Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Max-Planck-Institut für Intelligente Systeme
Autonomous Learning Group

Masterarbeit
Machine Learning

Colored Noise Exploration in Reinforcement Learning

Onno Eberhard

Erstprüfer:
Dr. Georg Martius
Autonomous Learning Group
Max Planck Institute for
Intelligent Systems

Zweitprüfer:
Prof. Dr. Philipp Hennig
Methods of Machine Learning
Wilhelm-Schickard-Institut
University of Tübingen

October 4, 2022

Erklärung

Abstract

In off-policy deep reinforcement learning with continuous action spaces, exploration is often implemented by injecting action noise into the action selection process. Popular algorithms based on stochastic policies, such as SAC or MPO, inject white noise by sampling actions from uncorrelated Gaussian distributions. In many tasks, however, white noise does not provide sufficient exploration, and temporally correlated noise is used instead. A common choice is Ornstein-Uhlenbeck (OU) noise, which is closely related to Brownian motion (red noise). Both red noise and white noise belong to the broad family of *colored noise*. In this work, we perform a comprehensive experimental evaluation on MPO, SAC and TD3 to explore the effectiveness of other colors of noise as action noise. We find that pink noise, which is halfway between white and red noise, significantly outperforms white noise, OU noise, and other alternatives on a wide range of environments. Thus, we recommend it as the default choice for action noise in continuous control.

Contents

Chapter 1

Introduction

Figure 1.1 shows a rather bleak landscape. A car sits near the bottom of a sine-shaped valley. On the right, at the hill's top, is a flag indicating that this is where the car is supposed to go. Unfortunately, however, the hill is so steep that the car's motor is not powerful enough to climb it if the car starts from rest near the valley's bottom. So what is the car to do? It seems like not such a hard problem. The car simply has to first drive back for a bit before attempting to climb the mountain, such that it already has some momentum when driving through the valley. Indeed, this is the correct solution. The added momentum from this maneuver is enough to carry the car up the mountain and solve the task. This is the "MountainCar" problem, first introduced by Moore (1990). Though it seems like a trivial problem, artificial intelligence research has proved again and again that tasks which seem easy to us, are not necessarily easy for computers (while tasks that are difficult for us, like Chess playing, are not necessarily challenging to a computer; this observation is also known as Moravec's paradox (Moravec 1988)).

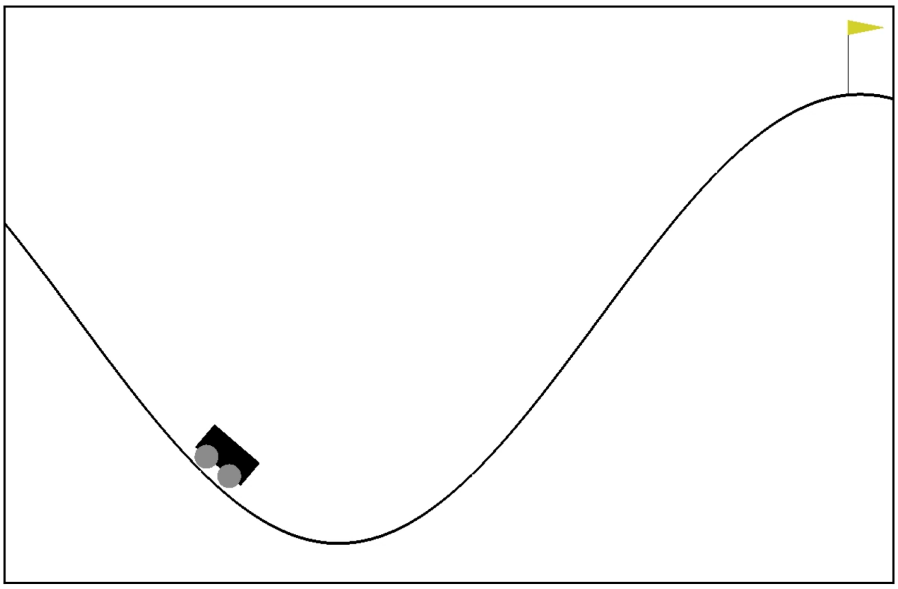

Figure 1.1: The *MountainCar* reinforcement learning task. Screenshot from OpenAI Gym (Brockman et al. 2016).

To understand why this task is challenging, we have to look at the problem in the same way an algorithm designed to solve it does. The algorithm (also called the *agent*) does not actually see the landscape; this would not be of much use, as it does not know how to interpret the pixels, never having seen a hill or a car (or an abstract representation of either). Instead, the state of the environment is encoded in two features: the car's position along the x-axis, and its speed. The agent's task is then to control the car, by choosing a force to apply to it (which we call the agent's *action*), based on the current state. This, of course, changes the car's state, and the agent gets to observe the new state, and respond with a new action. This is known as the "sensorimotor loop", and the problem is essentially closed-loop control. However, we do not now want to manually design a controller, but instead use *reinforcement learning* to learn a controller, by trial and error.

How should the agent choose its actions? Seeing the little flag at the top of the hill was enough for us to infer the task's goal from our experiences, but the agent needs to be told in another way. In reinforcement learning, at every time step, the agent is given not only the current state of the environment, but also a scalar reward signal. The goal is then defined as maximizing the sum of rewards over the whole episode (also called the *return*), which we assume to be of finite length. The reward has to encode the information of whether the goal has been achieved or not, so in the MountainCar example, if a certain x-position has been exceeded (i.e., the car has reached the flag), then the reward is 1 and the episode terminates. Setting the reward to 0 at every other time step would be enough to encode the original goal: if the flag is not reached within the time limit, the episode finishes with a return of 0. However, the actual MountainCar reward is slightly different: at every time step, the reward is actually

$$r(s,a) = [\![x \geq x_{\text{Flag}}]\!] - 0.1a^2, \tag{1.1}$$

where $s = (x, \dot{x})$ is the state, a is the action, and $[\![\cdot]\!]$ denotes the Iverson bracket (`bool2int` function). The first term in the reward encodes the goal of reaching the flag. The second term, which penalizes large actions, can be thought of as encoding that, among all solutions which reach the goal, the most "fuel-efficient" one is to be preferred. In effect, this term acts as a regularizer to shrink the set of optimal solutions.

So why is this problem challenging? It still seems like a very simple task, and indeed there are very simple solutions (see, for example, Section C for a method which exploits the problem's structure by finding the hill's resonance frequency). However, if modern state-of-the-art deep reinforcement learning algorithms, like SAC or MPO (see Sec. 2.1), are applied to this problem with their default settings, they fail most of the time.[1] The problem is that the reward function in Eq. (1.1) defines a local optimum. At the beginning of training, an agent does not know anything about how to behave. It starts out near the bottom of the valley and selects actions according to its randomly initialized policy. It is very unlikely that this random selection of actions "accidentally" solves the task, because, as we discussed above, this requires the coordinated motion of first accelerating to the

[1]The quantifier "most of the time" is necessary, because in (deep) reinforcement learning, everything depends heavily on the random initialization (Henderson et al. 2018).

left to build up some potential energy, and then accelerating to the right to climb the hill. Thus, at the beginning of training, the reward is approximately equal to $-0.1a^2$. The reinforcement learning algorithm picks up on this, and learns to apply less force. This makes it even more unlikely that the goal will be reached, and so, over time, the algorithm learns to apply no force at all.

This situation can be avoided by *exploration*. Exploration means that the agent selects a different action to the one recommended by the policy. Thus, even if the policy has learned to apply no force, exploratory actions are still taken. However, the exploratory actions should not be too different from the actions proposed by the policy, because then the agent will always be led to "off-policy" states far away from the ones it would visit without exploration. If this happens, the agent might not learn enough about the "on-policy" states, which are the ones visited in non-exploratory rollouts, and the policy performance will suffer. In the continuous control setting we consider, which means that the space of valid actions $\mathcal{A} \subset \mathbb{R}^d$ is not discrete[2], exploration is usually performed by adding some noise to the policy's action choice. The most common way to do this, is to sample each action from a Gaussian distribution centered at the policy's proposal, where the variance is either kept fixed, or learned as part of the policy. This is the default exploration strategy of all state-of-the-art deep RL algorithms we use for our experiments, with TD3 using a fixed variance and SAC and MPO learning it.

The MountainCar example is particularly nice to discuss exploration, because if the policy has learned to apply no force ($a = 0$), then the agent's behavior is completely determined by the exploration strategy. If the action noise is sampled from independent Gaussian distributions at every time step, then the exploration noise is known as *white noise*. The reason why these algorithms struggle with MountainCar, is that white noise is uncorrelated, which makes it very slow at exploring the state space (more details in Section 2.3). It is simple to see why temporal correlation should help in getting farther: imagine yourself as a 1-dimensional agent standing on a line, with the goal to explore the space. Your action space is $[-1, 1]$, and the action determines the direction and size of the next step you will take from your current position. Exploration behavior is typically symmetric, because without prior information it is impossible to know whether positive or negative actions are to be preferred. If your exploration strategy is uncorrelated and symmetric, and you choose the action a_t in time step t, which we can assume to be positive without loss of generality, then in the next time step you will be just as likely to undo your action and go back by choosing a negative action, as you will be to go further by choosing a positive action. If neighboring actions are *negatively* correlated, then you are more likely to undo your previous action than to build on it (which is generally undesirable). However, if neighboring actions are *positively* correlated, then subsequent actions are likely to be similar, making it more likely to move further in the same direction.

This problem of white noise has been recognized before, and a commonly used strategy is to use temporally correlated Ornstein-Uhlenbeck (OU) noise instead

[2]Usually this set is actually closed and convex; the important property we use is that adding a small perturbation to an action is *still a valid action*. However, $\mathcal{A}$ does not have to be open (and indeed should be closed), as the actions are usually clipped to be valid (projected onto $\mathcal{A}$).

(see Section 2.3). However, on many environments the high correlation of OU noise is not necessary, and its use only leads to slower learning due to increased off-policy data. Thus, overall, white noise is the default choice.

In this work, we analyze the effectiveness of using a different kind of temporally correlated noise as action noise: *colored noise*. Colored noise generalizes white noise and introduces a "color" parameter β, which can be adjusted to tune the strength of the temporal correlation. It can also be related to Ornstein-Uhlenbeck noise, and we go into more detail in Section 2.4. Colored noise has already been applied to model-based reinforcement learning (Pinneri et al. 2020), where it has been shown to be very effective when β was tuned correctly. We address the problem of setting the color parameter correctly for a given environment in Chapters 3 and 4. In contrast to the findings of Pinneri et al. (2020), we find that in the model-free setting, it is not necessarily beneficial to adapt β to an environment. Instead, simply using *pink noise*, which can be understood as a noise half-way between white noise and OU noise, seems to work better than any other strategy we try out, including a color-schedule, and a Bayesian optimization procedure to select β. Interestingly, pink noise has also been observed in the movement of humans: the slight swaying of still-standing subjects, as well as the temporal deviations of musicians from the beat, have both been measured to exhibit temporal correlations in accord with pink noise (Duarte and Zatsiorsky 2001; Hennig et al. 2011). In Chapter 5, we try to explain why pink noise performs so well by analyzing its behavior on two simple environments.

Chapter 2

Background

2.1 Reinforcement Learning

Formally, a reinforcement learning problem, such as the MountainCar problem from Chapter 1, is described by a *Markov decision process* (MDP). A Markov decision process consists of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, an initial state distribution $p(s_0)$, and the environment dynamics $p(s_{t+1}, r_{t+1} \mid s_t, a_t)$. The stochastic decision rule the agent uses to select an action given a state is called the *policy*, and it is denoted by $\pi(a_t \mid s_t)$. Deterministic policies, commonly denoted $\mu(s_t)$, can be thought of as a Dirac distribution: $\pi(a_t \mid s_t) = \delta(a_t - \mu(s_t))$.

If the policy is fixed, then MDP and policy together form a stochastic process. Sampling from this process is done via the interaction loop described in Chapter 1:

1. The initial state s_0 is sampled from $p(s_0)$.
2. The policy selects its action a_0 by sampling from $\pi(a_0 \mid s_0)$.
3. The environment responds with a reward and next state: $s_1, r_1 \sim p(\cdot \mid s_0, a_0)$.
4. At every time step t, the policy selects its next action $a_t \sim \pi(\cdot \mid s_t)$ and the environment responds with a next state and reward $s_{t+1}, r_{t+1} \sim p(\cdot \mid s_t, a_t)$
5. This procedure continues until the episode finishes at time step T with the final state s_T and reward r_T.[1]

The sequence $\tau = (s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T)$ is called the *trajectory.* Sampling according to the procedure above samples from the trajectory distribution

$$p_\pi(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi(a_t \mid s_t) p(s_{t+1}, r_{t+1} \mid s_t, a_t). \tag{2.1}$$

This distribution is depicted as a directed graphical model in Figure 2.1.

[1]In this text, we only consider the finite-horizon case, and thus also do not mention discounts, although the use of discounts in finite episodes is perfectly fine in our methods.

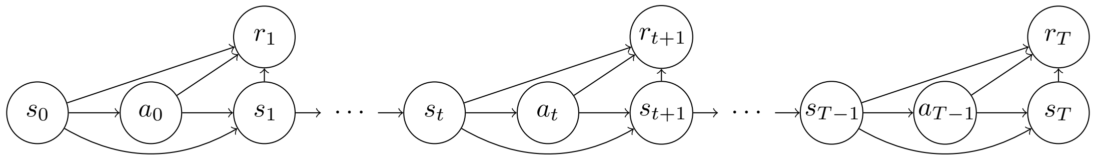

Figure 2.1: Directed graphical model showing the structure of a Markov decision process (MDP) with finite horizon T. This model defines the trajectory distribution $p_\pi(\tau)$.

The reinforcement learning objective is to find a policy which will yield the maximum expected return:

$$\pi^\star = \operatorname*{arg\,max}_{\pi \in \Pi} J(\pi) \quad \text{where} \quad J(\pi) = \mathbb{E}_{\tau \sim p_\pi}\left[\sum_{t=1}^{T} r_t\right]. \tag{2.2}$$

What sets reinforcement learning apart from optimal control, is that the environment dynamics are unknown to the algorithm. This makes the optimization problem in Eq. (2.2) fundamentally more difficult.

One approach to tackle the RL problem is by learning a model $\hat{p}(s', r \mid s, a)$ to approximate the environment dynamics. These methods are called *model-based* reinforcement learning. The model can be learned by running rollouts in the environment (using a data-collecting policy, e.g. random) to gather interaction tuples (s, a, r, s'), and then learn the model using standard techniques from supervised learning, like learning a Gaussian process model (Deisenroth and Rasmussen 2011) or a neural network model (Nagabandi et al. 2018). Once a model has been learned, it can be used for *planning*. A simple planning method is the *cross-entropy method* (CEM, Rubinstein 1999). In CEM, n actions are sampled for each of remaining time steps, from independent Gaussian distributions $p(a_t), \ldots, p(a_{T-1})$.[2] As sampling actions at random is not necessarily the best strategy for good behavior, these action sequences are not used directly, but instead their performances are evaluated using the learned model:

$$J(a_t, \ldots, a_{T-1} \mid s_t) = \mathbb{E}_{s,r \sim \hat{p}}\left[\sum_{t'=t+1}^{T} r_{t'} \;\middle|\; s_t\right]. \tag{2.3}$$

The best m action sequences are kept (where $m < n$), and the distributions $p(a_t), \ldots, p(a_{T-1})$ are refit to these actions. From the modified distributions, again n candidate action sequences are sampled, and these are again evaluated, and the best ones are again used to refit the sampling distributions. After repeating this procedure a few times (not necessarily until convergence), a promising action sequence will hopefully have been found. Instead of executing the complete action sequence, usually only the first action is executed. Then, the same planning procedure is repeated from the next state. Only using the first action of a planned sequence like this is called *model predictive control*, and it is especially useful in cases where the model is not perfect.

[2]The distributions could be non-Gaussian, but in the continuous control setting, Gaussian distributions are most common.

In this work, we don't use model-based methods, but instead focus on *model-free* reinforcement learning. If we select a specific function class Π to represent possible policies, we can perform *policy search* by optimizing the problem in Eq. (2.2) directly. Typically, the function class is described by parameters θ, which could, for example, represent the weights of a neural network of fixed architecture. In this case, we normally write the policy parameterized by θ as π_θ, and the optimization problem in Eq. (2.2) becomes

$$\theta^\star = \arg\max_\theta J(\pi_\theta). \tag{2.4}$$

If we choose a differentiable policy class, like a neural network, it would be nice to optimize the objective J using gradient ascent. For this, we would need to calculate the *policy gradient*

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T} r_t\right], \tag{2.5}$$

where the trajectory distribution induced by the policy π_θ is denoted as p_θ. This might seem hopeless without knowledge of the environment dynamics, but it turns out that it is possible to estimate the policy gradient simply by sampling trajectories using the policy π_θ:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \int \nabla_\theta p_\theta(\tau)\left(\sum_{t=1}^{T} r_t\right)\mathrm{d}\tau & (2.6)\\
&= \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau)\left(\sum_{t=1}^{T} r_t\right)\mathrm{d}\tau & (2.7)\\
&= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)\left(\sum_{t=1}^{T} r_t\right)\right] & (2.8)\\
&= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right)\left(\sum_{t=1}^{T} r_t\right)\right] & (2.9)
\end{aligned}$$

This result is known as the policy gradient theorem, and it uses the fact that

$$\nabla f(x) = f(x)\frac{\nabla f(x)}{f(x)} = f(x)\nabla \log f(x), \tag{2.10}$$

as well as that all environment dynamics terms in the trajectory distribution $p_\theta(\tau)$ (Eq. 2.1) disappear inside the expectation when taking the gradient with respect to θ. This admits a straightforward algorithm: simply run a few rollouts with the current policy, and use these trajectories to compute the Monte Carlo estimate of the expectation in Eq. (2.9). The computed quantity ($\hat{\nabla}_\theta J(\pi_\theta)$) is an estimate of the policy gradient, so we can use gradient ascent to update the policy parameters:

$$\theta \leftarrow \theta + \alpha\hat{\nabla}_\theta J(\pi_\theta) \tag{2.11}$$

This algorithm is called REINFORCE (Williams 1992), and its main problem is that the Monte Carlo estimate, while unbiased, in general can have very high variance. This problem is commonly addressed by using a baseline: a function which depends on the state, but not on the action, can be subtracted from the rewards in Eq. (2.9) to reduce variance. The most common choice for a baseline is the state value function $V(s)$:

$$V_t(s) = \mathbb{E}_{p_\pi}\left[\sum_{t'=t+1}^{T} r_{t'} \;\middle|\; s_t = s\right] \tag{2.12}$$

This function can also be learned from trajectories, by computing the Monte Carlo estimate of the expectation. A combination of the use of baselines, as well as adjustments to the update in Eq. (2.11) to improve stability, yields popular algorithms like Trust Region Policy Optimization (TRPO, Schulman et al. 2015) and Proximal Policy Optimization (PPO, Schulman et al. 2017) which can be used with neural network function approximators for π and V.[3]

By casting reinforcement learning as a problem of *approximate dynamic programming* (Bertsekas 2017), it becomes possible to use methods other than policy search: the value function itself can be used to solve the RL problem, by using approximations to the value or policy iteration algorithm from dynamic programming. The most famous of these methods is Q-learning (Watkins and Dayan 1992), an approximate value iteration method. In Q-learning, the policy is not parameterized itself. Instead, only the state-value function, or Q-function, is parameterized. This function ($Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$) is defined as giving the expected sum of future rewards, when starting in a given state by performing a certain action. The optimal Q-function, which is the Q-function of the optimal policy defined by Eq. (2.2), admits a recursive representation known as the *Bellman optimality equation*[4]:

$$Q^\star(s,a) = \mathbb{E}_{r,s' \sim p(\cdot|s,a)}\left[r + \max_{a' \in \mathcal{A}} Q^\star(s',a')\right] \tag{2.13}$$

Q-learning is the algorithm defined by turning this equation into an iterative procedure. During rollouts in the environment, collected interaction tuples (s, a, r, s') can be used to update a tabular Q-function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \max_{a'} Q(s',a') - Q(s,a)\Big). \tag{2.14}$$

The Q-function can also be represented by a function approximator, like a neural network. In this case, supervised learning can be used, where the training dataset is created from many interaction tuples:

$$\mathcal{D} = \{(x,y)_i\} = \left\{\Big((s_i,a_i), r_i + \max_{a'} Q(s'_i,a')\Big)_i\right\} \tag{2.15}$$

[3]Often, finite horizon tasks are simply treated as infinite horizon tasks. This makes it necessary to use discounts, but then only a single value function V has to be learned, instead of one for every time step.

[4]This equation really only makes sense in the infinite horizon setting. This technicality is mostly ignored in the (deep) reinforcement learning community, so we will ignore it as well. Usually though, at least a discount factor γ is placed before the max in Eq. (2.13).

In this form, the Q-learning algorithm is also called fitted Q-iteration. After the Q-function has been learned, a policy can be constructed to act greedily with respect to it by choosing the best action at every state:

$$\mu_Q(s) = \arg\max_{a \in \mathcal{A}} Q(s, a) \tag{2.16}$$

The great benefit of Q-learning over REINFORCE is its sample-efficiency. Q-learning is an *off-policy* algorithm: the training data consists of (s, a, r, s')-tuples. This data contains no information about the policy that was used to collect it, meaning that in principle any policy could be used to collect the data, as long as it approximately covers the same part of the state-action space as μ_Q. This fact is exploited in fitted Q-iteration: all interactions are stored in a so-called replay buffer, from which the dataset (Eq. 2.15) is sampled. This greatly improves sample efficiency, because old data can be reused many times. REINFORCE, on the other hand, is an *on-policy* algorithm, because the rollouts have to be sampled from the current policy π_θ for the policy gradient theorem to hold (Eq. 2.9). After one update of the parameters θ, all data collected with old parameters can be discarded.

The beginning of the *deep reinforcement learning* research field is marked by the Deep Q-Network algorithm (DQN, Mnih et al. 2015), which introduced several changes to the standard Q-learning algorithm to address stability issues arising with neural networks as function approximators. The DQN architecture is constructed for small finite action spaces: the deep neural network that represents the Q-function takes as input only the state, and outputs the Q-value of each action at that state. In this work, we are specifically interested in the continuous control setting, where such an approach is impossible. Reverse architectures, where the state-action pair is an input to the neural network, are possible, but then the maximization in Eq. (2.16), which has to be performed at every time step, becomes prohibitively expensive. This problem is addressed by the Deep Deterministic Policy Gradients algorithm (DDPG, Lillicrap et al. 2016). Instead of manually performing the optimization step, a second neural network μ is trained to perform an approximate optimization of Q. Thus, μ is a deterministic policy, and training is performed by gradient ascent on the deterministic policy gradient (DPG, Silver et al. 2014). Algorithms like this, which use both a parameterized value function and a parameterized policy are referred to as *actor-critic* algorithms (the policy is then called the actor, and the value function the critic).

A follow-up work on DDPG, called the Twin-Delayed Deep Deterministic Policy Gradient algorithm (TD3, Fujimoto et al. 2018), introduced several changes to DDPG to improve stability and overall performance. At the same time, several new off-policy algorithms emerged which made use of a framework called *maximum entropy reinforcement learning* (MaxEnt RL). In MaxEnt RL, the reinforcement learning objective (Eq. 2.2) is augmented to encourage a policy distribution with large entropy at the visited states (which, on the flip side, also encourages the agent to visit states where the policy has a large entropy). This idea can be related to the casting of optimal control and reinforcement learning as probabilistic inference problems (Levine 2018). In particular, the two popular algorithms Soft Actor-Critic (SAC, Haarnoja et al. 2018) and Maximum a Posteriori Policy Optimization (MPO, Abdolmaleki et al. 2018) build on this framework, both of which parameterize a stochastic policy (as opposed to DDPG and TD3). In this work, we use MPO, SAC and TD3 for all of our experiments. For SAC and TD3 we use the implementations from the Stable Baselines3 library (Raffin et al. 2021), and for MPO we make use of the Tonic RL library (Pardo 2020).

2.2 Bandits and Exploration

Exploration is defined as taking a different action than the “greedy” one:

$$a_t \neq \arg\max_{a \in \mathcal{A}} Q(s_t, a) \tag{2.17}$$

Not exploring, i.e. taking the greedy action, is referred to as *exploitation.* In action spaces with more than two choices it is often possible to explore more or less, and this is particularly the case in continuous spaces, where the degree of exploration can be measured by the distance (under an appropriate norm) of the chosen action from the greedy action. We have already touched upon the necessity for exploration in the first chapter. In most algorithms for continuous control, for instance the ones we consider (MPO, SAC, TD3), exploration is done by adding noise to the policy. This method of exploration is treated in the next section. In this section, we are going to take a look at the problem of exploration itself.

The “exploration vs. exploitation” dilemma is fundamental to reinforcement learning, and decision-making in general. A simple example by David Silver (Silver 2015) is restaurant selection: if you want to eat out, should you go to your favorite restaurant (which you know is very good), or should you try a new restaurant (which might be even better, but is also likely to be worse)? The former option corresponds to exploitation, and the latter to exploration.

The exploration problem in this example is very different from the exploration problem we described for the MountainCar task in the first Chapter. In the MountainCar problem, exploration was necessary to reach new parts of the state space. The restaurant selection example does not have a clearly defined state space. In fact, it turns out that the simplest way to model the restaurant selection problem is by what is called a *multi-armed bandit*: an MDP with only a single state and episode lengths of one step. These setups are called (multi-armed) bandits in suggestion of one-armed bandits (casino slot machines), which, upon pulling a lever, dispense money according to an unknown internal distribution. The multi-armed bandit problem can be thought of as trying to find the best slot machine out of several, each having a different internal distribution. In MDP terms, this means that a “rollout” consists of a single action (which arm to pull) and collecting a certain reward (amount of money). The “best” action (arm) is the one with the highest expected reward.

As the rewards are stochastic, it is not enough to simply try each action once (quickly “explore” to find the best one) and from then on exploit. Instead, all actions have to be taken several times to estimate their expected return. In fact, no finite number of samples is enough to get the precise value of an arm’s expected reward. Thus, if one continues pulling arms ad infinitum, one can never stop exploring. On the other hand, it is also important to exploit the knowledge that one already has gathered about these distributions. Even when exploring, it may often be better to pull an arm which has been observed to be almost as good as the one we think of as best.

These ideas are collected in a precise problem statement for bandits, which is to minimize *regret*, defined at iteration t as

$$\sum_{\tau=1}^{t} r(a^\star) - r(a_\tau), \tag{2.18}$$

where $a^\star$ is the unknown best action, a_τ is the action played at iteration τ and $r : \mathcal{A} \to \mathbb{R}$ is the unknown reward function, i.e. $r(a)$ is the expected reward for action a. Algorithms addressing the problem of minimizing regret are called bandit algorithms, and they compose a large research field independent of reinforcement learning (e.g. Lattimore and Szepesvári 2020). With the MDP reduced to only a single transition, the only problem that remains is the exploration vs. exploitation problem (and even here, it is not an easy one).

So, how does the bandit setting relate to the exploration problem we faced in MountainCar, if there is no state space? Even though these problems look very different on the surface, they are in fact quite similar. After all, the reason why we want to explore the state space, is to find high reward regions, so in both cases the problem is really about reward. The reason why the car should, at a given state s, go right instead of left, even though $Q(s, \text{left}) > Q(s, \text{right})$ (i.e. why it should explore), is for the possibility of achieving a higher episode return, just as in the bandit setting. The two settings are different, however, in the reason for why exploitation is necessary. In the bandit case, this is part of the problem definition (minimizing regret). In reinforcement learning, however, it is common to have a training stage, where an algorithm, such as MPO, is used to learn a policy from exploratory rollouts, after which the learning is "done", and the final policy can be used without exploration (as learning has finished). Translating this setting into bandit terms would be an objective of "having the best estimate for which arm is best" after t iterations, i.e. minimizing

$$r(a^\star) - r(\hat{a}_t^\star), \tag{2.19}$$

where $\hat{a}_t^\star$ is the estimated best arm after t interactions This is quite a different objective from regret minimization, one which does not punish a lack of exploitation. However, reinforcement learning agents also need to exploit, and the reason for this is that the algorithms (MPO and the rest) need the right data to learn good behavior. The final policy should perform well on the part of the state space it is likely to encounter during a non-exploratory trajectory. Data sampled from the non-exploratory policy is called *on-policy* data, while other data is called *off-policy*.[5] As is also the case in supervised learning, the training and testing data should come from the same distribution. In the context of RL this means that the (exploratory)

[5]At least that is how we use the terms here. Words like on-policy and off-policy are used with slightly different meanings in different contexts, and thus all definitions are somewhat wrong.

trajectories during the training phase should be approximately on-policy. Concretely, this shows that there is a need for exploitation: too much exploration will lead to highly off-policy trajectories, inhibiting learning and degrading evaluation performance.

As we want to tackle the exploration problem in RL, let us start with the simpler exploration problem described by the K-armed bandit setting. We have already discussed that an optimal strategy must keep pulling all arms to increase the accuracy of the expected reward estimates, but that it is also important to exploit the current knowledge to minimize regret. We can make the problem easier with a few assumptions on the bandit distributions. If we assume, for example, that all K reward distributions are Gaussian with equal and known standard deviation σ (a particularly convenient assumption), there are several algorithms to tackle the problem of regret minimization in an optimal way.[6] Optimal here means that the regret grows asymptotically as $\mathcal{O}(\log t)$ (Lai and Robbins 1985). Two methods which achieve logarithmic regret are Thompson sampling (Agrawal and Goyal 2012) and upper confidence bound (UCB) algorithms (Auer et al. 2002).

We will discuss Thompson sampling in a bit of detail, as we will use it again in Section 4.2. To reiterate, the setup is as follows: there are K possible actions (arms to pull), and upon pulling arm a, we will get a random reward r, sampled from that arm's distribution: $r \sim \mathcal{N}(r(a), \sigma)$. In Thompson sampling, we take a Bayesian approach and maintain belief distributions for the means of all K reward distributions. As the reward distributions are Gaussian, we can also model their means by Gaussian distributions (the corresponding conjugate prior). Thus, the prior belief distribution is initialized as $\mathcal{N}(\boldsymbol{\mu} \mid \boldsymbol{m}, \Sigma)$, where $\boldsymbol{\mu} \in \mathbb{R}^K$ and Σ can be chosen to be diagonal if we have no prior information about the covariance between the arms (more on this in Section 4.2). After an arm a has been pulled and a reward r collected, we can use Bayesian inference to compute the posterior updates for $\boldsymbol{m}$ and Σ:[7]

$$\begin{aligned} p(\boldsymbol{\mu}) &= \mathcal{N}(\boldsymbol{\mu} \mid \boldsymbol{m}, \Sigma) && (2.20)\\ p(r \mid \boldsymbol{\mu}, a) &= \mathcal{N}(r \mid \mathbf{e}_a^\top \boldsymbol{\mu}, \sigma) && (2.21)\\ p(\boldsymbol{\mu} \mid r, a) &= \mathcal{N}(\boldsymbol{\mu} \mid \boldsymbol{m} + \Sigma \mathbf{e}_a (\mathbf{e}_a^\top \Sigma \mathbf{e}_a + \sigma)^{-1} (r - \mathbf{e}_a^\top \boldsymbol{m}), \\ &\qquad \Sigma - \Sigma \mathbf{e}_a (\mathbf{e}_a^\top \Sigma \mathbf{e}_a + \sigma)^{-1} \mathbf{e}_a^\top \Sigma) && (2.22) \end{aligned}$$

The third equation follows from the first two by Bayes' rule for Gaussians. But how do we choose which arm we pull? In Thompson sampling, this is done by *sampling from the belief distribution*:

$$\boldsymbol{q} \sim \mathcal{N}(\boldsymbol{m}, \Sigma), \qquad (2.23)$$

where $\boldsymbol{m}$ and Σ are up-to-date with the history of past actions and rewards. Then, we choose the next arm as the one which had the best sampled value:

$$a = \operatorname*{arg\,max}_{k \in \{1, \dots, K\}} q_k. \qquad (2.24)$$

[6]These algorithms optimal under certain looser assumptions as well.

[7]Here, we write $r(a)$ as $\mathbf{e}_a^\top \boldsymbol{\mu}$, where $\mathbf{e}_a$ is the ath canonical basis vector, which is a convenient way to write $r(a)$ as a linear transformation of $\boldsymbol{\mu}$.

After pulling the arm and collecting the reward, the beliefs are updated and everything repeats.

The intuitive reason for why this procedure balances exploration and exploitation is that arms a with a high mean estimate m_a are more likely to be selected, as q_a is more likely to be high as well. On the other hand, sampling from these Gaussian distributions in principle makes it possible for all arms to be selected, but the higher the estimated mean, or the higher the variance, the more likely it is to be selected. This last point means that if an arm has been chosen comparatively few times, it is more likely to be selected, as its variance will be larger (Bayes' rule makes the variance of the selected arm shrink). This property is often described as *optimism in the face of uncertainty*: if we don't know a lot about an arm's reward distribution, being optimistic means we are more likely to choose it as "it might be good". A closely related algorithm to Thompson sampling, UCB (upper confidence bound algorithm), is also an optimistic method: the arm is chosen not by sampling from the belief distribution, but by explicitly choosing the arm which has a high mean and a high variance:

$$a = \underset{k \in \{1,\dots,K\}}{\arg\max}\, m_k + \lambda \sqrt{\Sigma_{kk}}, \tag{2.25}$$

where λ is an optimism hyperparameter (the larger λ, the more will uncertainty be valued).

These bandit algorithms can solve the exploration vs. exploitation problem optimally. In small finite MDPs, it is also possible to optimally solve exploration, by augmenting the states to include the agent's belief about the environment. This gives rise to the Bayes-Adaptive MDP formalism (Duff 2002), where an optimal exploration strategy can be derived. However, this quickly becomes intractable when the state space grows, and so is not applicable to the continuous control setting we are faced with. Still, it is possible to draw inspiration from bandit techniques to tackle the exploration problem in deep reinforcement learning. For example, in exploration based on pseudo-counts (Bellemare et al. 2016; Tang et al. 2017), an uncertainty bonus is added to the reward function, similarly to UCB. Other methods, like Bootstrapped DQN (Osband et al. 2016), take inspiration from Thompson sampling: here, Q-functions are sampled in the same way as mean estimates are sampled in Thompson sampling to achieve exploration. There are many other methods researchers have come up with to tackle the exploration problem, such as adding noise to the policy parameters (Plappert et al. 2018; Mania et al. 2018), or parameterizing policies based on complex distributions like normalizing flows (Mazoure et al. 2019). However, the simplest and most common exploration strategy for deep RL in continuous action spaces is simply *adding noise to the actions*.

2.3 Noisy Exploration

The actions proposed by a policy can be perturbed in many elaborate ways. There are methods which generate a function for each episode that deterministically alters the action selection (Raffin and Stulp 2020), which learn correlations between the action and state space dimensions to induce increasing excitation in the environment (Schumacher et al. 2022), or which learn an action prior from task-agnostic demonstrations (Bagatella et al. 2022). The most common approach, however, is to simply sample an action noise signal from some random process. In particular, this is the technique employed by all deep RL algorithms we consider in this work.

In algorithms like DDPG and TD3, where the learned policy μ is deterministic, the action noise signal $\varepsilon_{0:T-1} = (\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{T-1})$ is simply added to the policy:

$$a_t = \mu(s_t) + \sigma\varepsilon_t, \tag{2.26}$$

where σ is a scale parameter. If ε_t is sampled independently at every time step, e.g. from a Gaussian distribution, then the signal $\varepsilon_{0:T-1}$ is called *white noise* (WN). This is the prevailing choice of action noise, though it is also common to use temporally correlated Ornstein-Uhlenbeck noise ($\varepsilon_{0:T-1} \sim \mathrm{OU}_T$) (Uhlenbeck and Ornstein 1930).

Algorithms which parameterize a stochastic policy, such as SAC and MPO, also use action noise. In continuous action spaces, the most common policy distribution is a diagonal Gaussian, represented by the functions $\mu : \mathcal{S} \to \mathcal{A}$ and $\sigma : \mathcal{S} \to \mathcal{A}$. Here, actions are sampled at every time step t from the policy distribution according to

$$a_t \sim \mathcal{N}(\mu(s_t), \mathrm{diag}(\sigma(s_t))). \tag{2.27}$$

This can equivalently be written as

$$a_t = \mu(s_t) + \sigma(s_t) \odot \varepsilon_t, \tag{2.28}$$

where $\varepsilon_t \sim \mathcal{N}(0, I)$. This is known as the *reparameterization trick*, and in principle this technique also applies to other distributions which have a “location” and a “scale” parameter, as is described by Kingma and Welling (2014). In this case, the action noise $\varepsilon_{0:T-1}$ is again Gaussian white noise, which is scale-modulated by the function σ.

White noise is not correlated over time ($\mathrm{cov}[\varepsilon_t, \varepsilon_{t'}] = 0$). In some environments, this leads to very slow exploration, which in turn leads to inadequate state space coverage, leaving high reward regions undiscovered. Why is white action noise slow? We can examine this question theoretically, by making two simplifying assumptions:

1. The environment is an integrator: after receiving a sequence of actions a_t as input, the state is given by $x_t = \int_0^t a_\tau \,\mathrm{d}\tau$. These are the dynamics of a velocity- or step-controlled particle, and are particularly easy to analyze theoretically.

2. The agent is *purely noise* (i.e. the policy to which the noise is added is the 0-function). This is the situation we encountered with MountainCar in Chapter 1, and it allows us to analyze the exploration behavior in isolation of a policy. Like the first assumption, this assumption is very strong, but makes our lives much easier.

Thus, we are interested in the distance that *integrated white noise* (with noise scale $\sigma = 1$) achieves as a function of time. In continuous time[8], white noise is the (generalized) time derivative of the Wiener process $w(t)$ and is thus written as $\dot{w}(t)$. We want to calculate:

$$\mathbb{E}[|x_t|] = \mathbb{E}\left[\left|\int_0^t \dot{w}(\tau)\,\mathrm{d}\tau\right|\right] \tag{2.29}$$

$$= \mathbb{E}[|w(t)|] \tag{2.30}$$

We will use two properties of the Wiener process:

1. $w(0) = 0$ (almost surely)
2. $w(t) - w(t') \sim \mathcal{N}(0, t - t')$ for any $0 \leq t' < t$

From these properties it follows that

$$w(t) = w(t) - 0 = w(t) - w(0) \sim \mathcal{N}(0, t - 0) = \mathcal{N}(0, t). \tag{2.31}$$

Thus, $x_t \sim \mathcal{N}(0, t)$, and the expectation above is equal to

$$\mathbb{E}[|x_t|] = \int_{-\infty}^{\infty} |x|\,\mathcal{N}(x \mid 0, t)\,\mathrm{d}x \tag{2.32}$$

$$= 2\int_0^{\infty} x\,\mathcal{N}(x \mid 0, t)\,\mathrm{d}x \tag{2.33}$$

The last expression is the expected value of the half-normal distribution, which is known to be $\sqrt{\frac{2}{\pi}}\sigma$ (Leone et al. 1961), where σ is the standard deviation of the Gaussian (here, $\sqrt{t}$). Thus, we can conclude

$$\mathbb{E}[|x_t|] = \sqrt{\frac{2}{\pi}t}. \tag{2.34}$$

In other words, the exploration behavior induced by white action noise on the integrator environment will only reach distance proportional to the square root of the number of time steps. It is easy to see why on many environments this exploration behavior is not sufficient to discover far away high reward regions in the state space.

There is, however, a simple solution: temporally correlated action noise (such that $\mathrm{cov}[\varepsilon_t, \varepsilon_{t'}] > 0$). One example of a temporally correlated noise process is *Brownian noise*, which is just another name for integrated white noise, which we just analyzed. So, what would happen if we were to use Brownian noise instead of white noise as action noise on the integrator environment? In this case, the actions themselves are sampled from a Wiener process: $a_t = w(t)$. To calculate the expected distance reached under these actions, we can first write the resulting state

[8]In retrospect, these derivations would have been easier and more relevant in discrete time.

as an Itô integral:

$$\begin{aligned}
x_t &= \int_0^t a_\tau \,\mathrm{d}\tau && (2.35)\\
&= \int_0^t w(\tau)\,\mathrm{d}\tau && (2.36)\\
&= (\tau - t)w(\tau)\Big|_{\tau=0}^{t} - \int_0^t (\tau - t)\dot{w}(\tau)\,\mathrm{d}\tau && (2.37)\\
&= \int_0^t (t - \tau)\,\mathrm{d}w(\tau)\,. && (2.38)
\end{aligned}$$

Here we used partial integration with $f(\tau) := \tau - t, f'(\tau) = 1$ in step (2.37), and in step (2.38) we used the property $w(0) = 0$ from above and wrote $\dot{w}(\tau)\,\mathrm{d}\tau$ as $\mathrm{d}w(\tau)$. It is clear that the state is still distributed according to a zero-mean Gaussian, as the integration (2.35) can be seen as the limit of a sum of zero-mean Gaussians. The variance can be found by making use of Itô isometry:

$$\begin{aligned}
\mathrm{var}[x_t] &= \mathbb{E}\left[\left(\int_0^t (t-\tau)\,\mathrm{d}w(\tau)\right)^2\right] && (2.39)\\
&= \int_0^t \mathbb{E}\big[(t-\tau)^2\big]\,\mathrm{d}\tau && (2.40)\\
&= \int_0^t (t-\tau)^2\,\mathrm{d}\tau && (2.41)\\
&= \frac{1}{3}t^3 && (2.42)
\end{aligned}$$

To find the expected distance, we again make use of the known expected value of the half-normal distribution:

$$\begin{aligned}
\mathbb{E}[|x_t|] &= \int_{-\infty}^{\infty} |x|\,\mathcal{N}(x \mid 0, \frac{1}{3}t^3)\,\mathrm{d}x && (2.43)\\
&= \sqrt{\frac{2}{3\pi}t^3}. && (2.44)
\end{aligned}$$

Thus, Brownian action noise explores the state space much faster than white noise.

However, it turns out that Brownian noise is unsuitable to use as action noise in most situations. The reason for this is that it is not stationary: the variance increases without bounds over time, as can be seen in Eq. (2.31). As most real action spaces are bounded, this behavior is not ideal. Thus, it would be great if we could augment Brownian noise somehow to make it stationary. This is exactly what Ornstein-Uhlenbeck (OU) noise (Uhlenbeck and Ornstein 1930) does. OU noise is defined by the stochastic differential equation (SDE)

$$\mathrm{d}x_t = -\theta x_t\,\mathrm{d}t + \sigma\,\mathrm{d}w_t\,, \tag{2.45}$$

where w_t is again the Wiener process and σ is a scale parameter. If $\theta = 0$, then this equation defines Brownian noise (integrated white noise). However, setting $\theta > 0$

makes this noise stationary, and thus suitable for use as action noise (a typical choice is $\theta = 0.15$). OU noise was actually recommended by Lillicrap et al. (2016) as the default type of action noise for DDPG, and has been shown to lead to a significant increase in state space coverage (Hollenstein et al. 2022)

The easiest way to sample from an OU process is to just simulate the SDE using Euler's method, i.e. by simply discretizing:

$$x[t + \Delta t] = x[t] - \theta x[t]\Delta t + \sigma\varepsilon, \tag{2.46}$$

where $\varepsilon \sim \mathcal{N}(0, \Delta t)$ (see property 2 of the Wiener process). Indeed, this is the method used by the Stable-Baselines3 library (Raffin et al. 2021) which we use in our experiments. Thus, an Ornstein-Uhlenbeck action noise signal of length T ($\varepsilon_{0:T-1} \sim \mathrm{OU}_T$) can be simply generated by sampling T steps from Eq. (2.46). The exploration behavior of this signal, when added as action noise to a policy, depends crucially on the choice of the discretization parameter Δt. We have not found a discussion of this hyperparameter in the context of reinforcement learning, e.g. Lillicrap et al. (2016) only mention the parameter θ. In Figure 2.2 we show a comparison of the exploration speeds of OU noise under different choices of Δt and θ on the integrator environment. The mean-reverting behavior of OU noise only acts on a certain timescale: in the plots we can see that the different θ parameters only start having different effects after some time. On smaller timescales (small $t\Delta t$), OU noise is almost purely Brownian, which can be seen in the plots as the OU lines having the same slope as the Brownian noise. On the other hand, on large time scales (large $t\Delta t$), OU noise behaves like white noise, as can also be seen in the plots when comparing to the white noise slope. Thus, small choices of Δt both makes the noise more Brownian, but for the cost of very slow exploration in the beginning. The default choice in Stable-Baselines3 of $\Delta t = 0.01$ strikes a decent balance, which is also the one we use in all of our experiments (together with $\theta = 0.15$). OU noise also has a second connection to white noise: it can be interpreted as a "leaky integration" of white noise, i.e. white noise passed through a non-ideal low-pass filter. How "leaky" this integrator is, is controlled by the parameter θ: if θ is very small, then the OU process closely resembles Brownian noise (i.e., the integrator is ideal), and if θ is very large, then the OU process will be closer to white noise (i.e., there is no integrator). This can also be clearly seen in the plot.

As a short aside, we should discuss why it is even valid to add action noise to the algorithms we consider, especially as we suggest changing the sampling procedure for stochastic policy algorithms like SAC and MPO, which is a very non-standard thing to do. Of course, we have to assume that the environment dynamics are continuous enough that a slight perturbation of the action will not lead to a vastly different state, but without this assumption, learning itself would be hopeless anyway. The simple answer is that these algorithms are off-policy (they only rely on transitions of the form $(s_t, a_t, r_{t+1}, s_{t+1})$, see Sec. 2.1), so we can actually select arbitrary actions and completely ignore the policy if we wish. Thus adding action noise (no matter which random process was used to generate it) should present no problem at all. However, this is not completely true. Even though actions may come from any policy, all algorithms still require the state-visitation distribution of the

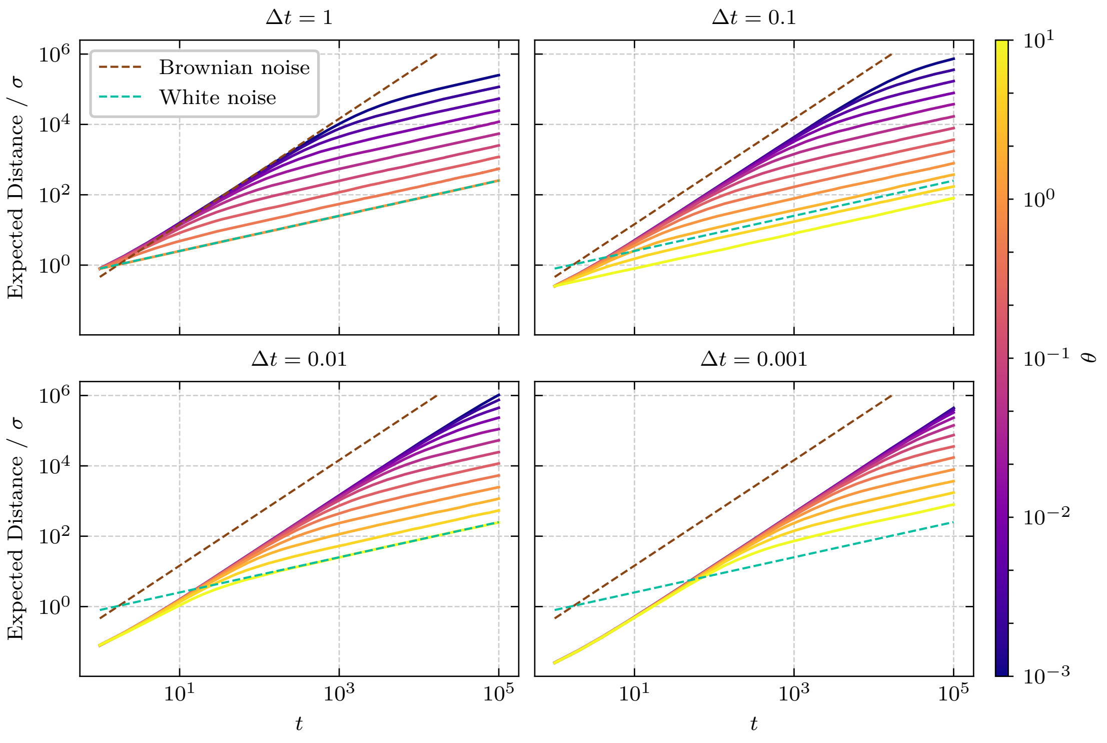

Figure 2.2: The choice of θ, as well as the discretization parameter Δt, both have a significant influence on the exploration speed of OU noise on the integrator environment. The dashed lines for Brownian and white noise are the theoretical results of Eqs. (2.44) and (2.34), respectively.

sampled trajectories to be approximately on-policy (this is the need for exploitation, which we discussed in the previous section). This is not exactly the case if the actions are chosen off-policy, which induces a different trajectory distribution. This distributional shift can lead to problems with strongly off-policy actions, which is especially the case when highly correlated (e.g. OU) action noise is used.

For this reason, Ornstein-Uhlenbeck noise was, after it was recommended for DDPG, abandoned again as the default choice in favor of the more simple white noise for TD3 (Fujimoto et al. 2018). However, as many environments do require more exploration than white noise delivers, a common strategy is to use white noise by default, and alternatives like OU noise only when necessary. In this work, our goal is to find a better strategy, by considering noises with intermediate temporal correlation, in the hope that these work well both on environments where white noise is enough, and on those which require increased exploration. In particular, we consider the general *colored noise* family of temporally correlated noises, which generalizes both white noise and Brownian noise (in this context called *red noise*).

2.4 Colored Noise

Definition 1 (Colored noise)**.** A stochastic process is called a *colored noise* process with color parameter β, if the signals $x(t)$ drawn from it have the property that

$$|\hat{x}(f)|^2 \propto f^{-\beta}, \tag{2.47}$$

where $\hat{x}(f) = \mathcal{F}[x(t)](f)$ denotes the Fourier transform of $x(t)$ and $|\hat{x}(f)|^2$ is called the power spectral density (PSD).

The color parameter β controls the amount of temporal correlation in the signal. If $\beta = 0$, then the signal is uncorrelated, and the PSD is flat, meaning that all frequencies are equally represented. Noise of this kind is called *white noise*, and the uncorrelated white noise that we encountered in the last section, where all time steps were independently sampled from a Gaussian distribution, is one example of colored noise with $\beta = 0$. The reason why this noise is called white noise is in an analogy to light, where a signal with equal power on all visible frequencies is perceived as white.

Colored noise has an interesting property: integrating a colored noise signal with parameter β again yields a colored noise signal, only with parameter $\beta + 2$. We can easily prove this by making use of the property of the Fourier transform that an integration in the time domain corresponds to a multiplication with $(i2\pi f)^{-1}$ in the frequency domain. Let $v(t)$ be the original colored noise signal with $|\hat{v}(f)|^2 \propto f^{-\beta}$. Then the PSD of $x(t) = \int_0^t v(\tau)\,\mathrm{d}\tau$ is

$$|\hat{x}(f)|^2 = \left|\mathcal{F}\left[\int_0^t v(\tau)\,\mathrm{d}\tau\right](f)\right|^2 = \left|\frac{1}{i2\pi f}\hat{v}(f)\right|^2 \propto f^{-2}|\hat{v}(f)|^2 \propto f^{-(\beta+2)}. \tag{2.48}$$

From this, and the definition of white noise as colored noise with $\beta = 0$, it follows that Brownian noise (integrated white noise) is also colored noise with parameter $\beta = 2$. In this context, Brownian noise is often called *red noise*, because it has more weight on lower frequencies, which in light corresponds to the red part of the spectrum. In the center between the uncorrelated white noise ($\beta = 0$) and the strongly correlated red noise ($\beta = 2$), lies the intermediately correlated *pink noise* ($\beta = 1$). A few colored noise signals are shown and compared to Ornstein-Uhlenbeck noise in Figure 2.3.

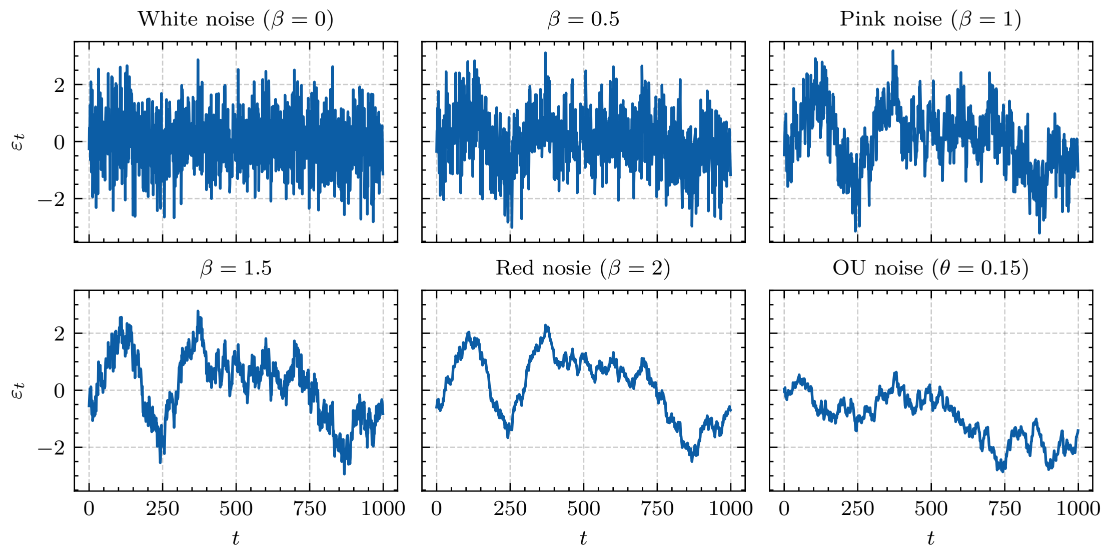

Figure 2.3: Sampled signals from various noise processes with noise scale $\sigma = 1$. It can be seen that red noise and OU noise look very similar.

The signals in Fig. 2.3 were generated according to the procedure described by Timmer and Koenig (1995), which allows us to efficiently sample a complete colored noise signal of a given length T (i.e. $\varepsilon_{0:T-1} \sim \mathrm{CN}_T(\beta)$). This method is very fast, as it only requires sampling a Gaussian signal in frequency space, where the PSD is then shaped, and then transforming it to the time domain via the Fast Fourier Transform (FFT) (Cooley and Tukey 1965). The implementation we use in our experiments is provided by the `colorednoise` Python package.[9] An important property of this noise generation method is that it produces *stationary* signals. This makes colored noise of all β suitable to use as action noise according to Equations (2.26) and (2.28), as the variance stays constant over time. In particular, this is also true for colored noise with $\beta = 2$ if generated this way. Thus, for clarity, in the remainder of this thesis, we will call $\beta = 2$ noise *red noise* if generated this way and *Brownian noise* if we mean integrated white noise (which, as we have seen, is not stationary). The PSDs of the signals from Figure 2.3 are shown in Figure 2.4. On the right plot, it can be seen that red noise is very similar to Ornstein-Uhlenbeck noise. This makes, sense as both can be viewed as stationary versions of Brownian noise.

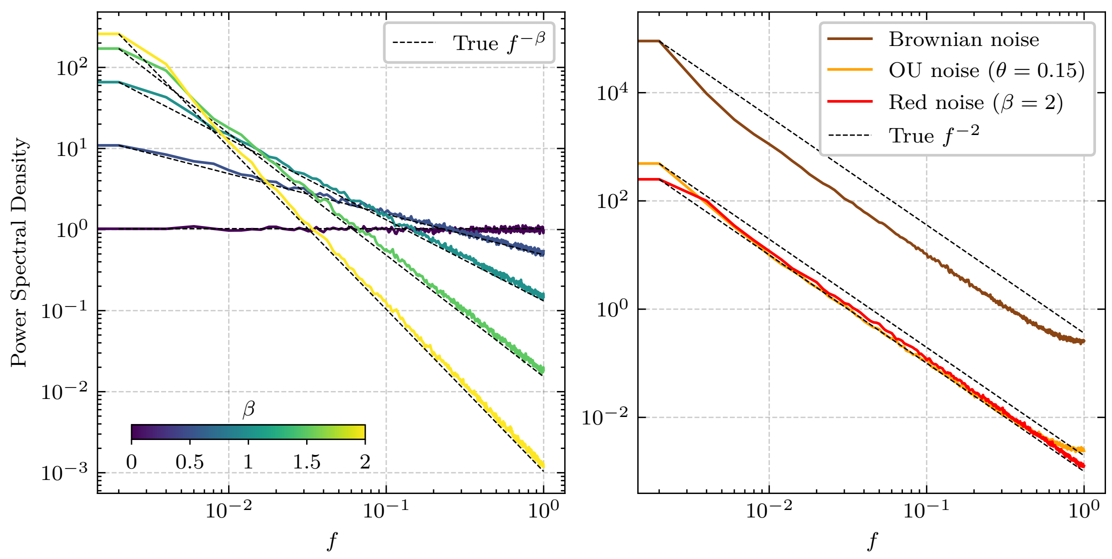

Figure 2.4: *Left*: The power law trends can be seen in the PSDs of sampled colored noise signals. *Right*: Brownian noise, generated by integrating white noise sampled from $\mathcal{N}(0,1)$, is compared to two related stationary noises: Ornstein-Uhlenbeck noise, and red noise (both with noise scale $\sigma = 1$). The similarity between OU and red noise is again very visible (compare to Figure 2.3). Each line shows the average PSD of 500 signals of length $T = 1000$ each.

Besides stationarity, another nice property of the generation method described above is that it produces *Gaussian* colored noise: the signals are marginally identical to standard Gaussian distributions, i.e. if $\varepsilon_{0:T-1} \sim \mathrm{CN}_T(\beta)$, then

$$p(\varepsilon_t) = \int p(\varepsilon_{0:T-1})\, \mathrm{d}\varepsilon_0\, \mathrm{d}\varepsilon_1 \cdots \mathrm{d}\varepsilon_{t-1}\, \mathrm{d}\varepsilon_{t+1} \cdots \mathrm{d}\varepsilon_{T-1} = \mathcal{N}(\varepsilon_t \mid 0, 1). \tag{2.49}$$

[9] `https://github.com/felixpatzelt/colorednoise`. The Python implementation contained a bug, which among other things made it so the generated "white noise" was temporally correlated. Our fix of this bug is included as of version 2.1.0 of the package.

For one thing, this means that white noise sampled as colored noise with $\beta = 0$ is exactly the same as sampling $\varepsilon_t \sim \mathcal{N}(0,1)$ independently at every time step, which is the current default strategy for sampling action noise. Furthermore, it means that the only difference between other colors and white noise is that other colors are temporally correlated (i.e. $p(\varepsilon_t, \varepsilon_{t'}) \neq p(\varepsilon_t)p(\varepsilon_{t'})$). This is important, as we want to use colored noise to study the effects of temporal correlation in the action noise signal. The "marginally $\mathcal{N}(0,1)$" property of the colored noise we use ensures us that our results are only due to a change in the correlation of the action noise, not in the scale or shape of the distribution, as this is the same as of regular Gaussian white noise. In Figure 2.5, the stationarity and "marginally $\mathcal{N}(0,1)$" properties of colored noise are shown empirically on the example of pink noise, and compared to Gaussian white noise (generated via independent sampling) and Ornstein-Uhlenbeck noise. It can be seen that OU noise is stationary, but not marginally $\mathcal{N}(0,1)$, as its marginal distribution is much broader.

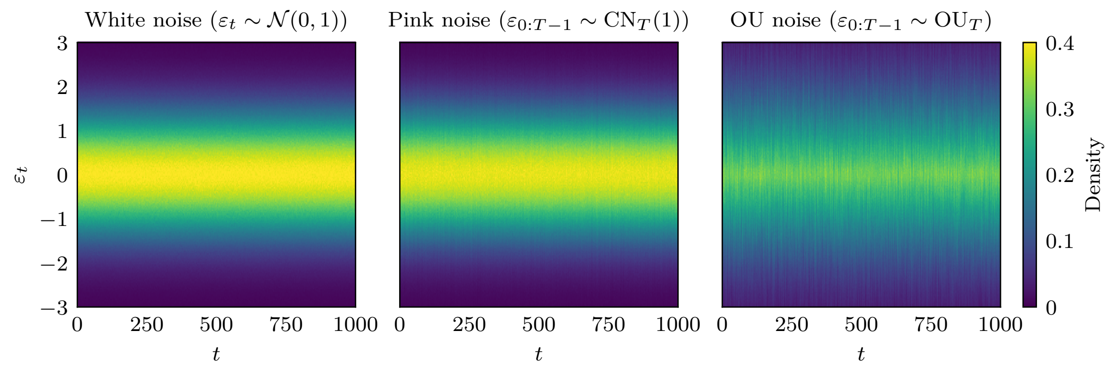

Figure 2.5: The colored noise we use as action noise has the same marginal distribution as independent Gaussian samples. We sampled 3×10^5 action noise signals of length $T = 1000$ from each of the following random processes: independent Gaussian samples (white noise, left), pink noise (center), Ornstein-Uhlenbeck noise (right). At every time step t we show a histogram density estimate over action noise values ε_t. This shows that our results are only due to the increased temporal correlation of the action noise signals, as the marginal distributions remain unchanged from white noise.

Like OU noise, colored noise has two parameters: the color parameter β, as well as the sequence length or "chunk size" T: to get a signal of length t, we can either generate a signal of length $T \geq t$ and use a sub-signal of the correct length, or we could generate several signals ("chunks") of length $T < t$ and stitch them together. T plays a similar role to the discretization parameter Δt we discussed for OU noise: if T is very small (and we stitch several chunks together to get a signal of length t), the correlation between time steps gets smaller, and the signal will more closely resemble white noise, and if T is very large, then the temporal correlation will also be large. The effect of choosing different chunk sizes T on the pure noise agent's behavior on the integrator environment is shown in Figure 2.6. If we are faced with an infinite horizon task, then we would indeed have to stitch several (finite-length) action noise signals together. However, as we only consider the finite horizon setting, where a rollout is always a fixed number of time steps long, we will simply choose this length as our sampling length, i.e. $T =$ episode length.

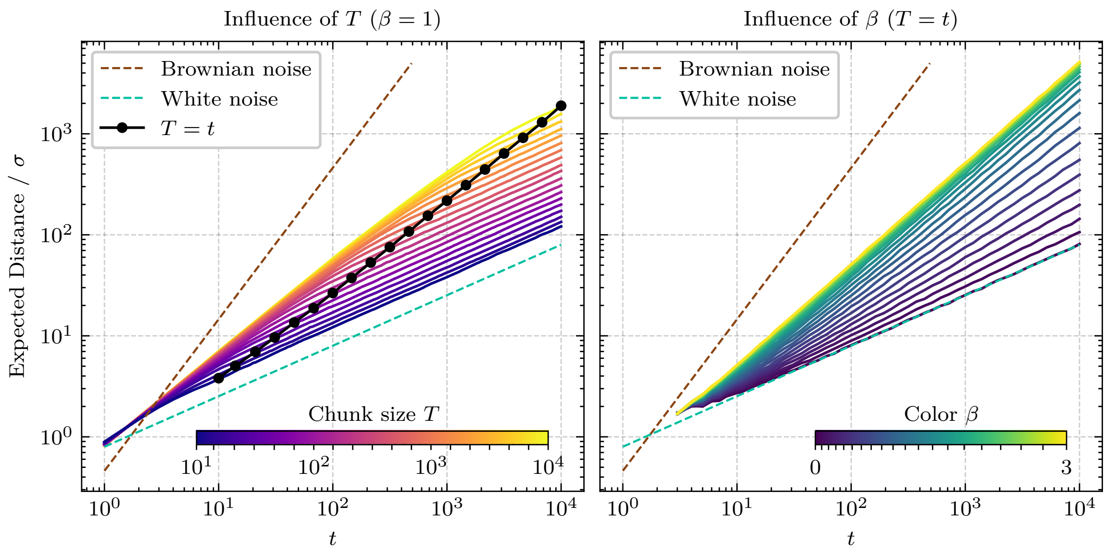

Figure 2.6: *Left*: The chunk size T has a considerable effect on the generated colored noise signal (here, pink noise). In our experiments we always use $T = t$, where t is the rollout length. *Right*: The higher the color parameter is, the faster will colored noise explore the space of the integrator environment.

The arguably more important parameter is the color β, whose impact on the integrator environment is also shown in Figure 2.6. As this parameter controls the amount of temporal correlation, a larger β leads to faster exploration, while a smaller β leads to slower exploration. Thus, colored noise has brought us what we wanted: a noise type which is valid as action noise due to its stationarity, which can be efficiently generated, and whose temporal correlation can be tuned to be higher than white noise's, but lower than Ornstein-Uhlenbeck noise's. By setting $0 < \beta < 2$ to get intermediate temporal correlation between white and red noise (which, as we have seen, is very similar to OU noise), colored noise might present us with a better default action noise type than white noise. However, it is still not clear how exactly we should set the color parameter β. After all, our suggestion to "use colored noise instead of white noise" is actually just the suggestions to "use a different β". So the important question that remains is: which β?

Our inspiration to use colored noise as action noise in model-free reinforcement learning comes from the model-based setting. Pinneri et al. (2020) introduced the *improved cross-entropy method* (iCEM). This method is closely related to the (original) cross entropy method (CEM), which we discussed briefly in Section 2.1. One of iCEM's main improvements to CEM was to use temporally correlated colored noise for action selection instead of the uncorrelated Gaussian samples used in CEM. Thus, to see how we should choose β, the natural place to start is to see how Pinneri et al. (2020) chose β for iCEM. The answer is that they kept it as a hyperparameter, choosing it individually for each task. They found that it is important to choose the correct value for β for each task, as each one has a certain "preference". For example, the "HalfCheetah Running" task required a low β, with the best performance at $\beta = 0$, whereas the "Humanoid StandUp" task performed poorly with $\beta = 0$ and needed a higher value for good results (Pinneri et al. 2020, Figure S8). These results are not too surprising. In the Cheetah task, the goal is to run fast. For this motion, the joints have to move at very high frequencies, so it makes sense that action noise with larger power in higher frequencies is preferred. On the other hand, this sort of motion is not required by the Humanoid task, where the goal is only to stand up. Here, it makes sense that noise which is highly correlated performs better, as a stand-up process consists mostly of low-frequency movements.

However, despite these findings in iCEM, it is still possible that it is not necessary to tune the color parameter in the context of model-free reinforcement learning. This setting is very different from trajectory optimization: the colored noise samples are not only shifted and scaled as in iCEM, to directly yield the action sequence. Instead, we only add a bit of action noise to the actions which are generated by the policy from a sequence of states. As the states are naturally correlated over time, so is the action sequence, even without adding correlated noise. In Chapter 3, we will investigate the necessity of adapting β to the environment and see if it is possible to find a solution which works well across many tasks without adaptation.

A different question that arises in the RL setting is whether it might be beneficial to change β over the course of training. If we think again about the HalfCheetah environment, it might be that the high frequencies are actually more important once the policy has learned how to move forward at all, for which low frequencies might even be better. This “curriculum learning” intuition could justify selecting β according to a schedule or adapting it online while learning the policy.

These considerations lead us to four different schemes for selecting β, which are also outlined in Figure 2.7. In Chapter 3, we don’t let β vary with time, but instead keep it constant throughout training. Here, we investigate whether it is necessary to adapt β to each environment (Sec. 3.2), as is done in iCEM manually, or if we can find a constant value for β which works well across tasks (Sec. 3.1). In Chapter 4, we then look at ways to vary β over the course of training. We begin in Section 4.1 by looking at non-adaptive fashions to vary β, such as using a schedule. Finally, in Section 4.2, we then try to actively adapt beta to the environment while interacting with it. For this, we introduce a Bayesian optimization method based on the Thompson sampling algorithm from Section 2.2.

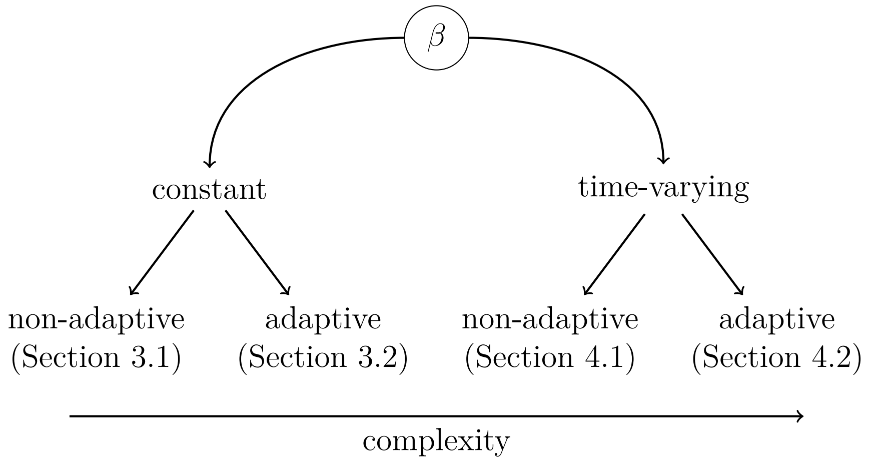

Figure 2.7: The most important choice in using colored noise for exploration is the color parameter β. We can either keep β constant throughout training, or vary it with time. Additionally, we can either adapt it to the task at hand, or apply a non-adaptive strategy across all tasks. This leads to four schemes which are discussed, in approximate order of increasing complexity, in Chapters 3 and 4 of this thesis.

Chapter 3

Constant Color

Let us start with the easy case: letting β be constant over the course of training. The best-case scenario would be if there is a value for β which works well across many tasks and environments, without any need of adaptation. This would make it very easy to use in practice. We will start by searching for such a task-agnostic β, as this will give us a simple baseline which we can then try to improve upon in the remainder of this thesis.

3.1 Non-Adaptive Methods

We proceed by running a set of experiments to find out how well this scheme can work, using a variety of environments and color parameters. The continuous control environments we use include the MountainCar implementation in OpenAI Gym (Brockman et al. 2016), a door opening tasks from the Adroit hand suite (Rajeswaran et al. 2018), as well as 8 different tasks from the DeepMind Control Suite (DMC, Tassa et al. 2018). All environments are depicted in Figure 3.1, and the respective sources and exact IDs are compiled in Table 3.1.

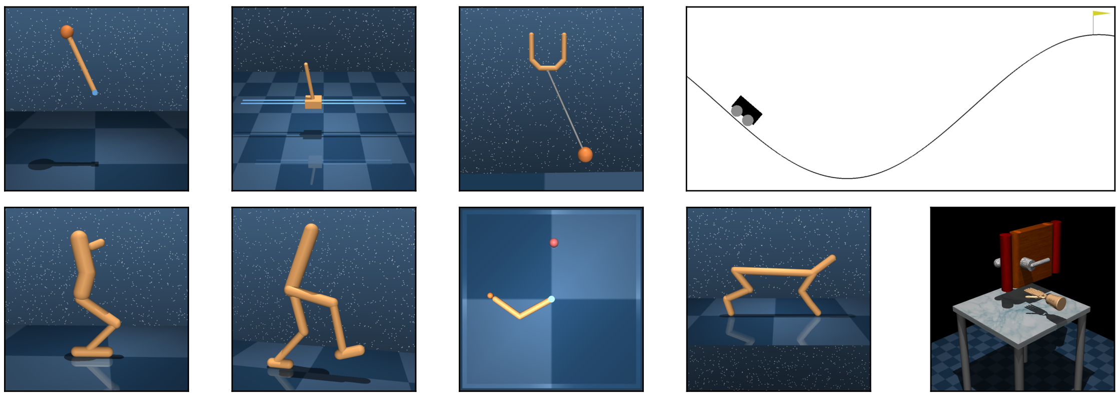

Figure 3.1: The environments we use: Pendulum, CartPole (balance + swingup tasks), Ball-In-Cup, MountainCar, Hopper, Walker, Reacher, Cheetah, Door. Images partly taken from Tassa et al. (2018) with permission.

Environment	Source	ID
Pendulum	DMC	`pendulum (swingup)`
CartPole (b.)	DMC	`cartpole (balance_sparse)`
CartPole (s.)	DMC	`cartpole (swingup_sparse)`
Ball-In-Cup	DMC	`ball_in_cup (catch)`
MountainCar	Gym	`MountainCarContinuous-v0`
Hopper	DMC	`hopper (hop)`
Walker	DMC	`walker (run)`
Reacher	DMC	`reacher (hard)`
Cheetah	DMC	`cheetah (run)`
Door	Adroit	`door-v0`

Table 3.1: Environments used in this work (see also Fig. 3.1).

Sometimes we need to talk about the whole set of environments, which we then denote as

$$\mathcal{E} = \{E_1, E_2, \dots, E_{10}\}.$$

We chose this rather large set of environments mainly because of its diversity. As explained in Section 2.4 with the HalfCheetah and Humanoid examples, we suspect that each environment has a certain preference for the color parameter. Some tasks will probably be served very well with the default white noise, but others may not. This diversity of tasks makes it less likely that we get misleading results, for example by accidentally choosing environments which all happen to work very well with a specific value of β. We chose all these environments before doing the experiments, and did not discard any additional ones.

The second variable in these experiments is the color parameter β. We can only do finitely many experiments, so we need to choose a finite set of values to try out. We should definitely include white noise ($\beta = 0$) to check if the performance is comparable to the default implementations which draw Gaussian noise independently at each time step. Red noise ($\beta = 2$) should also be included, to compare to Ornstein-Uhlenbeck noise, as these two noises are very similar (see Sec. 2.3). Should we include any $\beta > 2$? As discussed previously, our goal is to find an action noise of *intermediate* temporal correlation, so our main focus should lie on the interval $\beta \in [0, 2]$. Additionally, if we assume that the environment can be modelled as an integrator (such as in the last chapter), or as a double integrator (i.e. it converts forces or torques to positions), it turns out that colored noise of $\beta > 2$ does not actually manage to significantly increase the speed of exploration compared to red noise (at least, not when generated according to the method described in Sec. 2.4). This is shown in Figure 3.2. Thus, we will not consider any β higher than 2.

What about negative values of β? These signals are well-defined, though they are a bit strange conceptually: a colored noise signal with $\beta = -2$ can be seen as the derivative of white noise. As we are interested in noise with intermediate temporal correlation (as explained in Section 2.3), we will not consider any negative β, and restrict ourselves to the interval $[0, 2]$.

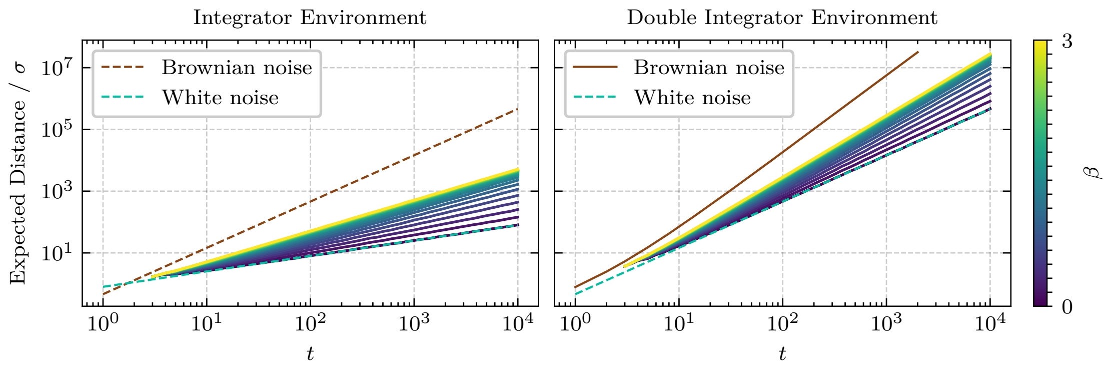

Figure 3.2: When generating colored noise as described in Section 2.4, the exploration speed does not increase indefinitely with higher β. It can be seen that setting $\beta > 2$ makes little difference in speed compared to $\beta = 2$. Thus, we don't try out any $\beta > 2$ in our experiments. Here, dashed lines are theoretical results (see Sec. 2.3), and solid lines are empirical results.

As we can see in Figure 3.2, a perturbation to a $\beta \ll 2$ will cause a much greater change in the final distance than a perturbation to a β close to 2. For this reason we don't choose an equidistant list of values, but instead go for

$$B = \{0, 0.1, 0.2, 0.35, 0.5, 0.75, 1, 1.5, 2\}.$$

Now that we have selected (hopefully appropriate) values for the two variables, it is time to train some agents and analyze their performance! We use MPO, SAC, and TD3 for all experiments, but only show the results for MPO in the main text. The results for SAC and TD3 can be found in Section A. The base implementation of the MPO algorithm that we use comes from the Tonic library (Pardo 2020) and the base SAC and TD3 implementations from Stable-Baselines3 (Raffin et al. 2021). We repeat these experiments with 20 random seeds. The main reason for this large amount of seeds is that deep reinforcement learning is notoriously difficult to reproduce (Henderson et al. 2018). Using many seeds helps to make sure that the results we see are not only due to randomness, but in fact allow us to draw conclusions about the effectiveness of our methods.

One last thing we need to discuss before coming to the results is how we evaluate performance. The most important metric for any reinforcement learning method is arguably the performance of the final policy. This is normally measured by running several evaluation rollouts with the policy and averaging the returns of these rollouts. In an evaluation rollout, no exploration is performed. For deterministic policies, this means that no action noise is added, and for stochastic unimodal policies it means that the mode of the distribution over actions is taken instead of sampling from it. A different metric than the final policy performance is the time it takes to reach this performance. This is the sample efficiency of the algorithm: how many interactions with the environment are necessary to train until convergence. We are interested in both final performance and sample efficiency, so we take a hybrid approach: every 10,000 interactions with the environment, we run 5 evaluation rollouts and take the average of all 5 returns. A complete training always lasts exactly 1,000,000 environment interactions, and we measure performance by averaging the outcome of all evaluation rollouts. This is akin to calculating the area under the learning curve. Thus, for high performance, it is not only important to have high final performance, but also to be sample efficient and reach this final performance quickly. To be clear about what exactly is meant by "performance", we introduce a new notation. By

$$\mathrm{perf}(E, \beta)$$

we mean the performance (according to the averaging of evaluation returns, as explained above) that results from training an agent (in the main text, MPO) on environment E with colored noise exploration of parameter β. This is a random variable, as both the environment E and the training procedure are nondeterministic. By specifying a random seed s, we can sample from this random variable and get the deterministic quantity

$$\mathrm{perf}(E, \beta, s).$$

Thus, our experiments will, for every environment $E \in \mathcal{E}$, and every $\beta \in B$, result in a set of performances $\{\mathrm{perf}(E, \beta, s) \mid s = 1, 2, \ldots, 20\}$.

In this section, our goal is to find a single value of β that performs well across many environments. How do we measure performance across environments? We want something like the average performance over all environments. To compute this, however, we must first make the performances comparable across environments. The DeepMind Control Suite tasks are all constructed such that the maximum rollout return is 1000 and the minimum return is 0. However, the other two tasks (MountainCar and door opening) have different scales and can also have negative-valued returns. To make all performances comparable we standardize the performance on each environment using the sample mean $\mu(E)$ and standard deviation $\sigma(E)$ computed from *all performances recorded on that environment*[1] (using the same agent, i.e. MPO in the main text). We call this standardized quantity the "relative performance" perf′:

$$\mathrm{perf}'(E, \beta) = \frac{\mathrm{perf}(E, \beta) - \mu(E)}{\sigma(E)}, \tag{3.1}$$

where the sample corresponding to the random seed s is again denoted $\mathrm{perf}'(E, \beta, s)$. It is debatable whether this is the best technique to make the performances comparable, but it ensures that all environments are on the same scale, and using all experiments avoids the problem where one "outlier" dominates averages of these performances. Using this definition, we now want to know the average (relative) performance over all environments, or

$$\mathrm{perf}(\beta) = \frac{1}{|\mathcal{E}|} \sum_{E} \mathrm{perf}'(E, \beta). \tag{3.2}$$

We could sample from this random variable simply by using each random seed once:

$$\mathrm{perf}(\beta, s) = \frac{1}{|\mathcal{E}|} \sum_{E} \mathrm{perf}'(E, \beta, s). \tag{3.3}$$

[1]This includes all experiments from the whole thesis, not just from this section.

However, if we sample with replacement, we can get a nicer distribution for $\mathrm{perf}(\beta)$: by sampling the seed s_{ij} uniformly from $\{1, 2, \ldots, S\}$, where S is the number of seeds (in out case, $S = 20$), for each $i \in \{1, 2, \ldots, N\}$ and $j \in \{1, 2, \ldots, |\mathcal{E}|\}$, we can get N approximate samples

$$\mathrm{perf}(\beta, s_i) = \frac{1}{|\mathcal{E}|} \sum_j \mathrm{perf}'(E_j, \beta, s_{ij}) \tag{3.4}$$

for the average performance $\mathrm{perf}(\beta)$. We can also use sampling with replacement to get a bootstrap distribution for the expected value of $\mathrm{perf}(\beta)$. If we take S such samples (where S is again the number of seeds), i.e. we choose s_{ijk} uniformly from $\{1, 2, \ldots, S\}$, where now $k \in \{1, 2, \ldots, S\}$, and take the average of all S samples, we get a bootstrapped estimate for the expected value of $\mathrm{perf}(\beta, s_i)$:

$$\hat{\mathbb{E}}\,\mathrm{perf}(\beta, s_i) = \frac{1}{S|\mathcal{E}|} \sum_k \sum_j \mathrm{perf}'(E_j, \beta, s_{ijk}). \tag{3.5}$$

Repeating this N times gives us a bootstrap distribution for the expected average performance.

Now we can finally come to the results of the experiments! In Figure 3.3, the sampled average performances for each $\beta \in B$ (Eq. 3.4) as well as the bootstrap distribution for the expected average performances (Eq. 3.5) are shown and compared to the two baselines white noise (i.e. the default noise) and Ornstein-Uhlenbeck noise. We can directly see that on this selection of environments, a constant colored action noise with parameter $\beta = 1$, i.e. pink noise, outperforms both baselines by quite a bit. This is good news: we can improve average performance already by simply using pink noise instead of white noise as a default choice.

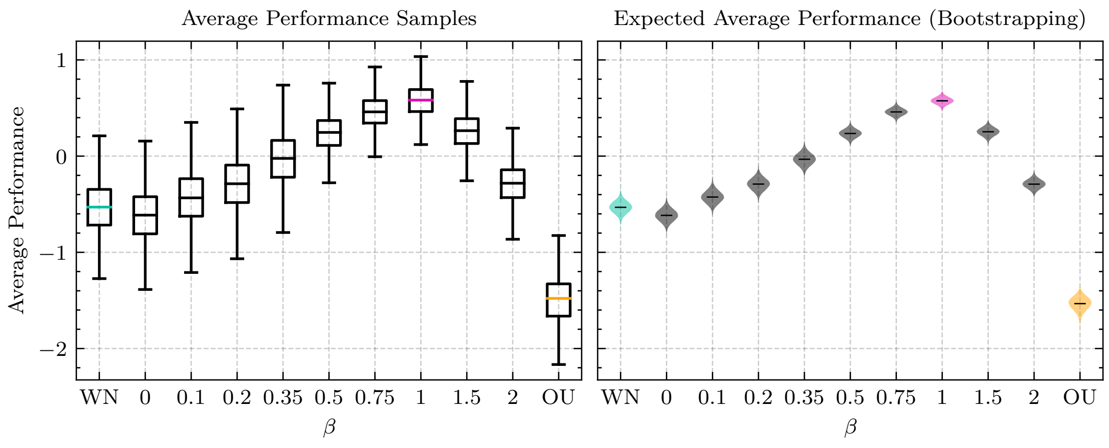

Figure 3.3: *Left*: Samples of MPO's average performance across environments when using different action noise types. *Right*: Bootstrap distributions for the expected average performances. Highlighted are white noise (WN), pink noise ($\beta = 1$), and Ornstein-Uhlenbeck noise (OU). We used $N = 10^5$.

Clearly recognizable in Figure 3.3 is a distinct shape in how the chosen β influences the performance. The performance seems to smoothly rise with increasing β, up to the point where $\beta \approx 1$. Then it drops again, and $\beta = 2$ performs similarly to white noise. It seems unlikely that this shape is just a random fluctuation in the data. In fact, we see a similar shape in the results from SAC and TD3, though not quite as pronounced (see Sec. A). This gives us further confidence in the conclusion that pink noise is indeed the better choice, but it also raises the question of why we see this shape. To answer this, it is important to look at the performances for the individual environments, to see if the same curve appears on each environment, or if it is instead an artifact of averaging over many environments. Of course, the underlying idea is that we want to know if we should adapt beta to the environments, or if indeed all environments prefer pink noise. The results for the individual environments are shown in Figure 3.4.

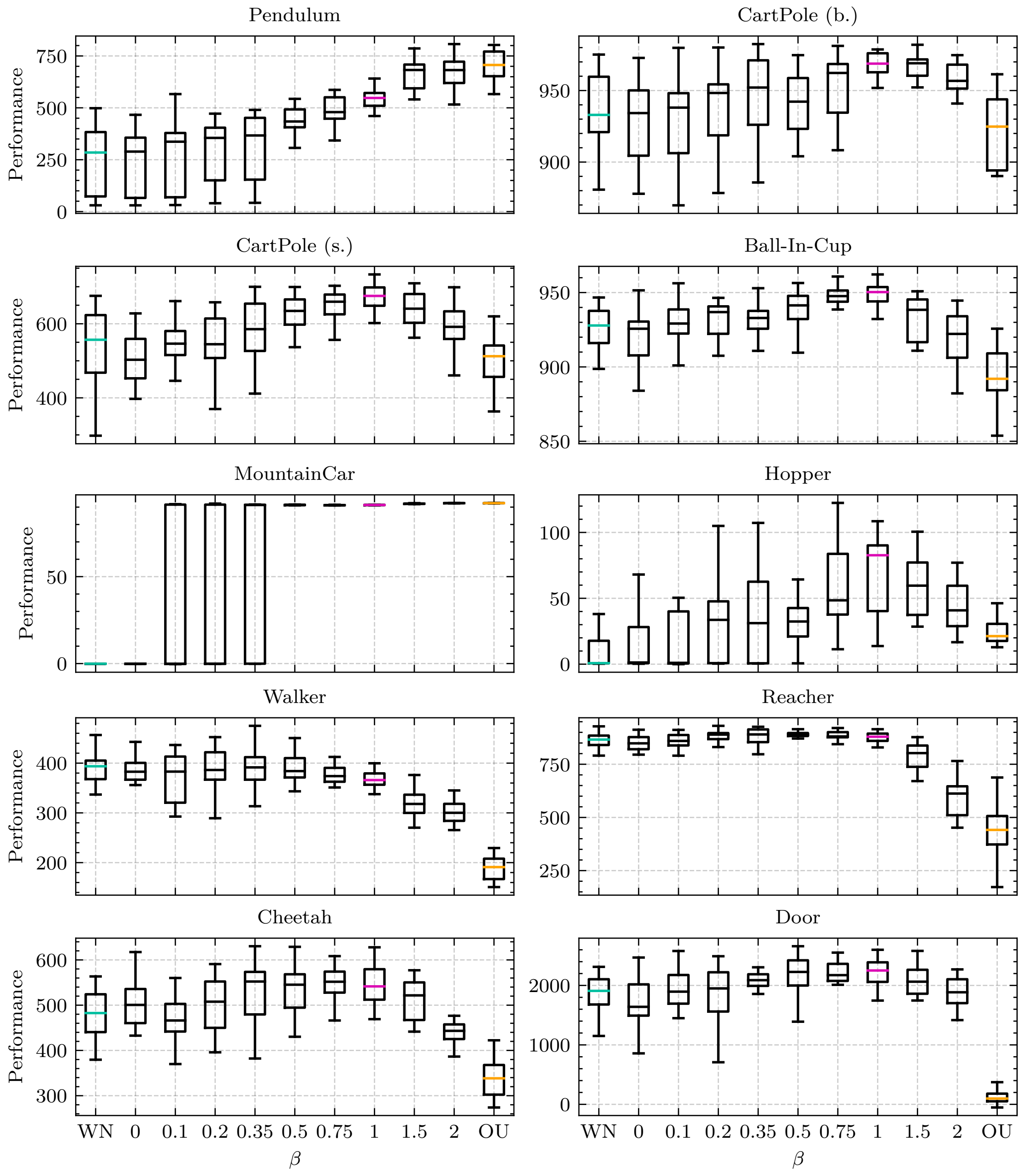

Figure 3.4: Performances of MPO on each environment in our selection when using different action noise types. Highlighted are white noise (WN), pink noise ($\beta = 1$), and Ornstein-Uhlenbeck noise (OU).

The first thing we see is that we do not see the same curve from Figure 3.3 everywhere. Instead, each environment looks very different. When looking carefully at these results, it looks broadly like there are three different types of environment:

1. Higher $\beta \to$ higher performance.
   This type includes the MountainCar and Pendulum environments.
2. Higher $\beta \to$ lower performance.
   This type includes the Walker and Reacher environments.
3. Peak near $\beta = 1$.
   On the rest of the environments, the relationship between performance and β is a bit less pronounced, but on most of these a bell-shaped curve comparable to that of Figure 3.3 is recognizable. Indeed, on four out of the six remaining environments, pink noise achieves the best median performance. Of course, these results are noisier and the bell curve is much less pronounced than in Figure 3.3, but the underlying cause could still be the same.

So why do we see these curves? The first type is easy to explain. We have already discussed why MountainCar might benefit from high values of β in Chapter 1. The Pendulum task is very similar: in both of these tasks the challenge is to "swing up" a mass with limited force (a form of underactuation), with a sparse reward once a certain threshold has been passed. It makes sense that exploration noise which is dominated by low-frequency components will make it easier to "accidentally" accomplish this. One way to look at this is that for successfully swinging an oscillator up, the best force signal should excite the oscillator at its resonant frequency. In these settings, the resonant frequencies of both the mountain in MountainCar and of the pendulum are relatively low, such that white noise alone will not be able to achieve a high amplitude. We will return to this discussion in Chapter 5. The intrinsic correlation of the policy's action sequence can sometimes still solve the task, as is seen from the white noise results on Pendulum, but the correct action noise can make a very large difference. In general, tasks which only really require exploration can be expected to benefit from highly correlated action noise, be it either in the underactuated "swing-up" setting described here, or in settings where it is important to reach a high distance in a short amount of time.

What about the second type? This one is not quite as straightforward. We have already discussed the main problem with strongly correlated noise in Section 2.2: high action noise correlation will lead to a more off-policy trajectory distribution. This off-policy data then hinders learning which results in low performance. To make this more concrete, how exactly is this distributional shift manifested in the two environments of "second type", Walker and Reacher? The reacher consists of an arm with a single joint in the middle (see Fig. 3.1). The task is to reach a small circle somewhere in the arm's two-dimensional range, with a sparse reward once the arm's tip is inside the circle. For early rollouts it is not clear if colored noise makes any difference, but once the policy has approximately learned to reach the circle, highly correlated action noise might lead to a situation where a good policy almost never hits the circle, getting 0 reward. If this happens, it would tell the policy that it is doing something wrong, and it will not be able to improve itself. For the walker environment, highly correlated noise might cause the walker to fall over. If this does not happen on-policy this is again a serious distributional shift, explaining why learning becomes difficult. Theses are only guesses about what the concrete problems of highly off-policy data are in these environments. We can observe that high correlation actually leads to problems on all environments, except the two of first type. So what is really special about the two environments is not that high β perform badly, but that low β perform well.

This is not the case on the majority of environments: the “third type”. There is a simple explanation for why we see curves which peak near $\beta = 1$. These tasks might simply suffer from the difficulty of underactuation (making low β perform poorly) as well as requiring data that is not too off-policy (making high β perform poorly). A different point of view is that high β and low β lead to different kinds of exploration. High β, being good at bringing an agent far and discovering high reward regions, are useful for a sort of *global exploration*. On the other hand, low values of β will not lead the agent far from where the policy would go without noise. But it does provide some variance in the trajectory distribution, which can (through a sort of *local exploration*) optimize finer details of the trajectory. This might paint a somewhat simplified picture, but it is still reasonable to expect most environments to require both global and local exploration. As pink noise has intermediate temporal correlation, it can be expected that it can provide a mixture of both types of exploration, making it a better choice on these environments than white noise or OU noise, which are both only good at providing one of the two types. We will return to this discussion in Section 4.1, as well as in Chapter 5.

How do these results on the individual environments relate to the general trend we saw in Figure 3.3? This trend can be easily explained now: if some environments prefer high β, some prefer low β and others prefer $\beta \approx 1$, then averaging these preferences will naturally yield the bell-shaped trend line that we observe. A different hypothesis for why the preferences of environments of third type look similar to the overall preferences, is that it may be for the same reason. Maybe these environments are a bit more complex than the environments of first and second type, and already encompass different “subtasks” which themselves prefer only either high or low β. Then the reason for why some environments themselves prefer $\beta \approx 1$ may be that they can be seen as an “average” over more simple subtasks, and so the same bell-shaped curve appears. We will return to this hypothesis in Chapter 5.

What final conclusions can we draw from this set of experiments? It seems to be the case that pink noise is a very good choice in general. On almost all environments it outperforms both white noise and Ornstein-Uhlenbeck noise. The learning curves comparing pink noise to both baselines are shown in Figure 3.5. However, even though some environments prefer pink noise over the other colors, many prefer a different noise, such as white noise or Brownian noise. Thus, it seems to be a good idea to try and find out which noise an environment prefers and then adapt β accordingly. This is the subject of the next section.

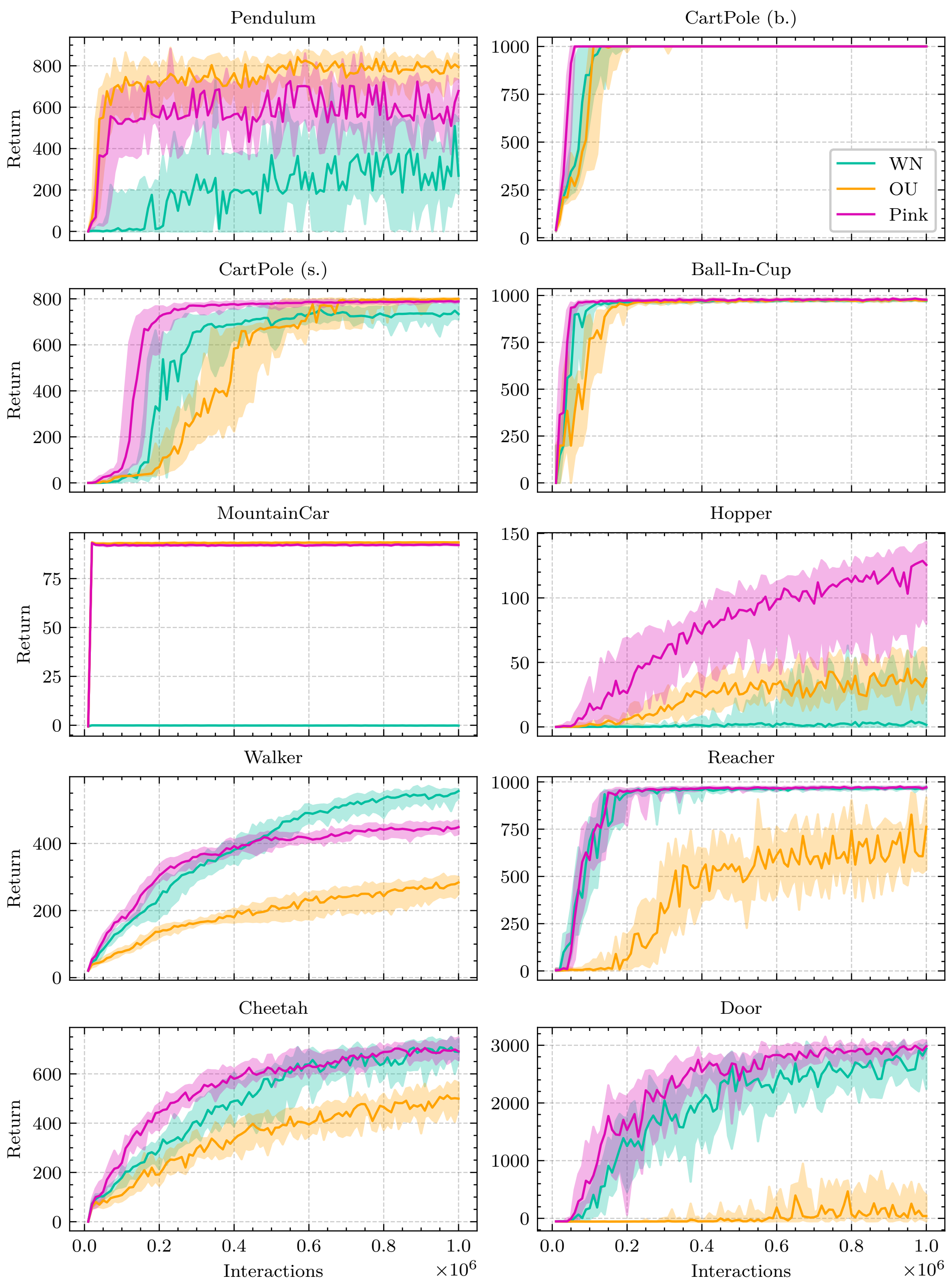

Figure 3.5: Learning curves on MPO (median and interquartile range of evaluation returns) of the two baseline action noise types white noise (WN) and Ornstein-Uhlenbeck (OU) noise, as well as of pink noise. It can be seen that pink noise, while not being better than both baselines on all environments, is the best default choice. It is never outperformed by both white noise and OU noise, and routinely outperforms white noise (e.g. MountainCar), OU noise (e.g. Door), or both (e.g. Hopper).

3.2 Adaptive Methods

In the last section we saw that each environment has its own specific "β preferences", which we broadly split into three different categories. Another way to say this is that β is a hyperparameter of the algorithm to which most environments are sensitive. This is also what Pinneri et al. (2020) found with iCEM. However, instead of tuning β manually through an inefficient method like grid search, it would be great if we can somehow infer a good value for beta directly from the interactions with the environment, i.e. have some function which takes in trajectories and infers an optimal value for β.

Before we devise a concrete technique to do this, let us first check how good we can hope to be with such a strategy. As we still only consider values for β that remain constant over the whole training, we can actually find this out quite easily by reusing the data from the experiments of the last section. This data lets us create an "oracle" estimate for an environment's best β setting:

$$\hat{\beta}^\star_E = \arg\max_\beta \operatorname{median}\{\operatorname{perf}(E, \beta, s) \mid s = 1, 2, \dots, 10\}. \tag{3.6}$$

We estimate the best β for environment E by simply taking the one which performed best on the first 10 seeds. We can then evaluate the performance of this β on the remaining 10 seeds; this strategy avoids the sampling bias that would incur if we were to choose $\beta^\star_E$ in accordance to the same seeds that we evaluate on. As we saw that pink noise ($\beta = 1$) was not the preferred choice on all environments, we should expect this oracle scheme to outperform the "using only pink noise" scheme. In Figure 3.6, the average performance of this oracle is compared to pink noise, white noise, and OU noise, as well as to an "anti-oracle", in which the arg max in Eq. (3.6) is replaced with an arg min.

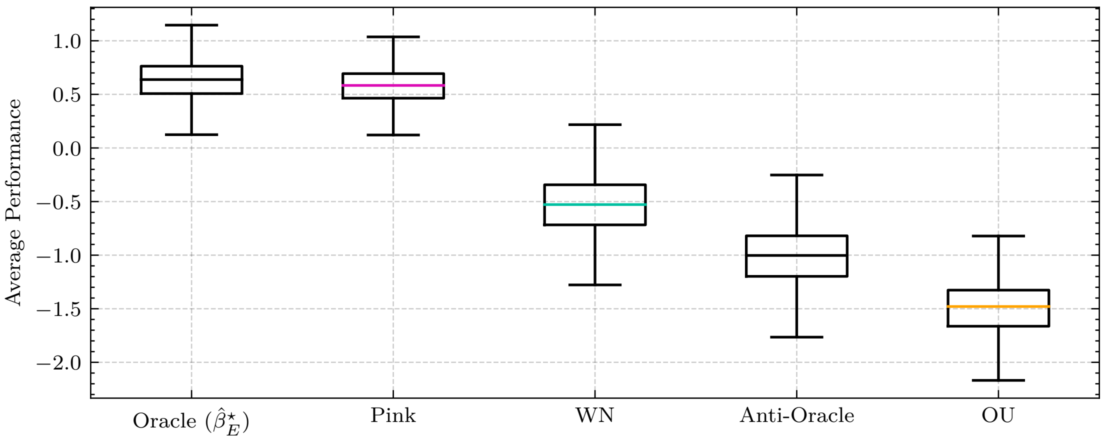

Figure 3.6: The average performance of an oracle β selection scheme is not significantly better than simply using pink noise.

Interestingly, it looks as if pink noise actually performs almost as good as the oracle scheme! Ornstein-Uhlenbeck noise even performs worse than the "worst β" selection, which can be thought of as the most unlucky pick of β for each environment possible. The learning curves corresponding to the oracle β selection are shown in Figure 3.7, where the 4 environments with $\hat{\beta}^\star_E = 1$ (i.e. pink noise is already the best noise) are omitted. Again, it looks like pink noise and the "best β" colored noise perform virtually identically on most environments (certainly on the four not shown). The only two environments where the best β seems to truly improve the performance are Pendulum and Walker, although the improvement is not enough to be reflected in the average performance.

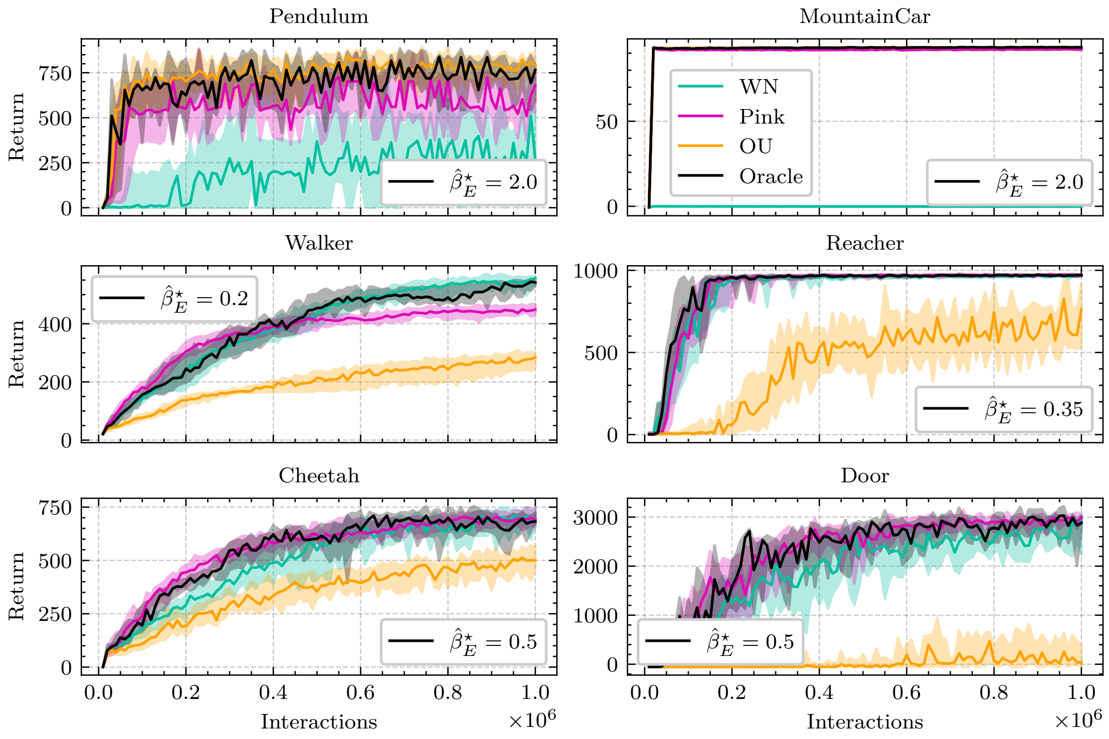

Figure 3.7: Learning curves on MPO. Pink noise compares favorably to an oracle β selection scheme on most environments.

If we cannot even improve performance over pink action noise by *running 10 complete trainings with each β candidate and choosing the best performing one*, it seems that there is no point in trying to adapt a constant β to an environment, and we should just use pink noise everywhere. But why do we get this result? If we look back at Figure 3.4, we can actually see that Pendulum and Walker are the only two environments where pink noise is not (almost) the best. This explains why these are the only environments with significant improvement. But it does not explain why fitting β to each environment in this way does not improve performance at all. Looking at Figure 3.4, we see that most environments don't prefer $\beta = 1$, and of course this is also why $\beta^\star_E \neq 1$ in most cases. So why don't the results improve?

The reason is randomness. In the oracle scheme, we select β by using the one which performs best in a number of trial runs (the first 10 seeds). But, as randomness plays a significant role in the outcome of any deep RL experiment (Henderson et al. 2018), we are not guaranteed that the best β in the trial runs is equal to the best β in the test runs (the remaining 10 seeds). To assess if adapting a constant β is generally worth it, let us examine the randomness involved here.

The quantity we are interested in is the "best β" on a given environment, measured in terms of its performance $\mathrm{perf}(E, \beta)$. As explained in the last section, the performance is a real-valued random variable (of unknown distribution) which we have access to through a set of i.i.d. samples $\{\mathrm{perf}(E, \beta, s) \mid s = 1, 2, \ldots, 20\}$. Thus, the "best β" on a given environment is also a random variable:

$$\arg\max_{\beta} \mathrm{perf}(E, \beta).$$

This random variable is categorical, as there is a certain probability for each β in the given list of colors B to be the best, which we call that beta's "best-ratio" BR:

$$\mathrm{BR}(E, \beta) := \mathbb{P}\left[\arg\max_{\beta' \in B} \mathrm{perf}(E, \beta') = \beta\right].$$

These probabilities are the parameters of the categorical distribution. They can be interpreted by considering a simplified experiment: if we are given an environment E and a list B of colors and train once with each $\beta \in B$, then

$$\mathrm{BR}(E, \beta) = \mathbb{P}[\mathrm{perf}(E, \beta) \geq \mathrm{perf}(E, \beta'), \forall \beta' \in B]. \tag{3.7}$$

We can estimate these probabilities using the data from our experiments, using a similar technique to the estimation of the average performance in Section 3.1. If we sample the seed s_{ij} uniformly from $\{1, 2, \ldots, 20\}$ for each $i \in \{1, 2, \ldots, N\}$ and $j \in \{1, 2, \ldots, |B|\}$, we can get N approximate samples

$$\hat{\mathrm{BR}}(E, \beta) := \frac{1}{N} \sum_{i=1}^{N} \left[\!\left[\arg\max_{j} \mathrm{perf}(E, \beta_j, s_{ij}) = \beta\right]\!\right]. \tag{3.8}$$

These estimated best-ratios are shown in Figure 3.8.

We wanted to know whether the best β of the trial runs is also the best β of the test runs. We can simplify this question a bit and only consider the situation where we have a single trial and a single test run. Let us denote the performances of the trial run as $\mathrm{perf}_{\mathrm{tr}}(E, \beta)$ and the performances of the test run as $\mathrm{perf}_{\mathrm{ts}}(E, \beta)$. These are both i.i.d. random variables; the quantity of interest is thus:

$$\begin{aligned}
&\mathbb{P}\left[\arg\max_{\beta \in B} \mathrm{perf}_{\mathrm{tr}}(E, \beta) = \arg\max_{\beta \in B} \mathrm{perf}_{\mathrm{ts}}(E, \beta)\right] && (3.9)\\
&= \mathbb{E}_{\beta_{\mathrm{tr}}, \beta_{\mathrm{ts}} \sim \arg\max_{\beta} \mathrm{perf}(E, \beta)} [\![\beta_{\mathrm{tr}} = \beta_{\mathrm{ts}}]\!] && (3.10)\\
&= \sum_{\beta_{\mathrm{tr}} \in B} \mathbb{P}\left[\arg\max_{\beta \in B} \mathrm{perf}(E, \beta) = \beta_{\mathrm{tr}}\right] \sum_{\beta_{\mathrm{ts}} \in B} \mathbb{P}\left[\arg\max_{\beta \in B} \mathrm{perf}(E, \beta) = \beta_{\mathrm{ts}}\right] [\![\beta_{\mathrm{tr}} = \beta_{\mathrm{ts}}]\!] && (3.11)\\
&= \sum_{\beta \in B} \mathbb{P}\left[\arg\max_{\beta' \in B} \mathrm{perf}(E, \beta') = \beta\right]^2 && (3.12)\\
&= \sum_{\beta \in B} \mathrm{BR}(E, \beta)^2 && (3.13)
\end{aligned}$$

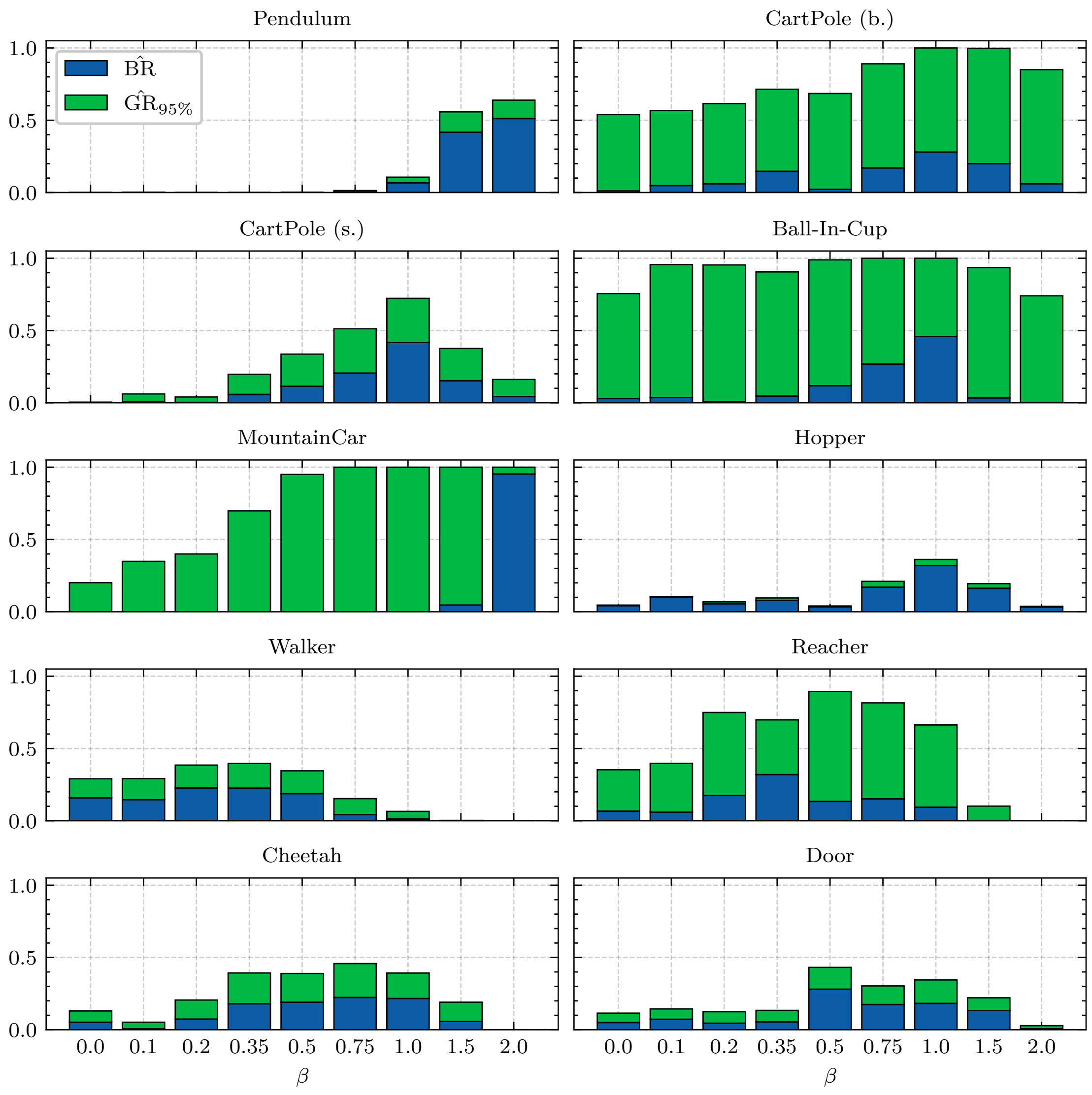

Figure 3.8: Estimated best-ratios ($\hat{\mathrm{BR}}$, blue) and 95%-good-ratios ($\hat{\mathrm{GR}}_{95\%}$, green) of all β values on all environments using MPO. ($N = 10^5$)

This gives us two ways to estimate this probability: either by Monte Carlo estimation of the expectation in Eq. (3.10):

$$\begin{aligned}&\mathbb{E}_{\beta_{\mathrm{tr}},\beta_{\mathrm{ts}}} \llbracket \beta_{\mathrm{tr}} = \beta_{\mathrm{ts}} \rrbracket \\ &\approx \frac{1}{S^2} \sum_{s_1=1}^{S} \sum_{s_2=1}^{S} \left\llbracket \operatorname*{arg\,max}_{\beta \in B} \mathrm{perf}(E, \beta, s_1) = \operatorname*{arg\,max}_{\beta \in B} \mathrm{perf}(E, \beta, s_2) \right\rrbracket, \qquad (3.14)\end{aligned}$$

where $S = 20$ is the number of seeds, or by replacing BR in Eq. (3.13) with its estimate $\hat{\mathrm{BR}}$. As the latter method introduces a further source of error due to the resampling procedure, we report the probabilities computed according to Eq. (3.14) in Table 3.2, but a nice interpretation of the best-ratios in Figure 3.8 is that the sum of their squares is the probability that $\beta_{\mathrm{tr}} = \beta_{\mathrm{ts}}$ on a given environment.

We can see that MountainCar has by far the highest probability of equality of β_{tr} and β_{ts}. Most environments have an equality probability of around 20%, but of

Environment	$\hat{\mathbb{P}}[\beta_{\text{tr}} = \beta_{\text{ts}}]$
Pendulum	0.46
CartPole (b.)	0.18
CartPole (s.)	0.25
Ball-In-Cup	0.24
MountainCar	0.91
Hopper	0.20
Walker	0.21
Reacher	0.19
Cheetah	0.19
Door	0.20

Table 3.2: Estimated probability that the best β of a trial run is also the best β in a test run, computed according to Eq. (3.14).

course this also depends on the size of B. For us, $|B| = 9$, so if the performance of each β were completely random (uniform), we would expect that

$$\mathbb{P}[\beta_{\text{tr}} = \beta_{\text{ts}}] = \frac{1}{9} \approx 11\%. \tag{3.15}$$

For some environments, the true equality probability is not much higher than this! This means that these environments are very stochastic: the measured performance depends heavily on the chosen random seed, which influences which β performs best. This makes tuning or adapting β to an environment a flawed idea: the environments don't have a clear "best β".

But this reasoning does not show the whole picture. We actually don't care about finding the "best β", we care about good performance. In Figure 3.8 we see that MountainCar has a clear best β: $\beta = 2$. But we already know from Figure 3.4 that we achieve very good performance on MountainCar with any $\beta > 0.5$. So instead of looking at a color's best-ratio, maybe we should consider whether a given color performs *almost* as good as the best β. One way to define this is what we call the q-good-ratio GR_q of a β on a given environment:

$$\begin{aligned} \text{GR}_q(E, \beta) &:= \mathbb{P}\left[\text{perf}(E, \beta) \geq q \max_{\beta' \in B} \text{perf}(E, \beta')\right] && (3.16) \\ &= \mathbb{P}[\text{perf}(E, \beta) \geq q\, \text{perf}(E, \beta'), \forall \beta' \in B]. && (3.17) \end{aligned}$$

If we train one agent with each $\beta \in B$ on the environment E, then $\text{GR}_q(E, \beta)$ tells us the probability that the performance of the agent trained with β is q times as good as the performance of the best agent. If we choose q large enough, e.g. $q = 0.95$, then this can be interpreted as the probability that β is "almost" as good as the best β. Looking at Eq (3.7) we see that the good-ratio generalizes the concept of the best-ratio and that

$$\text{GR}_q(E, \beta) \geq \text{BR}(E, \beta),$$

with equality holding when $q = 1$. The good-ratios can be estimated in a very similar way to the best-ratios in Eq. (3.8):

$$\hat{\mathrm{GR}}_q(E, \beta_k) := \frac{1}{N} \sum_{i=1}^{N} \left[\!\!\left[\mathrm{perf}(E, \beta_k, s_{ik}) \geq q \max_j \mathrm{perf}(E, \beta_j, s_{ij}) \right]\!\!\right], \tag{3.18}$$

where s_{ij} are defined as above. The computed GRs for $q = 0.95$ are shown in Figure 3.8 alongside the best-ratios.

Here we see what we would expect for MountainCar: the probability of a β being "good" increases and is approximately 1 for $\beta > 0.5$. The good-ratio gives a new perspective on whether an environment can benefit from β-adaptation: the more uniform the good-ratios for a task are, the less it matters which β is chosen, as all are equally likely to be "good". But there are two different reasons for why the good-ratios might be uniform. On the one hand, there are environments like Ball-In-Cup. In this case, all GRs are roughly equal, and they are also all very large. This means that, in an experiment where all colors are used to train an agent each, they will all likely achieve very similar performance. This is not to say that this performance is necessarily good (they can all be 0), but it means that no β would be a "bad choice", because each one has a high probability of performing at least 95% as well as the "best choice". On this kind of environment, β adaptation is thus simply unnecessary.

But something else is going on when all good-ratios are roughly uniform, but have low value. As an example, we can look at the Hopper environment. The GRs are a bit less uniform, but still nonzero for all colors. What jumps out is that the good-ratios are almost equal to the best-ratios here, which is not at all the case on the Ball-In-Cup environment. This means that the performance on Hopper is very stochastic. If we train an agent with each color on a task like this, then, if a β performs "well", i.e. it is 95% as good as the best β, it is very likely that it indeed *is* the best β. Because the good-ratios are still rather uniform, this means that in every such experiment an almost random β will outperform the others considerably, making it very unlikely to select the "best" color beforehand. Even though the underlying cause is quite different, the effect of this is the same as on Ball-In-Cup: on both of these tasks, we cannot really hope to improve performance by cleverly adapting β, either because all β are good choices anyway (Ball-In-Cup) or because all β are bad choices most of the time (Hopper).

The only two environments which are truly selective about the color, in the sense that there are multiple values of β s.t. $\mathrm{GR}_{95\%}(E, \beta) \approx 0$, and also the only ones where pink noise is not (almost) the best choice, are Pendulum and Walker. These are exactly the two environments where we saw a slight increase in performance when using the oracle scheme above. On these environments, we have to be somewhat careful in choosing β. On this selection of environments, pink noise is evidently the best general choice, and the only color with non-zero 95%-good-ratio on all environments. We will continue the discussion of adapting beta on the basis of interactions in Section 4.2, where we see if this is worth it if we let β vary with time, instead of keeping it constant.

Chapter 4

Time-Varying Color

Throughout Chapter 3, we have searched for a good action noise color under the assumption that we should use this color for the whole training process. But, as alluded to in Section 2.4, it might be beneficial to let β vary with time. In other words, it is possible that different stages of learning can benefit from different exploration strategies, i.e. different colors. In Section 4.1, we will look at non-adaptive time-varying strategies, where we choose β as a function of the rollout number ahead of training: instead of choosing a single β, we select a list $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_M)$, where M is the number of rollouts. We don't try to adapt this list to an environment, but instead want to find a list which works well across environments, similar to our goal in Section 3.1. In the last section we concluded that there is in general no benefit to adapting a constant β to an environment. This conclusion does not transfer to the case where we let β vary over time. Additionally, there is no "easy test", like we applied in the last section, to rule out whether this is a good idea. Thus, in Section 4.2, we will try to develop a Bayesian optimization method which adapts β online to the environment on the basis of interactions.

4.1 Non-Adaptive Methods

We have seen that using pink noise as exploration noise generally leads to improved performance over noise of both higher and lower correlation. To explain the improvement over white noise, we have argued that many tasks require "global exploration": higher temporal correlation in the action sequence generally leads to a greater state space coverage, which makes it possible to reach certain high reward regions much more quickly than when using white noise. Why does increasing correlation even further have a detrimental effect on performance? We have also tried to explain this. Higher correlation makes the agent move to regions further away from where it would go without noise. This is of course exactly why it is beneficial in discovering unknown high reward regions. But this also means that the trajectories are more off-policy: the state-marginal distributions of the noisy policy is different from that of the deterministic policy. The higher the correlation is, the higher will be the difference between these distributions. This distributional shift is a problem, because at test time, we want the policy to perform well on the states from its deterministic state-marginal distribution. But the policy has not actually seen these states that much, because it always collected trajectories with the correlated noise. In other words, the data used to train the actor and critic networks does not come from the same distribution as the data at test time. This is a common problem also in supervised learning, and here we see it in the context of reinforcement learning. Thus, what is needed for good performance is not just global exploration, but also "local exploration". Some trajectories must be generated close to on-policy, such that the state-marginal distributions from these trajectories cover the states which are likely to be seen during test time. This local exploration can be achieved by using action noise with low correlation, such as white noise.

These two points show why in practice, we need a mixture of both local and global exploration. White noise ($\beta = 0$) is very good at local exploration, but not very good at global exploration, because it is slow at reaching distance (as we showed in Sec. 2.3). It can still perform global exploration of course, otherwise it could not be used as action noise at all, at least in sparse reward settings. However, if there are high reward regions very far from the deterministic policy's trajectories, it will be unlikely that white noise is enough to reach these. On the opposite end of the color spectrum[1], we have red noise: good for global exploration, but not very good for local exploration, as it produces strongly off-policy trajectories. The good performance that we observe when using pink noise can be explained by the fact that pink noise in some sense strikes a compromise between these two types of exploration. But maybe there is a different way to achieve this compromise and reach even higher performance, by letting β vary over time.

The parameter β is similar in certain ways to the action noise scale σ, which is a parameter in algorithms that use a deterministic policy, such as TD3 (see Eq. 2.26). Setting σ to a large value lets the action noise drive the agent further away from the no-noise deterministic trajectory. This leads to higher state-space coverage, but also makes the data more off-policy. In other words, the effects are very similar to what we see when we increase β instead of σ. On the other hand, decreasing σ has a similar effect to decreasing β: the global exploration capabilities drop, but the collected data is more on-policy. As an aside, this also explains why Ornstein-Uhlenbeck noise performs so much worse than red noise, despite the fact that these noises are very similar. The reason is that, even though they are very similar with respect to their temporal correlation (see Fig. 2.4), OU noise has a much broader marginal distribution than the stationary colored noise we use (incl. red noise). This can be seen in Figure 2.5, and this makes the trajectories of OU noise even more off-policy. On most tasks, this leads to worse performance, though not on MountainCar and Pendulum (the two environments of "first type"), as can be seen in Figure 3.4.

Of course, for σ the same reasoning as above holds: we need not limit our choice to a single value of σ. Instead, we can let it vary over time. The most common way of varying σ is by using a schedule: gradually decrease σ over the course of training. This is in accordance with the "curriculum learning" analogy we presented

[1]As every child knows, the rainbow goes white-pink-red.

in Section 2.4: it seems natural that an agent should conduct global exploration at the beginning of training to find high reward regions, and perform local exploration at the end of training to fine-tune its trajectories. Indeed, Hollenstein et al. (2022) found that using such a schedule often improves final performance over using a constant value of σ. It stands to reason that a similar scheme should work for β as well: we can let β go linearly from 2 to 0 over the course of training:

$$\beta_m = 2\Big(1 - \frac{m}{M}\Big), \tag{4.1}$$

where m and M are the current rollout and the total number of rollouts respectively, and we keep β constant for every individual rollout. Hopefully, it will then gain the power of red noise at the beginning of training to find high reward regions, as well as the local optimization power of white noise at the end, to fine-tune trajectories with nearly on-policy data. On average, we can expect this method to work like pink noise, as this is the average β. So let's just try it out! We keep the same number of seeds, as well as the same sets B and $\mathcal{E}$ that were described in Section 3.1. The resulting learning curves are shown in Figure 4.1, where we compare to white noise, OU noise and pink noise; average performances are compared in Figure 4.2. It can be seen that pink noise clearly outperforms the schedule.

One explanation for this outcome is that, even though the scheduling approach provides a mixture of local and global exploration (and hence outperforms both white and OU noise), its behavior still very different from that of pink noise. Looking at the learning curve on the Walker environment, we can see that the performance starts improving more quickly at a point rather late in training. As we know that the Walker task prefers low values of β, it is possible that the reason for this is that at this point β_m becomes low enough, such that effective learning becomes possible. Looking at the Pendulum environment, which we know to prefer high values of β, we see no such problem. Maybe the average exploration behavior in this scheduling scheme is comparable more to red noise than to pink noise, and we should find a different method which brings down β more quickly, for the benefit of environments like Walker.

We have been looking at the parameter β as a measure of exploration behavior. A different and perhaps more appropriate metric is that of "speed". When comparing white noise to red noise, it is clear that red noise is "faster": on the integrator environment it is capable of reaching a much greater distance than white noise in a given length of time. We have discussed this in Sections 2.3 and 2.4, and it can be seen in Figure 2.6. The same also holds true for the double integrator environment, as is evident from Figure 3.2. What we have also already noticed in these plots is that the difference in the average distance reached by a "pure noise" agent when using a color β vs. a different color β' is much greater when β and β' are small ($\ll 2$) than when they are large ($\gg 0$), when the distance $|\beta - \beta'|$ stays the same. This was the reason why we chose a non-equidistant list B of color parameters to test out. One measure of the "speed" of a color β is the slope of the average distance log-log plot (which is shown here again in Figure 4.3):

$$\text{slope} = \frac{\log \Delta x}{\log \Delta t} \tag{4.2}$$

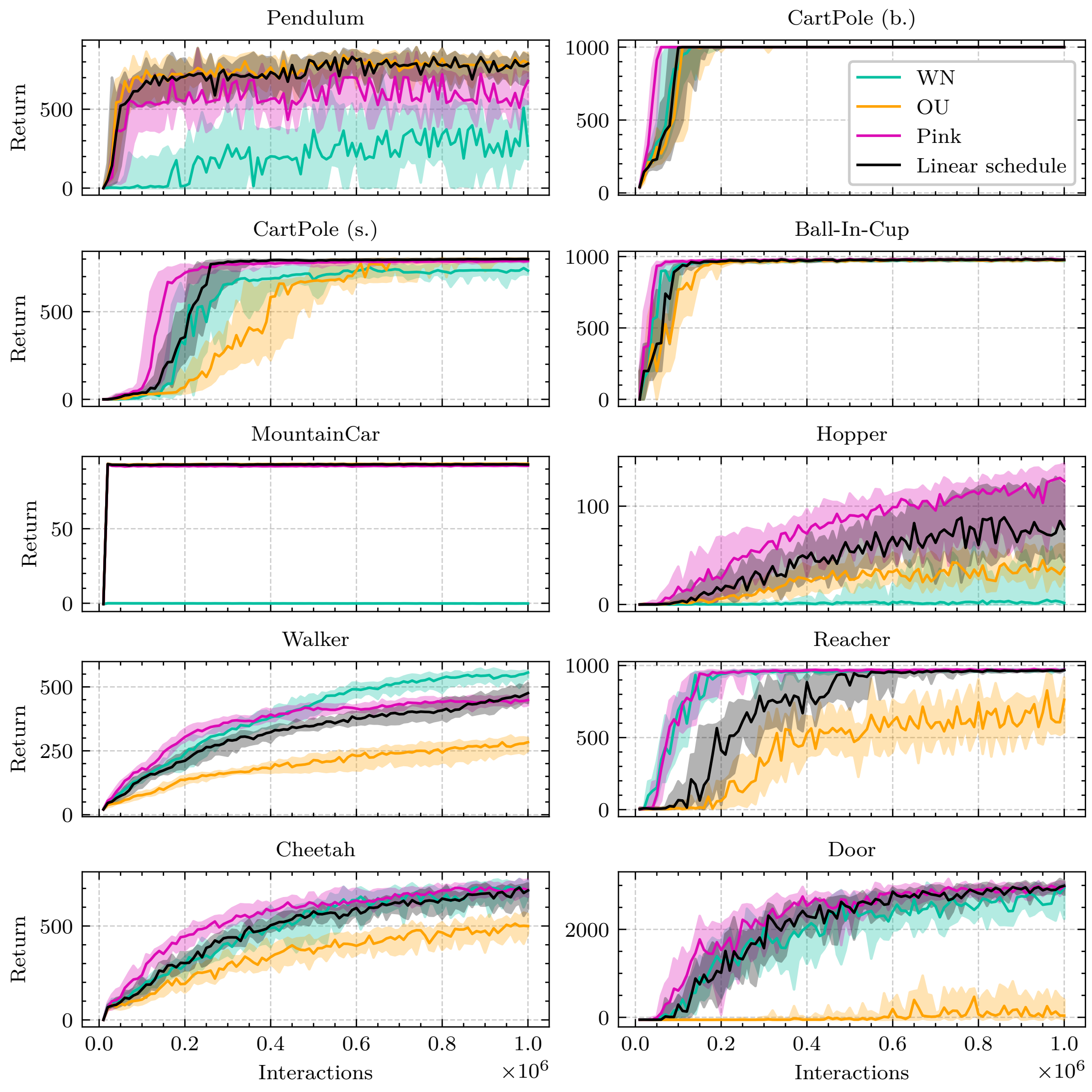

Figure 4.1: Learning curves of the linear color schedule action noise on MPO. It can be seen that pink noise generally outperforms this method.

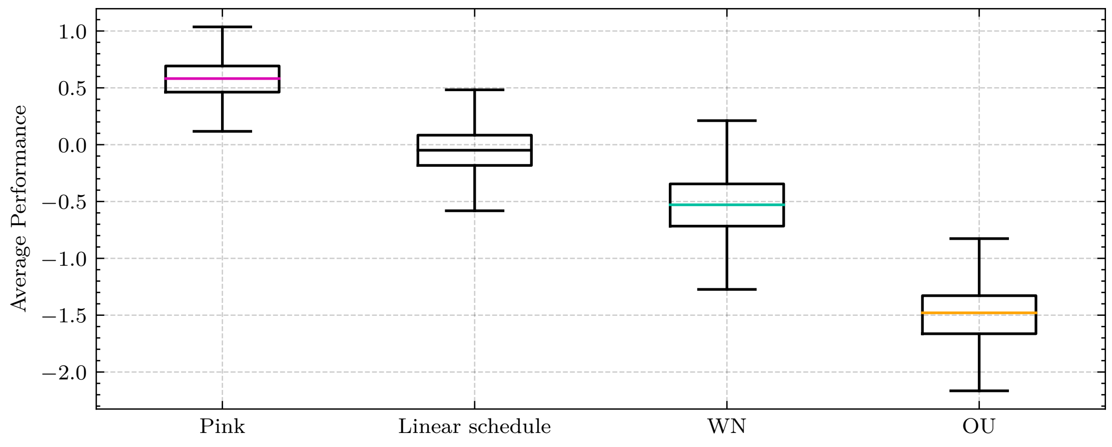

Figure 4.2: The linear color schedule achieves higher performance than the two baselines of white and OU noise, but is still outperformed by constant pink noise. (MPO, $N = 10^5$)

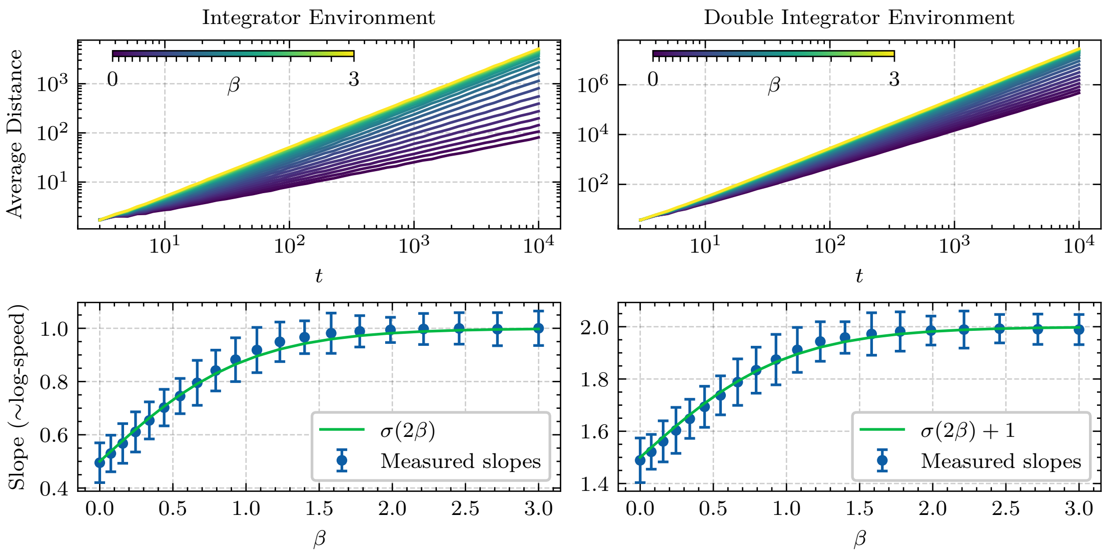

Figure 4.3: The influence of β on an agent's speed on the integrator (left) and double integrator (right) environments is non-linear. The log-speed is well approximated by a tanh function.

These lines are straight, but also noisy, as they are obtained through random sampling. We can estimate the underlying slope by taking the average of the individual (straight) line segments. These measured "individual slopes" and their averages are plotted for a set of β values in Figure 4.3. We can see that the averages follow a smooth curve, which, on the integrator environment, is well (though not perfectly) approximated by a sigmoid function:

$$\text{slope}_{\int} \approx \sigma(2\beta) = \frac{1}{2}(\tanh\beta + 1), \tag{4.3}$$

as shown in the plots. On the double integrator environment, we see a very similar picture. This time, the slopes are offset by a constant of 1, and are well approximated by

$$\text{slope}_{\int\int} \approx \sigma(2\beta) + 1 = \frac{1}{2}(\tanh\beta + 3). \tag{4.4}$$

If we fix the temporal distance Δt between measured positions, then these slopes are affine transformations of the log-speed:

$$\log\frac{\Delta x}{\Delta t} = \log\Delta x - \log\Delta t = \log\Delta t(\text{slope} - 1). \tag{4.5}$$

Thus, with the hyperbolic tangent approximation above, the log-speed can be seen as an affine transformation of $\tanh\beta$, suggesting that in some settings $\tanh\beta$ is physically more meaningful than β. In fact, we chose B in Section 3.1 such that the tanh-transformed values are approximately equidistant (see Figure 4.4).

Now that we have a new measure of exploration behavior ($\tanh\beta$), maybe it would be a good idea to design a schedule that is linear in $\tanh\beta$. Letting $\tanh\beta$ go from

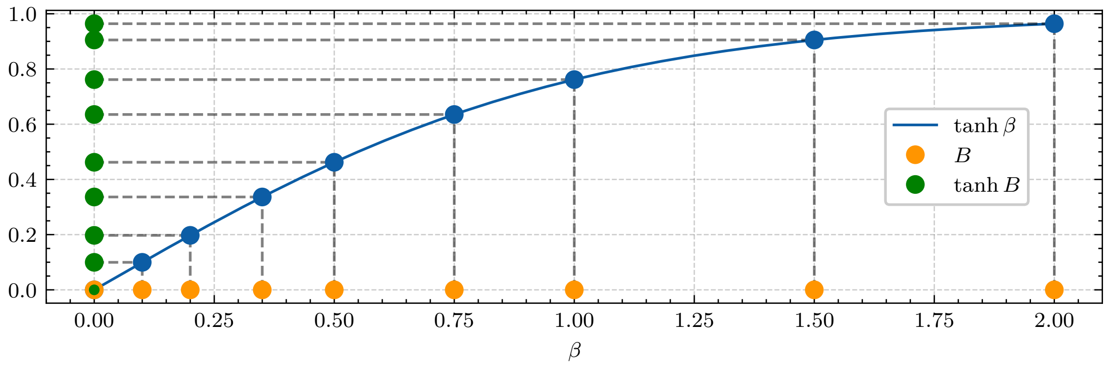

Figure 4.4: The set B of colors that we chose in Sec. 3.1 is approximately equidistant on a $\tanh\beta$ scale (and thus, so is their log-speed on the integrator/double integrator environments).

(close to) 1 to 0, we can choose β_m according to

$$\tanh\beta_m = 1 - \frac{m}{M} \tag{4.6}$$

$$\iff \beta_m = \operatorname{artanh}\left(1 - \frac{m}{M}\right). \tag{4.7}$$

A comparison of the linear schedule and this artanh schedule is shown in Figure 4.5. When using the artanh schedule, the average β is now not $\beta = 1$ like the linear schedule, but[2]

$$\int_0^1 \operatorname{artanh}(1-x)\,\mathrm{d}x = \ln 2 \approx 0.69. \tag{4.8}$$

This gives us hope that the performance on the Walker environment will improve, as this schedule provides more local exploration behavior.

The results of the artanh schedule experiments are shown in Figures 4.6 (learning curves) and 4.7 (average performances). Overall, the performances are not much better than those of the linear schedule. Interestingly, it looks like this schedule may actually outperform pink noise on both tasks where pink noise is not ideal: Walker and Pendulum (see Figure 4.6). On most environments however, the schedule is

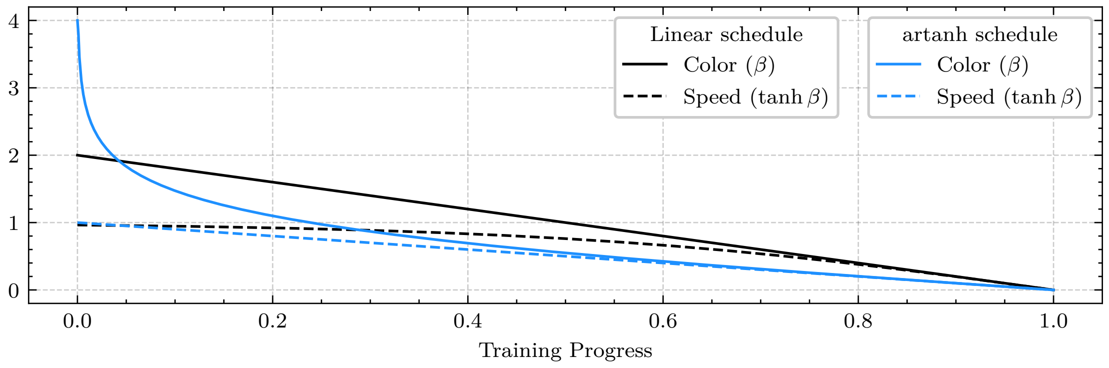

Figure 4.5: A comparison of the linear and artanh schedules.

[2]This improper integral is a piece of cake for modern computer algebra systems.

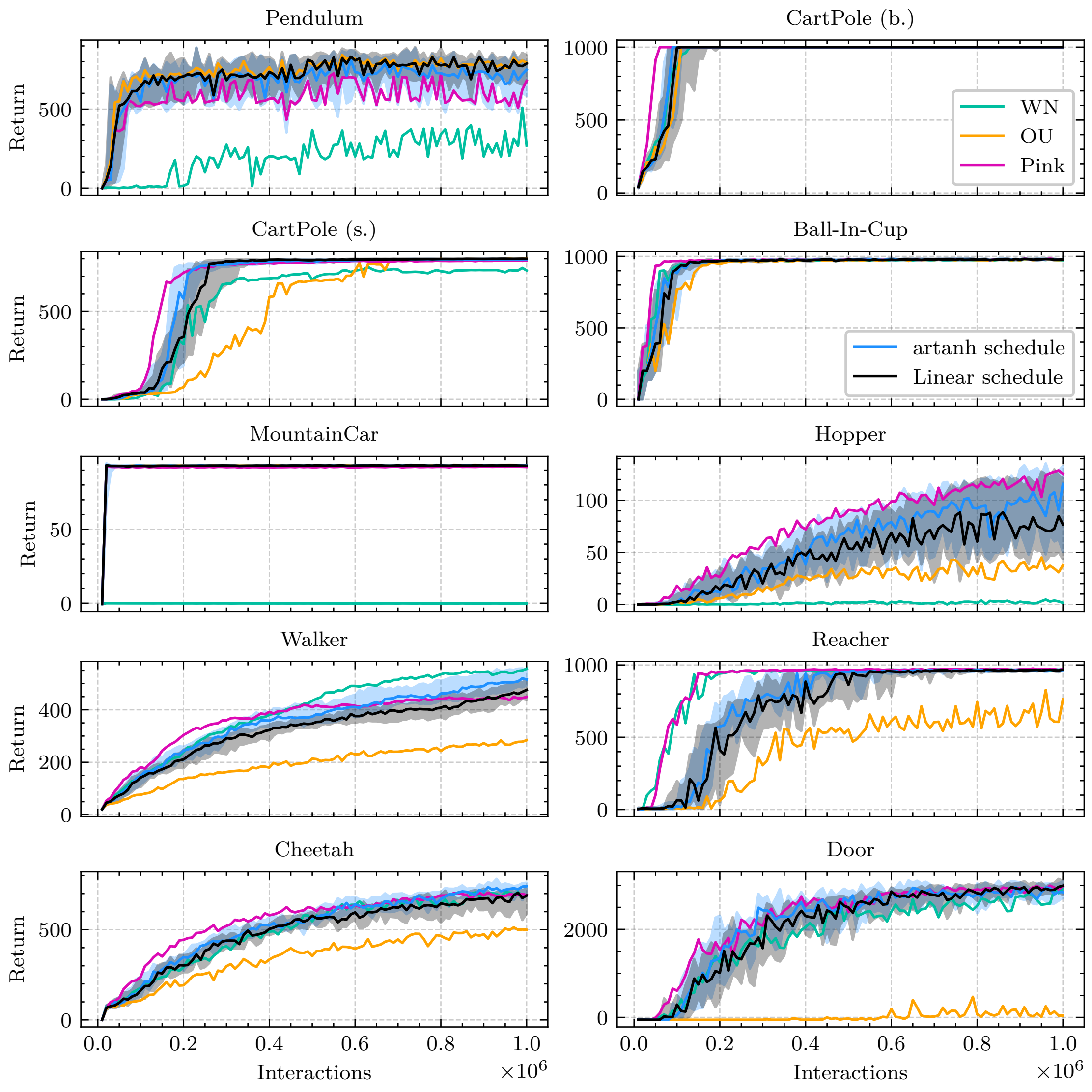

Figure 4.6: Learning curves of the artanh color schedule action noise on MPO. It can be seen that pink noise also outperforms this approach, and that the results are very similar to the linear schedule.

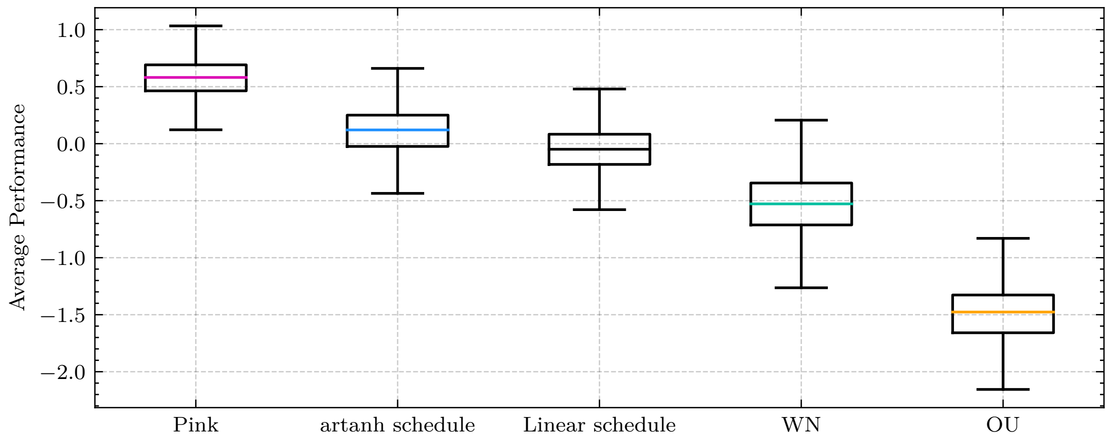

Figure 4.7: The artanh color schedule seems to outperform the linear schedule, but only very slightly. (MPO, $N = 10^5$)

outperformed by either both pink and white noise or by both pink and OU noise. It looks like on some tasks, scheduling gets the "best of both worlds" from white noise and red noise, whereas pink noise performs in-between, while on other tasks, pink noise gets the best of both worlds, and scheduling performs in-between.

It seems that the curriculum learning analogy we put forward is flawed. If we believe the reason for pink noise's high performance is indeed a good mix of global and local exploration, then the comparatively low average performance of the scheduling approach suggests that it is more beneficial to have both types of exploration at all points in training, rather than to slowly transition from one to the other. One way to ensure this is to simply select β randomly for every rollout. Then some rollouts will explore locally, others will explore globally. The idea behind this is again that what is needed during training for good performance is a diversity in exploration (i.e. local and global) and that pink noise, due to being in the "middle of the exploration spectrum" achieves this approximately, but that increasing the diversity of exploration strategies (i.e. using more than one β) may be beneficial. To try this idea out, we first need to select an appropriate distribution to sample β_m from. The simplest choice is to simply select one β from our list B uniformly at random. Another choice would be to sample β such that the sampled "log-speed" or $\tanh \beta$ is uniform. To do this, we simply sample a number uniformly from the open interval $(0, 1)$:

$$z \sim \mathcal{U}(0, 1), \tag{4.9}$$

and then transform it such that $z = \tanh \beta$:

$$\beta = \operatorname{artanh} z. \tag{4.10}$$

One new thing we can try with random sampling is to let different actuators use different exploration processes in the same rollout. We can achieve this simply by sampling a value for β independently for each action dimension. This is sensible if our goal is (as stated above) to increase the diversity of exploration strategies. So far, we have always used the same β for every action dimension. But this was only because it was unclear how to choose β individually for each dimension. If we select β by sampling, it becomes very easy to do this, and it definitely increases the diversity of exploration!

More concretely, why should this kind of exploration be helpful? Imagine that the environment we want to solve consists of a two-dimensional surface, where the agent starts at the origin and is empowered to move around freely using two actions: horizontal and vertical velocity. The goal of the task (which is encoded in a sparse reward signal) is to reach a certain "finish region" which happens to be placed on the x-axis, i.e. it is only necessary to move horizontally. In this case, a large exploration speed (i.e. large β) is required along the x-axis, such that it is possible to reach the goal accidentally. On the other hand, if the goal region is not extended vertically, it will be beneficial to have a small β for the y-velocity. This might seem like a contrived example, but in a given environment it might well be the case that it is important to vary one action dimension more than another, and then this kind of "independent β sampling" might be beneficial.

The independent sampling can be applied to both the list and the artanh sampling strategies, so we simply test both methods with and without independent sampling. The resulting average performances are shown in Figure 4.8. Again, we see that pink noise remains the top performing option. The random sampling methods perform very similarly to the schedule. Independently sampling β for each action dimension does not seem to have any impact on the performance. These results seem to indicate that pink noise performs well for different reasons than just "a good mix of local and global exploration", but as these terms are not defined precisely, this is a bit unclear. We will come back to discussing the reason for pink noise's good performance in Chapter 5.

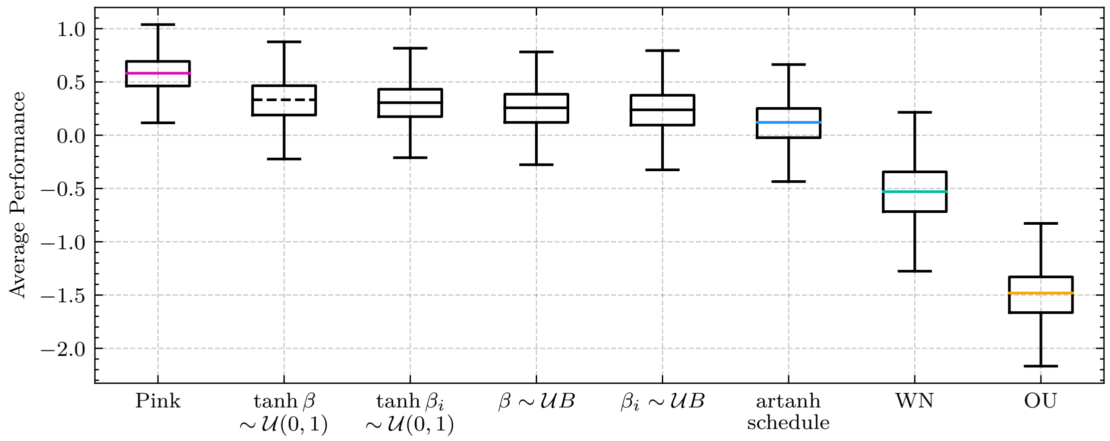

Figure 4.8: All four random β selection methods perform very similarly, and are clearly outperformed by pink noise. Sampling each action dimension's β_i independently, vs. using the same β for all action dimensions, seems to not make any difference. (MPO, $N = 10^5$)

There are, of course, many other methods we could try to vary β over time, but as those that we did try out performed basically identically, and as there is no obvious method to try out next, we will stop here. Indeed, if we were to continue to search for a list $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_M)$ without changing the set of environments $\mathcal{E}$, observed improvements will increasingly get less meaningful, as we overfit the colors $\boldsymbol{\beta}$ to the environments at hand. In the next section we will instead try to adapt β_m to the environment, based on the interactions from all preceding rollouts $(1, 2, \ldots, m-1)$.

4.2 Adaptive Methods

We have discovered in Section 3.1 that pink action noise improves performance across many environments without any need for adaptation. Indeed, in the following section we saw that adaptation of a constant β to an environment does not lead to any noticeable increase in performance, even when using an oracle to choose β. In the last section we then found out that we are also not able to improve performance much by letting β vary over the course of training, e.g. by using a schedule. So why should combining these two approaches (adapting to an environment and varying β with time) that *honestly didn't work very well* be a good idea?

The first thing to restate is that the conclusion at the end of Section 3.2 about the ineffectiveness of adapting β applies only to constant values of β. It is of course possible that there really is no better colored noise strategy than "constant pink". On the other hand, we have seen in Section 3.2 that the best choice for a constant β depends heavily on the random seed. If this is true for a constant value of β then it is probably even more true for a time-varying β. In other words, if the distribution

$$p(\beta^\star \mid E) = \mathbb{P}\left[\beta^\star = \arg\max_\beta \operatorname{perf}(E, \beta)\right] \tag{4.11}$$

is spread out, why should we expect that the distribution

$$p(\beta_m^\star \mid E) \tag{4.12}$$

of the best beta for rollout m is any less so? The randomness in these cases comes not only from the transition dynamics of the environment, but also from the stochasticity of the policy and its action noise, as well as potential randomness in choosing $(\beta_1, \beta_2 \dots, \beta_{m-1})$. It seems like it should be much more definite what the best constant β for an environment is (e.g. "Walker prefers $\beta = 0$"), than what the best β_m in rollout m is, given only the environment (Which β_{42} does the Cheetah environment prefer in its 42nd rollout?). Choosing the sequence $\boldsymbol{\beta}$ ahead of time is similar to open-loop control, only that there is no model of how the RL algorithm reacts to a sequence choice. Abstractly, the RL algorithm can be seen as a black box, taking in a list of colors $\boldsymbol{\beta}$ (and a random seed s) and producing a performance $\operatorname{perf}(E, \boldsymbol{\beta}, s)$. So far, we have chosen $\boldsymbol{\beta}$ heuristically, but we might get much better results by "closing the feedback loop" and selecting β_m using all the information contained in the previous rollouts $\tau_1, \tau_2, \dots, \tau_{m-1}$, as well as the previous color choices $\beta_{1:m-1}$.

How do we do this? We want some function f that takes in previous trajectories $\tau_{1:m-1}$ and colors $\beta_{1:m-1}$ and spits out the next color β_m. This situation is shown in Figure 4.9. There is no clear way to design such a function heuristically. If we can't design a function, maybe we can learn it using machine learning! Supervised learning is unfortunately out of the question, as there is no expert data of well selected colors β_m from the available data $\tau_{1:m-1}, \beta_{1:m-1}$. However, reinforcement learning certainly does apply here. This problem can be described by a Markov decision process[3] with episode lengths of M. The action space is $\mathcal{A}' = B$, where B does not have to be the same set of colors we considered before, but can also be

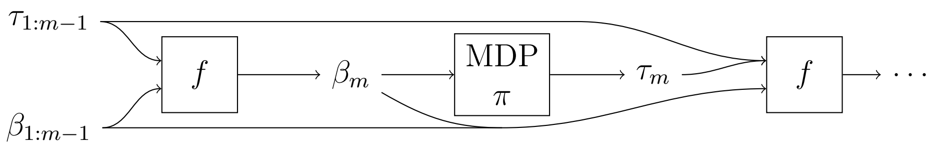

Figure 4.9: How can past trajectories $\tau_{1:m-1}$ and color choices $\beta_{1:m-1}$ be transformed into a new color choice β_m?

[3]As we don't consider the random seed s as observed, we are technically in a *contextual* MDP (Hallak et al. 2015), but we ignore this technicality.

continuous. The state space is

$$\mathcal{S}' = \{(\tau_{1:m-1}, \beta_{1:m-1}) \mid m \in \{1, \ldots, M\}, \beta_n \in B, \tau_n \in \mathcal{T}, \forall n \in \{1, \ldots, m-1\}\} \tag{4.13}$$

with

$$\mathcal{T} = \{(s_0, a_0, r_1, s_1, \ldots, r_T, s_T) \mid s_t \in \mathcal{S}, a_t \in \mathcal{A}, r_t \in \mathbb{R}, \forall t \in \{0, \ldots, T\}\}, \tag{4.14}$$

where $\mathcal{S}$ and $\mathcal{A}$ are the state and action space of the original (environment) MDP, respectively. The reward signal is

$$\rho_m = \begin{cases} 0 & \text{if } m < M \\ \text{perf}(E, \beta_{1:M}, s) & \text{otherwise} \end{cases} \tag{4.15}$$

If we can solve this RL problem, then f is the resulting policy. However, it turns out that this RL problem way too hard to solve. There are several reasons for this:

- The state-space is enormous. Thus, many rollouts would be required to make progress on this task.
- The reward is sparse. This makes the credit assignment problem more difficult and again, more rollouts are required.
- A rollout corresponds to one complete training of an agent on the given environment. As we want to train only once, this means that the policy must be done training before the first rollout has finished, which is especially impossible because the first sparse reward signal comes only at the end of training. Even if we allowed multiple trainings, it would never be enough to satisfy the first two points.

These contradictory requirements make the RL problem of course completely intractable.

We are forced to simplify the problem. One way to do this is to get rid of the state space entirely. This is the bandit situation we encountered in Section 2.2: at the beginning of each rollout we select β according to belief distributions over the expected reward we would obtain with each color choice and a bandit strategy such as Thompson sampling or UCB. For this, we also have to modify the reward signal, such that learning becomes possible before the end of training when the sparse performance-reward arrives. A natural choice here is to design a reward function which calculates a score for one single rollout (in the environment) to measure the quality of exploration. After each rollout, the belief distributions can then be updated using this score and the process repeats. The hope is that designing this score function is easier than designing the function f itself, but it is not obvious how to infer good exploration from a trajectory.

Perhaps the simplest choice for a score function is to simply take the episode return:

$$S_{\text{ret}}(\tau) = \sum_{t=1}^{T} r_t, \tag{4.16}$$

where $\tau = (s_0, a_0, r_1, s_1, \ldots, s_T, r_T)$ is taken to be the latest trajectory, i.e. that whose exploration quality is to be judged. This has the advantage that we would directly optimize the quantity we care about most: performance. It is also easy to see how such a measure can perform well in sparse reward settings: here, the objective of exploration is to "reach the goal". Thus, the best exploration strategy is the one which gets the agent to the goal, which will also yield the highest return. However, in many settings, exploration is inherently costly. If an okay policy has been found, deviating from it might result in lower return, even if this would improve the policy in the long run. This is the exploration-exploitation trade off that we discussed in Section 2.2.

A good exploration strategy should lead the agent into states where the policy or Q-function has high uncertainty. Using something like "average uncertainty" of the actor or critic on states visited in the trajectory as a score might indeed be a very good choice. However, as we want this method to be compatible with standard (non-Bayesian) deep RL methods, we cannot easily estimate uncertainty. What we can do though, is compare the rollout to previous trajectories. If the rollout is very similar to a previous one (according to an appropriate measure of similarity), then the information gained from this rollout is probably minimal, which means that the exploration was not very good. This is connected to the idea of uncertainty because, even though we don't have a quantitative measure of uncertainty, visiting the same states multiple times should decrease uncertainty in the critic and actor on these states. To use this idea, we have to propose a method of measuring similarity between trajectories, as well as a way to turn this similarity into a score.

There are many ways in which the similarity, or equivalently dissimilarity, between two rollouts can be measured. One might, for example, measure the average distance $\|s_t - s'_t\|$ between visited states in two rollouts τ and τ'. However, two rollouts can be *effectively* the same even if the visited states look very different. To account for this, one might instead use something like the average difference between Q-values $|Q(s_t) - Q(s'_t)|$. We choose instead to use a much simpler metric, the difference in rollout return $|G - G'|$, where $G = \sum_{t=1}^{T} r_t$. Thus, two trajectories are considered to be "similar", if they achieve a similar total reward. The reasoning here is that if an agent always collects the same return, then these trajectories are probably achieving this in the same way, meaning that the agent is not acting very differently. If however, the agent receives a return that is different from those seen before, then no matter if this new return is higher or lower than the previous returns, the exploration behavior definitely found something new. This situation is the one we want to reward.

The way we turn this idea into a score function, is by first constructing a distribution over previously observed returns, and then looking at the likelihood of observing the new return G under this distribution. If the likelihood is high, then the new return is similar to the previous ones, and the episode score is low. If the likelihood is low, then this episode is not similar to previous ones, as the return is different. Thus, a different behavior has been found, and this situation is rewarded with a high score. Of course, it could be that the new behavior is not due to exploration, but because the policy was just updated; we do not account for this.

More concretely, we approximate the distribution of past returns using kernel density estimation. We choose a Gaussian density kernel which can be thought of as using a dissimilarity measure of $(G - G')^2$. Given a set $\{G_i\}_{i=1}^N$ of past rollout returns, the fitted distribution then has the form

$$p(G) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{N}(G \mid G_i, h^2). \tag{4.17}$$

As the bandwidth parameter we use $h = \hat{\sigma}(\frac{4}{3N})^{\frac{1}{5}}$ with $\hat{\sigma} = 1.4826$ MAD as a robust estimate for the standard deviation, where MAD is the mean absolute deviation of $\{G_i\}_{i=1}^N$. This is a common choice for h (Murphy 2022, Sec. 16.3.3). Finally, the score can be calculated as the (continuous) Shannon information of the observed return $G = \sum_{t=1}^T r_t$ under this distribution:

$$S_{\text{info}}(\tau) = -\log p(G). \tag{4.18}$$

This "info-score" has exactly the property listed above, namely rewarding only those rollouts that achieve an unlikely (novel) return.

Equipped with a method of scoring rollouts, we can now use a bandit algorithm to select and optimize β. We start with the finite multi-armed bandit setting that we already discussed in Section 2.2. The first thing we have to do is to select a list of colors ("bandit arms") to search over: $B = (\beta_1, \beta_2, \dots, \beta_K)$ with $\beta_k \in [0, 2], \forall k$. In our experiments we choose the same list B that we used in the previous sections. To be able to make use of a standard bandit algorithm, such as the Thompson sampling algorithm we introduced in Section 2.2, we start with a rather strong assumption, namely that the bandit rewards (= rollout scores) are distributed according to K Gaussian distributions with known common standard deviation σ. Under this assumption, we can use exact Bayesian inference to estimate the means ($\boldsymbol{\mu} \in \mathbb{R}^K$) of the reward distributions (see Eq. 2.22). In Algorithm 1, we show how Thompson sampling can be applied in this setting to tackle the β-optimization problem ($\mathbb{S}_+^K$ denotes the set of positive semi-definite $K \times K$ matrices). The relationships between the random variables are shown in the Bayesian network in Figure 4.10a.

There is a second strong assumption in the Thompson sampling algorithm (and similarly for other algorithms like UCB): it assumes that the reward distributions are stationary, i.e. that they don't change over time. This is not the case in the context of reinforcement learning! Assume, for example, that the rollout scores are defined as their return, i.e. $\rho_m = S_{\text{ret}}(\tau_m)$. Then, if the reinforcement learning algorithm works, it should naturally be the case that the policy improves over time, and thus, on average, $\rho_m > \rho_n$ for $m \gg n$.

Algorithm 1: Thompson sampling

```
Input: Arms $B = (\beta_1, \dots, \beta_K)$,
       Reward distributions std $\sigma$
Initialize $\boldsymbol{m} \in \mathbb{R}^K, \Sigma \in \mathbb{S}_+^K$
for $i \in \mathbb{N}$ do
    Sample $\boldsymbol{q} \sim \mathcal{N}(\boldsymbol{m}, \Sigma)$
    $\alpha_i \leftarrow \arg\max_{k \in \{1,\dots,K\}} q_k$
    $\tau_i \leftarrow$ Run rollout with $\beta_{\alpha_i}$
    $\rho_i \leftarrow$ score of rollout $\tau_i$
    Do Bayesian update of $\boldsymbol{m}, \Sigma$
      using $\{\alpha_j, \rho_j\}_{j=1}^i, \sigma$ (Eq. 2.22)
end
```

This setting of non-stationary bandit distributions can be addressed by using a sliding-window approach (e.g. Garivier and Moulines 2008): instead of updating the

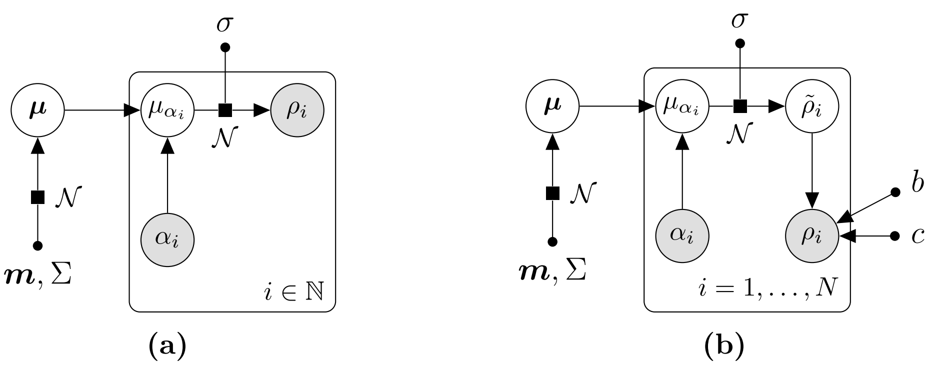

Figure 4.10: **(a)** A Bayesian bandit with Gaussian reward distributions. The rewards from arm k are sampled from $\mathcal{N}(\mu_k, \sigma)$. Thompson sampling (Alg. 1) can infer $\boldsymbol{\mu}$ while trading off exploration and exploitation. **(b)** By introducing the constants b and c, the algorithm can be made scale invariant by performing Thompson sampling with respect to the normalized reward $\tilde{\rho}_i = (\rho_i - b)/c$.

belief parameters $\boldsymbol{m}$ and Σ with respect to the whole history of observations, only keep a window of the last N rollouts, in which we assume that the distributions are approximately stationary.

There remains one other problem: how do we choose the prior parameters $\boldsymbol{m}$ and Σ and the variance σ^2 of the reward distributions? For Σ, the easiest solution is to assume independent arms, i.e. make Σ diagonal. This is not necessarily the most efficient solution, as one can imagine that two similar β values will also perform similarly in their rollouts. We will explore a different approach taking this into account later in this section. For $\boldsymbol{m}$, the non-stationarity becomes a problem: again assuming we use the rollout return as a score, these scores will probably be much lower at the beginning of training than at the end. Additionally, we might not even know the scale of returns in a task. To account for this, it would be necessary to make the prior variances Σ_{kk} very large (uninformed). Similarly, σ needs to be large, to account for the unknown scale of the bandit reward spread. However, this would mean that many more samples (rollouts) are necessary to tighten the belief distributions. This is a problem, especially because we only have a small set of N rollouts when using the sliding-window method.

The ideal would be a bandit method which is invariant with respect to affine transformations of the rewards, in the sense that it would make no difference if all rewards ρ were transformed to be $b\rho + c$ for some constants $b > 0$ and $c \in \mathbb{R}$ for all arms. In Figure 4.10b, this situation is shown in a Bayesian network. Here, the generative process is almost the same as before (see Fig. 4.10a), except that the reward $\tilde{\rho}_i$ is scaled and translated by $\rho_i = b\tilde{\rho}_i + c$ before observation. If, as shown, the constants b and c are independent of the chosen arm and stay constant within the window, it is possible to optimize them via maximum marginal likelihood, given the window of past observations of ρ_i.

The bandit inference task is to infer the distributional means $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_k)$ from the actions (color indices) $\boldsymbol{\alpha} = (\alpha_i)_{i=1}^N$ and rewards (rollout scores) $\boldsymbol{\rho} = (\rho_i)_{i=1}^N$. We set the prior means of the belief distributions to 0 ($\boldsymbol{m} = \mathbf{0}$), because we want the normalized reward distributions to be centered around 0. For now, don't fix Σ, but let it be any positive semi-definite $K \times K$ matrix. The generative model for $\boldsymbol{\rho}$ is defined via the following prior and likelihood function:

$$p(\boldsymbol{\mu} \mid \Sigma) = \mathcal{N}(\boldsymbol{\mu} \mid \mathbf{0}, \Sigma) \tag{4.19}$$

$$p(\boldsymbol{\rho} \mid \boldsymbol{\mu}, \boldsymbol{\alpha}, b, c, \sigma) = \prod_i \mathcal{N}(\rho_i \mid b\mu_{\alpha_i} + c, (b\sigma)^2) \tag{4.20}$$

These lead us to the following evidence (marginal likelihood) function:

$$\begin{aligned}
p(\boldsymbol{\rho} \mid \boldsymbol{a}, b, c, \sigma, \Sigma) &= \prod_i p(\rho_i \mid \alpha_i, b, c, \sigma, \Sigma) && (4.21)\\
&= \prod_i \int p(\rho_i \mid \boldsymbol{\mu}, \alpha_i, b, c, \sigma)\, p(\boldsymbol{\mu} \mid \Sigma)\, \mathrm{d}\boldsymbol{\mu} && (4.22)\\
&= \prod_i \int \mathcal{N}(\rho_i \mid b\mathbf{e}_{\alpha_i}^\top \boldsymbol{\mu} + c, (b\sigma)^2)\, \mathcal{N}(\boldsymbol{\mu} \mid \mathbf{0}, \Sigma)\, \mathrm{d}\boldsymbol{\mu} && (4.23)\\
&= \prod_i \mathcal{N}(\rho_i \mid b\mathbf{e}_{\alpha_i}^\top \mathbf{0} + c, (b\sigma)^2 + b\mathbf{e}_{\alpha_i}^\top \Sigma b\mathbf{e}_{\alpha_i}) && (4.24)\\
&= \prod_i \mathcal{N}(\rho_i \mid c, b^2(\sigma^2 + \Sigma_{\alpha_i\alpha_i})), && (4.25)
\end{aligned}$$

where we used canonical basis vectors to represent $\mu_{\alpha_i} = \mathbf{e}_{\alpha_i}^\top \boldsymbol{\mu}$. For maximization, it is convenient to work with the log-evidence:

$$\begin{aligned}
\log p(\boldsymbol{\rho} \mid \boldsymbol{a}, b, c, \sigma, \Sigma) &= \log \prod_i \mathcal{N}(\rho_i \mid c, b^2(\sigma^2 + \Sigma_{\alpha_i\alpha_i})) && (4.26)\\
&= \sum_i -\frac{1}{2} \log\bigl(2\pi b^2(\sigma^2 + \Sigma_{\alpha_i\alpha_i})\bigr) - \frac{(c-\rho_i)^2}{2b^2(\sigma^2 + \Sigma_{\alpha_i\alpha_i})} && (4.27)\\
&=: L(b, c) && (4.28)
\end{aligned}$$

We can now maximize the evidence with respect to b and c by simply setting the partial derivatives to 0:

$$\partial_c L(b, c) \propto \sum_i (c - \rho_i) = 0 \tag{4.29}$$

$$\partial_b L(b, c) = \sum_i \frac{-1}{b} + \frac{(c-\rho_i)^2}{b^3(\sigma^2 + \Sigma_{\alpha_i\alpha_i})} = 0 \tag{4.30}$$

Solving these equations gives us

$$c = \frac{1}{N} \sum_i \rho_i \tag{4.31}$$

$$b^2 = \frac{1}{N} \sum_i \frac{(c-\rho_i)^2}{\sigma^2 + \Sigma_{\alpha_i\alpha_i}}. \tag{4.32}$$

Using these values, we can "reconstruct" the unscaled or "normalized" reward

$$\tilde{\rho}_i = \frac{\rho_i - c}{b} \tag{4.33}$$

and perform Thompson sampling with respect to $\tilde{\rho}_i$. This *normalized Thompson sampling* algorithm, including the sliding window modification, is presented in Algorithm 2.

Algorithm 2: Normalized TS

```
Input: Arms $B = (\beta_1, \ldots, \beta_K)$,
       Window size $N$
Initialize
 $\boldsymbol{m} \leftarrow \mathbf{0} \in \mathbb{R}^K, \Sigma \in \mathbb{S}_+^K, \sigma \leftarrow 1$
for $l \in \mathbb{N}$ do
    $i \leftarrow l \mod N$
    $M \leftarrow \min\{l, N\}$
    Sample $\boldsymbol{q} \sim \mathcal{N}(\boldsymbol{m}, \Sigma)$
    $\alpha_i \leftarrow \arg\max_{k \in \{1,\ldots,K\}} q_k$
    $\tau_i \leftarrow$ Run rollout with $\beta_{\alpha_i}$
    $\rho_i \leftarrow$ score of rollout $\tau_i$
    $c \leftarrow \frac{1}{M} \sum_{j=1}^{M} \rho_j$
    $b \leftarrow \sqrt{\frac{1}{M} \sum_{j=1}^{M} \frac{(c-\rho_j)^2}{\sigma^2 + \Sigma_{\alpha_j \alpha_j}}}$
    $\tilde{\rho}_i \leftarrow \frac{\rho_i - c}{b}$
    Do Bayesian update of $\boldsymbol{m}, \Sigma$
      using $\{\alpha_j, \tilde{\rho}_j\}_{j=1}^{M}, \sigma$
end
```

With this reward normalization, the prior parameters m (of $\boldsymbol{m} = m\mathbf{1}$) and s (of $\Sigma = s^2 I$) become redundant. We have already set $\boldsymbol{m} = \mathbf{0}$, and we now also set the prior variances Σ_{kk} to 1. This encourages the algorithm to keep the normalized mean estimates μ_k approximately $\mathcal{N}(0,1)$-distributed. The "likelihood" parameter σ remains to be tuned, but it is now not necessary to account for the large uncertainty in the reward scale, as σ is only concerned with the normalized reward. In our experiments, we always set $\sigma = 1$.

Next, we want to show that this method is indeed invariant to affine transformations of the bandit reward.

Proposition 1. *The posterior distribution over $\boldsymbol{\mu}$ in the normalized bandit algorithm (Alg. 2) is identical for the observations $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_N)$ and $\boldsymbol{\rho}' = b'\boldsymbol{\rho} + c'$, for all $b' > 0$ and $c' \in \mathbb{R}$. In other words, the algorithm is invariant to a scaling and translation of the rewards.*

Proof. In this setting, the observed rewards ρ_i are normalized to

$$\tilde{\rho}_i = \frac{\rho_i - c(\boldsymbol{\rho})}{b(\boldsymbol{\rho})} \tag{4.34}$$

with

$$c(\boldsymbol{\rho}) = \frac{1}{N} \sum_{i=1}^{N} \rho_i \tag{4.35}$$

$$b(\boldsymbol{\rho}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \frac{(c(\boldsymbol{\rho}) - \rho_i)^2}{\sigma^2 + \Sigma_{\alpha_i \alpha_i}}}. \tag{4.36}$$

To prove the invariance of the algorithm, we will simply show that this normalized reward is the same for both sets of observations, i.e. that $\tilde{\boldsymbol{\rho}} = \tilde{\boldsymbol{\rho}}'$. Then, clearly, the posteriors $p(\boldsymbol{\mu} \mid \tilde{\boldsymbol{\rho}})$ and $p(\boldsymbol{\mu} \mid \tilde{\boldsymbol{\rho}}')$ will also be the same. Expanding $\tilde{\boldsymbol{\rho}}'$, we get:

$$\tilde{\boldsymbol{\rho}}' = \frac{\boldsymbol{\rho}' - c(\boldsymbol{\rho}')}{b(\boldsymbol{\rho}')} \tag{4.37}$$

$$= \frac{b'\boldsymbol{\rho} + c' - c(b'\boldsymbol{\rho} + c')}{b(b'\boldsymbol{\rho} + c')} \tag{4.38}$$

$$= \frac{b'\boldsymbol{\rho} + c' - \frac{1}{N}\sum_{i=1}^{N}(b'\rho_i + c')}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\frac{(\frac{1}{N}\sum_{j=1}^{N}(b'\rho_j + c') - (b'\rho_i + c'))^2}{\sigma^2 + \Sigma_{\alpha_i\alpha_i}}}} \tag{4.39}$$

$$= \frac{b'\boldsymbol{\rho} + c' - b'\frac{1}{N}\sum_{i=1}^{N}\rho_i - c'}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\frac{(b'\frac{1}{N}\sum_{j=1}^{N}\rho_j + c' - b'\rho_i - c')^2}{\sigma^2 + \Sigma_{\alpha_i\alpha_i}}}} \tag{4.40}$$

$$= \frac{b'(\boldsymbol{\rho} - c(\boldsymbol{\rho}))}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\frac{b'^2(c(\boldsymbol{\rho}) - \rho_i)^2}{\sigma^2 + \Sigma_{\alpha_i\alpha_i}}}} \tag{4.41}$$

$$= \frac{\boldsymbol{\rho} - c(\boldsymbol{\rho})}{b(\boldsymbol{\rho})} \tag{4.42}$$

$$= \tilde{\boldsymbol{\rho}} \tag{4.43}$$

Thus, we can conclude that the reward normalization indeed guarantees invariance to affine reward transformations in algorithms such as Thompson sampling. □

In Section B, we test out this algorithm on a few toy problems.

As mentioned above, it might be advantageous to consider the bandit arms not as independent, but as correlated: if a rollout with $\beta = 1.5$ performs very well, this suggests that $\beta = 1.51$ would probably also perform well. This reasoning can be integrated into the Thompson sampling algorithm simply by using a different prior covariance matrix. A simple approach would be to use something like an RBF kernel:

$$\Sigma_{kk'} = k_{\mathrm{RBF}}(\beta_k, \beta_{k'}) = \exp\left(-\frac{(\beta_k - \beta_{k'})^2}{2\ell^2}\right) \tag{4.44}$$

This would ensure that the information from every rollout is used to update the means for each β. How much a rollout with β influences the posteriors for another β' depends on the distance $|\beta - \beta'|$ and on the chosen length scale ℓ.

We have already discussed in Section 4.1 that β might not be the best metric of exploration behavior. Instead, we found that the "speed" of a certain color of noise is more meaningfully described by $\sigma(2\beta)$ (or, equivalently, $\tanh\beta$). We can incorporate this prior knowledge into the kernel function by transforming the inputs:

$$k(\beta, \beta') = k_{\mathrm{RBF}}(\sigma(2\beta), \sigma(2\beta')). \tag{4.45}$$

This "color kernel" is still a valid positive definite kernel, as both inputs are transformed by the same function (see e.g. Bishop 2006, Eq. 6.19). The color kernel is visualized in Figure 4.11, where it can be compared to the normal RBF kernel. It can be seen how the generalization is larger for high β and smaller for low β.

As noted before, some environments are better described as double integrators than as single integrators, and the speed of a color is then $\sigma(2\beta) + 1$ (see Sec. 4.1).

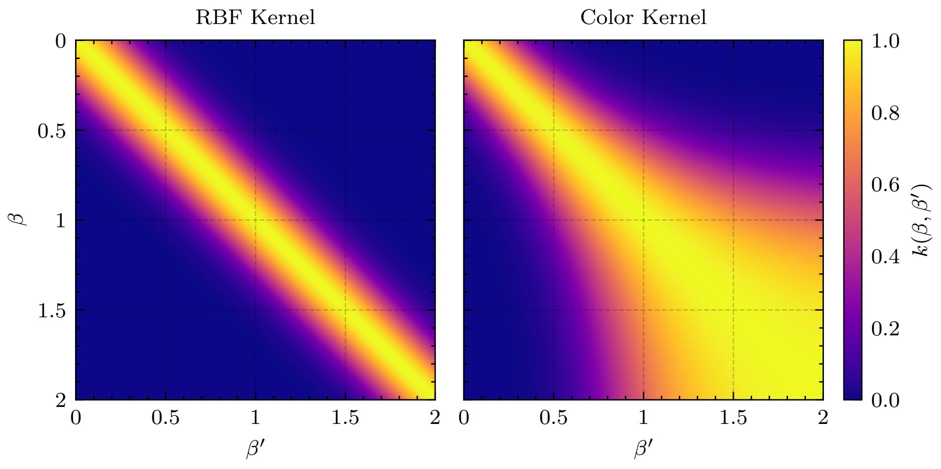

Figure 4.11: Comparison of the RBF kernel (Eq. 4.44, $\ell = 0.2$) and the color kernel (Eq. 4.45, $\ell = 0.1$).

Should we use a different kernel function k' for these environments? Luckily, this is not necessary, as the color kernel is equally valid in these settings:

$$\begin{aligned} k'(\beta, \beta') &= k_{\text{RBF}}(\sigma(2\beta) + 1, \sigma(2\beta') + 1) && (4.46)\\ &= k_{\text{RBF}}(|\sigma(2\beta) + 1 - (\sigma(2\beta') + 1)|) && (4.47)\\ &= k_{\text{RBF}}(|\sigma(2\beta) - \sigma(2\beta)|) && (4.48)\\ &= k(\beta, \beta'), && (4.49) \end{aligned}$$

where we have used that the RBF kernel is stationary and can thus be written as $k_{\text{RBF}}(x, x') = k_{\text{RBF}}(|x - x'|)$.

We can now use the kernel function to account for covariance between the bandit arms by encoding this information in the initialization of Σ. However, using a kernel also opens up a new possibility: we can skip the discretization step and instead directly perform Bayesian optimization on the complete interval $\beta \in [0, 2]$. This can be done using the Gaussian process Thompson sampling algorithm, which is shown in Algorithm 3. It is very similar to the original Thompson sampling algorithm (Alg. 1), and all the changes we discussed, namely using a sliding window, normalizing the reward for scale-invariance, and the color kernel, can be used in the same way.

Algorithm 3: GP-TS

```
Input: Domain $B = [\beta_a, \beta_b]$,
       Reward distributions std $\sigma$
Initialize $m\colon B \to \mathbb{R}, k\colon B^2 \to \mathbb{R}$ p.d.
for $i \in \mathbb{N}$ do
    Sample $q \sim \mathcal{GP}(m, k)$
    $\beta_i \leftarrow \arg\max_{\beta \in B} q(\beta)$
    $\tau_i \leftarrow$ Run rollout with $\beta_i$
    $\rho_i \leftarrow$ score of rollout $\tau_i$
    Do Bayesian update of $m, k$ using
      $\{\beta_j, r_j\}_{j=1}^{i}, \sigma$
end
```

Let us now finally try out these methods in the reinforcement learning setting. We perform a suite of 6 experiments (all of them with the normalized Thompson sampling method), 3 each with both of the proposed scoring functions S_{ret} (episode return) and S_{info} ("info-score"):

- Independent arms ($\Sigma = I$) with $B = \{0, 0.1, 0.2, 0.35, 0.5, 0.75, 1, 1.5, 2\}$,

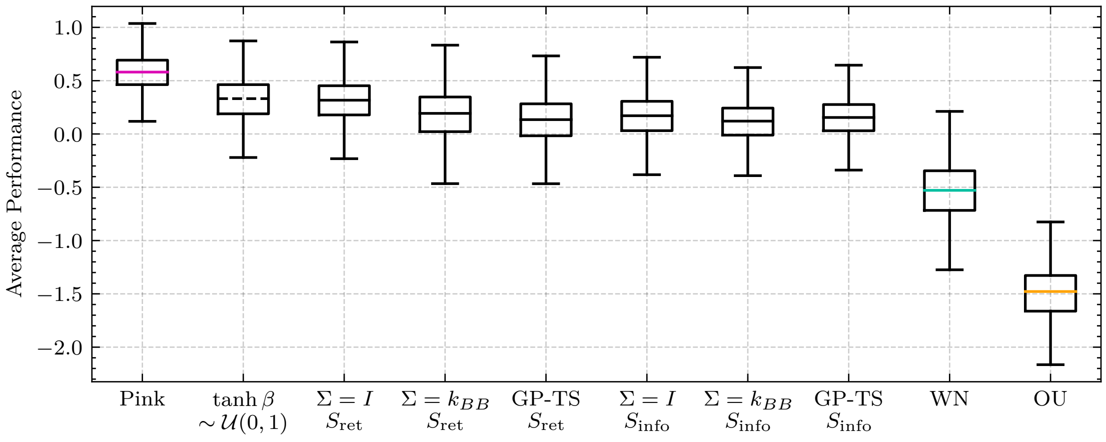

Figure 4.12: Results of the bandit color selection methods. No method manages to come close to pink noise's performance, and they don't even outperform the simple random β selection baseline, shown with a dashed line. (MPO, $N = 10^5$)

- Correlated arms ($\Sigma_{ij} = k(\beta_i, \beta_j)$), where k is the color kernel ($\ell = 0.2$), and
- Bayesian optimization on the interval $B = [0, 2]$ using the color kernel with Gaussian process Thompson sampling (GP-TS).

The results of these experiments are shown in Figure 4.12. It can be seen that the bandit approach does not outperform pink noise, and that all 6 methods perform virtually identically. Perhaps most importantly, the bandit methods do not outperform the random color selection from the last section. This shows that the bandit method does not work as intended, as "random arm selection" should be an easy baseline to outperform. As we have seen that the bandit algorithms do work on simple non-stationary tasks (see Sec. B), the reason for this is probably due to the bandit signals, S_{ret} and S_{info}, not being informative enough as a bandit reward signal.

Having now tested all approaches outlined in Figure 2.7, it seems that no clever scheme will outperform constant pink action noise. In the next chapter, we will try to explain why pink noise seems to perform so well.

Chapter 5

Elementary Dynamics

Why is pink noise such a good default noise type? We have now seen that this is the case, but what quality sets it apart from white noise and OU noise to be well suited to so many environments, even if it is not always the top performing option? In Section 4.1, we discussed the concepts of local and global exploration, and hypothesized that the best exploration behavior provides a balance of the two, such that high reward regions will be found, while trajectories are not too off-policy.

To analyze how different noise types behave, we will look at a simplified *bounded integrator* environment, which can be thought of as a particle moving in a 2-dimensional box, where the actions provide the velocity vector $(\dot{x}, \dot{y})$. If we control this particle purely by noise, we can analyze the exploration behavior in isolation of a policy, similarly to our analysis in Section 2.3. As a first test, we run 20 episodes of 1000 steps in an environment of size 250×250 with white noise, pink noise, and OU noise (all with scale $\sigma = 1$, x- and y-velocity controlled independently). The resulting trajectories are shown in Figure 5.1. It can be seen that white noise does not reach far enough in this environment, and would not be able to collect a sparse reward at the edges: it performs only local exploration. On the other hand, OU noise only explores globally, and gets stuck at the edges. Of the three only pink noise provides a balance of local and global exploration, and covers the state-space more uniformly than the other two.

Of course, this is only true for this specific size of the environment. If the environment were much smaller, then white noise would be enough to cover the space and the pink noise trajectories would look similar to the OU trajectories here. On the other hand, if the environment were bigger, then pink noise would not reach the edges and OU noise would explore better. The best exploration behavior seems to be the one which covers the state space most uniformly: reaching, but not getting stuck at, the edges. The uniformity of the state space coverage is measured by the entropy of the state-visitation distribution induced by the trajectory distribution. This distribution (and hence the entropy) can be estimated by a histogram density approximation: partitioning the state-space into a number of boxes (we choose $50 \times 50 = 2500$ boxes), sampling many trajectories (we choose 10^4), and then counting the number of sampled points in each box. If we now vary the environment size

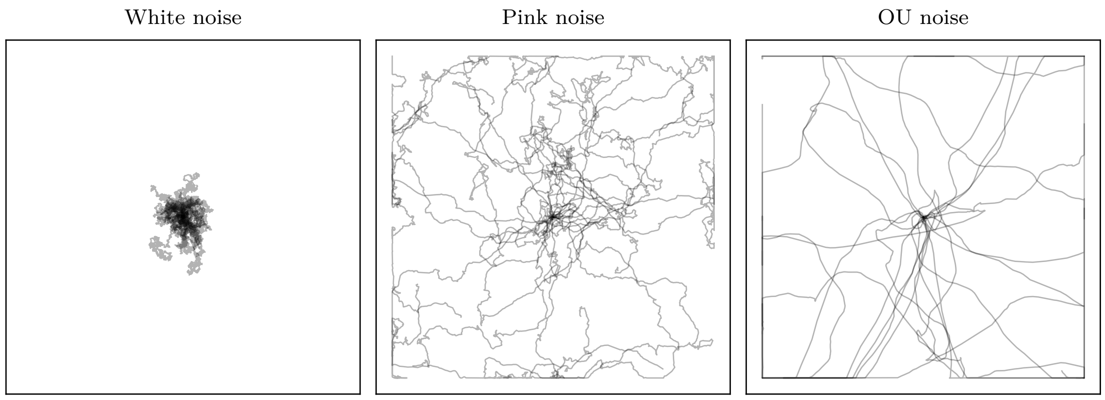

Figure 5.1: Trajectories of pure noise agents on the bounded integrator environment. It can be seen that white action noise (left) only explores locally, OU noise only explores globally, and only pink noise (center) provides both local and global exploration.

and compute the entropy for all three noise types, we should expect that for very small environments, white noise has the highest entropy, and that for very large environments, OU noise has the highest entropy. For medium-sized environments, however, we can expect pink noise to have higher entropy, as we saw in Figure 5.1. We choose the range of environment sizes for this experiment to reflect the complete sensible range for episode lengths of 1000 steps and noise scales of $\sigma = 1$: from very small (50×50) to very large (2000×2000). The results of this experiment are shown on the left in Figure 5.2, and we see exactly what we expected. Pink noise is not "special" in the sense that it performs best on all environments, as we already saw in the previous sections. However, it seems to perform best on "medium scales", as determined by the episode length, and if we do not know where on this spectrum a given environment lies, then pink noise is clearly a better default choice than white noise or OU noise!

The bounded integrator environment, which describes the dynamics of a velocity-controlled particle moving in a box, can be seen as an abstract description of the partial dynamics of many real environments. A different part of the dynamics of many environments is oscillation. Oscillation is the dominant part of the dynamics of the Pendulum and MountainCar environments[1], but is also integral to other environments, like Ball-In-Cup or CartPole. To model these dynamics, we construct a second environment: a driven harmonic oscillator. The oscillator environment, which we make available online as a Gym environment[2], models the 1-dimensional motion of a particle of mass m, attached to the origin by an ideal spring of stiffness k, damped with friction coefficient b, and driven by a force F. The state space consists of the mass's position x and velocity $\dot{x}$, and the action describes the force F that is applied to the mass. The goal is now to drive the oscillator in such a way that as much energy as possible is delivered to the system. This is a very similar goal as in the MountainCar and Pendulum tasks, where an action noise with this property is necessary to collect the sparse reward under an uninformed policy.

[1]See Section C for a simple method exploiting this property to solve MountainCar.
[2]`https://github.com/onnoeberhard/oscillator-gym`

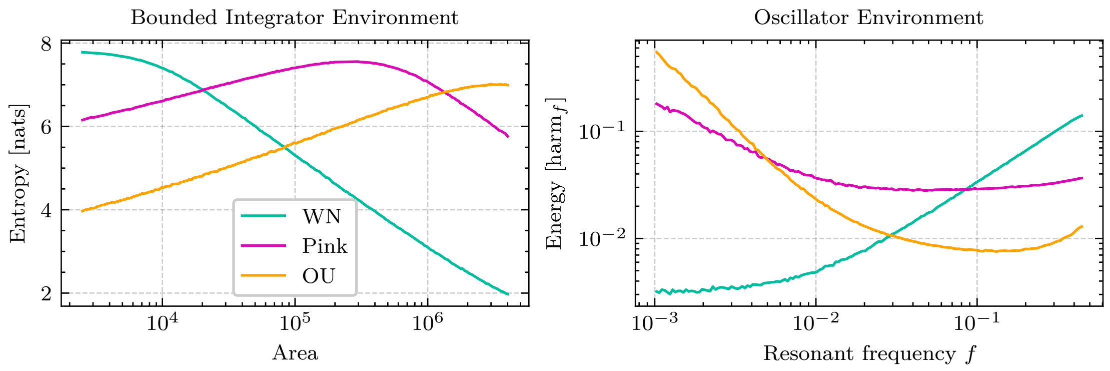

Figure 5.2: Pink noise strikes a favorable middle ground between white noise and Ornstein-Uhlenbeck noise on a wide range of environments. On both a bounded integrator environment parameterized by its size (left), and on a simple harmonic oscillator environment parameterized by its resonant frequency (right), it is much more general in terms of the range of parameters which yield good results, and performs well on the complete range of reasonable parameterizations. We argue that this quality is what makes it a good default.

The oscillator environment is parameterized by the resonant frequency f of the system. How can we set this resonant frequency? The oscillator's motion is described by the ordinary differential equation

$$m\ddot{x} = F - b\dot{x} - kx, \tag{5.1}$$

where x is the particle's position. In our experiments we set the friction coefficient b to zero, i.e. the system is undamped. This setup is then called a *simple harmonic oscillator*. The energy of the oscillator is the sum of kinetic and potential energy:

$$E = \frac{1}{2}m\dot{x}^2 + \frac{1}{2}kx^2. \tag{5.2}$$

The resonant frequency is:

$$f = \frac{1}{2\pi}\sqrt{\frac{k}{m}}. \tag{5.3}$$

As we want to configure the oscillator to have a given resonant frequency f, we need to set m and k accordingly. To get a unique solution, we impose a second constraint: the energy at $x = 1$ and $\dot{x} = 0$ should be $E = 2\pi^2$. If we now solve the two equations (5.2) and (5.3) for m and k, imposing the constraint on E, we get the solution

$$k = 4\pi^2 \tag{5.4}$$

$$m = \frac{1}{f^2} \tag{5.5}$$

to set the resonant frequency. In Figure 5.3, a few pure-noise trajectories (akin to Fig. 5.1) are shown on the oscillator environment.

Figure 5.2 shows the average energy in the oscillator system (over episodes of 1000 steps) as a function of the resonant frequency f, which we vary from very low

($f = \frac{1}{1000}$, episode length = 1 period) to very high ($f = \frac{1}{2}$, Nyquist frequency), when driven by white noise, pink noise, and OU noise. The energy is measured relative to the average energy achieved by optimal actions (sinusoidal excitation at the resonant frequency), denoted harm_f. Even though this is a completely different setup to the bounded integrator, and we are measuring a completely different quantity, the two plots look remarkably similar. Again, this shows the power of pink noise as a default action noise: if we do not know the resonant frequency of the given environment, pink noise is the best choice.

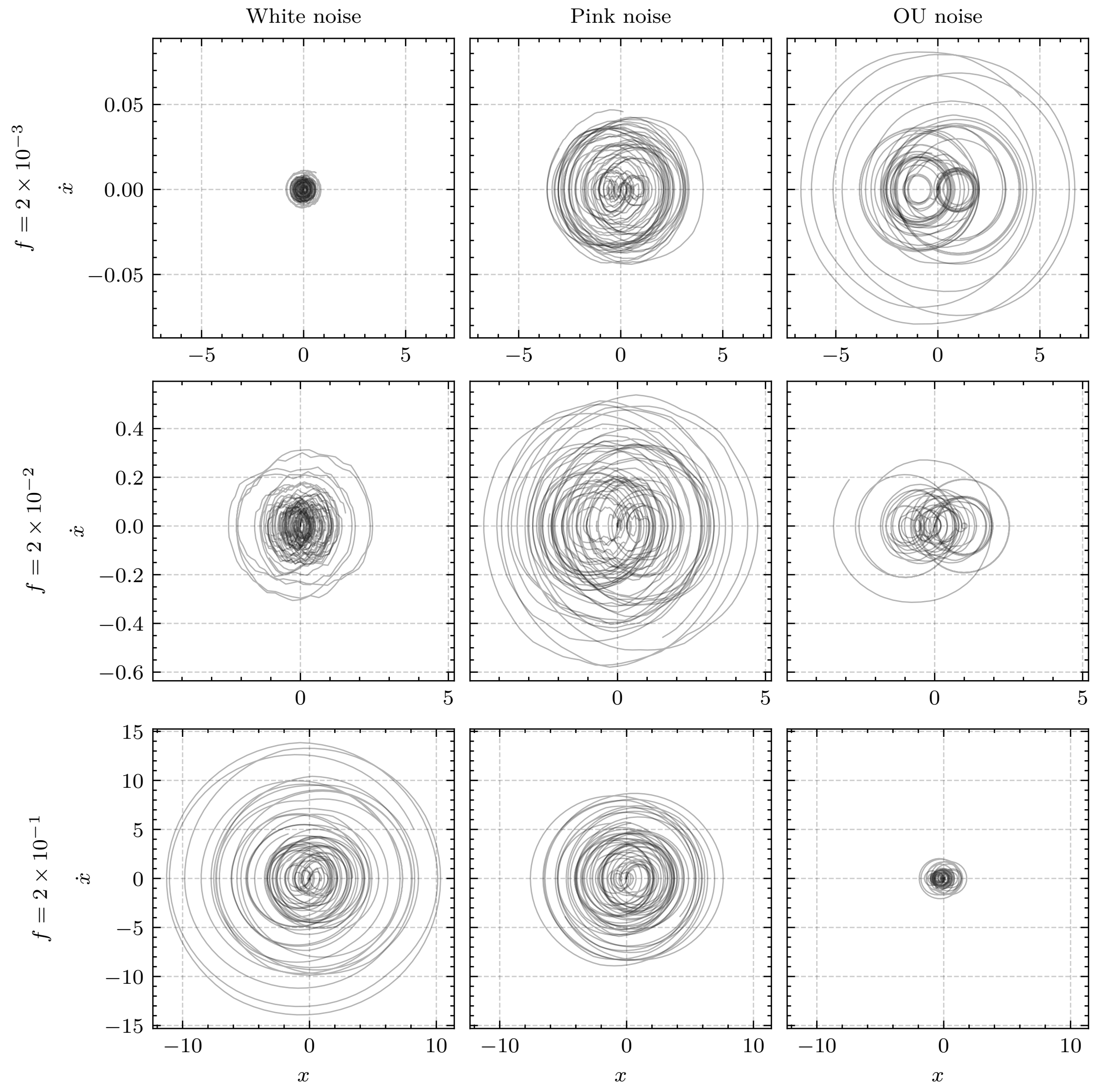

Figure 5.3: Trajectories on the oscillator environment. For each of the 3 resonance frequencies $f \in \{0.002, 0.02, 0.2\}$, we sample 5 action noise signals of length $\frac{10}{f}$ of white noise, pink noise and OU noise. We can see that pink noise is much less sensitive to the parameterization than white noise and OU noise, and always manages to excite the oscillator up to a certain amplitude. White noise and OU noise only work well in the high- and low-frequency regime, respectively.

These two environments (bounded integrator and oscillator) are rather simplistic. However, the dynamics of many real systems undoubtedly contain parts which resemble oscillations (when a spring or pendulum is present), single or double integration (when velocities/steps or forces/torques are translated into positions) or contact dynamics (such as the box in the bounded integrator). If an environment's dynamics are very complex, i.e. they contain many such individual parts, then the ideal action noise should score highly on each of these "sub-tasks". However, if all these individual parts have different parameters (like the environment size or resonant frequency above), it stands to reason that the best single action noise would be the one which is general enough to play well with all parameterizations, i.e. *pink noise*. On the flip side, the average performance in Figure 3.3 over all environments may be interpreted as the performance over a very complicated environment, with the sub-tasks being the "actual" environments. This might explain why we see this curve: all sub-tasks have very different parameters, and require different action noises (as we see in Fig. 3.4), but pink noise is general enough to work well on all sub-tasks, and thus easily outperforms noise types like white noise or OU noise, which are only good on very specific environments (see Fig. 5.2). These ideas are related to our discussion at the end of Section 3.1, where we hypothesized that pink noise might also be the best noise on certain individual environments (those of "third type"), because these environments are complex enough that they require an action noise, such as pink noise, which performs well on all different sub-tasks.

Chapter 6

Conclusion

In this work we performed a comprehensive experimental evaluation of colored noise as action noise in deep reinforcement learning for continuous control. We compared a variety of colored noises with the standard choices of white noise and Ornstein-Uhlenbeck noise, and found that pink noise outperformed all other noise types when averaged across a selection of standard benchmarks. Pink noise performs equally well to an oracle selection of the noise type, and is also not outperformed by more sophisticated methods that change the noise type over the course of training: color-schedules, bandit methods, and random selection schemes. As no method outperforms pink noise, our recommendation is to *use pink noise as the default action noise.* Finally, we studied the behaviors of pure noise agents on two simplified environments: a bounded integrator and a harmonic oscillator. The results showed that pink noise is much more general with respect to the environment parameterization than white noise and OU noise, which sheds some light on why it performs so well as the default choice.

Bibliography

Abdolmaleki, Abbas et al. (2018). "Maximum a Posteriori Policy Optimisation". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. URL: `https://openreview.net/forum?id=S1ANxQW0b` (cit. on p. 9).

Agrawal, Shipra and Navin Goyal (2012). "Analysis of Thompson Sampling for the Multi-armed Bandit Problem". In: *COLT 2012 - The 25th Annual Conference on Learning Theory, June 25-27, 2012, Edinburgh, Scotland*. Vol. 23, pp. 39.1–39.26. URL: `http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf` (cit. on p. 12).

Auer, Peter, Nicolò Cesa-Bianchi, and Paul Fischer (2002). "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Mach. Learn.* 47.2-3, pp. 235–256. DOI: `10.1023/A:1013689704352`. URL: `https://doi.org/10.1023/A:1013689704352` (cit. on p. 12).

Bagatella, Marco, Sammy Joe Christen, and Otmar Hilliges (2022). "SFP: State-free Priors for Exploration in Off-Policy Reinforcement Learning". In: *Transactions on Machine Learning Research*. URL: `https://openreview.net/forum?id=qYNfwFCX9a` (cit. on p. 14).

Bellemare, Marc G. et al. (2016). "Unifying Count-Based Exploration and Intrinsic Motivation". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 1471–1479. URL: `https://proceedings.neurips.cc/paper/2016/hash/afda332245e2af431fb7b672a68b659d-Abstract.html` (cit. on p. 13).

Bertsekas, Dimitri (2017). *Dynamic programming and optimal control: Volume I*. Fourth. Vol. 1. Athena Scientific (cit. on p. 8).

Bishop, C.M. (2006). *Pattern Recognition and Machine Learning*. Springer (cit. on p. 57).

Brockman, Greg et al. (2016). "OpenAI Gym". In: *CoRR* abs/1606.01540. URL: `http://arxiv.org/abs/1606.01540` (cit. on pp. 1, 25).

Cooley, James W. and John W. Tukey (1965). "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19, pp. 297–301 (cit. on p. 20).

Deisenroth, Marc Peter and Carl Edward Rasmussen (2011). "PILCO: A Model-Based and Data-Efficient Approach to Policy Search". In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pp. 465–472. URL: `https://icml.cc/2011/papers/323%5C_icmlpaper.pdf` (cit. on p. 6).

Duarte, Marcos and Vladimir M. Zatsiorsky (2001). "Long-range correlations in human standing". In: *Physics Letters A* 283.1, pp. 124–128. DOI: `https://doi.org/10.1016/S0375-9601(01)00188-8`. URL: `https://www.sciencedirect.com/science/article/pii/S0375960101001888` (cit. on p. 4).

Duff, Michael O'Gordon (2002). *Optimal Learning: Computational procedures for Bayes-adaptive Markov decision processes.* University of Massachusetts Amherst (cit. on p. 13).

Fujimoto, Scott, Herke van Hoof, and David Meger (2018). "Addressing Function Approximation Error in Actor-Critic Methods". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Vol. 80, pp. 1582–1591. URL: `http://proceedings.mlr.press/v80/fujimoto18a.html` (cit. on pp. 9, 18, 75).

Garivier, Aurélien and Eric Moulines (2008). "On Upper-Confidence Bound Policies for Non-Stationary Bandit Problems". In: DOI: `10.48550/ARXIV.0805.3415`. URL: `https://arxiv.org/abs/0805.3415` (cit. on p. 53).

Haarnoja, Tuomas et al. (2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Vol. 80, pp. 1856–1865. URL: `http://proceedings.mlr.press/v80/haarnoja18b.html` (cit. on p. 9).

Hallak, Assaf, Dotan Di Castro, and Shie Mannor (2015). "Contextual Markov Decision Processes". In: *CoRR* abs/1502.02259. URL: `http://arxiv.org/abs/1502.02259` (cit. on p. 50).

Henderson, Peter et al. (2018). "Deep Reinforcement Learning That Matters". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pp. 3207–3214. URL: `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16669` (cit. on pp. 2, 27, 35).

Hennig, Holger et al. (2011). "The Nature and Perception of Fluctuations in Human Musical Rhythms". In: *PLOS ONE* 6.10, pp. 1–7. DOI: `10.1371/journal.pone.0026457`. URL: `https://doi.org/10.1371/journal.pone.0026457` (cit. on p. 4).

Hollenstein, Jakob et al. (2022). "Action Noise in Off-Policy Deep Reinforcement Learning: Impact on Exploration and Performance". In: *CoRR* abs/2206.03787. DOI: `10.48550/arXiv.2206.03787`. URL: `https://doi.org/10.48550/arXiv.2206.03787` (cit. on pp. 17, 43).

Kingma, Diederik P. and Max Welling (2014). "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. URL: `http://arxiv.org/abs/1312.6114` (cit. on p. 14).

Lai, T.L and Herbert Robbins (1985). "Asymptotically efficient adaptive allocation rules". In: *Advances in Applied Mathematics* 6.1, pp. 4–22. DOI: `https://doi.org/10.1016/0196-8858(85)90002-8`. URL: `https://www.sciencedirect.com/science/article/pii/0196885885900028` (cit. on p. 12).

Lattimore, Tor and Csaba Szepesvári (2020). *Bandit algorithms*. Cambridge University Press (cit. on p. 11).

Leone, Fred C, Lloyd S Nelson, and RB Nottingham (1961). "The folded normal distribution". In: *Technometrics* 3.4, pp. 543–550 (cit. on p. 15).

Levine, Sergey (2018). "Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review". In: *CoRR* abs/1805.00909. URL: `http://arxiv.org/abs/1805.00909` (cit. on p. 9).

Lillicrap, Timothy P. et al. (2016). "Continuous control with deep reinforcement learning". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. URL: `http://arxiv.org/abs/1509.02971` (cit. on pp. 9, 17).

Mania, Horia, Aurelia Guy, and Benjamin Recht (2018). "Simple random search of static linear policies is competitive for reinforcement learning". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 1805–1814. URL: `https://proceedings.neurips.cc/paper/2018/hash/7634ea65a4e6d9041cfd3f7de18e334a-Abstract.html` (cit. on p. 13).

Mazoure, Bogdan et al. (2019). "Leveraging exploration in off-policy algorithms via normalizing flows". In: *3rd Annual Conference on Robot Learning, CoRL 2019, Osaka, Japan, October 30 - November 1, 2019, Proceedings*. Vol. 100, pp. 430–444. URL: `http://proceedings.mlr.press/v100/mazoure20a.html` (cit. on p. 13).

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nat.* 518.7540, pp. 529–533. DOI: `10.1038/nature14236`. URL: `https://doi.org/10.1038/nature14236` (cit. on p. 9).

Moore, Andrew William (1990). "Efficient Memory-based Learning for Robot Control" (cit. on p. 1).

Moravec, Hans (1988). *Mind children: The future of robot and human intelligence*. Harvard University Press (cit. on p. 1).

Murphy, Kevin P. (2022). *Probabilistic Machine Learning: An introduction*. MIT Press. URL: `probml.ai` (cit. on p. 53).

Nagabandi, Anusha et al. (2018). "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning". In: *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane, Australia, May 21-25, 2018*, pp. 7559–7566. DOI: `10.1109/ICRA.2018.8463189`. URL: `https://doi.org/10.1109/ICRA.2018.8463189` (cit. on p. 6).

Osband, Ian et al. (2016). "Deep Exploration via Bootstrapped DQN". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 4026–4034. URL: `https://proceedings.neurips.cc/paper/2016/hash/8d8818c8e140c64c743113f563cf750f-Abstract.html` (cit. on p. 13).

Pardo, Fabio (2020). "Tonic: A Deep Reinforcement Learning Library for Fast Prototyping and Benchmarking". In: *CoRR* abs/2011.07537. URL: `https://arxiv.org/abs/2011.07537` (cit. on pp. 10, 27).

Pinneri, Cristina et al. (2020). "Sample-efficient Cross-Entropy Method for Real-time Planning". In: *4th Conference on Robot Learning, CoRL 2020, 16-18 November 2020, Virtual Event / Cambridge, MA, USA*. Vol. 155, pp. 1049–1065. URL:

https://proceedings.mlr.press/v155/pinneri21a.html (cit. on pp. 4, 22, 34).

Plappert, Matthias et al. (2018). “Parameter Space Noise for Exploration”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* URL: https://openreview.net/forum?id=ByBAl2eAZ (cit. on p. 13).

Raffin, Antonin and Freek Stulp (2020). “Generalized State-Dependent Exploration for Deep Reinforcement Learning in Robotics”. In: *CoRR* abs/2005.05719. URL: https://arxiv.org/abs/2005.05719 (cit. on p. 13).

Raffin, Antonin et al. (2021). “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *J. Mach. Learn. Res.* 22, 268:1–268:8. URL: http://jmlr.org/papers/v22/20-1364.html (cit. on pp. 10, 17, 27).

Rajeswaran, Aravind et al. (2018). “Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations”. In: *Robotics: Science and Systems XIV, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, June 26-30, 2018.* DOI: 10.15607/RSS.2018.XIV.049. URL: http://www.roboticsproceedings.org/rss14/p49.html (cit. on p. 25).

Rubinstein, Reuven (1999). “The cross-entropy method for combinatorial and continuous optimization”. In: *Methodology and computing in applied probability* 1.2, pp. 127–190 (cit. on p. 6).

Schulman, John et al. (2015). “Trust Region Policy Optimization”. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015.* Vol. 37, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html (cit. on p. 8).

Schulman, John et al. (2017). “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347. URL: http://arxiv.org/abs/1707.06347 (cit. on p. 8).

Schumacher, Pierre et al. (2022). “DEP-RL: Embodied Exploration for Reinforcement Learning in Overactuated and Musculoskeletal Systems”. In: DOI: 10.48550/ARXIV.2206.00484. URL: https://arxiv.org/abs/2206.00484 (cit. on p. 14).

Silver, David (2015). *Lectures on Reinforcement Learning.* URL: https://www.davidsilver.uk/teaching/ (cit. on p. 10).

Silver, David et al. (2014). “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014.* Vol. 32, pp. 387–395. URL: http://proceedings.mlr.press/v32/silver14.html (cit. on p. 9).

Tang, Haoran et al. (2017). “#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 2753–2762. URL: https://proceedings.neurips.cc/paper/2017/hash/3a20f62a0af1aa152670bab3c602feed-Abstract.html (cit. on p. 13).

Tassa, Yuval et al. (2018). “DeepMind Control Suite”. In: *CoRR* abs/1801.00690. URL: http://arxiv.org/abs/1801.00690 (cit. on p. 25).

Timmer, Jens and Michel Koenig (1995). “On generating power law noise.” In: *Astronomy and Astrophysics* 300, p. 707 (cit. on p. 20).

Uhlenbeck, G. E. and L. S. Ornstein (1930). “On the Theory of the Brownian Motion”. In: *Phys. Rev.* 36 (5), pp. 823–841. DOI: `10.1103/PhysRev.36.823`. URL: `https://link.aps.org/doi/10.1103/PhysRev.36.823` (cit. on pp. 14, 16).

Watkins, Christopher JCH and Peter Dayan (1992). “Q-learning”. In: *Machine learning* 8.3, pp. 279–292 (cit. on p. 8).

Williams, Ronald J. (1992). “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Mach. Learn.* 8, pp. 229–256. DOI: `10.1007/BF00992696`. URL: `https://doi.org/10.1007/BF00992696` (cit. on p. 7).

Appendix A

SAC and TD3 results

In addition to the experiments on MPO discussed in the main text, we also performed all experiments on SAC and TD3. MPO and SAC parameterize a stochastic policy, meaning they learn the action noise scale as a function $\sigma(s)$ of the state. TD3, on the other hand, uses a deterministic policy, and the action noise is added independently of the state. Usually, the noise scale σ is kept fixed over the course of training, and this how we handle it in our experiments as well. However, σ is an important hyperparameter, and there is no single value that works well on all environments. Thus, we repeat our TD3 experiments with the values $\sigma \in \{0.05, 0.1, 0.3, 0.5, 1\}$, and 10 different random seeds. For SAC, we use 20 seeds, just as with MPO.

In Figure A.1, the results of the SAC and TD3 experiments with constant noise type are shown in the form of bootstrap distributions for the expected average performance (see Sec. 3.1), and compared to the same experiments on MPO, as well as to a combination of all results, where the influence of the algorithm has been normalized out. As there is an additional hyperparameter (σ), we first average the TD3 performance over all σ values, before computing the average performance across tasks. The beneficial effect of pink noise can be clearly seen on TD3 and SAC as well. Incidentally, this figure also confirms Fujimoto et al. (2018)'s results that, on TD3, white noise and OU noise perform similarly. In Figure A.2, we show the learning curves on TD3 and SAC (the MPO learning curves are shown in Fig. 3.5).

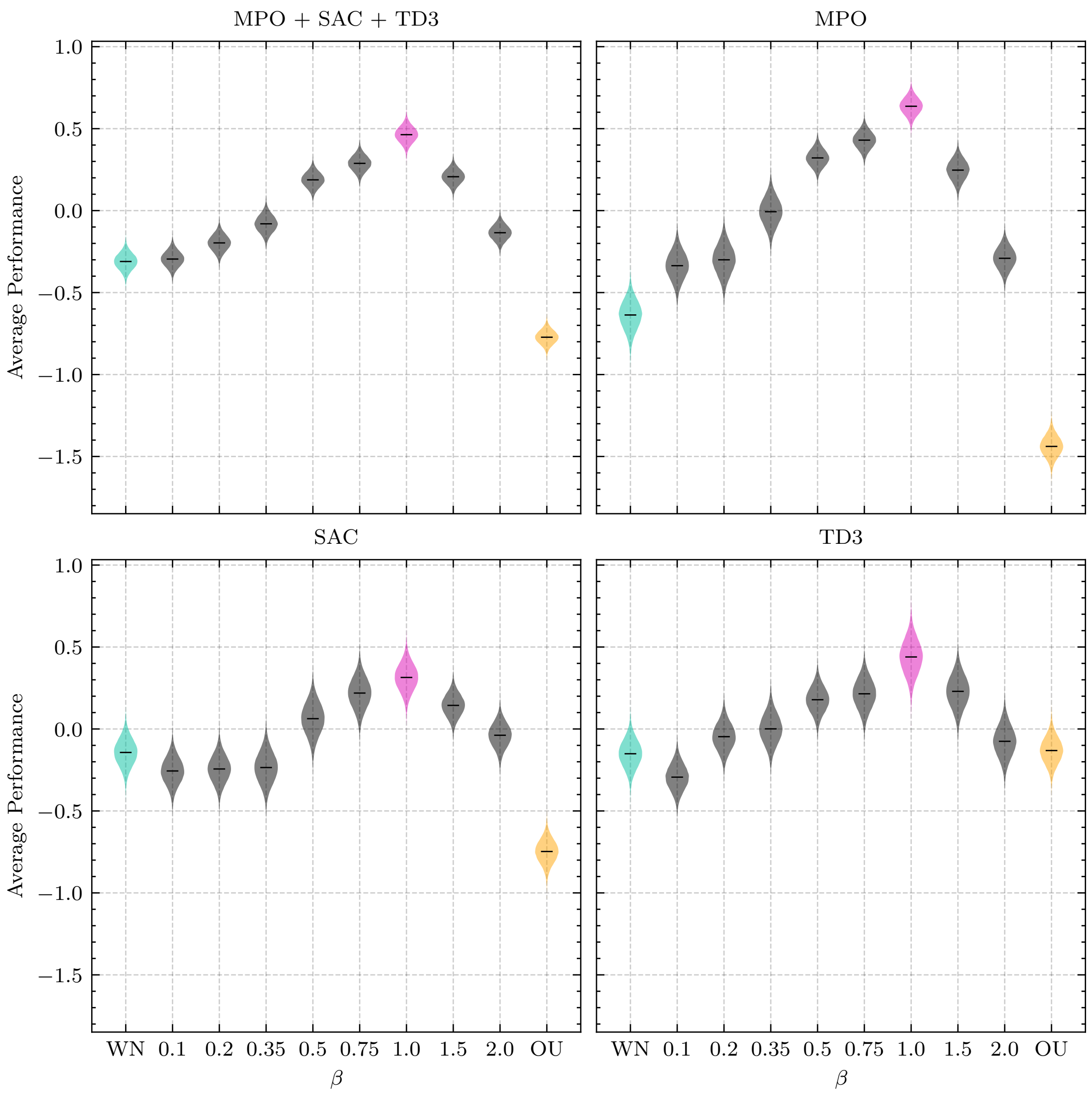

Figure A.1: All three algorithms (MPO, SAC, TD3) show a clear preference for pink action noise as measured by the average performance over the environments of Figure 3.1.

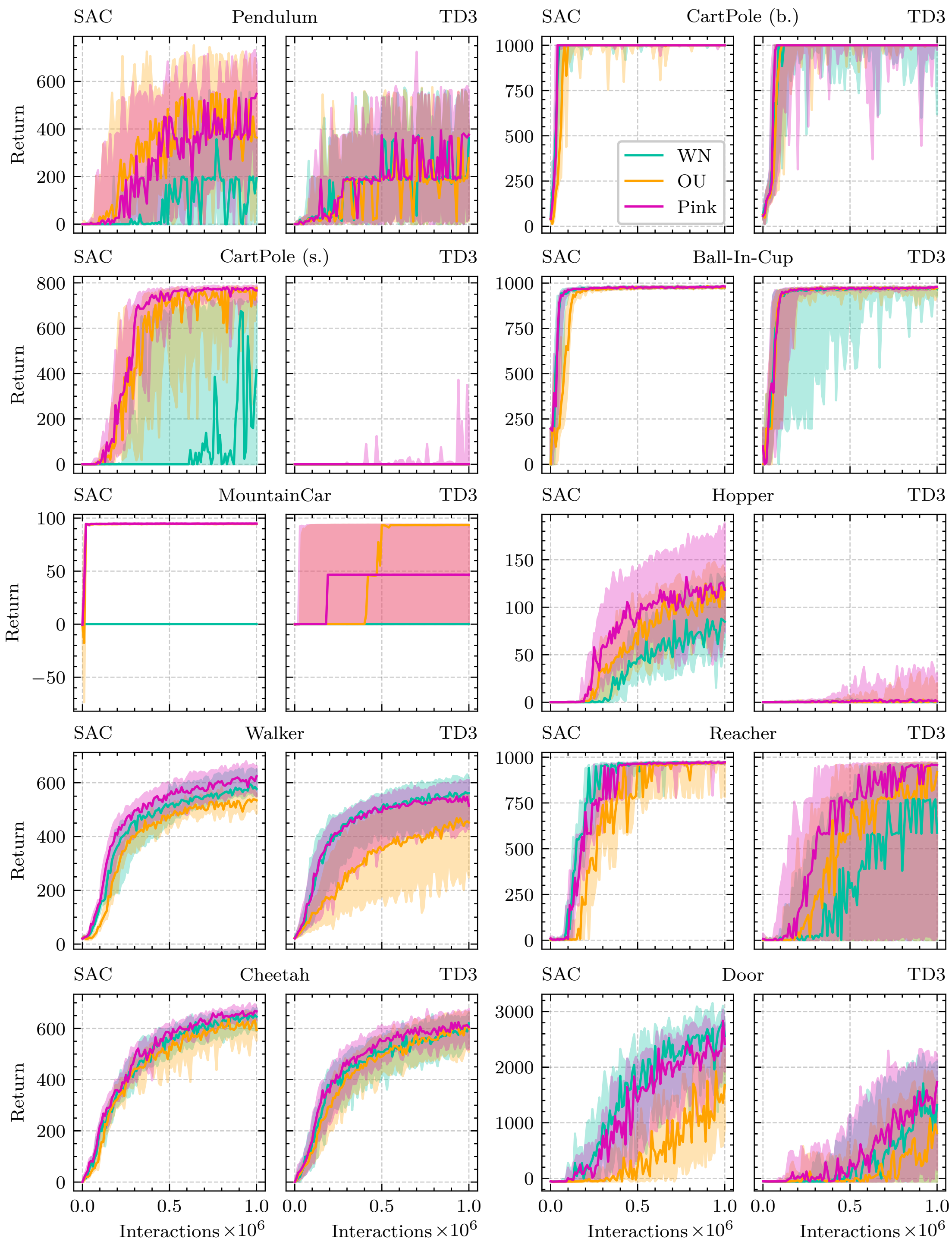

Figure A.2: Learning curves on SAC and TD3 (median and interquartile range of evaluation returns)of the two baseline action noise types white noise (WN) and Ornstein-Uhlenbeck (OU) noise, as well as of pink noise. Here, the noise scale σ on TD3 is ignored here and treated identically to the random seeds (i.e. where for SAC there are 20 lines per experiment, there are 50 lines for TD3). It can be seen that pink noise, while not being better than both baselines on all environments, is the best default choice, as it is never outperformed by both white noise and OU noise.

Appendix B

Normalized Thompson sampling

In Figure B.1, six non-stationary bandit problems are shown. Each bandit has four arms (represented by four primary colors), and the reward distributions are all Gaussian, though they change over time (i.e. they are non-stationary). These distributions are supposed to represent (non-realistically), how the rollout scores might change over time, depending on which arm (β) is selected. The biggest challenge with this setup is not necessarily that these problems are non-stationary. As explained in Section 4.2, this can be addressed by using a sliding window approach. Instead, the biggest challenge is that we do not normally know the scale of the rewards. As can be seen in the figure, these scales vary by a large amount from distribution to distribution, as can also be the case with reinforcement learning environments. Thus, without the reward normalization of Algorithm 2, we will have to carefully select a likelihood noise scale σ, as well as the prior mean m (where $\boldsymbol{m} = m\mathbf{1}$) and standard deviation s (where $\Sigma = s^2 I$). This is not easy, as all distributions are very different. We choose the parameters $\sigma = 500$, $m = 1000$ and $s = 1000$. These might not be the very best choices, but as one normally does not even have a nice picture like this to decide these hyperparameters, we think these are as good as one might choose. The "trainings" last for 10,000 steps, and we set the sliding window size to 1000.

In Figure B.2, the results of this (unnormalized) approach are shown. For each of the six problems, two plots are created. The first one shows a scatter plot of the *history*: At every time step, a dot indicates the received reward and, by its color, the arm that was chosen. Below the scatter plots, a colored horizontal line indicates for every time step, which arm (color) is the best one, in terms of the mean of the reward distribution. Thus, if the scatter plots have the same color as the horizontal line below, the bandit algorithm is selecting the best arms. The second plot for every bandit problem shows the *mean estimates*. Here, for each of the four arms a, the current value of the posterior mean estimate m_a is plotted. Below these lines, there are two horizontal lines. The lower one is the same as in the history plots, indicating which arm has the highest *actual* mean. The upper one shows which arm has the highest *estimated* mean. Thus, if both lines are colored identically, this is a good sign. The title of the history plot also reveals the total regret R at the end of training, as defined by Eq. (2.18), as well as the "optimality ratio": how many

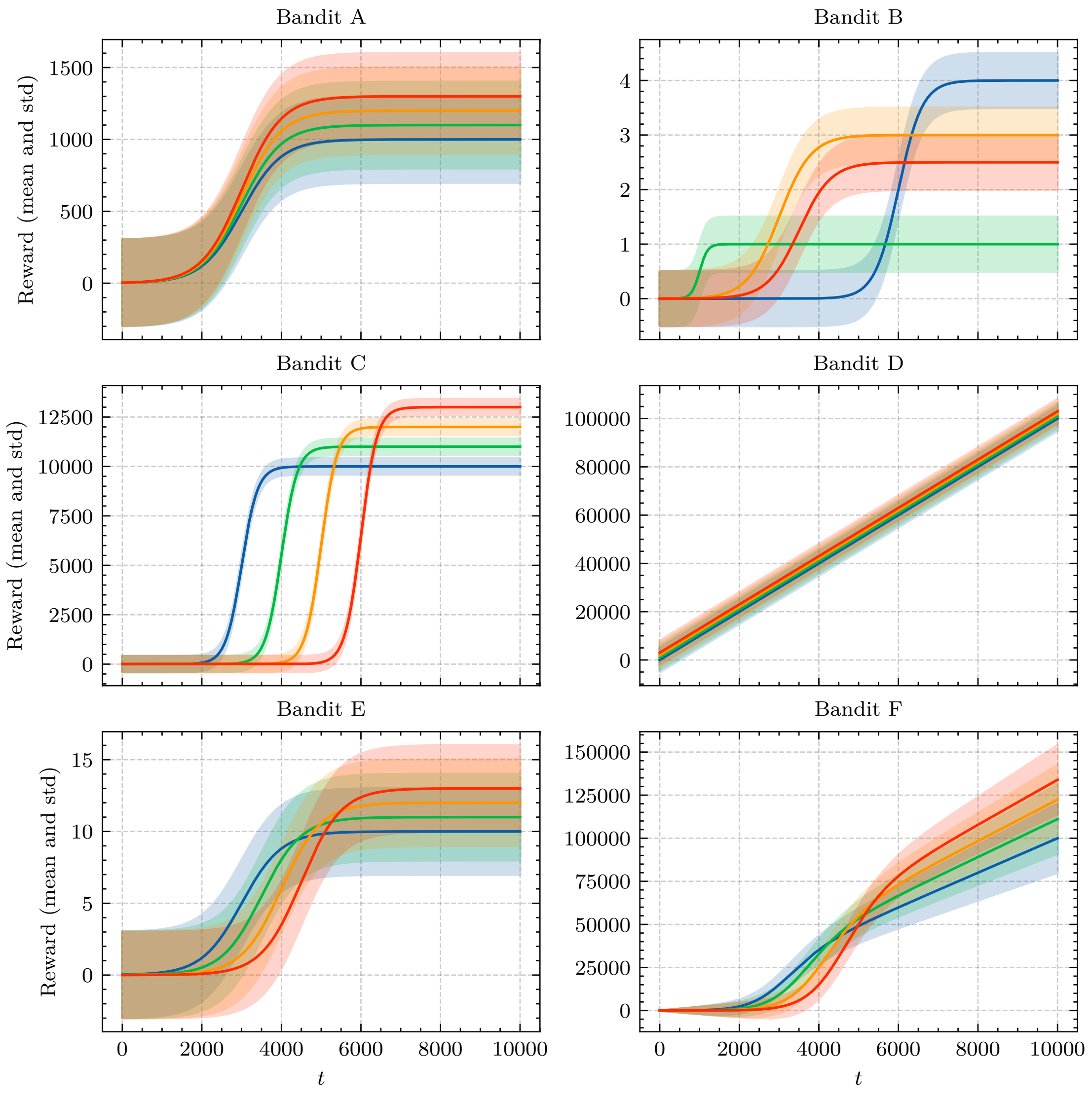

Figure B.1: A few non-stationary bandit toy problems

of the bandit algorithm's choices were optimal, in the sense that, at that point in time, the chosen arm was the best one. Keep in mind that randomly selecting an arm at every interaction will yield an optimality of $\frac{1}{4} = 25\%$.

The results of the normalized Thompson sampling algorithm (Alg. 2) are shown in Figure B.3 (here, the default hyperparameters of $m = 0, s = \sigma = 1$ were used). It can be seen that the unnormalized approach performs very poorly when compared to the normalized method, which outperforms the unnormalized method on each of the six toy problems.

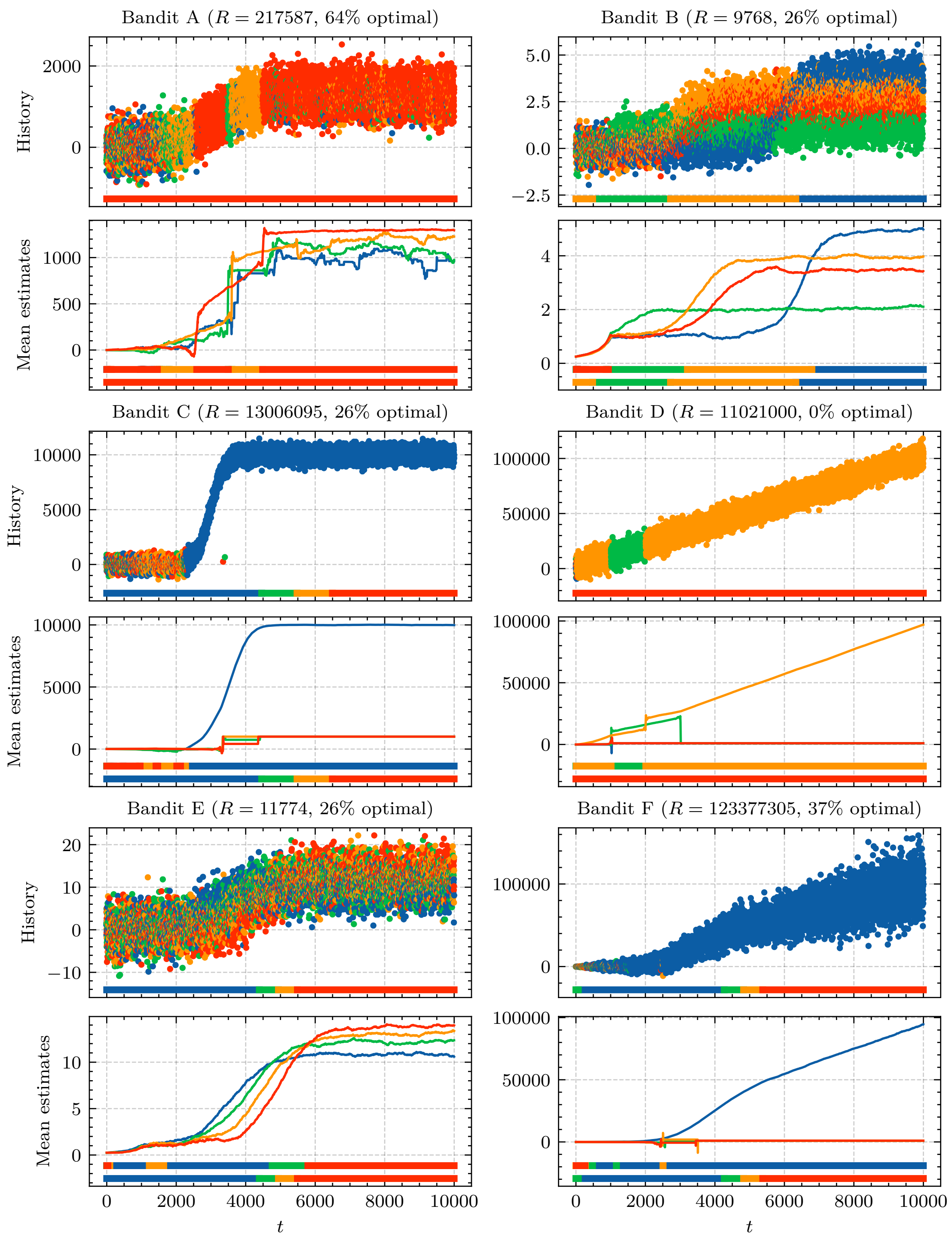

Figure B.2: Results of the unnormalized Thompson sampling method (see text for details). It can be seen that on some problems, the algorithm is too certain and chooses the wrong arm with confidence (C, D, F), while on other problems, it is not confident enough and explores too much (B, E).

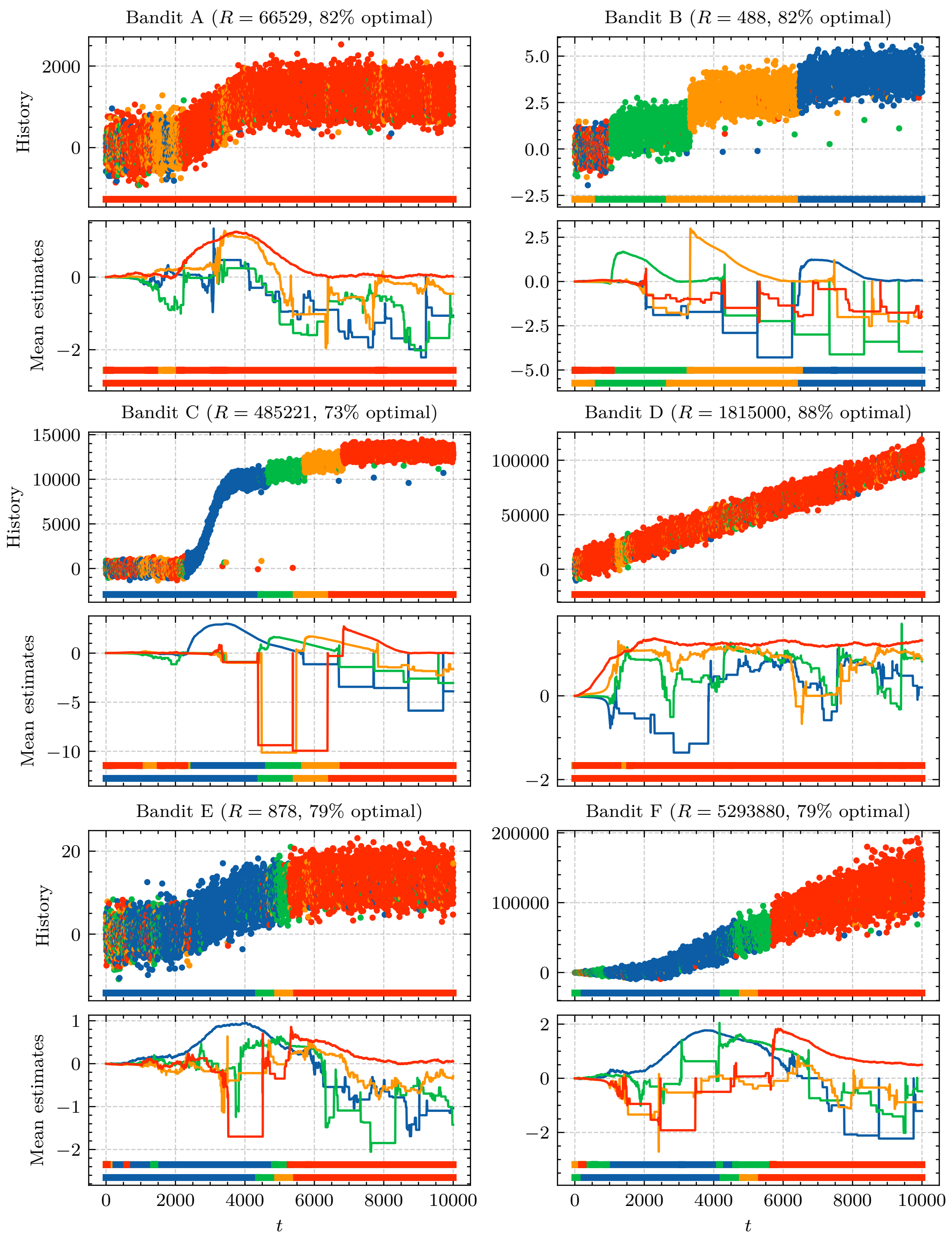

Figure B.3: Results of the normalized Thompson sampling method (see text for details). The invariance to the reward scale (guaranteed by Prop. 1) makes this method outperform the unnormalized version on every one of these toy problems.

Appendix C

Solving MountainCar by FFT

MountainCar is a very simple environment. Although its dynamics are almost those of a harmonic oscillator, there is a difference to the oscillator environment from Chapter 5: MountainCar's oscillation dynamics are non-linear. At the bottom of MountainCar's valley (see Fig. 1.1), the small-angle approximation of a non-linear oscillator may be used, but for the motion to go up to the top, the behavior is different from simple harmonic motion. Nevertheless, we can use this insight to develop a very simple open-loop control algorithm to solve this environment, by running one rollout without applying any action (just letting the mountain make the car go back and forth a bit), then analyzing the resulting trajectory and inferring the hill's (small-angle) resonant frequency (via the Fast Fourier Transform algorithm). Finally, we can control the car by simply swinging it back and forth at the resonant frequency. This algorithm, which works very well on this task, is shown on the next page, and the trajectory resulting from the rollout without applying any force is shown in Figure C.1.

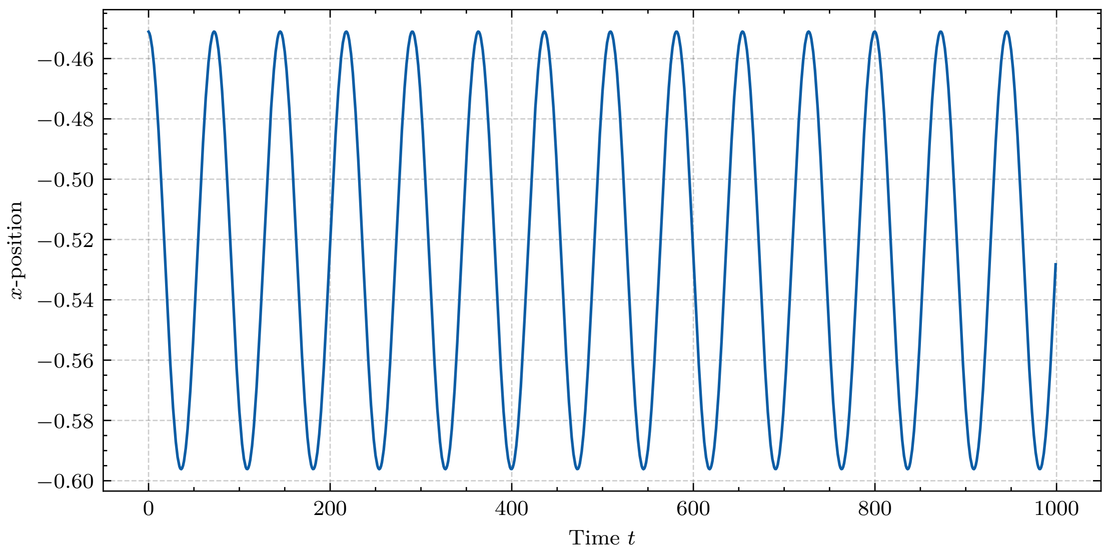

Figure C.1: Trajectory (x-coordinate) resulting from running one full episode (1000 steps) on MountainCar without applying any force ($a_t = 0, \forall t$).

```python
import gym
import numpy as np
from scipy.fft import rfft

# Initialize environment
env = gym.make('MountainCarContinuous-v0')
T = env._max_episode_steps

# Run a single rollout with no force. Save x-coordinate to `x`.
obs = env.reset()
x = [obs[0]]
for t in range(T):
    obs, *_ = env.step([0])
    x.append(obs[0])

# Find resonant frequency = highest peak of FFT (excluding DC)
f = (np.argmax(abs(rfft(x))[1:]) + 1) / (T + 1)

# Action plan (harmonic excitation)
a = np.sin(2*np.pi*f * np.arange(T))

# Test on 1000 rollouts
N = 1000
solved = 0
for i in range(N):
    env.reset()
    for t in range(T):
        _, r, _, _ = env.step([a[t]])
        if r > 0:
            solved += 1
            break

print(f"Solved: {solved/N * 100:.0f}%.")  # prints "Solved: 100%."
```