Final Report

Diffusion Policy Policy Optimization GMM-SAC Variant

by

Elton Chun-Chai Li

Submitted in fulfilment of the requirements for

Machine Learning Center of Excellence Summer Associate

in the

2025-2026

Date of submission: 03 February 2025

Table of Contents

1 Abstract

- State the goal: Replicating and extending Diffusion Policy Policy Optimization, specifically addressing the challenge of entropy estimation in diffusion policies for a Soft Actor-Critic (SAC) framework.
- Summarize Part 1 achievements: Successful replication of the DPPO paper using TensorFlow Agent, highlighting key insights gained and clarification of unclear paper sections.
- Highlight Part 2's core contribution: Development and implementation of a novel Diffusion Policy Soft Actor-Critic (DP-SAC) algorithm using a Gaussian Mixture Model (GMM) for entropy estimation and a modified loss function inspired by DACER, addressing the limitations of standard SAC with diffusion policies.
- Briefly mention successful testing and key findings regarding the performance and characteristics of your DP-SAC algorithm.
- Emphasize TensorFlow implementation and rigorous testing throughout.

2 Introduction

Reinforcement learning (RL) has emerged as a powerful paradigm for training intelligent agents across diverse domains, ranging from robotics, autonomous systems and financial modelling. Its ability to learn optimal policies through trial-and-error interaction with an environment has fuelled significant advancements. However, traditional RL approaches often grapple with fundamental challenges, including sample inefficiency, particularly in complex, high-dimensional environments, and limited policy expressiveness, especially when dealing with tasks requiring multi-modal or diverse behaviours. Furthermore, exploration and effective generalization across tasks remain active areas of research.

In response to these challenges, researchers have increasingly explored integrating concepts from other machine learning domains into RL. One particularly promising direction involves the incorporation of diffusion models. Diffusion models, renowned for their exceptional performance in generative modelling tasks such as image and audio synthesis, offer unique advantages when applied to reinforcement learning. Specifically, diffusion policies can effectively model complex, multi-modal action distributions, overcoming the limitations of traditional unimodal Gaussian policies. They also hold the potential to facilitate improved exploration due to their inherent stochasticity and offer a framework for robust generalization. Various approaches have emerged to leverage diffusion models in RL, encompassing their use as planners, policies, and data augmentation tools.

The recent paper "Diffusion Policy Policy Optimization" (DPPO) introduced an innovative approach by demonstrating the effective fine-tuning of pre-trained diffusion policies using policy gradient methods. DPPO leverages the sequential nature of the diffusion denoising process within a Markov Decision Process framework, achieving impressive results in continuous control and robotic manipulation tasks. The paper highlights DPPO's strong performance, training stability, and sim-to-real transfer capabilities. However, DPPO is based on Proximal Policy Optimization (PPO), an on-policy RL algorithm. While on-policy methods like PPO are known for their stability and relative ease of implementation, they can be sample inefficient compared to off-policy approaches and may struggle with exploration in complex, sparse-reward environments. Conversely, off-policy methods, such as Soft Actor-Critic (SAC), offer the potential for greater sample efficiency and more effective exploration through experience replay and entropy maximization, although they can sometimes be more challenging to stabilize.

Motivated by the potential benefits of off-policy learning in conjunction with diffusion policies, this report presents a comprehensive investigation into extending the DPPO framework to an off-policy setting. This work details the replication and thorough analysis of the DPPO algorithm using TensorFlow Agent, addressing unclear aspects of the original paper and rigorously testing the implementation. Furthermore, and as the central contribution, this report introduces a novel Diffusion Policy Soft Actor-Critic (DP-SAC) algorithm. This DP-SAC algorithm tackles the challenge of entropy estimation for diffusion policies by employing a Gaussian Mixture Model (GMM) estimator and incorporates a modified loss function inspired by DACER to effectively integrate GMM entropy estimation within the SAC framework. This report will present a detailed description of the DP-SAC algorithm, along with an analysis of its potential advantages and implementation considerations.

This report is structured as follows, first details the replication of the DPPO algorithm and addresses the questions posed in the assessment. Then presents the proposed DP-SAC algorithm, including its theoretical foundation, implementation details, and a discussion of its potential. Finally, the report concludes with a comprehensive discussion of the findings and potential future directions.

3 Replication and Analysis of DPPO

3.1 Literature Review

The “Diffusion Policy Policy Optimization” (DPPO) paper addresses the challenge of efficiently fine-tuning pre-trained diffusion policies for continuous control and robotic tasks using policy gradient (PG) methods. While diffusion policies are powerful for representing complex behaviours, traditional PG methods were thought to be inefficient for them due to potential high action variance and extended effective horizons. DPPO surprisingly demonstrates that, contrary to this belief, PG methods can be highly effective for fine-tuning diffusion policies when applied within a carefully designed framework.

The core innovation of DPPO lies in the concept of the “Diffusion MDP”. DPPO reformulates the denoising process of a diffusion policy as a Markov Decision Process (MDP). This "Diffusion MDP" is then embedded within the standard Environment MDP of the RL task, creating a two-layer structure. In this “Diffusion MDP”:

- **States ($\tilde{s}_{t(t,k)}$):** Represent the noisy action at each denoising step k, combined with the environment state s_t
- **Actions ($\tilde{a}_{t(t,k)}$):** Are the denoised actions a_t^k generated at each step.
- **Rewards ($R_{\tilde{t}(t,k)}$):** Are primarily obtained from the Environment MDP only at the final denoising step ($k = 0$), corresponding to the execution of action a_t^0 in the environment.

By treating the denoising process as an MDP, DPPO makes it possible to apply Proximal Policy Optimization (PPO), a stable and effective on-policy PG algorithm, to fine-tune diffusion policies. DPPO leverages the fact that the policy within the Diffusion MDP $\pi_\theta\left(\tilde{a}_{t(t,k)} \middle| \tilde{s}_{t(t,k)}\right)$, so can be formulated with a Gaussian likelihood, making it compatible with policy gradient updates.

Furthermore, the DPPO paper identifies several best practices that are crucial for achieving strong performance:

- **Fine-tuning only the last K' denoising steps:** Focusing computation on the final, most critical denoising steps accelerates training.
- **Utilizing DDIM sampling**: Employing DDIM sampling during fine-tuning significantly reduces the number of denoising steps, further enhancing efficiency.
- **Careful diffusion noise scheduling:** Clipping the diffusion noise to a higher minimum value during action sampling, and likelihood evaluation improves exploration and training stability.

Through extensive experiments, DPPO demonstrates strong training stability, high performance, and robust sim-to-real transfer capabilities in various challenging robotic and control tasks. Its key contribution is showing that policy gradient methods, when applied within the Diffusion MDP framework and with the identified best practices, can be a powerful and efficient approach for fine-tuning diffusion policies, overcoming previously held concerns about their inefficiency.

3.2 Replication Methodology

To ensure a robust and objective evaluation, this replication effort was undertaken using TensorFlow, as mandated by the assessment criteria, starting from the publicly available PyTorch codebase of the DPPO paper. The original codebase is well-structured with a modular design, facilitating experimentation and automated logging. The core strategy for replication involved a direct translation of individual modules into TensorFlow, denoted by appending a *"_tf"* suffix to filenames (e.g., diffusion.py became diffusion_tf.py). This approach allowed for the reuse of the well-designed codebase structure while ensuring a TensorFlow implementation.

The process of replication extended beyond mere code translation. To fully leverage TensorFlow's optimization capabilities, it was necessary to adopt TensorFlow-specific operations. This included refactoring data handling to utilize tf.data.Dataset for efficient data pipelines and employing tf.while_loop for control flow within TensorFlow's graph execution. To validate performance against the original PyTorch implementation, TensorFlow Profiler was employed to identify and address potential bottlenecks. This involved analysing hardware resource consumption and optimizing TensorFlow operations to mitigate performance limitations.

Given the breadth of the original DPPO codebase, which encompasses various diffusion-based RL algorithms (e.g., PPO, AWR, DQL) and environments (e.g., Gym, Furniture, D3TIL), the replication scope was deliberately focused. This effort concentrated on replicating the pre-training, fine-tuning, and evaluation workflows specifically for the diffusion policy optimized with PPO. The OpenAI Gym Hopper-v2 environment was chosen as the primary benchmark for replication. The success metric for replication was defined as achieving comparable trends and quantitative values in key log metrics obtained from running the experiments using the original PyTorch codebase.

The structure of the replicated codebase mirrors the original PyTorch repository, maintaining the following directory organization:

Directory	**Description**
agent	Contains customized scripts for experiment execution, including training and evaluation loops.
cfg	Stores configuration files (.yaml) for various experiments.
model	Houses core machine learning modules such as MLPs, Diffusion MLPs, and other neural network components.
script	Provides entry point scripts for initiating different experiments.
util	Offers utility functions for reward scaling, learning rate scheduling, timers, and other helper functionalities.

The following table shows the complete replicated code in TensorFlow version.

Files
agent.dataset.sequence_tf.py
agent.eval.eval_agent_tf.py

agent.eval.eval_diffusion_agent_tf.py
agent.finetune.train_agent_tf.py
agent.finetune.train_ppo_agent_tf.py
agent.finetune.train_ppo_diffusion_agent_tf.py
agent.finetune.train_sac_diffusion_agent_tf.py
agent.pretrain.train_agent_tf.py
agent.pretrain.train_diffusion_agent_tf.py
cfg.gym.eval.hopper-v2.eval_sac_diffusion_mlp_tf.yaml
cfg.gym.eval.hopper-v2.eval_ppo_diffusion_mlp_tf.yaml
cfg.gym.finetune.hopper-v2.ft_ppo_diffusion_mlp_tf.yaml
cfg.gym.finetune.hopper-v2.ft_sac_diffusion_mlp_tf.yaml
cfg.gym.pretrain. hopper-medium-v2.pre_diffusion_mlp_tf.yaml
model.common.ciritc_tf.py
model.common.mlp_tf.py
model.common.modules_tf.py
model.diffusion.diffusion_ppo_tf.py
model.diffusion.diffusion_sac_tf.py
model.diffusion.diffusion_tf.py
model.diffusion.diffusion_vpg_tf.py
model.diffusion.mlp_diffusion_tf.py
model.diffusion.modules_tf.py
model.diffusion.sampling_tf.py
script.run_tf.py
util.scheduler_tf.py

3.3 Replication Results

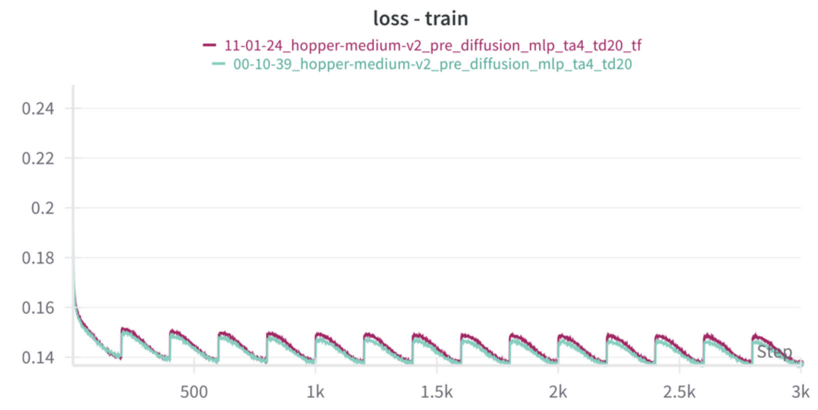
To validate the replication, we evaluated the training loss metrics of our TensorFlow implementation against the reference PyTorch version during both pre-training and fine-tuning phases, using identical configurations.

3.3.1 Pre-Training

Pre-training, a crucial initial phase, involves training a diffusion policy model (DiffusionModel with DiffusionMLP) on expert demonstration data. This step is essential for providing a strong prior for subsequent Reinforcement Learning (RL) fine-tuning. Pre-training offers several benefits: it provides a meaningful initialization, enhances sample efficiency and convergence during fine-tuning, improves training stability, and facilitates structured exploration by capturing complex action distributions from expert examples.

Pre-training is achieved through supervised learning, minimizing the Mean Squared Error (MSE) loss (implicitly within `DiffusionModel.p_losses` function) between the model's predicted noise and the noise added to expert actions. Our TensorFlow implementation mirrored the PyTorch version in network architecture (DiffusionMLP), optimization (AdamW), learning rate schedule (Cosine Annealing with Warmup Restarts), and batch training procedures.

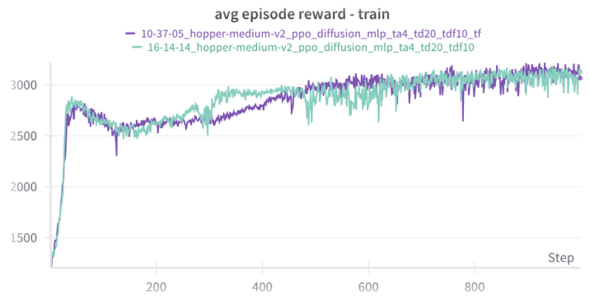
As shown in the training loss plot (purple: TensorFlow version, light blue: PyTorch version), the training loss curves for both implementations are nearly identical. The characteristic *"zip-zap"* shape in the loss is attributed to the "CosineAnnealingWarmupRestarts" learning rate scheduler, designed to aid in escaping local optima. This close alignment of training loss indicates a successful replication of the pre-training phase in TensorFlow.

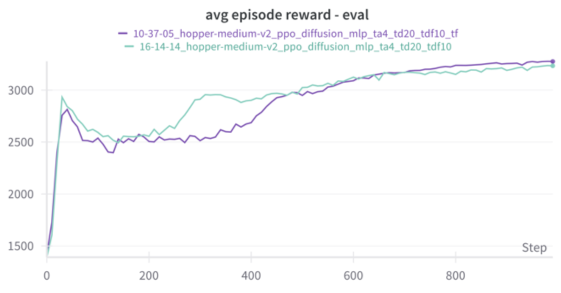


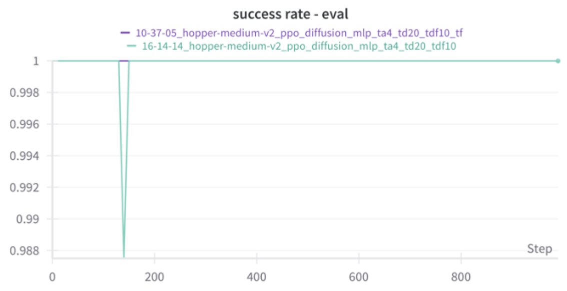
3.3.2 Fine-Tuning

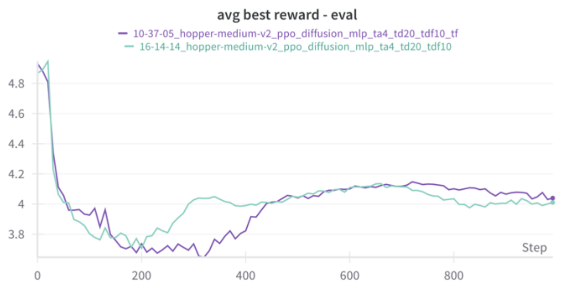
Following the crucial pre-training phase, the DPPO methodology employs a fine-tuning stage to optimize the pre-trained diffusion policy for reward maximization within the task environment. While pre-training establishes a robust prior based on expert demonstrations, fine-tuning leverages Reinforcement Learning to adapt the policy to the specifics of the task reward structure and environment dynamics. This is achieved using the Proximal Policy Optimization (PPO) algorithm, a stable on-policy method, with a carefully designed loss function tailored for the Diffusion MDP framework.

To validate the successful replication of the fine-tuning process, we compared key performance metrics of our TensorFlow implementation against the original PyTorch version. The core metrics for evaluating fine-tuning success are the **average reward per episode**, **success rate**, and **average best reward per episode**, as these directly reflect the agent's learning progress and task proficiency within the environment. These metrics were meticulously logged and compared throughout the fine-tuning process for both implementations.









As depicted in the comparative plots (purple: TensorFlow version, light blue: PyTorch version), the training and evaluation curves for average reward per episode exhibit a strikingly similar trend and achieve comparable quantitative values at step 1000. This close alignment is further reinforced by the near-identical trajectories observed in the success rate and average best reward per episode plots. Both the TensorFlow and PyTorch implementations demonstrate consistent upward trends in these key performance indicators, indicating successful learning and task improvement during fine-tuning.

The highly consistent trends and quantitative alignment across all key performance metrics—average reward, success rate, and average best reward—strongly substantiate the successful replication of the DPPO fine-tuning phase within TensorFlow. The TensorFlow implementation demonstrably mirrors the learning behaviour and performance characteristics of the original PyTorch codebase, validating the accurate translation and optimization of this critical algorithm component.

The highly consistent trends and quantitative alignment across all key performance metrics—average reward, success rate, and average best reward—strongly substantiate the successful replication of the DPPO fine-tuning phase within TensorFlow. The TensorFlow implementation demonstrably mirrors the learning behaviour and performance characteristics of the original PyTorch codebase, validating the accurate translation and optimization of this critical algorithm component.

3.4 Discussion

3.4.1 High-Level Pseudocode

To provide a complete understanding of the Diffusion Policy Policy Optimization (DPPO) algorithm, it's essential to clarify both its action generation mechanism and the fine-tuning process that enables policy improvement. The pseudocode below details DPPO's action generation, highlighting the iterative denoising process and the strategic roles of the frozen and fine-tuned actor networks. Further elaboration on the fine-tuning loop, which leverages policy gradient optimization within the Diffusion MDP framework, will follow in subsequent sections. This combined explanation aims to offer a more detailed and step-by-step view of DPPO's algorithmic mechanics than is available in the original paper.

3.4.1.1 DPPO Denoising Process for Action Generation

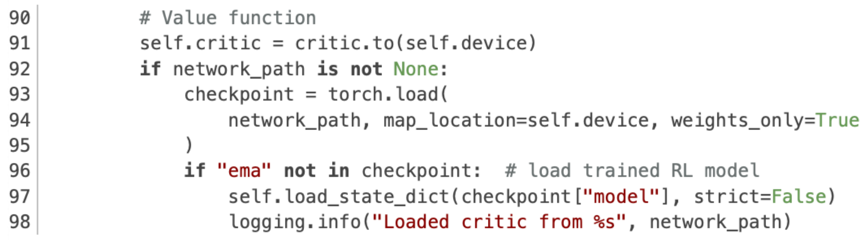
```
# DPPO Denoising Process for Action Generation (High-Level Pseudocode)
# Emphasizing Frozen and Fine-Tuned Actors

Function: GenerateAction(environment_state)

Input:
    environment_state: Current state of the environment (s_t)

Output:
    final_action: Action to be executed in the environment (a_t^0)

Initialization:
    frozen_actor_network: Pre-trained diffusion network (weights are frozen)
    fine_tuned_actor_network: Fine-tuned diffusion network (weights are updated during RL)
    denoising_steps_fine_tune_start:  Step index from which fine-tuning starts
    maximum_denoising_steps: Total number of denoising steps (K)

Steps:

    # 1. Initialize Noisy Action
    current_action = Sample random noise from a Gaussian distribution (a_t^K ~ N(0, I))
    denoising_step_index = maximum_denoising_steps # Start at maximum denoising step (K)

    # 2. Start Denoising Loop (Iterate from K-1 down to 0)
    For each denoising step k from maximum_denoising_steps - 1 down to 0:

        # 2.1. Prepare Inputs for Diffusion Network
        network_input = {
            "noisy_action": current_action,  # Action from the previous denoising step (a_t^{k+1})
            "denoising_timestep": denoising_step_index, # Current denoising step index (k+1)
            "environment_condition": environment_state # Current environment state (s_t)
        }

        # 2.2. Select Actor Network based on Denoising Step
        If denoising_step_index >= denoising_steps_fine_tune_start:
            # Use FINE-TUNED Actor Network for later (fine-tuning) steps
            actor_network_to_use = fine_tuned_actor_network
        Else:
            # Use FROZEN Actor Network for earlier steps
            actor_network_to_use = frozen_actor_network

        # 2.3. Run Selected Diffusion Network (Actor Network)
        guidance_information = Run actor_network_to_use(network_input)
        # guidance_information is typically predicted noise (epsilon)

        # 2.4. Apply Denoising Formula to Refine Action
        # Use the guidance information and denoising schedule to refine the action
        # Example (DDPM-like step):
        #   mean, variance = CalculateDenoisingDistribution(current_action, guidance_information, denoising_
        #   next_action = Sample from Gaussian distribution N(mean, variance)
        next_action = Apply denoising formula using current_action and guidance_information
        # next_action is now less noisy (a_t^k)

        # 2.5. Update Current Action
        current_action = next_action
        denoising_step_index = denoising_step_index - 1 # Move to the next denoising step

    # 3. Final Action
    final_action = current_action # Action after all denoising steps (a_t^0)

    Return final_action

```

3.4.1.2 DPPO fine-tuning loop

```
# DPPO Fine-Tuning Loop (High-Level Pseudocode)

Initialize:
    - Pre-trained Diffusion Policy (Actor)
    - Critic Value Function
    - PPO hyperparameters (clip ratio, GAE lambda, etc.)
    - Environment

For each iteration in range(number of training iterations):
    Set policy to evaluation mode (for data collection, but can be training mode as well)

    # 1. Data Collection (Rollout)
    Initialize empty buffer for experiences (observations, actions, rewards, etc.)
    Reset environment for each parallel environment

    For each step in range(number of steps per iteration):
        For each parallel environment:
            Get current observation from environment

            # Sample action from Diffusion Policy (Denoising process)
            Sample action from Diffusion Policy given observation
            (This involves running the reverse diffusion process for a number of steps)
            Store the chain of denoised actions (for logprob calculation later)

            Apply action to environment
            Get next observation, reward, done flag from environment

            Store experience in buffer:
                - Current observation
                - Action taken
                - Reward received
                - Next observation
                - Done flag
                - Chain of denoised actions (for logprob calculation)

            Update current observation to next observation

    Set policy to training mode

    # 2. Advantage Calculation
    Calculate Returns and Advantages for collected experiences:
        - Estimate Value function for all states in the collected trajectories
        - Calculate TD-residuals (temporal difference errors)
        - Compute Advantages using Generalized Advantage Estimation (GAE) or similar method
        - Normalize Advantages (optional)

    # 3. Policy and Critic Updates (PPO Optimization)
    For each update epoch in range(number of update epochs):
        Shuffle collected experiences
        For each mini-batch of experiences:
            Extract mini-batch of data (observations, actions, advantages, old log probabilities, returns, old values)

            # Calculate loss
            Calculate Policy Loss (PPO Clipped Objective):
                - Calculate new log probabilities of actions under the current policy (through Diffusion MDP)
                - Calculate probability ratio (current policy prob / old policy prob)
                - Calculate clipped and unclipped policy loss terms
                - Policy Loss = Max(unclipped policy loss, clipped policy loss)

            Calculate Value Loss (MSE between predicted value and returns):
                - Calculate value function for current observations
                - Value Loss = Mean Squared Error(predicted values, returns) (optionally clipped)

            Calculate Total Loss = Policy Loss + Value Loss + Entropy Bonus (optional) + BC Loss (optional)

            # Update networks
            Zero out gradients of Actor and Critic optimizers
            Calculate gradients of Total Loss
            Clip gradients (optional)
            Update Actor and Critic networks using their respective optimizers

            Check for early stopping based on KL divergence (optional)

        Update learning rate schedulers (if used)
        Anneal diffusion parameters (e.g., minimum noise level, denoising steps - if annealing schedule is used)

    # 4. Logging and Evaluation
    Evaluate policy performance (e.g., calculate average reward, success rate)
    Log training metrics (losses, rewards, KL divergence, clip fraction, etc.) to console/wandb/tensorboard
    Save model checkpoint (periodically or at the end of training)
    Render episode videos (periodically)

    Increment iteration counter
Training Finished.  Model saved. Results logged.
```

3.4.2 Potential Mistake

From the original implementation of the code *dppo.py* in torch version, it will load the model checkpoint if the network path is provided. At line 97, if the ema is not in the checkpoint, it will load the model to the current instance. However, its very misleading to output “Loaded critic from …” because its not just the critic instead its loading the whole mode, which is expected to includes other object, like actor, actor_ft.



```
90          # Value function
91          self.critic = critic.to(self.device)
92          if network_path is not None:
93              checkpoint = torch.load(
94                  network_path, map_location=self.device, weights_only=True
95              )
96              if "ema" not in checkpoint:  # load trained RL model
97                  self.load_state_dict(checkpoint["model"], strict=False)
98                  logging.info("Loaded critic from %s", network_path)
```

3.4.3 Paper Review

As a reviewer, I would recommend acceptance of the Diffusion Policy Policy Optimization (DPPO) paper, leaning towards a strong accept. This work presents a novel and significant contribution by introducing the Diffusion MDP framework, which ingeniously reframes the diffusion denoising process as a Markov Decision Process, thereby enabling the application of policy gradient methods.

The paper's strengths are manifold. Firstly, the Diffusion MDP abstraction itself is a conceptually novel and valuable contribution. Secondly, DPPO demonstrates a remarkably effective integration of diffusion policies with the PPO algorithm, skilfully adapting PPO's components to the Diffusion MDP structure and yielding a practical and high-performing algorithm. Thirdly, the empirical validation is extensive and compelling, showcasing DPPO's superior performance across a diverse set of challenging benchmarks and in sim-to-real transfer scenarios. Furthermore, the paper effectively addresses and refutes the prior conjecture regarding the inefficiency of policy gradients for diffusion policies. Finally, the evidence for structured exploration and policy robustness adds further merit.

While the paper is generally well-written, some weaknesses warrant consideration. The rationale for selective fine-tuning, while effective, lacks broad theoretical justification beyond PPO-style algorithms. DPPO's reliance on pre-training data raises questions about its real-world applicability in data-scarce scenarios. The inherent computational cost of diffusion models, though mitigated, remains a factor. Additionally, a deeper exploration of DPPO algorithm variations could further strengthen the analysis. However, these weaknesses are relatively minor in light of the paper's significant contributions. This paper provides a good initiative for this area of research.

In conclusion, the novelty of the Diffusion MDP framework, the robust empirical validation, and the convincing demonstration of efficient fine-tuning for diffusion policies make this paper a valuable and noteworthy contribution deserving of acceptance at a major conference.

4 Diffusion Policy with SAC

4.1 Possibility of Constructing SAC on Diffusion Policy MDP

As explored in the DPPO paper, Policy Gradient methods can be effectively applied to Diffusion Policy MDP. However, DPPO, being an on-policy algorithm based on PPO, inherently exhibits limitations in sample efficiency due to its on-policy nature. Off-policy algorithms, such as Soft Actor-Critic (SAC), offer a compelling alternative, potentially improving sample efficiency through experience replay and enhancing exploration via entropy maximization. While DPPO explored offline RL baselines like Advantage-Weighted Regression (AWR) and Deep Q-Learning (DQL), it did not investigate the integration of SAC with the Diffusion Policy MDP. Therefore, a natural extension is to consider the feasibility of constructing a SAC-type algorithm tailored for the Diffusion Policy MDP framework.

4.2 Entropy Estimation for Diffusion Policies

A core component of Soft Actor-Critic (SAC) is the principle of "softness", achieved through entropy regularization. Early SAC formulations incorporated entropy into the objective function to encourage exploration, with the temperature parameter (α) controlling the balance between reward maximization and entropy. Initially, α was often treated as a fixed hyperparameter, requiring manual tuning. However, to dynamically adapt the exploration-exploitation trade-off, SAC evolved to incorporate learnable temperature. This refinement transformed the objective into constrained optimization: maximizing reward while maintaining policy entropy above a desired threshold. In essence, contemporary SAC algorithms, often referred to as SAC with learnable temperature, automatically adjust exploration levels during learning.

In standard SAC implementations for continuous action spaces, a Gaussian policy is typically employed, parameterized by a neural network outputting the mean and standard deviation: $\pi(s|a) = \mathcal{N}(a; \mu(s), \sigma(s))$. For such Gaussian policies, a closed-form expression for entropy exists, facilitating straightforward integration into the SAC loss function. However, directly applying this SAC framework to the Diffusion Policy MDP presents a significant challenge. Diffusion policies, unlike simple Gaussian policies, generate actions through a complex, iterative denoising process. This process lacks a closed-form probability density function, rendering direct entropy calculation impossible. Therefore, estimating the entropy of a diffusion policy is a critical hurdle in constructing a Diffusion Policy SAC algorithm.

4.3 GMM Estimation and DACER as an Alternative Approach

To address the challenge of entropy estimation for diffusion policies in SAC, the paper "Diffusion Actor-Critic with Entropy Estimation" proposes using a Gaussian Mixture Model (GMM) estimator. This approach fits a GMM to actions sampled from the diffusion policy, approximating its complex distribution and enabling entropy estimation. While GMM entropy estimation solves the quantification problem, directly incorporating this estimated entropy into the standard SAC loss function alongside a diffusion policy presents significant hurdles.

Problem 1: Loss of Exploration Incentive. SAC's objective function balances reward and entropy to encourage exploration. If we directly substitute the GMM-estimated entropy as a constant value, the entropy term becomes independent of policy parameters. This removes the gradient signal that encourages exploration, leading to deterministic policies and potential local optima. The crucial exploration incentive of SAC's entropy term is lost.

Problem 2: Non-Differentiable GMM Estimation. The GMM entropy estimator relies on the iterative Expectation-Maximization (EM) algorithm, which is inherently non-differentiable. SAC relies on efficient gradient-based optimization. Backpropagating through the complex, iterative GMM fitting process is computationally infeasible and unstable, preventing end-to-end learning. Directly incorporating GMM entropy into the SAC objective breaks the gradient flow and renders optimization impractical.

Therefore, directly using GMM-estimated entropy in the standard SAC objective is not ideal. DACER offers a solution by using the GMM entropy estimate to guide a separate, differentiable exploration mechanism (learned temperature-controlled noise injection), thus incorporating entropy regularization practically. This necessitates a modified loss function and algorithm, inspired by DACER, for a viable Diffusion Policy SAC implementation with GMM entropy estimation.

4.4 Implementation Details

This describes a modified version of a Diffusion Policy Soft Actor-Critic (DP-SAC) algorithm, drawing inspiration from the "Diffusion Actor-Critic with Entropy Estimation" (DACER) paper to address the challenges of constructing an off-policy SAC algorithm for Diffusion Policy MDPs. Building upon the DPPO framework, this algorithm aims to enhance sample efficiency and exploration compared to on-policy methods by adopting an off-policy SAC approach. Key modifications are incorporated, inspired by DACER, to facilitate entropy estimation for diffusion policies and adapt the SAC loss function accordingly.

Algorithm Design - Key Modifications Inspired by DACER:

This DP-SAC algorithm incorporates several key modifications inspired by the DACER approach, adapted for the Diffusion Policy MDP context:

1. **Modified Loss Function Calculation**: Following DACER, the standard SAC loss function is modified. For example: Instead of directly incorporating a closed-form entropy term, the loss function is adapted to guide a learned temperature and incorporate a GMM-related term. This modification is crucial to address the non-differentiable nature of direct diffusion policy entropy estimation and to maintain an exploration incentive.
2. **Entropy/Noise Injection at Diffusion Policy Output**: Consistent with DACER's approach, noise is injected at the output of the diffusion policy. This noise is not simply fixed Gaussian noise, but its variance is adaptively controlled by a learned temperature parameter. Similar to DACER, Gaussian noise is added to the action sampled from the diffusion policy. The standard deviation of this noise is regulated by a learnable temperature parameter, which is adjusted based on the GMM-estimated entropy, allowing for dynamic control of exploration.

3. **GMM Entropy Estimator**: To address the challenge of estimating diffusion policy entropy, a Gaussian Mixture Model (GMM) estimator is employed. During training, actions are sampled from the diffusion policy. A GMM is then fitted to these sampled actions to approximate the policy's distribution.

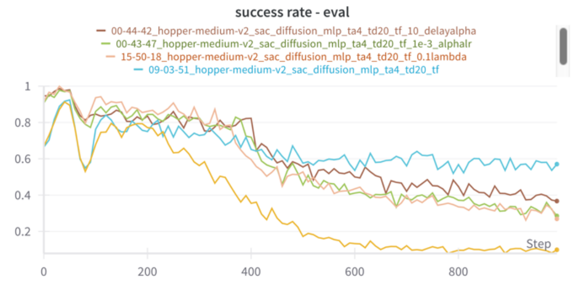
To maintain a controllable experimental setup and manage implementation complexity, some aspects of DACER and DPPO were simplified in this implementation:

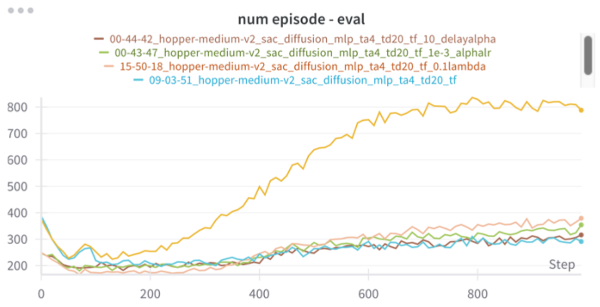
- **Critic Network**: A simple neural network is used for the critic, instead of DACER's Distributed Q-Network.
- **Q-Value Averaging**: The Exponential Moving Average (EMA)-like approach for Q-values, as used in DACER, is not implemented. Standard target networks are used for stabilization.
- **Full Denoising Step Fine-tuning:** Unlike DPPO's ability to fine-tune only the last few denoising steps (achieved through specific PPO loss redesign), this DP-SAC algorithm fine-tunes the entire diffusion model across all denoising steps. This is due to the nature of the SAC loss function and the focus on adapting SAC directly to the Diffusion Policy MDP, rather than redesigning the loss for partial fine-tuning.

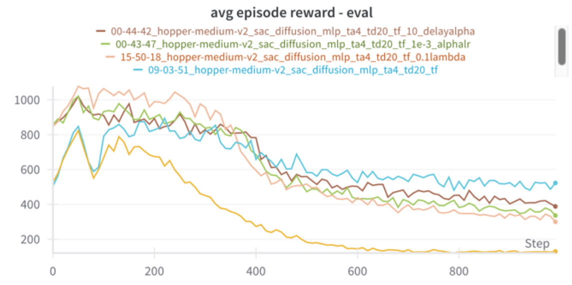
Hyperparameter Tuning

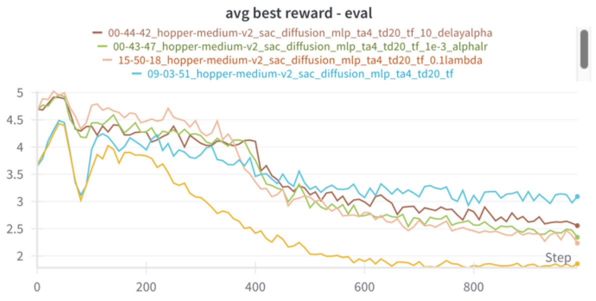
Hyperparameter tuning was performed to evaluate the algorithm's performance under various settings (Delay Alpha, Alpha lr, lambda, Target entropy). The table below summarizes the performance observed with different hyperparameter combinations:

Plot	**Delay Alpha**	**Alpha lr**	**lambda**	**Target entropy**
Blue	5	3e-2	0.15	-0.9
Brown	10	3e-2	0.15	-0.9
Green	5	1e-3	0.15	-0.9
Pink	5	3e-2	0.1	-0.9
Yellow	5	3e-2	0.15	-0.3









Here are some insights drawn from the result.

- **Hyperparameter Sensitivity:** Further hyperparameter optimization may be necessary to achieve optimal performance. More extensive tuning of parameters specific to DACER and GMM integration, beyond those explored, could be beneficial.
- **Impact of Simplified Critic:** The simplification of the critic network (not using Distributed Q-Network) may have limited performance. Exploring the Distributed Q-Network architecture from DACER could be a direction for future improvement.
- **Replay Buffer and Sampling:** The replay buffer implementation and sampling strategy might need refinement for optimal off-policy learning in this context.
- **Full vs. Partial Fine-tuning:** Fine-tuning the entire diffusion model across all denoising steps, instead of a partial fine-tuning approach like DPPO, may have implications for training efficiency or stability in the SAC setting. Further investigation into adapting a partial fine-tuning strategy for DP-SAC could be valuable.

Conclusion

The modified Diffusion Policy Soft Actor-Critic (DP-SAC) algorithm, incorporating GMM entropy estimation and DACER-inspired loss function modifications to address the challenges of applying SAC to Diffusion Policy MDPs. While preliminary results show some promise, further research and refinement, particularly in hyperparameter tuning, critic architecture, and exploration strategies, are needed to fully realize the potential of this approach and achieve robust and high-performance off-policy learning with diffusion policies.

5 Testing Plan

To rigorously validate the reliability and accuracy of our TensorFlow DPPO implementation, we have adopted a comprehensive, multi-layered testing strategy. This strategy encompasses unit tests, regression tests, and integration tests, designed to systematically verify both individual component functionality and their seamless interaction within the complete system.

Unit Tests: Component-Level Verification

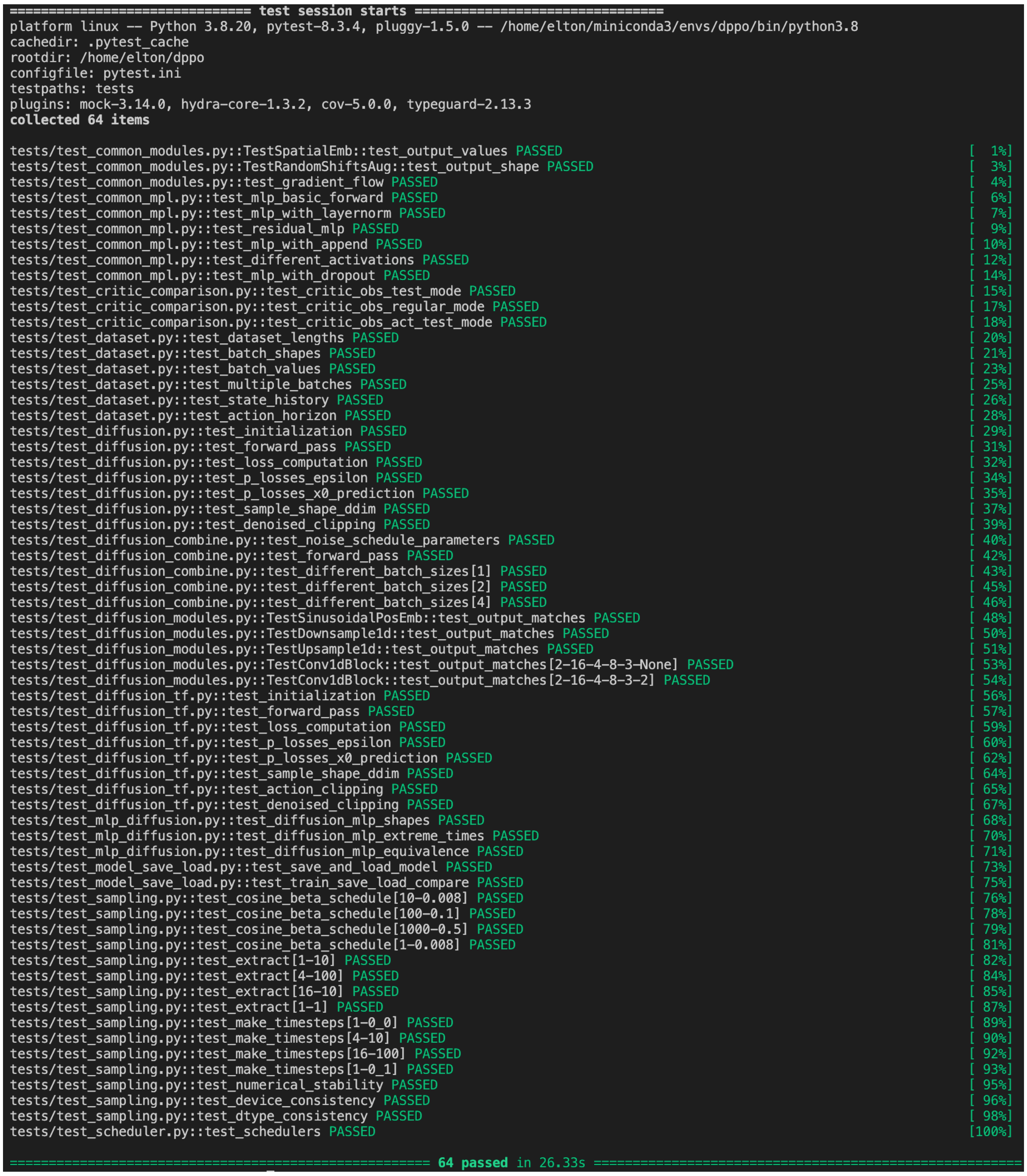
Unit tests are designed to ensure the functional correctness and numerical stability of fundamental DPPO components in isolation. These tests meticulously examine:

- **Sampler Module:** Unit tests for the sampler verify the correct generation of action samples, ensuring outputs are within expected numerical ranges and adhere to defined specifications. These tests focus on computational efficiency and accurate implementation of sampling procedures.
- **Cosine Scheduler:** Unit tests validate the cosine scheduler's accurate generation of beta values across the timestep range, confirming adherence to the cosine annealing schedule and correct parameterization.
- **Policy Network:** Unit tests for the policy network verify the successful forward pass for various input dimensions and batch sizes, and validate that the network produces outputs of the expected shape and data type, ensuring basic network functionality.

Integration Tests: System-Level Validation

Integration tests are crucial for validating the harmonious interaction between different DPPO components and the overall system architecture. These tests are designed to verify:

- **DPPO Architecture Assembly**: Integration tests confirm the correct assembly and instantiation of the complete DPPO architecture within TensorFlow Agent, ensuring all components are properly connected and initialized.
- **End-to-End Training Pipeline Functionality**: These tests validate the complete training pipeline, from data loading and environment interaction to forward and backward passes and optimization steps. They ensure the training loop executes without errors and that training metrics are correctly tracked.
- **Checkpoint Management Integrity**: Integration tests verify the checkpoint saving and loading mechanisms, ensuring the model state can be reliably saved and restored, preserving training progress.



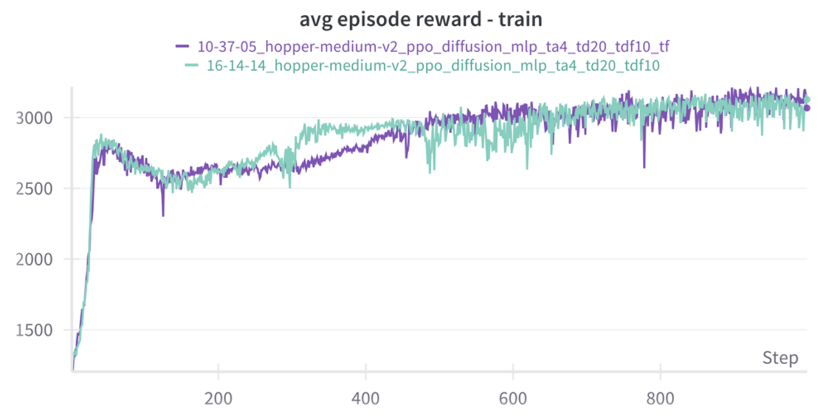
```
================================ test session starts ================================
platform linux -- Python 3.8.20, pytest-8.3.4, pluggy-1.5.0 -- /home/elton/miniconda3/envs/dppo/bin/python3.8
cachedir: .pytest_cache
rootdir: /home/elton/dppo
configfile: pytest.ini
testpaths: tests
plugins: mock-3.14.0, hydra-core-1.3.2, cov-5.0.0, typeguard-2.13.3
collected 64 items

tests/test_common_modules.py::TestSpatialEmb::test_output_values PASSED                                              [  1%]
tests/test_common_modules.py::TestRandomShiftsAug::test_output_shape PASSED                                          [  3%]
tests/test_common_modules.py::test_gradient_flow PASSED                                                              [  4%]
tests/test_common_mpl.py::test_mlp_basic_forward PASSED                                                              [  6%]
tests/test_common_mpl.py::test_mlp_with_layernorm PASSED                                                             [  7%]
tests/test_common_mpl.py::test_residual_mlp PASSED                                                                   [  9%]
tests/test_common_mpl.py::test_mlp_with_append PASSED                                                                [ 10%]
tests/test_common_mpl.py::test_different_activations PASSED                                                          [ 12%]
tests/test_common_mpl.py::test_mlp_with_dropout PASSED                                                               [ 14%]
tests/test_critic_comparison.py::test_critic_obs_test_mode PASSED                                                    [ 15%]
tests/test_critic_comparison.py::test_critic_obs_regular_mode PASSED                                                 [ 17%]
tests/test_critic_comparison.py::test_critic_obs_act_test_mode PASSED                                                [ 18%]
tests/test_dataset.py::test_dataset_lengths PASSED                                                                   [ 20%]
tests/test_dataset.py::test_batch_shapes PASSED                                                                      [ 21%]
tests/test_dataset.py::test_batch_values PASSED                                                                      [ 23%]
tests/test_dataset.py::test_multiple_batches PASSED                                                                  [ 25%]
tests/test_dataset.py::test_state_history PASSED                                                                     [ 26%]
tests/test_dataset.py::test_action_horizon PASSED                                                                    [ 28%]
tests/test_diffusion.py::test_initialization PASSED                                                                  [ 29%]
tests/test_diffusion.py::test_forward_pass PASSED                                                                    [ 31%]
tests/test_diffusion.py::test_loss_computation PASSED                                                                [ 32%]
tests/test_diffusion.py::test_p_losses_epsilon PASSED                                                                [ 34%]
tests/test_diffusion.py::test_p_losses_x0_prediction PASSED                                                          [ 35%]
tests/test_diffusion.py::test_sample_shape_ddim PASSED                                                               [ 37%]
tests/test_diffusion.py::test_denoised_clipping PASSED                                                               [ 39%]
tests/test_diffusion_combine.py::test_noise_schedule_parameters PASSED                                               [ 40%]
tests/test_diffusion_combine.py::test_forward_pass PASSED                                                            [ 42%]
tests/test_diffusion_combine.py::test_different_batch_sizes[1] PASSED                                                [ 43%]
tests/test_diffusion_combine.py::test_different_batch_sizes[2] PASSED                                                [ 45%]
tests/test_diffusion_combine.py::test_different_batch_sizes[4] PASSED                                                [ 46%]
tests/test_diffusion_modules.py::TestSinusoidalPosEmb::test_output_matches PASSED                                    [ 48%]
tests/test_diffusion_modules.py::TestDownsample1d::test_output_matches PASSED                                        [ 50%]
tests/test_diffusion_modules.py::TestUpsample1d::test_output_matches PASSED                                          [ 51%]
tests/test_diffusion_modules.py::TestConv1dBlock::test_output_matches[2-16-4-8-3-None] PASSED                        [ 53%]
tests/test_diffusion_modules.py::TestConv1dBlock::test_output_matches[2-16-4-8-3-2] PASSED                           [ 54%]
tests/test_diffusion_tf.py::test_initialization PASSED                                                               [ 56%]
tests/test_diffusion_tf.py::test_forward_pass PASSED                                                                 [ 57%]
tests/test_diffusion_tf.py::test_loss_computation PASSED                                                             [ 59%]
tests/test_diffusion_tf.py::test_p_losses_epsilon PASSED                                                             [ 60%]
tests/test_diffusion_tf.py::test_p_losses_x0_prediction PASSED                                                       [ 62%]
tests/test_diffusion_tf.py::test_sample_shape_ddim PASSED                                                            [ 64%]
tests/test_diffusion_tf.py::test_action_clipping PASSED                                                              [ 65%]
tests/test_diffusion_tf.py::test_denoised_clipping PASSED                                                            [ 67%]
tests/test_mlp_diffusion.py::test_diffusion_mlp_shapes PASSED                                                        [ 68%]
tests/test_mlp_diffusion.py::test_diffusion_mlp_extreme_times PASSED                                                 [ 70%]
tests/test_mlp_diffusion.py::test_diffusion_mlp_equivalence PASSED                                                   [ 71%]
tests/test_model_save_load.py::test_save_and_load_model PASSED                                                       [ 73%]
tests/test_model_save_load.py::test_train_save_load_compare PASSED                                                   [ 75%]
tests/test_sampling.py::test_cosine_beta_schedule[10-0.008] PASSED                                                   [ 76%]
tests/test_sampling.py::test_cosine_beta_schedule[100-0.1] PASSED                                                    [ 78%]
tests/test_sampling.py::test_cosine_beta_schedule[1000-0.5] PASSED                                                   [ 79%]
tests/test_sampling.py::test_cosine_beta_schedule[1-0.008] PASSED                                                    [ 81%]
tests/test_sampling.py::test_extract[1-10] PASSED                                                                    [ 82%]
tests/test_sampling.py::test_extract[4-100] PASSED                                                                   [ 84%]
tests/test_sampling.py::test_extract[16-10] PASSED                                                                   [ 85%]
tests/test_sampling.py::test_extract[1-1] PASSED                                                                     [ 87%]
tests/test_sampling.py::test_make_timesteps[1-0_0] PASSED                                                            [ 89%]
tests/test_sampling.py::test_make_timesteps[4-10] PASSED                                                             [ 90%]
tests/test_sampling.py::test_make_timesteps[16-100] PASSED                                                           [ 92%]
tests/test_sampling.py::test_make_timesteps[1-0_1] PASSED                                                            [ 93%]
tests/test_sampling.py::test_numerical_stability PASSED                                                              [ 95%]
tests/test_sampling.py::test_device_consistency PASSED                                                               [ 96%]
tests/test_sampling.py::test_dtype_consistency PASSED                                                                [ 98%]
tests/test_scheduler.py::test_schedulers PASSED                                                                      [100%]

====================================================== 64 passed in 26.33s ======================================================
```

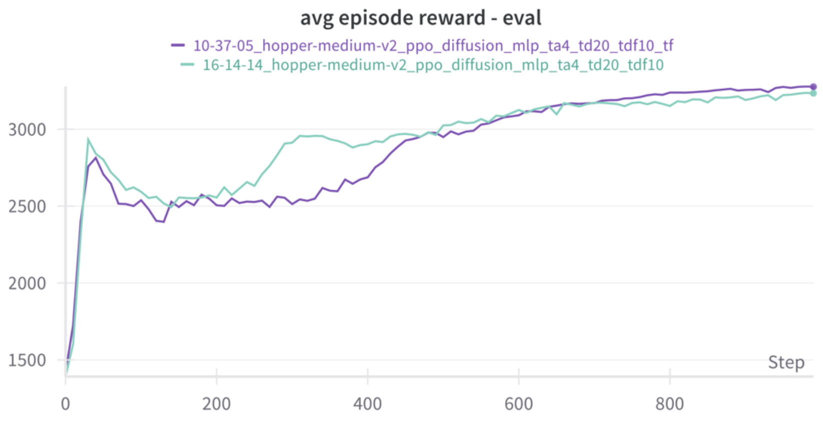
6 Appendix

6.1 Appendix 1 – Complete DPPO fine-tuning log

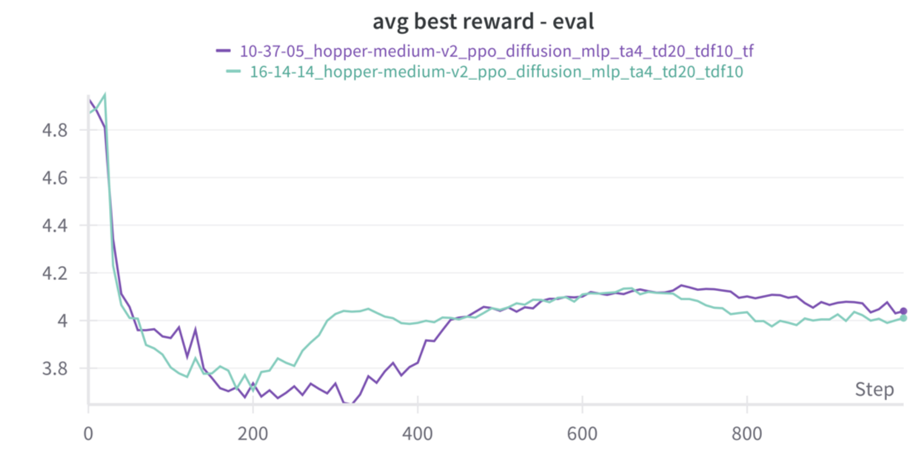
Performance Metrics

- **avg episode reward – train / avg episode reward – eval**
  - **Definition:** The average cumulative reward obtained per episode, calculated across all environments.
  - **Indication: Primary performance metric.** Higher values generally indicate better policy performance in terms of achieving task goals and accumulating reward in the environment. Compare - train vs. - eval to assess generalization.

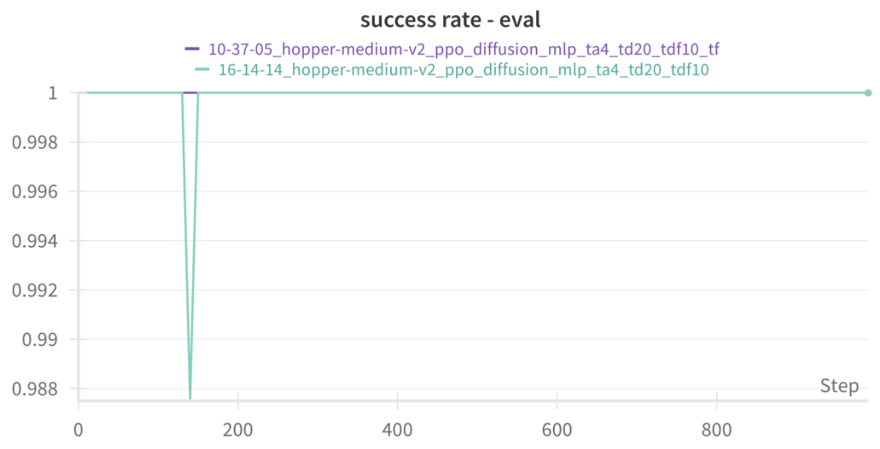




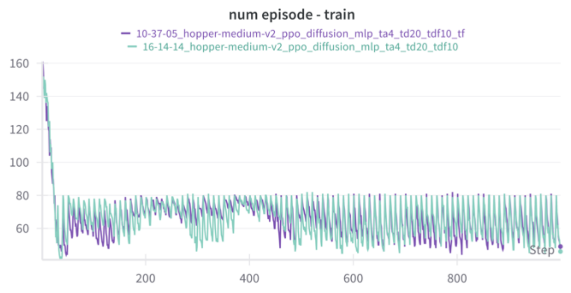
- **avg best reward – eval**
  - **Definition:** The average of the maximum reward achieved within each episode (divided by act_steps for normalization, in this specific implementation), calculated during evaluation. This metric is often used in tasks where reward is sparse, or task completion is more important than cumulative reward.
  - **Indication: Another performance metric, often more sensitive to task completion**. Higher values indicate better policies in terms of achieving high rewards within an episode, even if episodes are short. Useful for sparse reward settings. Higher is better.

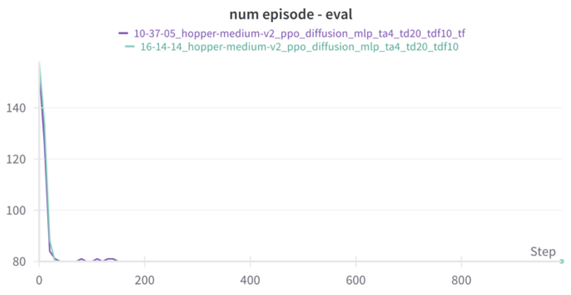


- **success rate – eval**
  - **Definition:** The percentage of evaluation episodes that are considered “successful”, based on whether the avg best reward - eval exceeds a predefined threshold.
  - **Indication: Key metric for task completion**. Specifically measures how often the policy reliably achieves a certain level of performance considered "successful" for the task. Higher is better, ideally approaching 1.0 or 100%.



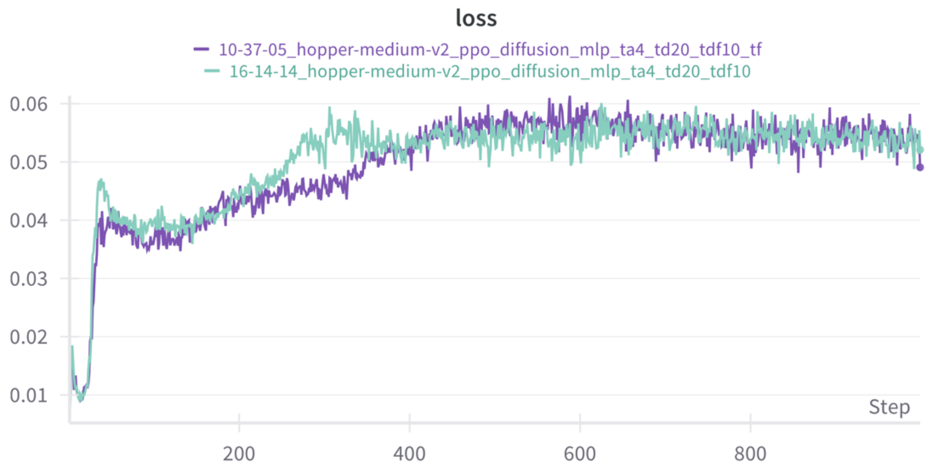
- **num episode – train / num episode – eval:**
  - **Definition:** The number of episodes that completed (terminated or truncated) within a training iteration or evaluation phase.
  - **Indication: Sample efficiency and data collection volume.** Can indicate how many full episodes were used for training or evaluated. Higher values might suggest more data collection, but the ideal value depends on the task and iteration length. Not necessarily "better" to be higher or lower in isolation.



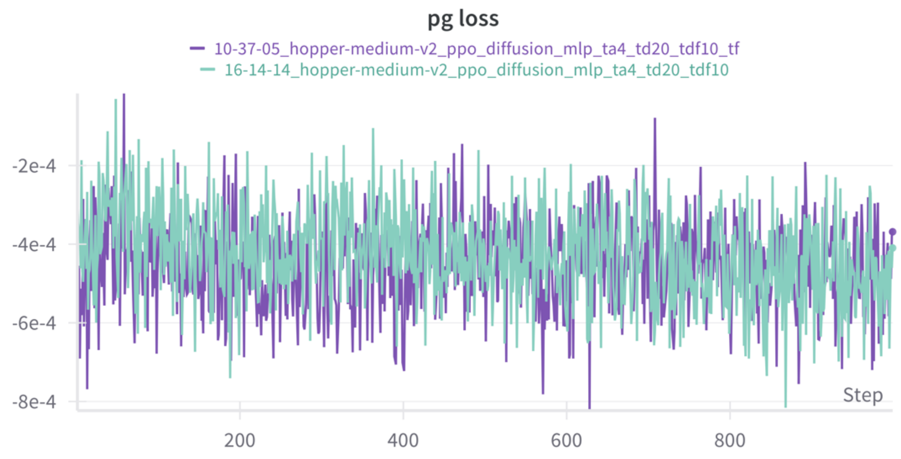


DPPO Training Loss Metrics

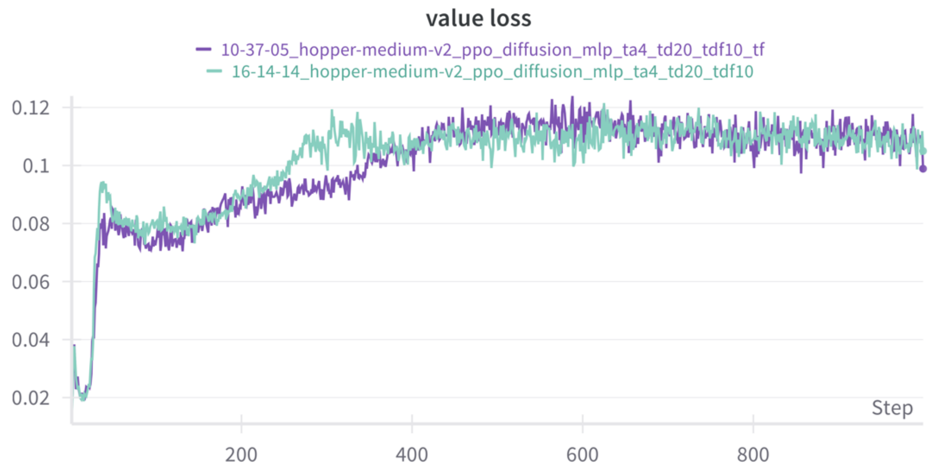
- **loss**
  - **Definition:** The total PPO loss, which is the sum of the policy gradient loss, entropy loss (scaled by ent_coef), value function loss (scaled by vf_coef), and behavior cloning loss (scaled by bc_loss_coeff, if used).
  - **Indication: Overall training objective being minimized**. Lower values generally indicate better training progress in terms of satisfying the combined PPO objective. Monitor for decreasing trend over training.



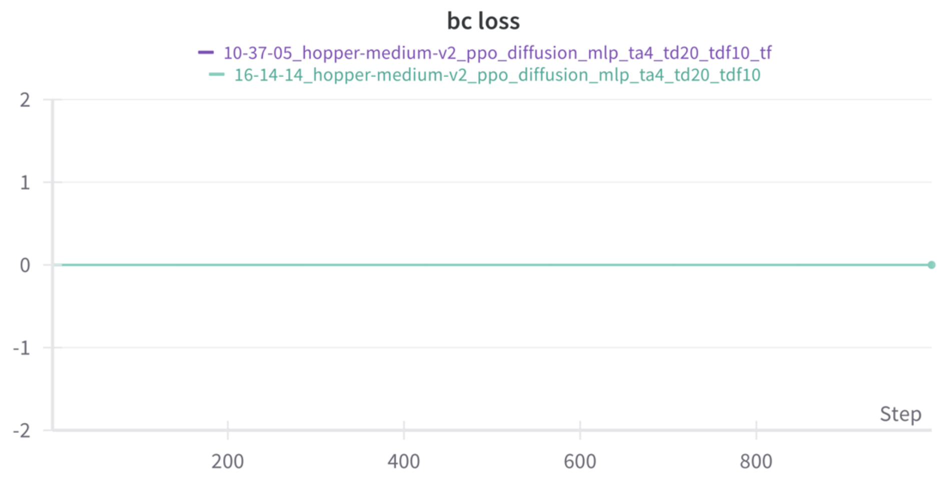
- **pg loss**
  - **Definition:** Policy gradient loss component of the PPO objective. This is the clipped surrogate objective that PPO uses to update the policy, encouraging actions with higher advantages and penalizing large policy changes.
  - **Indication: Policy improvement signal**. Lower values (or values close to zero) generally indicate that the policy is improving according to the PPO objective, or that policy updates are becoming smaller as the policy converges. Monitor for decreasing trend, but not necessarily driving it to zero.



- **value loss**
  - **Definition:** Value function loss component, typically Mean Squared Error (MSE) between predicted state values and target returns (using GAE)
  - **Indication: Critic accuracy in predicting state values**. Lower values indicate that the critic is better at estimating the expected future rewards (state values). Monitor for decreasing trend, indicating improved value function learning.

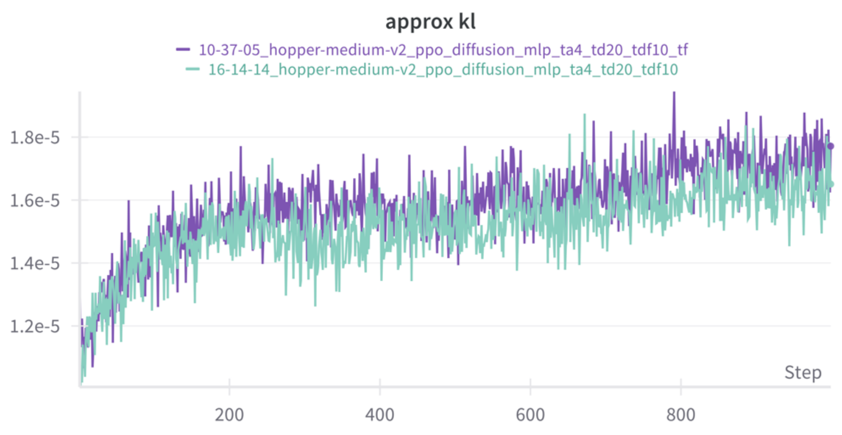


- **bc loss**
  - **Definition:** Behavior cloning loss component (if use_bc_loss is enabled). Measures how well the fine-tuned policy's actions align with the base (pre-trained) policy's actions.
  - **Indication: Regularization strength and similarity to base policy**. Lower values indicate better alignment with the base policy. Used to encourage the fine-tuned policy to stay close to the pre-trained behavior, which can be helpful for stability or sim-to-real transfer. Monitor if used, but the target value depends on the desired trade-off between RL and BC.

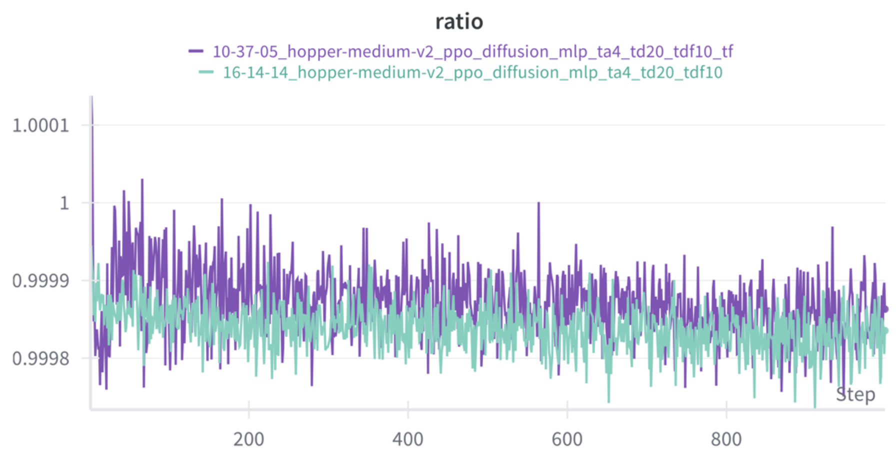


PPO Internal Status

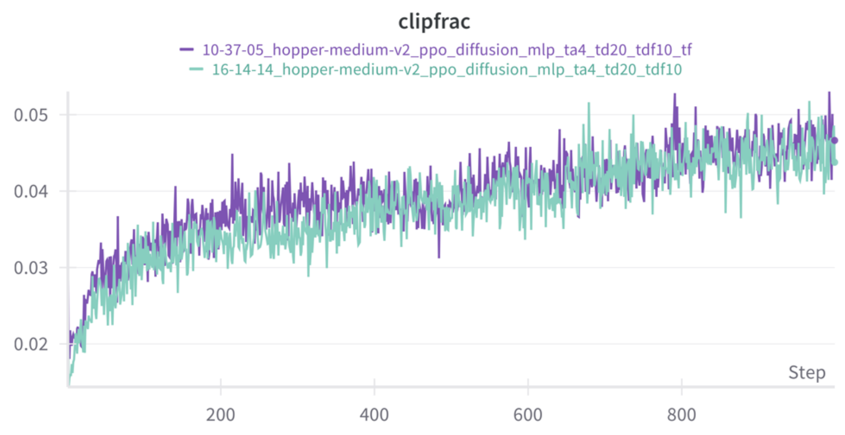
- **approx kl**
  - **Definition:** Approximate Kullback-Leibler (KL) divergence between the old policy (before update) and the new policy (after update).
  - **Indication: Magnitude of policy update.** Monitors how much the policy is changing in each update. PPO aims to keep KL divergence within a certain range to ensure stable updates. Useful for diagnosing instability if KL diverges or becomes too large. Should ideally be within a reasonable range (e.g., not excessively large or small).



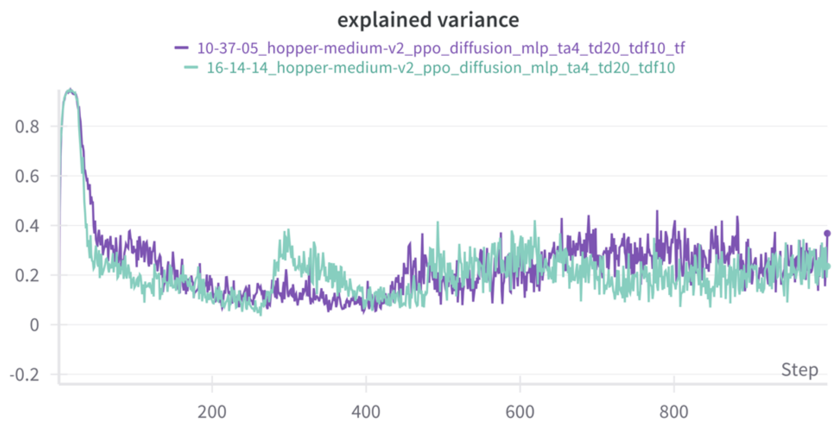
- **ratio**
  - **Definition:** Mean of the policy probability ration $\pi_\theta(a|s)/\pi_{\theta_{old}}(a|s)$
  - **Indication: Magnitude and direction of policy change.** Values close to 1 indicate small policy changes. Values significantly greater or less than 1 indicate larger changes. Useful for understanding the extent of policy updates.



- **clipfrac**
  - **Definition:** Clipping fraction - the fraction of data samples in a PPO batch where the policy probability ratio is clipped
  - **Indication: Extent of clipping applied during PPO update.** Higher values mean more clipping is happening, which can indicate that policy updates are becoming too large or aggressive for the clipping value

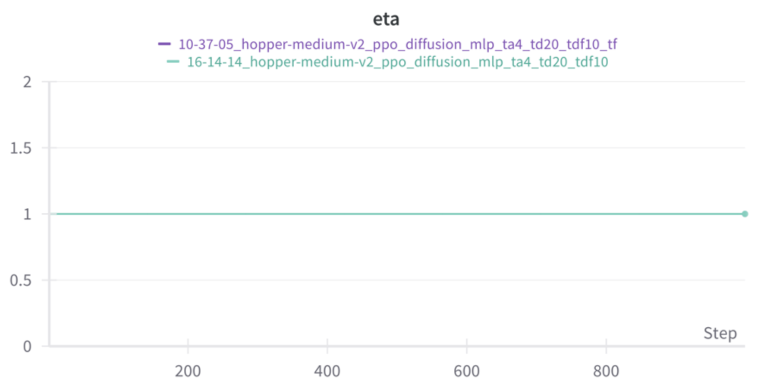


- **Explained variance**
  - **Definition:** Explained variance of the returns by the value function. Measures how well the value function predicts the actual returns (discounted future rewards).
  - **Indication: Critic's predictive power and how well it captures the variance in returns.** Values closer to 1 indicate that the value function explains a large portion of the variance in returns, suggesting a more accurate value estimate. Higher is generally better, but depends on the stochasticity of the environment and task.

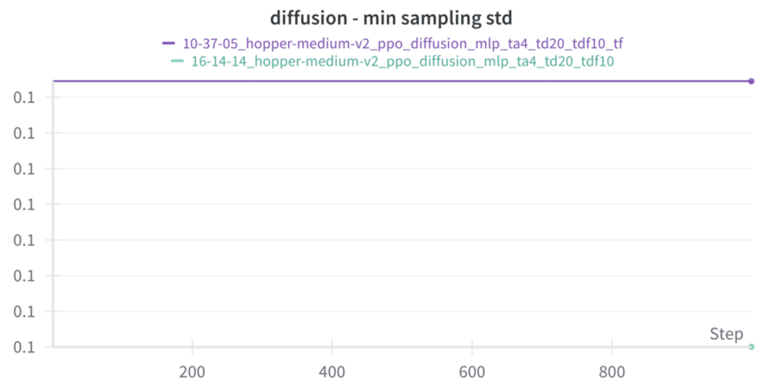


Diffusion Specific Metrics

- **eta**
  - **Definition:** Entropy of the diffusion policy distribution. In the context of these scripts, it likely refers to the average entropy across the denoising steps and batch.
  - **Indication: Policy stochasticity and exploration.** Higher entropy generally indicates a more stochastic policy, which can be beneficial for exploration. However, excessive entropy might lead to less consistent or less exploitative behavior. Optimal range depends on the task and desired exploration-exploitation trade-off.

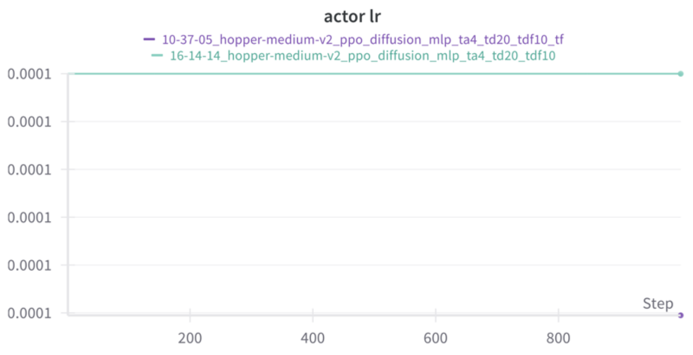


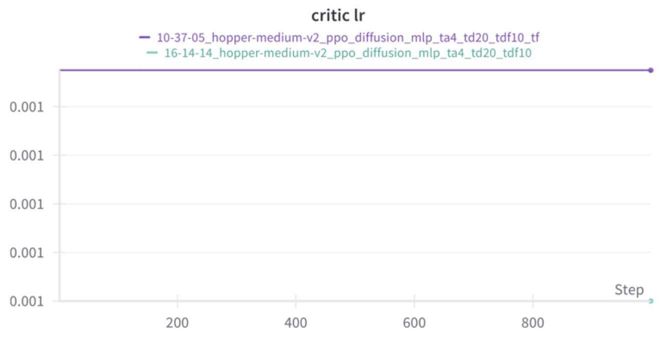
- **min sampling std**
  - **Definition:** The minimum standard deviation used for sampling actions from the diffusion policy during the denoising process. This is a parameter that controls the minimum level of noise added for exploration.
  - **Indication: Exploration noise level.** Higher values introduce more exploration noise. Can be tuned to balance exploration and exploitation. Higher values for more exploration, lower values for more exploitation (more deterministic behaviour).



Learning Rates

- **actor lr / critic lr**
  - **Definition:** Current learning rate of the actor (critic) optimizer (AdamW)
  - **Indication: Learning rate schedule**. Typically decreases over training (e.g., due to cosine annealing scheduler). Monitor to ensure learning rate is decaying as expected.


actor lr
10-37-05_hopper-medium-v2_ppo_diffusion_mlp_ta4_td20_tdf10_tf
16-14-14_hopper-medium-v2_ppo_diffusion_mlp_ta4_td20_tdf10
Step


critic lr
10-37-05_hopper-medium-v2_ppo_diffusion_mlp_ta4_td20_tdf10_tf
16-14-14_hopper-medium-v2_ppo_diffusion_mlp_ta4_td20_tdf10
Step