part1 Q6

Could you please write a note commenting the pros and cons of this paper?
Pros.

- **Experiments**: The empirical results clearly demonstrate the effectiveness of the proposed framework on real-world datasets.
  - **P1-1:** The authors report performance improvements over baselines in long-horizon and sparse reward tasks.
  - **P1-2:** Their method also leverages environment parallelization, enabling faster training compared to off-policy RL methods. This is a notable strength, as off-policy RL techniques typically do not take full advantage of parallelization, limiting the use of high-throughput simulators.
  - **P1-3:** The paper includes comprehensive comparisons with other RL fine-tuning methods, including two novel baselines of the authors' own design, and thorough ablation studies.

  - **P1-4:** The experiments, including comparisons with diffusion policies and other fine-tuning RL policies, provide valuable insights and contribute significantly to the research field.
  - **P1-5:** DPPO consistently outperforms benchmarks, particularly in tasks involving pixel-based observations and long-horizon rollouts.
  - **P1-6:** DPPO shows impressive zero-shot transfer performance from simulation to real-world robotics tasks, with a minimal sim-to-real performance gap.
  - **P1-7:** The paper provides extensive findings on fine-tuning diffusion policies via PPO, covering topics like denoising steps, network architecture, GAE variants, and the impact of expert data. These results are highly valuable for the community.
  - **P1-8:** The authors also offer an empirical explanation for why their method outperforms alternatives, focusing on structural exploration.
- **Framework Design**:
  - **P2-1:** The paper introduces a novel dual-layer MDP framework, using PPO to fine-tune diffusion policies, which enhances performance across a variety of tasks.
  - **P2-2:** Several best practices for DPPO are proposed, such as fine-tuning only the last few denoising steps and replacing some denoising steps with DDIM sampling, improving both efficiency and performance.
  - **P2-3:** Fine-tuning diffusion models with RL methods is a valuable and original contribution. The experimental results presented in this work are insightful and ground-breaking.
- **New Topics**:
  - **P3-1:** The impressive results and the fact that this paper is among the first to leverage high-throughput simulators for diffusion policy fine-tuning is a key contribution to both the robotics and machine learning communities.

Cons.

- **Writing**: The writing lacks clarity and structure, making the paper difficult to follow. The authors should provide a more explicit list of contributions.
  - **C1-1:** The main sections (pages 1-10) are poorly structured, with the results presented without a clear focus, making it hard to understand the contributions and experimental outcomes.
- **Motivation**: The motivation behind the approach is not adequately explained. Readers expect a clear understanding of the intuition behind the idea, which is somewhat lacking in this paper.
  - **C2-1:** While the typical goal of fine-tuning is to improve sample efficiency and reduce online interactions, DPPO does not significantly improve sample efficiency, as its convergence speed is comparable to other online RL methods.
- **Baselines**: Some sections, such as the initial part of Section 6 and Figure 8, are redundant and unnecessary.
  - **C3-1:** Previous diffusion policy papers have already shown that diffusion models outperform GMM and Gaussian policies in capturing multi-modal distributions, making this result no longer novel.
  - **C3-2:** The experiments only compare DPPO against other diffusion-based methods and do not include comparisons with state-of-the-art Offline2Online or Sim2Real algorithms, such as Uni-O4 and O3F, missing an important opportunity for benchmarking.
  - **C3-3:** Including comparisons with established offline-to-online methods would improve the evaluation and strengthen the findings.
- **Novelty**: While the technical contribution is solid, the novelty is relatively low.
  - **C4-1:** The approach follows a well-established pattern of "applying diffusion policy to task X and fine-tuning with method Y," which has been explored in the robotics learning community.

- **C4-2:** The paper integrates a denoising process as a multi-step MDP into the environmental MDP, which is a concept already proposed in previous works. The PPO adaptation is also not particularly innovative, with similar ideas explored in GAE.
- **C4-3:** Compared to QSM, DPPO does not introduce groundbreaking concepts but instead offers empirical adjustments like fine-tuning certain denoising steps and replacing value estimation with advantage estimation.

4.7 Part2 Q1

Consider the Diffusion Policy MDP of DPPO paper, is it possible to construct a soft actor critic (SAC) type algorithm on this special MDP[8].

Answer: Constructing a Soft Actor-Critic (SAC) type algorithm on the Diffusion Policy MDP presented in the DPPO paper (Training Diffusion Policy with Policy Gradient) is theoretically feasible but requires addressing several challenges, primarily related to the entropy estimation and the unique structure of the diffusion policy.

- **Entropy Estimation Challenges:** In SAC, the entropy of the policy $\pi(a \mid s)$ plays a central role in the objective function:

  $$J(\pi) = \mathbb{E}_{(s_t,a_t)\sim\pi}\left[r\left(s_t, a_t\right) + \alpha\mathcal{H}\left(\pi\left(\cdot \mid s_t\right)\right)\right]$$

  For diffusion policies, the action distribution $\pi(a \mid s)$ lacks a closed-form expression because it results from a reverse diffusion process. This complicates the direct calculation of entropy, which SAC requires for balancing exploration and exploitation.
  Gaussian Mixture Models (GMM), as used in DACER[30], could be adapted to approximate the entropy of the diffusion policy. By fitting the generated actions to a GMM, the entropy can be estimated and incorporated into the SAC framework, which will be discussed in Section.4.8.
- **Action Sampling Efficiency:** SAC typically relies on efficient action sampling for both the actor and critic updates. The multi-step denoising process in diffusion models introduces additional computational overhead, potentially affecting training efficiency.
  In DACER, action generation efficiency is improved by reducing the number of diffusion steps to lower computational costs, using GMM-based entropy estimation with adaptive noise scaling to maintain policy diversity, and incorporating Q-value feedback to guide the diffusion model towards generating high-quality, diverse actions, effectively balancing exploration and efficiency in complex reinforcement learning tasks.

To adapt SAC to the Diffusion Policy MDP, several modifications are required:
The SAC objective needs to incorporate entropy estimation tailored to the diffusion process:

$$J\left(\pi_\theta\right) = \mathbb{E}_{(s,a)\sim\pi_\theta}\left[Q_\phi(s,a) - \alpha\hat{\mathcal{H}}\left(\pi_\theta(a \mid s)\right)\right],$$

where $\hat{\mathcal{H}}\left(\pi_\theta\right)$ is the approximated entropy via GMM or other methods.

The value function (often Q-function) can still be updated using the Bellman residual minimization, as in traditional SAC:

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s')}\left[\left(Q_\phi(s,a) - \left(r + \gamma\mathbb{E}_{a'\sim\pi_\theta}\left[Q_{\phi'}\left(s', a'\right) - \alpha\hat{\mathcal{H}}\left(\pi_\theta\left(a' \mid s'\right)\right)\right]\right)\right)^2\right]$$

where ϕ represents the parameters of the Q-value network; γ is the discount factor; α is the temperature parameter that balances the trade-off between reward and entropy (exploration). $\log \pi\left(a' \mid s'\right)$ is the policy entropy term. A key innovation of SAC is incorporating entropy into the

Bellman equation, encouraging the policy to find a balance between high returns and high entropy (exploration).

The policy update step needs to backpropagate gradients not only through the Q-value but also through the entire diffusion chain:

$$\nabla_\theta J\left(\pi_\theta\right)=\mathbb{E}_{s\sim\mathcal{B},a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a \mid s)\left(Q_\phi(s,a)-\alpha\hat{\mathcal{H}}\left(\pi_\theta(a \mid s)\right)\right)\right] \tag{3}$$

Base on Equation.3, the policy gradient in the diffusion chain can be formulated by considering the multi-step reverse denoising process that generates the action a_0. The diffusion policy is expressed as:

$$\pi_\theta\left(a_0 \mid s\right)=p\left(a_T\right)\prod_{t=1}^{T} p_\theta\left(a_{t-1} \mid a_t, s\right),$$

where $a_T \sim \mathcal{N}(0, I)$ is the initial Gaussian noise and each $p_\theta\left(a_{t-1} \mid a_t, s\right)$ represents a denoising step parameterized by θ.

Thus, the policy gradient through the diffusion chain becomes:

$$\nabla_\theta J^{diff}\left(\pi_\theta\right)=\mathbb{E}_{s\sim\mathcal{B},a_0\sim\pi_\theta}\left[\left(Q_\phi\left(s,a_0\right)-\alpha\mathcal{H}\left(\pi_\theta\left(a_0 \mid s\right)\right)\right)\sum_{t=1}^{T}\nabla_\theta \log p_\theta\left(a_{t-1} \mid a_t, s\right)\right],$$

where the gradient is propagated through each denoising step in the diffusion process, allowing the policy to be optimized by adjusting the parameters at every stage of the reverse diffusion chain.

The final SAC-like algorithm would benefit from the robust exploration capabilities of diffusion models while maintaining the stability and sample efficiency characteristic of SAC. This hybrid approach could potentially outperform traditional SAC in environments that benefit from complex, multimodal policy representations.

4.8 Part2 Q2

One problem in the construction of the soft actor critic algorithm is the estimation of the entropy of the diffusion policy. Is it possible to employ some kind GMM estimation like this paper[30]. If you think that I cannot construct an SAC algorithm on the diffusion policy MDP, do you think this paper provides a way to construct such kind of algorithm? This paper claims that they are practically the same as the SAC with a diffusion policy. However, a few details are missing. Could you please more detail description of the algorithm?

Answer: Addressing the feasibility of constructing a Soft Actor-Critic (SAC) algorithm based on the Diffusion Policy MDP involves overcoming the challenge of entropy estimation, as the diffusion policy lacks an explicit analytical distribution. This difficulty is effectively tackled in the paper [30], which introduces the Diffusion Actor-Critic with Entropy Regulator (DACER) method. DACER employs Gaussian Mixture Models (GMM) to approximate the diffusion policy distribution by sampling action points and applying the Expectation-Maximization (EM) algorithm to estimate GMM parameters, thereby facilitating entropy estimation. Beyond resolving entropy estimation, DACER enhances the policy by maximizing Q-values while adaptively regulating the exploration-exploitation trade-off through its entropy control mechanism.

Therefore, I will firstly analyze the algorithms and details in DACER, and then discuss the methodology to construct a Soft Actor-Critic (SAC) type algorithm based on the Diffusion Policy MDP framework outlined in the DPPO paper in later parts.

Diffusion Actor-Critic with Entropy Regulator(DACER). The specific steps of the DACER algorithm include:

Algorithm 3: Diffusion Actor-Critic with Entropy Regulator for Online RL

Input: $\lambda, \theta, \phi_1, \phi_2, \phi_1', \phi_2', \alpha, \beta_q, \beta_\alpha, \beta_\pi$, and ρ
for *each iteration* **do**
  for *each sampling step* **do**
    Sample $\boldsymbol{a} \sim \pi_\theta(\cdot \mid s)$ by Eq. 4
    Add noise $\boldsymbol{a} = \boldsymbol{a} + \lambda\alpha \cdot \mathcal{N}(0, \boldsymbol{I})$
    Get reward r and new state s'
    Store a batch of samples (s, a, r, s') in replay buffer $\mathcal{B}$
  for *each update step* **do**
    Sample data from $\mathcal{B}$
    Update critic networks using $\phi_i \leftarrow \phi_i - \beta_q \nabla_{\phi_i} \mathcal{L}_q(\phi_i)$ for $i = \{1, 2\}$
    Update diffusion policy network using $\theta \leftarrow \theta - \beta_\pi \nabla_\theta \mathcal{L}_\pi(\theta)$
    if *step* mod10000 == 0 **then**
      Estimate the entropy of diffusion policy $\hat{\mathcal{H}} = \mathbb{E}_{s\sim\mathcal{B}}[\mathcal{H}_s]$
      Update α using $\alpha \leftarrow \alpha - \beta_\alpha[\hat{\mathcal{H}} - \overline{\mathcal{H}}]$
    Update target networks using $\phi_i' = \rho\phi_i' + (1-\rho)\phi_i$ for $i = \{1, 2\}$

(1) **Action Generation:** The diffusion model generates actions from Gaussian noise, with noise levels dynamically adjusted based on entropy estimations.
(2) **Value Estimation:** The generated actions are evaluated using double Q-networks, minimizing the Bellman error to improve Q-value predictions.
(3) **Entropy Estimation and Adjustment:** The action distribution is fitted using a GMM to estimate entropy, and the exploration parameter (noise level) is adjusted to maintain the desired stochasticity.
(4) **Policy Update:** The diffusion model parameters are updated to maximize expected Q-values, incorporating feedback from both value estimation and entropy regulation.
(5) **Periodic Target Network Updates and Entropy Re-Estimation:** Target networks are periodically updated to stabilize learning, and entropy is re-estimated every 10,000 iterations to ensure continuous adaptation.

I summarize our implementation in Algorithm 3.

Further details:

Representation and Optimization Objective of Diffusion Policy. I use the reverse process of a conditional diffusion model as a parametric policy

$$\pi_\theta(\boldsymbol{a} \mid \boldsymbol{s}) = p_\theta(\boldsymbol{a}_{0:T} \mid \boldsymbol{s}) = p(\boldsymbol{a}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{a}_{t-1} \mid \boldsymbol{a}_t, \boldsymbol{s}) \tag{4}$$

where $p(\boldsymbol{a}_T) = \mathcal{N}(0, \boldsymbol{I})$, the end sample of the reverse chain, $\boldsymbol{a}_0$, is the action used for RL evaluation. Generally, $p_\theta(\boldsymbol{a}_{t-1} \mid \boldsymbol{a}_t, s)$ could be modeled as a Gaussian distribution $\mathcal{N}\left(\boldsymbol{a}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{a}_t, \boldsymbol{s}, t), \Sigma_\theta(\boldsymbol{a}_t, \boldsymbol{s}, t)\right)$. I choose to parameterize $\pi_\theta(\boldsymbol{a} \mid \boldsymbol{s})$ like DDPM [18], which sets $\Sigma_\theta(\boldsymbol{a}_t, \boldsymbol{s}, t) = \beta_t I$ to fixed time-dependent constants, and constructs the mean $\boldsymbol{\mu}_\theta$ from a noise prediction model as

$$\boldsymbol{\mu}_\theta(\boldsymbol{a}_t, \boldsymbol{s}, t) = \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{a}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\boldsymbol{a}_t, \boldsymbol{s}, t) \right)$$

where $\alpha_t = 1 - \beta_t, \bar{\alpha}_t = \prod_{k=1}^{t} \alpha_k$, and ϵ_θ is a parametric model. To obtain an action from DDPM, I need to draw samples from T different Gaussian distributions sequentially. The sampling process can be reformulated as

$$\boldsymbol{a}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{a}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta \left(\boldsymbol{a}_t, \boldsymbol{s}, t \right) \right) + \sqrt{\beta_t} \boldsymbol{\epsilon}$$

with the reparametrization trick, where $\epsilon \sim \mathcal{N}(0, I), t$ is the reverse timestep from T to $0, a_T \sim \mathcal{N}(0, I)$.

Q-Value Maximization and Diffusion Policy Learning. In integrating diffusion policy with offline RL, policy improvement relies on minimizing the behaviorcloning term. However, in online RL, without a dataset to imitate, I discarded the behavior-cloning term and the imitation learning framework. In this study, the policy-learning objective is to maximize the expected Q-values of the actions generated by the diffusion network given the state

$$\max_\theta \mathbb{E}_{\boldsymbol{s} \sim \mathcal{B}, \boldsymbol{a}_0 \sim \pi_\theta(\cdot|\boldsymbol{s})} \left[Q_\phi \left(\boldsymbol{s}, \boldsymbol{a}_0 \right) \right]$$

where $\mathcal{B}$ denotes the replay buffer containing experience data, and $Q_\phi \left(s, a_0 \right)$ represents the Q -value estimated using double Q-learning. The gradient information from the reverse diffusion chain is backpropagated to the policy parameters θ to optimize the policy.

- **Soft State-Action Return.** DACER leverages distributional reinforcement learning to model Soft State-Action Returns, enhancing value estimation accuracy. It also computes the global mean of Q-value standard deviations to stabilize uncertainty estimation and smooth the training process.
  The soft state-action return $Z^\pi(s, a)$ is defined as the random variable representing the cumulative return from state s and action a under policy π

  $$Z^\pi(s, a) := r + \gamma G_{t+1},$$

  where G_{t+1} is the future return starting from the next state. In traditional RL, I focus on its expectation $Q^\pi(s, a) = \mathbb{E}\left[Z^\pi(s, a) \right]$.
  In distributional RL, instead of modeling $Q^\pi(s, a)$ directly, I aim to model the entire distribution of $Z^\pi(s, a)$. The value distribution $\mathcal{Z}^\pi(s, a)$ represents the distribution of possible returns for a state-action pair (s, a)

  $$\mathcal{Z}^\pi(s, a) := P\left(Z^\pi(s, a) \mid s, a \right).$$

  This function captures the uncertainty, variance, and even the multi-modal nature of the returns, providing much more information than a simple expectation.
  The traditional Bellman equation is extended to operate over distributions. The distributional Bellman operator $\mathcal{T}^\pi$ updates the return distribution as follows

  $$\mathcal{T}^\pi \mathcal{Z}(s, a) \stackrel{D}{=} r + \gamma \mathcal{Z}\left(s', a' \right),$$

  where s' is the next state sampled from the environment and a' is the next action sampled from the policy π. This operator updates the full return distribution, rather than just the expected value.
- **Critic Update Gradient.** When the Diffusion model generates actions $a \sim \pi_\theta(a \mid s)$, these actions are passed to the Q-network (Critic). The parameters θ of the Diffusion model are updated by maximizing the Q-value (or minimizing the Actor-Loss). To ensure accurate Q-value estimation, the parameters ϕ of the Q-network are updated using the Critic Update Gradient, which is the derivative of the Q-Loss function with respect to ϕ.

Traditionally, the Q-network parameters are optimized by minimizing the Q-Loss function, which represents the Temporal Difference (TD) error between predicted and target Q-values. The standard form of this optimization is given by:

$$\min_{\phi_i} \mathbb{E}_{(s,a,s')\sim\mathcal{B}} \left[\left(r(s,a) + \gamma \min_{i=1,2} Q_{\phi_i'}(s',a') - Q_{\phi_i}(s,a) \right)^2 \right]$$

where a' is obtained by feeding s' into the Diffusion policy, and $\mathcal{B}$ denotes the replay buffer. This loss function measures the squared difference between the current Q -value estimation $Q_{\phi_i}(s,a)$ and the target Q-value computed from rewards and the next state.

In DACER, a modified Critic Update Gradient (drawing from techniques in DSAC [4][5]) is employed to mitigate the problem of Q-value overestimation, enhancing stability and performance, particularly when using Diffusion models. The Critic Update Gradient in DACER is defined as:

$$\begin{aligned}\nabla_\theta J_{\mathcal{Z}}(\theta) =& \mathbb{E}\left[\nabla_\theta \frac{(y_z - Q_\theta(s,a))^2}{2\sigma_\theta(s,a)^2} + \frac{\nabla_\theta \sigma_\theta(s,a)}{\sigma_\theta(s,a)} \right] \\ =& \mathbb{E}[\underbrace{-\frac{(y_z - Q_\theta(s,a))}{\sigma_\theta(s,a)^2}\nabla_\theta Q_\theta(s,a)}_{\text{mean-related gradient}} \underbrace{- \frac{(y_z - Q_\theta(s,a))^2 - \sigma_\theta(s,a)^2}{\sigma_\theta(s,a)^3}\nabla_\theta \sigma_\theta(s,a)}_{\text{variance-related gradient}}].\end{aligned}$$

where $Q_\theta(s,a)$ is the predicted mean Q-value for the state-action pair (s,a); y_z represents the target Q -value derived from rewards and the next state; $\sigma_\theta(s,a)$ is the predicted standard deviation of the Q-value, capturing the uncertainty in value estimation.

This gradient includes both a mean-related term, which adjusts Q-value predictions based on the prediction error, and a variance-related term, which fine-tunes the uncertainty estimation. By incorporating uncertainty modeling through $\sigma_\theta(s,a)$ and bounding the target Q-value, this approach prevents excessive updates when Q-value estimates are unreliable. This stabilizes training and improves the reliability of value estimation, particularly in complex environments like those involving Diffusion models.

Gaussian Mixture Model (GMM) Entropy Estimation. Since the diffusion policy lacks an analytical expression, DACER employs a Gaussian Mixture Model (GMM) to fit the policy distribution and estimate its entropy. The GMM is expressed as a weighted sum of multiple Gaussian distributions:

$$\hat{f}(\boldsymbol{a}) = \sum_{k=1}^{K} w_k \cdot \mathcal{N}\left(\boldsymbol{a} \mid \boldsymbol{\mu}_k, \Sigma_k\right) \tag{5}$$

where K is the number of Gaussian distributions, and w_k is the mixing weight of the k-th component, satisfying $\sum_{k=1}^{K} w_k = 1, w_k \geq 0.\boldsymbol{\mu}_k, \Sigma_k$ are the mean and covariance matrices of the k-th Gaussian distribution, respectively.

- **Expectation-Maximization (EM) Algorithm for GMM Parameter Estimation.** For each state, I use a diffusion policy to sample N actions, $\boldsymbol{a}^1, \boldsymbol{a}^2, \ldots, a^N \in \mathcal{A}$. The ExpectationMaximization algorithm is then used to estimate the parameters of the GMM. In the expectation step, the posterior probability that each data point $\boldsymbol{a}^i$ belongs to each component k is computed, denoted as

$$\gamma\left(\boldsymbol{z}_k^i\right) = \frac{w_k \cdot \mathcal{N}\left(\boldsymbol{a}^i \mid \boldsymbol{\mu}_k, \Sigma_k\right)}{\sum_{j=1}^{K} w_j \cdot \mathcal{N}\left(\boldsymbol{a}^i \mid \boldsymbol{\mu}_j, \Sigma_j\right)}$$

where $\gamma\left(z_k^i\right)$ denotes that under the current parameter estimates, the observed data $\boldsymbol{a}^i$ come from the k-th component of the probability. In the maximization step, the results of the Eq. (13) calculations are used to update the parameters and mixing weights for each component:

$$w_k = \frac{1}{N}\sum_{i=1}^{N}\gamma\left(z_k^i\right), \boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N}\gamma\left(z_k^i\right)\cdot \boldsymbol{a}^i}{\sum_{i=1}^{N}\gamma\left(z_k^i\right)}, \Sigma_k = \frac{\sum_{i=1}^{N}\gamma\left(z_k^i\right)\left(\boldsymbol{a}^i-\boldsymbol{\mu}_k\right)\left(\boldsymbol{a}^i-\boldsymbol{\mu}_k\right)^{\mathrm{T}}}{\sum_{i=1}^{N}\gamma\left(z_k^i\right)}$$

Iterative optimization continues until parameter convergence. Based on our experimental experience in the MuJoCo environments, a general setting of $K = 3$ provides a better fit to the action distribution.

- **Entropy Estimation.** I can estimate the entropy of the action distribution corresponding to the state by [13]. For a continuous-valued random vector $\boldsymbol{x} \in \mathbb{R}^N$ with probability density function $f(x)$, the differential entropy is defined as

$$\mathcal{H}(\boldsymbol{x}) = \mathrm{E}\{-\log f(\boldsymbol{x})\} = -\int_{\mathbb{R}^N} f(x)\cdot\log f(x)\mathrm{d}x, f(x) = \sum_{i=1}^{L}\omega_i\cdot\mathcal{N}\left(x;\mu_i,\mathbf{C}_i\right)$$

By replacing the logarithm with a multivariate Taylor-series expansion, the resulting integral can be solved in closed form. Therefore, the logarithm is expanded around the mean vector μ_k of each Gaussian component of $f(x)$, which leads to

$$\log f(x) = \sum_{k=0}^{R}\frac{1}{k!}\left((x-\mu_i)\odot\nabla\right)^k\log f(x)\Big|_{x=\mu_i} + O_R$$

for $i = 1, 2, \ldots, L$, where ∇ is the gradient with respect to x, O_R is the remainder term, and $\odot$ is the so-called matrix contradiction operator $c = \mathbf{A}\odot\mathbf{B} = \sum_i\sum_j A_{ij}\cdot B_{ij}$ for two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{N\times M}$, which consists of an element-wise matrix multiplication and a subsequent summation of all matrix elements.

It is obvious that the Taylor-series expansion is of infinite order since derivatives of a sum of exponential functions are considered. Thus, the expansion has to be truncated in order to maintain a practical solution. Truncating at order R yields the approximation of the entropy.

$$\mathcal{H}(\boldsymbol{x}) \approx -\sum_{i=1}^{L}\int_{\mathbb{R}^N}\omega_i\cdot\mathcal{N}\left(x;\mu_i,\mathbf{C}_i\right)\cdot\left(\sum_{k=0}^{R}\frac{1}{k!}\left((x-\mu_k)\odot\nabla\right)^k\log f(x)\Big|_{x=\mu_i}\right)\mathrm{d}x$$

This approximation can be evaluated analytically, as solving the integral for component I corresponds to determining the first R central moments of a Gaussian density. Essential parts of the proposed entropy approximation are the derivatives of a Gaussian density.

$$g(x) := \mathcal{N}(x;\mu,\mathbf{C}) = \frac{1}{\sqrt{|2\pi\mathbf{C}|}}e^{-\frac{1}{2}(x-\mu)^{\mathrm{T}}\mathbf{C}^{-1}(x-\mu)}$$

with respect to the vector x. In the following, the first-order up to the third-order derivatives are given. Employing $\frac{\partial}{\partial\mu}(x-\mu)^{\mathrm{T}}\mathbf{C}^{-1}(x-\mu) = -2\mathbf{C}^{-1}(x-\mu)$, the first-order derivative is (see [13] for comparison)

$$\nabla g(x) = \frac{\partial}{\partial x}\mathcal{N}(x;\mu,\mathbf{C}) = \mathbf{C}^{-1}(x-\mu)\cdot g(x).$$

The second-order derivative or Hessian of (5) is given by

$$\begin{aligned}\mathbf{H}(x) &= \frac{\partial^2}{\partial x \partial x^{\mathrm{T}}} \mathcal{N}(x; \mu, \mathbf{C}) \\ &= \mathbf{C}^{-1}(x-\mu)(\nabla g(x))^{\mathrm{T}} - \mathbf{C}^{-1} g(x).\end{aligned}$$

Using the Kronecker product $\otimes$ and matrix-vectorization #, the third-order derivative can be written as

$$\frac{\partial^3}{\partial x \partial x^{\mathrm{T}} \partial x} \mathcal{N}(x; \mu, \mathbf{C}) = \mathbf{H}(x) \otimes \left(\mathbf{C}^{-1}(x-\mu)\right) - \nabla g(x) \otimes \mathbf{C}^{-1} - \#\left(\mathbf{C}^{-1}\right) \cdot (\nabla g(x))^{\mathrm{T}}.$$

Based on these results, for $R = 0$, the zeroth-order Taylor-series expansion is given by

$$\begin{aligned}\mathcal{H}(\boldsymbol{x}) &\approx -\sum_{i=1}^{L} \int_{\mathbb{R}^N} \omega_i \cdot \mathcal{N}\left(x; \mu_i, \mathbf{C}_i\right) \cdot \log g\left(\mu_i\right) \mathrm{d}x \\ &= -\sum_{i=1}^{L} \omega_i \cdot \log f\left(\mu_i\right) \\ &=: \mathcal{H}_0(\boldsymbol{x}).\end{aligned}$$

As the first central moment of a Gaussian density is zero, this approximation of the entropy is identical to the first-order Taylor-series expansion. For $R = 2$:

$$\begin{aligned}\mathcal{H}(x) \approx& \mathcal{H}_0(x) - \sum_{i=1}^{L} \frac{\omega_i}{2} \int_{\mathbf{R}^N} \mathcal{N}\left(x'; \mu_i, \mathbf{C}_i\right) \cdot \mathbf{F}\left(\mu_i\right) \odot \left(x-\mu_i\right)\left(x-\mu_i\right)^{\mathrm{T}} \, \mathrm{d}x \\ =& \mathcal{H}_0(\boldsymbol{x}) - \sum_{i=1}^{L} \frac{\omega_i}{2} \mathbf{F}\left(\mu_i\right) \odot \mathbf{C}_i\end{aligned}$$

with

$$\mathbf{F}(x) = \frac{1}{f(x)} \sum_{j=1}^{L} \omega_j \cdot \mathbf{C}_j^{-1} \left(\frac{1}{f(x)} \left(x-\mu_j\right) \left(\nabla f(x)\right)^{\mathrm{T}} + \left(x-\mu_j\right) \left(\mathbf{C}_j^{-1}\left(x-\mu_j\right)\right)^{\mathrm{T}} - \mathbf{I} \right) \cdot \mathcal{N}\left(x^{\mathrm{T}} \mu_j, \mathbf{C}_j\right)$$

So According to Equation (5), I can estimate the entropy of the action distribution:

$$\mathcal{H}_s \approx -\sum_{k=1}^{K} w_k \log w_k + \sum_{k=1}^{K} w_k \cdot \frac{1}{2} \log\left((2\pi e)^d \left|\Sigma_k\right|\right)$$

where d is the dimensionality of the action space. The estimated entropy $\mathcal{H}$ is obtained by averaging the entropies over a batch of states.

Adaptive Adjustment of Exploration in Diffusion Policy. Similar to maximizing entropy RL, I learn a parameter α based on the estimated entropy. I update this parameter using

$$\alpha \leftarrow \alpha - \beta_\alpha [\hat{\mathcal{H}} - \overline{\mathcal{H}}]$$

where $\overline{\mathcal{H}}$ is target entropy. Finally, I use $\boldsymbol{a} = \boldsymbol{a} + \lambda\alpha \cdot \mathcal{N}(0, \boldsymbol{I})$ to adjust the diffusion policy entropy during training, where λ is a hyperparameter and a is the output of diffusion policy. Additionally, no noise is added during the evaluation phase.

4.9 Part2 Q3

Are you able to implement the algorithm or a modified version of the algorithm?

Yes, I have implemented a modified version of the algorithm. I called this version of algorithm Diffusion Policy Optimization with Entropy Regulator(DPOER).

Actually, my implementation strategy is firstly based on the DACER (Diffusion Actor-Critic with Entropy Regulator) framework and I replace the diffusion part of DACER with the Diffusion Policy MDP from the DPPO paper. The overall framework is implemented in TensorFlow. The specific steps of my implementation include:

Experimental Results: I compare DPOER with other baselines in the Gym and Robomimic tasks.

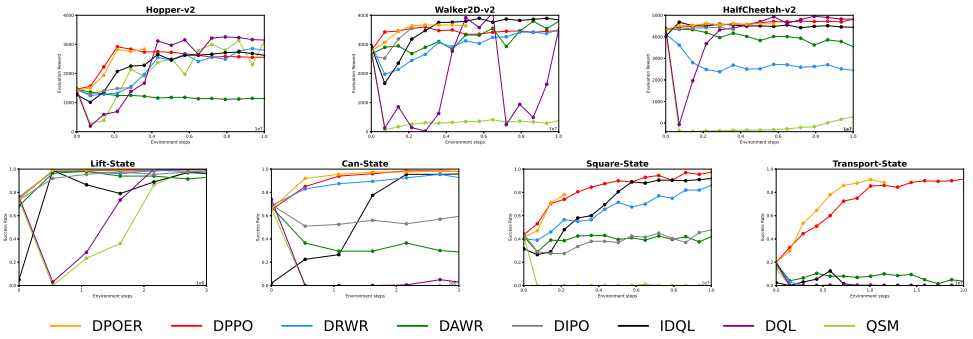


Fig. 7. Comparison between DPOER and other baselines in Gym and Robomimic tasks

4.10 Part2 Q4

Can you please provide a note describing either your algorithm in a) and b) or c) or your own version of diffusion policy SAC?

Inspired by your given idea, I have attempted soft actor critic (SAC) type algorithms on the special Markov Decision Process(MDP) of DPPO. Your idea leads me to develop the final version of the Diffusion Policy Optimization with Entropy Regulator(DPOER) algorithm.

First Attempts: First, similar to the other baselines including DPPO, I try to directly utilize the pretrained policy and apply the Soft Actor Critic(SAC) algorithms to fine-tune it.

- Actor: The diffusion process in DACER is replaced with the pre-trained Diffusion Policy from the DPPO paper.
- Critic: The Critic part still uses the double Q-network structure from DACER. This dual-Q structure helps estimate the state-action value function while avoiding overestimation of Qvalues. The Critic evaluates the actions generated by the Actor for further optimization.
- Entropy Estimation: To ensure the policy remains exploratory, I use Gaussian Mixture Models (GMM) for entropy estimation. Entropy is used to balance the exploration and exploitation to avoid the policy converging to a local optimum too early. Specifically, similar to DACER[30], when estimating the parameters of the GMM, I use the Expectation-Maximization (EM) Algorithm and Taylor-series expansion for entropy approximation.

We utilize the Critic Update Gradient in DACER, which is defined as:

$$\begin{aligned}\nabla_\theta J_{\mathcal{Z}}(\theta) =& \mathbb{E}\left[\nabla_\theta \frac{(y_z - Q_\theta(s,a))^2}{2\sigma_\theta(s,a)^2} + \frac{\nabla_\theta \sigma_\theta(s,a)}{\sigma_\theta(s,a)}\right] \\ =& \mathbb{E}[\underbrace{-\frac{(y_z - Q_\theta(s,a))}{\sigma_\theta(s,a)^2}\nabla_\theta Q_\theta(s,a)}_{\text{mean-related gradient}} \underbrace{-\frac{(y_z - Q_\theta(s,a))^2 - \sigma_\theta(s,a)^2}{\sigma_\theta(s,a)^3}\nabla_\theta \sigma_\theta(s,a)}_{\text{variance-related gradient}}].\end{aligned}$$

where $Q_\theta(s,a)$ is the predicted mean Q-value for the state-action pair (s,a); y_z represents the target Q-value derived from rewards and the next state; $\sigma_\theta(s,a)$ is the predicted standard deviation of the Q-value, capturing the uncertainty in value estimation.

The actions are generated from the diffusion policy. Then the double-Q network estimates the Q values from observations and actions. Finally, as the loss in the Soft Actor Critic(SAC), we take the maximum of the two values and use them as the loss to optimize.

However, even these lines of approches work pretty well in the offline setting. From our experiments, we observe that they are inferior in the setting of offline pretraining to online finetuning. Especially, similar to other baselines, it encounters the performance-dropping problems when I fine-tuned the actor from the pretrained base policy. At the same time, even I clipped the gradients to avoid sudden change of the actor. The actor is still easily updated to a bad performing agent after several epochs.

Diffusion Policy Optimization with Entropy Regulator: DPOER After this attempts, I realize the importance of controlling the stability in the update process of the actor policy.

Why is it hard to apply an SAC directly to the MDP of DPPO? The state in any two contiguous denoising steps $t+1$ and t is the same. Different from other baselines, which only takes the last action steps x_0 to optimize, DPPO utilize the denoising process of the diffusion models as pair-wise spatial-temporal sequential model. During this denoising steps, no actions are take for the agent and the states are the same. This cause difficulty to apply the SAC algorithm directly. Becasue the critic double Q network usually takes both the states and actions as the input. In DPPO, for the same state(with different denoising timestamp t), they regard intermediate denoising steps as actions to optimize the loss. In their implementation, for a sampled action pair from two contiguous denoising steps at time $t+1$ to t, DPPO adopts an GAE to estimate the advantages of action from $t+1$ to t. From the step at $t+1$ to the step t, they estimate the mean and variance of the Normal distribution from the denoising process of DDPM[11]. Then they measure the policy distribution shift between these two denoising timestamps and apply a PPO to control the magnitude of the policy change.

In the given paper Diffusion Actor-Critic with Entropy regulator(DACER)[30] , they address the challenge of entropy estimation in diffusion models by employing a Gaussian Mixture Model (GMM). Inspired by your given idea, I estimate the entropy of the actor(generated by the Diffusion Policy) by a Gaussian Mixture Model (GMM).

In DACER, they add another dimension with 200 and samples 200 times of the original batched action data. For example, given a batch of action data with batch size 128, and action dim 5. They samples actions of 200 samples dim, batch dim 128 and action dim 10 to constitute a $(200, 128, 5)$ tensor to estimate the entropy of the current policy.

Different from DACER, DPPO utilize a lot of data in the denoising process of the diffusion models. They regard the contiguous two actions in denoising step $t+1$ and denoising step t as an action pair. Furthermore, DPPO handles the long-horizon tasks better and they usually a data tensor with a long horizon steps. Thus with a denoising steps of 20, and a horizon steps of 5. Given the same batch size 128 and action dim 10 as the last example in DACER, DPPO could get a tensor of

Algorithm 4: Diffusion Policy Optimization with Entropy Regulator (DPOER)

Pre-train diffusion policy π_θ with offline dataset $\mathcal{D}_{\text{off}}$ using BC loss $\mathcal{L}_{\text{BC}}(\theta)$ Eq. (C.1).
Initialize value function V_ϕ.
for *iteration* $= 1, 2, \ldots$ **do**
  Initialize rollout buffer $\mathcal{D}_{\text{itr}}$.
  $\pi_{\theta_{\text{old}}} = \pi_\theta$.
  for *environment* $= 1, 2, \ldots, N$ *in parallel* **do**
    Initialize state $\bar{s}_{\bar{t}(0,K)} = \left(s_0, a_0^K\right)$ in $\mathcal{M}_{\text{DP}}$.
    for *environment step* t $= 1, \ldots, T$, *denoising step* $k = K-1, \ldots, 0$ **do**
      Sample the next denoised action $\bar{a}_{\bar{t}(t,k)} = a_t^k \sim \pi_{\theta_{\text{old}}}$.
      if $k = 0$ **then**
        then Run a_t^0 in the environment and observe $\bar{R}_{\bar{t}(t,0)}$ and $\bar{s}_{\bar{t}(t+1,K)}$.
      else
        Set $\bar{R}_{\bar{t}(t,k)} = 0$ and $\bar{s}_{\bar{t}(t,k-1)} = \left(s_t, a_t^k\right)$.
      Add $\left(k, \bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}, \bar{R}_{\bar{t}(t,k)}\right)$ to $\mathcal{D}_{\text{itr}}$.
  Compute advantage estimates $A^{\pi_{\theta_{\text{old}}}}\left(s_{\bar{t}(t,k=0)}, a_{\bar{t}(t,k=0)}\right)$ for $\mathcal{D}_{\text{itr}}$ using GAE Eq. (C.2).
  for *update* $= 1, 2, \ldots,$ *num_update* ▷ *Based on replay ratio* N_θ **do**
    for *minibatch* $= 1, 2, \ldots, B$ **do**
      Sample $\left(k, \bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}, \bar{R}_{\bar{t}(t,k)}\right)$ and $A^{\pi_{\theta_{\text{d}}}}\left(s_{\bar{t}(t,k)}, a_{\bar{t}(t,k)}\right)$ from $\mathcal{D}_{\text{itr}}$.
      Compute denoising-discounted advantage $\hat{A}_{\bar{t}(t,k)} = \gamma_{\text{DENOISE}}^k A^{\pi_{\theta_{\text{dat}}}}\left(s_{\bar{t}(t,0)}, a_{\bar{t}(t,0)}\right)$.
      Optimize π_θ using policy gradient loss $\mathcal{L}_\theta$ Eq. (C.3).
      Optimize V_ϕ using value loss $\mathcal{L}_\phi$ Eq. (C.4).
  return converged policy π_θ.

(128, 20, 5, 10). At the same time, considering the 50 parallel environments to generate the data, there are enough sampled data from these environments and denoising process. So I didn't sample more datas. I only constructs the GMM in the original data to estimate the entropy and accelerate the computation. With estimated entropy, we adjust the α parameters dynamically. This optimized α term controls the influence of the entropy term in the original actor loss of DPPO to better trade off the exploration and exploitation.

References

[1] Philip J. Ball, Laura Smith, Ilya Kostrikov, and Sergey Levine. 2023. Efficient Online Reinforcement Learning with Offline Data. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 1577–1594. https://proceedings.mlr.press/v202/ball23a.html

[2] Kevin Black, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. 2024. Training Diffusion Models with Reinforcement Learning. arXiv:2305.13301 [cs.LG] https://arxiv.org/abs/2305.13301

[3] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. 2024. Diffusion Policy: Visuomotor Policy Learning via Action Diffusion. arXiv:2303.04137 [cs.RO] https://arxiv.org/abs/2303.04137

[4] Jingliang Duan, Yang Guan, Shengbo Eben Li, Yangang Ren, Qi Sun, and Bo Cheng. 2022. Distributional Soft Actor-Critic: Off-Policy Reinforcement Learning for Addressing Value Estimation Errors. *IEEE Transactions on Neural Networks and Learning Systems* 33, 11 (Nov. 2022), 6584–6598. https://doi.org/10.1109/tnnls.2021.3082568

[5] Jingliang Duan, Wenxuan Wang, Liming Xiao, Jiaxin Gao, Shengbo Eben Li, Chang Liu, Ya-Qin Zhang, Bo Cheng, and Keqiang Li. 2025. Distributional Soft Actor-Critic With Three Refinements. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2025), 1–12. https://doi.org/10.1109/tpami.2025.3537087

[6] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. 2020. Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO. arXiv:2005.12729 [cs.LG] https://arxiv.org/abs/2005.12729

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv:1801.01290 [cs.LG] https://arxiv.org/abs/1801.01290

[8] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. 2019. Soft Actor-Critic Algorithms and Applications. arXiv:1812.05905 [cs.LG] https://arxiv.org/abs/1812.05905

[9] Philippe Hansen-Estruch, Ilya Kostrikov, Michael Janner, Jakub Grudzien Kuba, and Sergey Levine. 2023. IDQL: Implicit Q-Learning as an Actor-Critic Method with Diffusion Policies. arXiv:2304.10573 [cs.LG] https://arxiv.org/abs/2304.10573

[10] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [stat.ML] https://arxiv.org/abs/1503.02531

[11] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising Diffusion Probabilistic Models. arXiv:2006.11239 [cs.LG] https://arxiv.org/abs/2006.11239

[12] Hengyuan Hu, Suvir Mirchandani, and Dorsa Sadigh. 2024. Imitation Bootstrapped Reinforcement Learning. arXiv:2311.02198 [cs.LG] https://arxiv.org/abs/2311.02198

[13] Marco F. Huber, Tim Bailey, Hugh Durrant-Whyte, and Uwe D. Hanebeck. 2008. On entropy approximation for Gaussian mixture random vectors. In *2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*. 181–188. https://doi.org/10.1109/MFI.2008.4648062

[14] Xiaogang Jia, Denis Blessing, Xinkai Jiang, Moritz Reuss, Atalay Donat, Rudolf Lioutikov, and Gerhard Neumann. 2024. Towards Diverse Behaviors: A Benchmark for Imitation Learning with Human Demonstrations. arXiv:2402.14606 [cs.RO] https://arxiv.org/abs/2402.14606

[15] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Skq89Scxx

[16] Haitong Ma, Tianyi Chen, Kai Wang, Na Li, and Bo Dai. 2025. Soft Diffusion Actor-Critic: Efficient Online Reinforcement Learning for Diffusion Policy. arXiv:2502.00361 [cs.LG] https://arxiv.org/abs/2502.00361

[17] Mitsuhiko Nakamoto, Yuexiang Zhai, Anikait Singh, Max Sobol Mark, Yi Ma, Chelsea Finn, Aviral Kumar, and Sergey Levine. 2023. Cal-QL: Calibrated Offline RL Pre-Training for Efficient Online Fine-Tuning. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=GcEIvidYSw

[18] Alex Nichol and Prafulla Dhariwal. 2021. Improved Denoising Diffusion Probabilistic Models. arXiv:2102.09672 [cs.LG] https://arxiv.org/abs/2102.09672

[19] Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. 2019. Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning. arXiv:1910.00177 [cs.LG] https://arxiv.org/abs/1910.00177

[20] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. 2017. FiLM: Visual Reasoning with a General Conditioning Layer. arXiv:1709.07871 [cs.CV] https://arxiv.org/abs/1709.07871

[21] Frank Permenter and Chenyang Yuan. 2024. Interpreting and Improving Diffusion Models from an Optimization Perspective. arXiv:2306.04848 [cs.LG] https://arxiv.org/abs/2306.04848

[22] Jan Peters and Stefan Schaal. 2007. Reinforcement learning by reward-weighted regression for operational space control. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:11551208

[23] Michael Psenka, Alejandro Escontrela, Pieter Abbeel, and Yi Ma. 2024. Learning a Diffusion Model Policy from Rewards via Q-Score Matching. arXiv:2312.11752 [cs.LG] https://arxiv.org/abs/2312.11752

[24] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv:1505.04597 [cs.CV] https://arxiv.org/abs/1505.04597

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[26] Jiaming Song, Chenlin Meng, and Stefano Ermon. 2022. Denoising Diffusion Implicit Models. arXiv:2010.02502 [cs.LG] https://arxiv.org/abs/2010.02502

[27] Marcel Torne, Anthony Simeonov, Zechu Li, April Chan, Tao Chen, Abhishek Gupta, and Pulkit Agrawal. 2024. Reconciling Reality through Simulation: A Real-to-Sim-to-Real Approach for Robust Manipulation. arXiv:2403.03949 [cs.RO] https://arxiv.org/abs/2403.03949

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[29] Shenzhi Wang, Qisen Yang, Jiawei Gao, Matthieu Gaetan Lin, Hao Chen, Liwei Wu, Ning Jia, Shiji Song, and Gao Huang. 2023. Train Once, Get a Family: State-Adaptive Balances for Offline-to-Online Reinforcement Learning. arXiv:2310.17966 [cs.LG] https://arxiv.org/abs/2310.17966

[30] Yinuo Wang, Likun Wang, Yuxuan Jiang, Wenjun Zou, Tong Liu, Xujie Song, Wenxuan Wang, Liming Xiao, Jiang Wu, Jingliang Duan, and Shengbo Eben Li. 2024. Diffusion Actor-Critic with Entropy Regulator. arXiv:2405.15177 [cs.LG] https://arxiv.org/abs/2405.15177

[31] Zhendong Wang, Jonathan J Hunt, and Mingyuan Zhou. 2023. Diffusion Policies as an Expressive Policy Class for Offline Reinforcement Learning. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=AHvFDPi-FA

[32] Long Yang, Zhixiong Huang, Fenghao Lei, Yucun Zhong, Yiming Yang, Cong Fang, Shiting Wen, Binbin Zhou, and Zhouchen Lin. 2023. Policy Representation via Diffusion Probability Model for Reinforcement Learning. arXiv:2305.13122 [cs.LG] https://arxiv.org/abs/2305.13122

[33] Haichao Zhang, We Xu, and Haonan Yu. 2023. Policy Expansion for Bridging Offline-to-Online Reinforcement Learning. arXiv:2302.00935 [cs.AI] https://arxiv.org/abs/2302.00935