Reinforcement Learning

The following content is largely inspired by and adapted from the course 6.484(6.8200) Computational Sensorimotor Learning, taught by Prof. Pulkit Agrawal at MIT. The 6.484 lecture notes were written by Zhang-Wei Hong and Pulkit Agrawal.

Basics

Before we dive into the details of reinforcement learning (RL), let's begin by understanding a fundamental problem that captures the essence of RL: the Multi-Armed Bandit problem. This seemingly simple problem offers a clear perspective on the critical trade-off between exploration and exploitation, two pivotal concepts in RL.

Multi-Armed Bandit

The multi-armed bandit (MAB) problem captures a scenario in which an agent interacts with the environment by making a sequence of choices, and receives rewards based on those choices. Here, the agent faces a row of slot machines, each offering a different payout. Every time the agent chooses a machine, it receives a reward drawn from a fixed but unknown probability distribution specific to that machine. The objective is to maximize the expected cumulative reward over a series of pulls. We discuss four strategies the agent can take and analyze their performances, including random selection, explore-then-commit (ETC), ϵ-greedy and Upper Confidence Bound (UCB).

To help you understand, we provide an example at the end of each sub-section, but make sure to understand the generalized version of the problem as well! The derivations of expected reward bounds for more complex strategies are omitted here as they exceed the scope of this lecture. However, the key takeaway is to understand and evaluate the tradeoffs between exploration and exploitation. The specific example provided offers important insights for comparing various strategies.

Let K denote the number of slot machines. At each timestep, the agent takes an action $a_t \in \{1, \ldots, K\}$, which denotes the slot machine the agent chooses at timestep t. After taking action a_t, the agent receives a reward $r(a_t)$, which is the payout the agent receives from pulling slot machine a_t. This reward is randomly drawn from an independent underlying distribution of the chosen arm. Here, the goal is to maximize the cumulative reward over a finite time horizon T, which can be expressed as:

$$\max_{a_1,\ldots,a_T} \mathbb{E}\Big[\sum_{t=1}^{T} r(a_t)\Big]$$

Let μ_k be the true mean reward of taking action k (which is equivalent to the expected reward for pulling the k-th slot machine, i.e. $\mu_k = \mathbb{E}[r(k)]$), and $\mu^* = \max_k \mu_k$ be the optimal mean reward among the k machines. Another way of

looking at this problem is to consider minimizing the gap between the optimal expected reward and the expected reward obtained by our strategy over T time steps, which is characterized by *regret*, denoted as:

$$\textbf{Regret}(T) = T \cdot \mu^* - \mathbb{E}\big[\sum_{t=1}^{T} r(a_t)\big]$$

Example: Consider a scenario with $K = 10$ slot machines, each representing a unique arm of the MAB problem. The k-th slot machine has an underlying reward distribution of $B(0.1 \cdot k)$, where $B(p)$ denotes the Bernoulli distribution with parameter p. In other words, the first machine ($k = 1$) has a 10% chance of yielding a reward, the second machine ($k = 2$) has a 20% chance, and so on. Note that this underlying reward distribution is unknown to the agent, who seeks to devise a strategy that maximizes the expected total reward over a series of plays. Formally, it seeks to find the set of actions:

$$\arg\max_{a_1,\ldots,a_T} \mathbb{E}\big[\sum_{t=1}^{T} r(a_t)\big]$$

Given the distribution, we have $\mu_k = \mathbb{E}[r(a_t)] = 0.1 \cdot k$. The last arm ($k = 10$) has the optimal mean reward $\mu^* = \max_k \mu_k = 1$. Recall that this is unknown to the agent. Suppose the agent has the opportunity to engage with the environment over a duration of $T = 100$ timesteps, then the regret for each sequence of actions $a_1, \ldots, a_{100}$ is:

$$\textbf{Regret}(100) = 100 \cdot \mu^* - \mathbb{E}\big[\sum_{t=1}^{100} r(a_t)\big] = 100 - 0.1 \cdot \sum_{t=1}^{100} a_t$$

The regret is to capture the difference between the expected reward for taking actions $a_1, \ldots, a_{100}$ and what the agent could have earned by *always* choosing the slot machine with the optimal expected reward. Given that the last arm is guaranteed to give you a reward, the optimal strategy here is to always choose the last arm, which yields an expected reward of 100. Any deviation from this strategy will result in some regret as the expected reward is lower than the optimal expected reward.

Random Selection

We first consider the most naive strategy, in which we randomly select an arm at each turn. Since each arm is chosen with probability $\frac{1}{K}$, the expected reward at any time t can be calculated as:

$$\mathbb{E}\big[r(a_t)\big] = \sum_{k=1}^{K} \frac{1}{K} \mu_k$$

Therefore, we can obtain the expected cumulative reward and the regret over T time steps:

$$\mathbb{E}\big[\sum_{t=1}^{T} r(a_t)\big] = T \cdot \sum_{k=1}^{K} \frac{1}{K}\mu_k = T \cdot \frac{1}{K}\sum_{k=1}^{K} \mu_k$$

$$\textbf{Regret}(T) = T \cdot \mu^* - T \cdot \frac{1}{K}\sum_{k=1}^{K} \mu_k$$

Given that μ^* is the optimal mean reward, and the term $\frac{1}{K}\sum_{k=1}^{n} \mu_k$ is the average reward across all arms, the regret essentially captures the difference between always selecting the best arm and selecting arms uniformly at random. However, this strategy does not utilize any past information to influence future decisions, therefore does not "learn" over time. This leads to a higher regret as compared to other strategies that exploit past observations to make more informed decisions. Therefore, this *naive* strategy can also be considered to be pure exploration without any exploitation.

Using the previous example, we analyze the regret for the random selection strategy. Since each arm has an equal chance of being selected, the expected reward at each timestep is:

$$\mathbb{E}\big[r(a_t)\big] = \sum_{k=1}^{10} \frac{1}{10} \cdot \frac{1}{10}k = 0.55$$

In other words, using the random selection strategy, the expected reward at each timestep is 0.55, which is lower than the optimal expected reward of 1 (by selecting the last arm). Therefore, the regret over 100 timesteps is:

$$\textbf{Regret}(100) = 100 - 100 \cdot 0.55 = 45$$

Explore-then-commit (ETC)

Next, we consider the explore-then-commit (ETC) strategy, which consists of two phases:

- During the exploration phase the first $m \cdot K$ rounds are dedicated to exploration, where each action is attempted m times. Since each arm is selected the same number of times, the empirical mean of each arm should converge to its true mean reward as m increases. During this phase, we experience some regret when we pull an arm that is not the optimal arm.
- After exploration, the agent commits to the action with the highest empirical mean reward for the rest of the $T - m \cdot K$ rounds. Ideally, if the exploration phase has been sufficiently long, this would be the arm with the highest true mean reward.

Let $B_k(T) \triangleq \sum_{t=1}^{T} \mathbb{1}_{\{a_t=k\}}$ denote the number of times arm k is pulled over T timesteps according to the strategy, where $\mathbb{1}_{\{\cdot\}}$ is the indicator function. Observe that $\sum_{k=1}^{K} B_k(T) = T$. Therefore, we can formulate regret of a certain strategy over T timesteps as:

$$\begin{aligned}
\textbf{Regret}(T) &= T \cdot \mu^* - \mathbb{E}\Big[\sum_{t=1}^{T} r(a_t)\Big] \\
&= T \cdot \mu^* - \sum_{k=1}^{K} \mathbb{E}\big[B_k(T)\big] \cdot \mathbb{E}\big[r(k)\big] \\
&= \sum_{k=1}^{K} \mathbb{E}\big[B_k(T)\big]\Big(\mu^* - \mathbb{E}\big[r(k)\big]\Big) \\
&= \sum_{k=1}^{K} \Delta_k \mathbb{E}\big[B_k(T)\big]
\end{aligned}$$

where $\Delta_k \triangleq \mu^* - \mu_k$ be the suboptimality gap of action k, which indicates the expected loss of reward of pulling arm k instead of the optimal arm. In the first mK rounds, the policy is deterministic in which each arm is pulled exactly m times. In the exploitation phase, the agent chooses a single action that maximizes the average reward during exploration. Therefore,

$$\mathbb{E}\big[B_k(T)\big] \leq m + (T - mK) \cdot \mathbb{P}\Big(\hat{\mu}_k(mK) \geq \max_{j \neq k} \hat{\mu}_j(mK)\Big)$$

where $\hat{\mu}_k(t)$ indicates the empirical expected reward of arm k after t timesteps of exploration. The inequality here is to account for the case where multiple arms have the same empirical mean and arm k is not chosen due to tie-breaking rules.

In the remaining analysis of the Multi-Armed Bandit (MAB) problem, we assume the underlying reward function for each arm to be 1-subgaussian°. Intuitively, 1-subgaussian is a class of distributions whose probability of extreme values cannot be higher than that of a Gaussian distribuion with variance of 1. This class includes many common distributions such as Bernoulli distribution and any Gaussian distributions with $\sigma^2 \leq 1$. This assumption enables us to provide tighter bounds for the empirical expected reward for each arm, and make our computation tractable.

← **Optional**: Formally, a subgaussian distribution indicates that its tail is dominated by the tails of a Gaussian distribution. A random variable X is said to be σ-subgassian if for all t, it satisfies:

$$\mathbb{E}[e^{tX}] \leq e^{\sigma^2 t^2/2}$$

Under this assumption, the regret for the ETC algorithm is bounded by

$$\textbf{Regret}(T) \leq m \sum_{k=1}^{K} \Delta_k + (T - mK) \sum_{k=1}^{K} \Delta_k \exp\big(-\frac{m \cdot \Delta_k^2}{4}\big)$$

The two terms correspond to the regrets for the two phases respectively. The formal derivation of the regret term is beyond the scope of these lecture notes. Observe that the performance of ETC heavily depends on m, the number of times each arm is sampled during exploration. Setting m too small might mean the empirical means aren't close to true means, leading to a poor exploitation choice, while setting m too large means most of the rounds are spent exploring, which

isn't efficient either. A common practice is to set $m = T^{2/3}$, which provides a regret bound of $O(KT^{2/3})$. This balance ensures that enough rounds are spent exploring to get a good estimate of the arm rewards while leaving enough rounds for exploitation. ETC is a simple yet powerful strategy for the MAB problem, as it divides the problem into distinct exploration and exploitation phases, ensuring that past information is used to make future decisions.

We continue on the previous example. Consider the ETC strategy, in which the agent partitions 100 timesteps into 50 timesteps for exploration, then commits the remaining 50 timesteps to the arm with the highest empirical mean. During the exploration phase, each arm is sampled 5 times, resulting in a regret of:

$$\textbf{Regret}(50) = 50 - 50 \cdot 0.55 = 22.5$$

where 0.55 is the average expected reward across all arms. The agent keeps track of the reward received by pulling each arm and evaluates the empirical mean $\hat{\mu_k}$. Suppose the agent is lucky and is able to identify the optimal arm after the exploration phase. The agent then commits to the last arm for the remaining 50 timesteps, resulting in a regret of 0. Therefore, the total reward for this strategy is 22.5, which is lower than the random selection strategy. However, since each arm is only sampled 5 times during the exploration phase, it is highly possible that the empirical mean has not converged to the true reward mean. If the agent commits to an arm that is not the optimal arm, it will result in a non-zero regret during the commitment phase.

In this example, we can compute the probability where the agent is unable to identify the optimal arm after sampling each arm 5 times. Assume that in case of a tie, an arm with the highest empirical expected reward is selected at random. Since the last arm always succeeds on every pull, it will have a success count of 5 after the exploration phase. Given the probabilities, none of the other arms can surpass 5 successes because their success rates are all below 100%. Let A_k denote the event where arm k is selected as the optimal arm after the exploration phase. For arm $k \neq 10$ to be selected, the arm has to match the perfect success record during exploration, and be selected at random among the tied arms. We obtain the bound:

$$\mathbb{P}(A_k) \leq \frac{1}{2} \cdot (\frac{k}{10})^5$$

since for A_k to happen, arm k has to succeed at each pull and win the tie-breaker, in which there will always be another arm (the last arm) in the competition. Therefore, we have:

$$\mathbb{P}(A_{10}) \geq 1 - \sum_{k=1}^{9} \mathbb{P}(A_k) \approx 0.396$$

As shown above, after the exploration phase, there remains a significant probability of choosing a sub-optimal arm. Nonetheless, the arm selected is typically close to being optimal:

$$\mathbb{P}(A_8 \cup A_9 \cup A_{10}) \geq 1 - \sum_{k=1}^{7} \mathbb{P}(A_k) \approx 0.855$$

We can increase the probability of selecting the optimal arm by increasing the timesteps used for exploration, such that the empirical mean converges to the true reward mean, but at the expense of less timesteps during the commitment phase. For example, if we sample each arm 8 times instead, we have:

$$\mathbb{P}(A_{10}) \geq 0.661$$

such that we have a higher probability of selecting the optimal arm, but we only have 20 timesteps remaining for exploitation. If we consider a longer event horizon, in which we can sample each arm 30 times, we have:

$$\mathbb{P}(A_{10}) \geq 0.978$$

ϵ-greedy Algorithm

The ETC strategy builds upon the idea of balancing exploration and exploitation using two separate phases. We can integrate the two phases, which ensure continuous exploration and exploitation throughout the decision making process. In the ϵ-greedy algorithm, the agent, with probability ϵ, chooses a random action to explore, and with probability $1 - \epsilon$, chooses the action with the highest estimated reward so far. Intuitively, at each step, it explores by choosing a random arm with some probability, or exploits by pulling the arm with the highest current estimated reward. If we compare ϵ-greedy to ETC, we can see that with the same exploration rate (i.e. by exploring each arm $m = \frac{1}{k}T\epsilon$ times in the ETC algorithm), ETC is expected to outperform ϵ-greedy because (a) the exploration phases yield the same regret, and (b) The exploitation phase in ETC happens **after** the exploration phase, which implies that it utilizes the knowledge of all timesteps during exploration. Since exploitation happens at random in ϵ-greedy, each step of exploitation can only utilize the knowledge from prior exploration steps. In the beginning, the empirical expected reward is far from convergence, thus the probability of selecting the optimal arm is significantly lower, resulting in a higher regret.

As shown above, the primary limitation of the ϵ-greedy strategy is its constant exploration rate, which fails to account for the accumulated knowledge over time. As the agent learns more about the reward distributions of the arms, it continues to explore at a fixed rate, leading to unnecessary regret. A simple way of improving the algorithm is to introduce a decaying ϵ value, such as scaling ϵ

by $\frac{1}{t}$ or $\frac{1}{t^2}$, to dynamically adjust the exploration-exploitation tradeoff based on the agent's increasing understanding of the arms' rewards.

Continuing from the previous example, we examine the epsilon-greedy algorithm where the exploration parameter is set at $\epsilon = 0.5$. At each timestep, there is a 50% chance for the agent to explore and an 50% chance to exploit. During the 50 timesteps of exploration, the agent accumulates a regret of $0.45 \times 50 = 22.5$, where 0.45 is the expected regret for randomly selecting 1 arm. At each timestep of exploitation, the arm with the highest empirical mean is chosen:

$$k = \arg\max_k \hat{\mu_k} = \arg\max_k \frac{1}{|s_k|} \sum_{t \in s_k} r(a_t)$$

where s_k is the set of all previous timesteps in the exploration phase where the k-th slot machine is selected. For example, suppose at $t = 60$ we have explore 30 times, then s_1 represents the timesteps where the first slot machine was pulled among these 30 exploration pulls. Initially, the exploitation phase also results in significant regret, as the agent's knowledge of the true mean reward is still limited. However, as the empirical mean gradually aligns with the true mean reward, the incurred regret with each exploitation step decreases. Nevertheless, a constant 50% of the timesteps are still devoted to exploration, even after the empirical rewards have converged to the true mean.

In this example, bounding the regret is more challenging because the empirical expected reward is updated at each exploration time step, thereby the exploitation steps would not necessarily result in the same arm choice. Consider the reformulated regret:

$$\textbf{Regret}(T) = \sum_{k=1}^{K} \Delta_k \mathbb{E}\left[B_k(T)\right]$$

where $B_k(T)$ is the number of times arm k is pulled over T timesteps. Since the empirical mean reward is only updated during exploration, let $\tilde{B_k}(T)$ denote the number of times arm k is pulled over T **exploration timesteps only**. At each timestep t, we have:

$$\mathbb{E}[\tilde{B_k}(t)] = \epsilon \cdot \frac{1}{k} \cdot t = 0.05 \cdot t \leq 5$$

where we get the inequality because $t \leq T = 100$. For ETC, each arm has been sampled exactly 5 times to obtain the empirical mean reward before the exploitation phase. However, for each exploitation timestep t in the ϵ-greedy strategy, each arm has been sampled less than or equal to 5 times, therefore can deviate more from the true mean reward μ_k. As a result, the probability of selecting a sub-optimal arm to exploit is higher, leading to a higher regret overall. To optimize this approach, we can introduce a time-decaying strategy for ϵ. As confidence in the empirical mean increases, reducing the proportion of timesteps spent exploring and increasing those for exploitation becomes a more efficient strategy.

Upper Confidence Bound (UCB)

Lastly, we discuss the Upper Confidence Bound (UCB) algorithm, rooted in the principle of "optimism in the face of uncertainty". The idea of UCB is when facing the uncertainty regarding the expected reward of an action, we should overestimate its value. This is because there may be unexplored options that are potentially better than the currently known best option. By being optimistic about unexplored options, UCB ensures that all actions get a fair evaluation, thereby minimizing regret over time.

Acting optimistically means that we aim to construct an upper confidence bound $\hat{\mu}_k \geq \mu_k$ for each arm k, such that we are confident the true reward for the arm is upper bounded by $\hat{\mu}_k$. UCB first constructs these upper confidence bounds (similar to the upper bound of the confidence interval) on the expected rewards, then selects the arm with the higher upper bound. Initially, since we have zero knowledge of the reward distribution, the upper confidence bound for each arm is infinity. As we acquire more knowledge, the confidence bounds will eventually be closer to the true mean reward.

Under the 1-subgaussian distribution, we have:

$$\mathbb{P}(\mu_k \geq \hat{\mu}_k + \epsilon) \leq \exp\left(-\frac{B_k(t)\epsilon^2}{2}\right), \epsilon \geq 0$$

where $B_k(t)$ represents the number of time action k has been selected up over t timesteps, as defined above. Solving for the ϵ when equality holds, let δ be the confidence level, we have:

$$\mathbb{P}(\mu_k \geq \hat{\mu}_k + \sqrt{\frac{2\log(1/\delta)}{B_k(t)}}) \leq \delta, \delta \in (0,1)$$

and we construct the UCB for an action:

$$\hat{\mu}'_k := \hat{\mu}_k + \sqrt{\frac{2\log(1/\delta)}{B_k(t)}}$$

Therefore, at each time step, we select the action following the rule:

$$a_t = \arg\max_k \left[\hat{\mu}_k + \sqrt{\frac{2\log 1/\delta}{B_k(t)}}\right]$$

The term $\sqrt{\frac{2\log 1/\delta}{B_k(t)}}$ encourages the agent to select the arm that has not been selected sufficiently many times (i.e. when $B_k(t)$ is small, the exploration bonus is large). The confidence level δ controls how quickly we become certain about the expected reward of action k. Here, a small δ can cause the agent to prematurely stick to a sub-optimal action, while a excessively large δ can waste too much time exploring new actions. For the regret analysis, we let the confidence level decay as a function of time by setting $\delta = 1/t^2$. This is to reflect the behavior that as we obtain more knowledge about each arm, we become more certain about the

expected rewards. Here, we omit the details of deriving the regret for UCB, but we show that the regret is tigher than that of ETC.

$$\textbf{Regret}(T) \leq O(\sqrt{KT \log T})$$

Continuing from the previous example, consider deploying the UCB strategy. In the beginning, since no machine has been tried, the UCB is infinite for all machines (since $B_k(0) = 0$ for any k), ensuring that each machine is tried at least once. This helps to initialize the empirical means $\hat{\mu_k}$ for each machine. As each machine is tried over time, the algorithm will calculate a confidence bound for each machine's reward based on its empirical mean and the number of times it has been tried. The confidence level δ will impact the width of the confidence bound. In the beginning, when δ is large (recall that $\delta = 1/t^2$), the confidence intervals are wide, reflecting greater uncertainty about the true mean reward of each machine. At each timestep, the machine with the highest upper confidence bound is selected. This balances the exploration of machines with fewer trials (thereby wider confidence intervals) and the exploitation of machines with higher empirical means. Over time, as machines are tried more, the empirical means will converge to the true mean reward, and the confidence bounds will become narrower as δ decays with time. This will lead to less exploration and more exploitation of the best performing machine. Using the UCB strategy with the specific slot machine probabilities provided, the regret would be relatively low compared to strategies that do not adapt over time, because the UCB strategy effectively identifies and exploits the best machine fairly quickly. Since the UCB balances exploration and exploitation, it's likely to perform well in this scenario, finding the optimal arm quickly and exploiting it most of the time, thus keeping the regret low, especially after the initial exploration phase.

The Multi-Armed Bandit problem is a classic introduction to the exploration-exploitation tradeoff in many decision-making scenarios. We discussed various strategies, including the naive approach of random selection, Epsilon-Greedy, Explore-Then-Commit, and Upper Confidence Bound, each with its own way of addressing the challenge.

Sequential Decision Making

Reinforcement Learning (RL) is a branch of machine learning that focuses on sequential decision-making. Through iterative interactions with the environment, agents learn to make a series of decisions to maximize cumulative rewards. This process of making decisions over multiple timesteps is modeled within the framework of a Markov Decision Process (MDP), which includes the following components:

- State $s \in \mathcal{S}$ and action $a \in \mathcal{A}$: a state includes the current information regarding the environment. It acts as a snapshot of the system at timestep t,

providing the context in which decisions are made. a_t denotes the action chosen by the agent as time t.

- Transition Function $P(s_{t+1} = s' \mid s_t = s, a_t = a)$: This describes the dynamics of the environment. It indicates the probability that action a is state s at time t will lead to state s' at time $t+1$. Under the Markov assumption, we assume that the probability is only dependent on the previous state instead of the entire previous history of states.
- Reward $r(s, a, s')$ is the reward for transitioning from state s to state s' by taking the action a. In some other settings, the reward is formatted as $r(s, a)$ or $r(s, s')$.

In most RL frameworks, the interaction between the agent and the environment is described as a MDP. Aside from the MDP, RL includes the following components:

- Timestep t: In RL, actions are made sequentially across discrete moments, called timesteps. Each timestep signifies a unique decision-making point.
- Policy π: at each timestep t, the agent takes action $a_t = \pi(s_{1:t}, a_{1:t-1})$, which can be either deterministic or stochastic. It represents the agent's strategy, specifying which action the agent should take given its past history. The policy is typically parameterized by a neural network with parameters θ.
- Episode: an episode describes a full cycle of the agent's interactions with the environment, starting from an initial state and to a terminal state or until a specific condition is met.
- Trajectory τ: A trajectory is a sequence of states, actions, and rewards experienced by an agent during its interaction with the environment. Given a policy, we can "roll out" the policy to obtain a trajectory:

  $$\tau = \{(s_1, a_1, r_1), (s_2, a_2 r_2), \ldots, (s_T, a_T, r_T)\}$$

  We can also define the reward for the entire trajectory as:

  $$R(\tau) = \sum_{t=1}^{T} r_t = \sum_{t=1}^{T} r(s_t, a_t)$$

- Time Horizon T: The horizon defines the number of timesteps in consideration. A finite horizon implies a predetermined number of timesteps, while an infinite horizon considers the scenario in which the agent can interact with the environment forever.

In a process of sequential decision making, for a specific episode, the agent starts with initial state s_1, takes actions $a_{1:t}$, observes rewards $r(s_{1:t}, a_{1:t})$, and transitions to the next state s_{t+1} for each $t > 1$. At the end of the time horizon, the agent starts with a new episode and resets the state. The goal of the agent is to find the policy π that maximizes the expected cumulative reward.

Policy Gradient

We introduce a family of policy gradient algorithms to solve the sequential decision making problems, including reinforce, discount, baselines, and actor critic. The idea behind policy gradient is to optimize the policy by determining the gradient of the expected reward, enabling us to iteratively improve the policy. We can formulate the problem of finding the policy that maximizes the expected return of trajectories:

$$\max_{\theta} J(\theta), \quad J(\theta) = \mathbb{E}_{\pi_\theta}\left[R(\tau)\right]$$

where $R(\tau)$ denotes that reward of the trajectory τ, π_θ is the policy parameterized by θ. In practice, the policy is typically parameterized by a neural network, such that θ represents the network's parameters such as weights and biases.

Reinforce

Let $p_\theta(\tau)$ be the probability of a trajectory τ being sampled. We compute its derivative with respect to θ:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\pi_\theta}[R(\tau)] \\
&= \int_\tau \nabla_\theta p_\theta(\tau) R(\tau) d\tau \\
&= \int_\tau p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} R(\tau) d\tau \\
&= \int_\tau p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)) R(\tau) d\tau \qquad (\text{by } \frac{d \log x}{dx} = \frac{1}{x}) \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log(p_\theta(\tau)) R(\tau)]
\end{aligned}$$

The expression captures the idea of adjusting the probabilities of actions in the direction that maximizes expected rewards, as this gradient indicates how the performance function changes with a small change in the policy parameters θ. By climbing this gradient, we iteratively refine our policy to collect higher rewards, thus achieving the essence of policy gradient methods. This method is also referred as the vanilla reinforcement algorithm, which increases the likelihood of action sequences that yield higher returns at every update. The primary steps in this algorithm can be summarized as:

1. Obtain a trajectory rollout τ using the current policy π_θ.
2. Determine the gradient of the log-likelihood for all actions $g = \nabla_\theta \log \pi_\theta(\cdot|\cdot)$.
3. Multiply the obtained gradient by the corresponding returns, $g \cdot R(\tau)$. This ensures that actions leading to better outcomes have a more significant impact on the update.
4. Update the policy parameters by weighted gradients, $\nabla\theta = \alpha \cdot g \cdot R(\tau)$, where α is the learning rate.

The idea of policy gradient is to update the policy to increase the probability of good decisions, and decrease the probability of poor decisions. However, the

high variance in sampled trajectory returns $R(\tau_{t:T})$ makes it difficult to assign credits to each decision. Next, we introduce more advanced algorithms to address some of the issues in the most vanilla version of reinforce.

Discount

A common approach is to reduce the weight given to rewards expected in the distant future. In the reward function, a reward r_i that is further in the future gets a higher discount, represented by γ^{i-t}, where $0 < \gamma < 1$. This means that rewards expected later have a decreased impact, leading to reduced variance in the expected future rewards. However, this leads to a trade-off between bias and variance, where a very low value of γ can make the agent learn quickly due to the reduced variance, but also encourages the agent to prioritize short-term rewards.

Baselines

A baseline is essentially a reference value used to reduce the variance of the policy gradient estimate without introducing bias. It helps to differentiate trajectories that lead to rewards better than average. The primary purpose of a baseline is to amplify the probabilities of trajectories with rewards better than the baseline and dampen those with rewards worse than the baseline, which makes learning more stable and potentially faster. The term "advantage" quantifies the gap between the return from a trajectory and the average trajectory return $b(s_t) = \mathbb{E}\left[R^\gamma(\tau_{t:T})\right]$. In practice, it is common to use weighted average or the value function to obtain the baseline. Using advantage, we can rewrite policy gradient as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\left(R(t) - b(s_t)\right)\right]$$

Baselines reduce variance due to the following inequality:

$$\mathrm{Var}[\nabla_\theta \log p_\theta(\tau)\left(R^\gamma(\tau) - b(\tau)\right)] \leq \mathrm{Var}[\nabla_\theta \log(p_\theta(\tau))R^\gamma(\tau)]$$

By differentiating between above-average and below-average trajectories, the baselines method offers a systematic approach to guide the agent's learning process.

Actor-Critic

We introduce two actor-critic methods to reduce the variance of policy gradient. These methods leverage the value function to estimate trajectory returns instead of relying on Monte-Carlo approaches. First, the advantage actor-critic (A2C) model consists of an actor that is responsible for decision making, and a critic which ensures those decisions are heading in the right direction. Consider an analogy of a soccer match: the actor resembles the players making decisions, while the critic is the coach guiding their performance. The actor's decisions, governed by the policy, is evaluated by the critic, similar to a coach giving feedback to their strategy. Here, the critic uses the advantage value to compare the

action to the expected return value, such that a positive value increases the likelihood of such actions being sampled in the future, while a negative value reduces the probability of repeating the same actions. Through iterative training, the network is updated to make better decisions overtime.

Generalized Advantage Estimation (GAE) balances the bias and variance of advantage estimates by adeptly mixing Monte Carlo estimates with approximated estimations. The idea is to incorporate multiple steps of rewards from the environment to control the variance in the value prediction. For example, consider a single-step advantage estimation, which combines an immediate reward with a value prediction:

$$A^1(s_t, a_t) = r(s_t, a_t) + \gamma V^\pi(s_{t+1}) - V\pi(s_t)$$

By extending this approach, we can define an n-step advantage that includes more terms from the reward sequence, thereby refining our estimation. However, GAE does not stick to a specific n. Instead, it proposes a flexible approach by taking a weighted average of all n-step advantages using a parameter λ, which allows GAE to control the trade-off between bias and variance. Specifically, when $\lambda = 0$, GAE utilizes only the 1-step advantage, making it biased but with low variance. When $\lambda = 1$, GAE fully employs the MC estimation, which is unbiased but might exhibit high variance. Using this approach, the agent benefits from both immediate and future rewards, resulting in a more robust learning process.

Reward Shaping

In reinforcement learning, the reward function is the compass that guides an agent's learning journey. It is the primary signal that dictates what behaviors are desirable and which ones should be avoided. A well-crafted reward function can lead an agent swiftly to optimal or near-optimal policies, making it perform tasks effectively and efficiently. However, curating a good reward function is easier said than done. The challenge arises due to the complexity behaviors an agent can exhibit in complex environments. While we might craft a reward function intending to promote certain actions, an agent might find unintended shortcuts to maximize its cumulative reward. This phenomenon is often referred to as "reward hacking", in which agents exploit the reward function. For instance, an agent tasked with stacking blocks might find it more rewarding to scatter blocks around and stack only a subset, if that is what the reward function unintentionally incentivizes. In addition, slight modifications in reward functions can lead to vastly different agent behaviors. Over-specifying a reward can stifle an agent's ability to explore and find innovative solutions, while under-specifying can leave it wandering aimlessly.

Sparse vs. Dense Reward

Given the challenges in crafting an appropriate reward function, it becomes crucial to understand reward shaping, a technique to guide the agent's learning in

a more directed manner. Here, we use an example to demonstrate the process and challenges in designing a good reward function. Consider an agent moving in a 2-dimensional space from its initial position to a specific target location. The environment is designed with a discrete number of timesteps, such as 100 timesteps. The goal of the agent is to maximize the cumulative reward within these timesteps. To simplify the problem, we do not discount future rewards in this example. First, we consider a sparse reward function, in which the agent is given a positive reward only when it successfully reaches the target.

$$R(s) = \begin{cases} 1 & \text{if } \mathbf{x}_{\text{agent}} = \mathbf{x}_{\text{target}} \\ 0 & \text{otherwise} \end{cases}$$

With this reward structure, the agent receives no intermediate feedback on its performance until it reaches the goal. This can make the learning process slow and inefficient, as the agent has very little information to learn from. Without cues to guide its direction, the agent must explore the environment extensively and may only stumble upon the target by chance. To address this issue, we can design a dense reward function that provides more immediate feedback. At each timestep, the agent receives a reward which is equivalent to the negative squared distance between the agent and the target.

$$R(s) = -\|\mathbf{x}_{\text{agent}} - \mathbf{x}_{\text{target}}\|^2$$

At every timestep, the agent receives a signal that indicates how close or far it is from the target. The negative squared distance to the target serves as a smoother landscape for the agent to navigate, providing a gradient of rewards that guide the agent in the right direction. Consider the agent's experience with the dense reward function as walking through a terrain with a gradient that slopes upward as it moves away from the target and downward as it approaches the target. This gradient consistently tells the agent that descending will lead it closer to the goal. As a result, the agent can take informed steps towards the target, with each action adjusted based on the immediate feedback from the environment. The continuous feedback loop allows the agent to learn the correct direction to move towards the goal more efficiently. As a result, this function incentivizes the agent to make progress towards the target at each timestep.

Introducing Obstacle

Now that we are able to reach the target, let's introduce a more challenging problem. Consider adding an obstacle between the agent initial position and the target. How would the agent behave using the same reward function? Since the reward is purely based on the distance between the agent and the target, the agent is incentivized to minimize this distance at each step. Initially, this will guide the agent directly towards the target, as before. However, as the agent approaches the obstacle, it reaches a point where moving directly towards the

target is no longer possible, thereby not increasing the reward. When humans encounter a similar obstacle, we instinctively navigate around it, understanding that circumvention is necessary to reach the target. However, the reward function does not understand that an alternative path, while being initially longer, can eventually lead to the goal. The agent has no incentive to explore away from the direct path to the target, and gets stuck at a local minimum.

Collision Penalty

One thing we can do is to add a penalty term on the agent colliding with the obstacle. In the physical world, this is a common term to avoid damaging surrounding equipments. The dense reward function can be modified as:

$$R(s) = -\|\mathbf{x}_{\text{agent}} - \mathbf{x}_{\text{target}}\|^2 - P \cdot \mathbb{1}_{d(\text{agent,obstacle}) < \delta}$$

where $\mathbb{1}$ is the indicator function, P is the hyperparameter controlling how much penalty the agent receives when it is too close to the obstacle, and δ is the distance threshold. While the addition of the collision penalty makes the area around the obstacle less attractive, the agent lacks a reward structure that encourages the exploration of longer or more complex paths that would lead it around the obstacle to the target. Without an incentive to explore such trajectories, the agent can still end up in a situation where it stays in a safe zone, not close enough to incur the penalty but also not making progress towards the target—effectively a local minimum in terms of reward optimization. This illustrates the intricate balance required in reward shaping to both discourage unwanted behaviors and encourage the necessary exploration to achieve a goal.

Introducing Sub-goals

To overcome the local minimum problem and encourage the agent to navigate around the obstacle, we can implement a subgoal-based approach to reward shaping. This method involves defining intermediate targets that guide the agent on a path that circumvents the obstacle, thus leading it toward the final target. In our scenario, we place subgoals at locations that would naturally lead to the agent exploring the desired path. The agent's reward function is initially focused on reaching the nearest subgoal. Once the agent reaches a subgoal, the reward function is updated to focus on the next subgoal or the final target if all subgoals have been reached. The modified reward function with subgoals can be represented as follows:

$$R(s) = \begin{cases} -\|\mathbf{x}_{\text{agent}} - \mathbf{x}_{\text{subgoal}}\|^2 & \text{subgoal has not been reached} \\ -\|\mathbf{x}_{\text{agent}} - \mathbf{x}_{\text{target}}\|^2 & \text{otherwise} \end{cases}$$

To determine whether a subgoal has been reached, we can introduce a threshold distance. For each episode, if the agent comes within this threshold of a subgoal, we consider the subgoal to have been reached, and the reward function

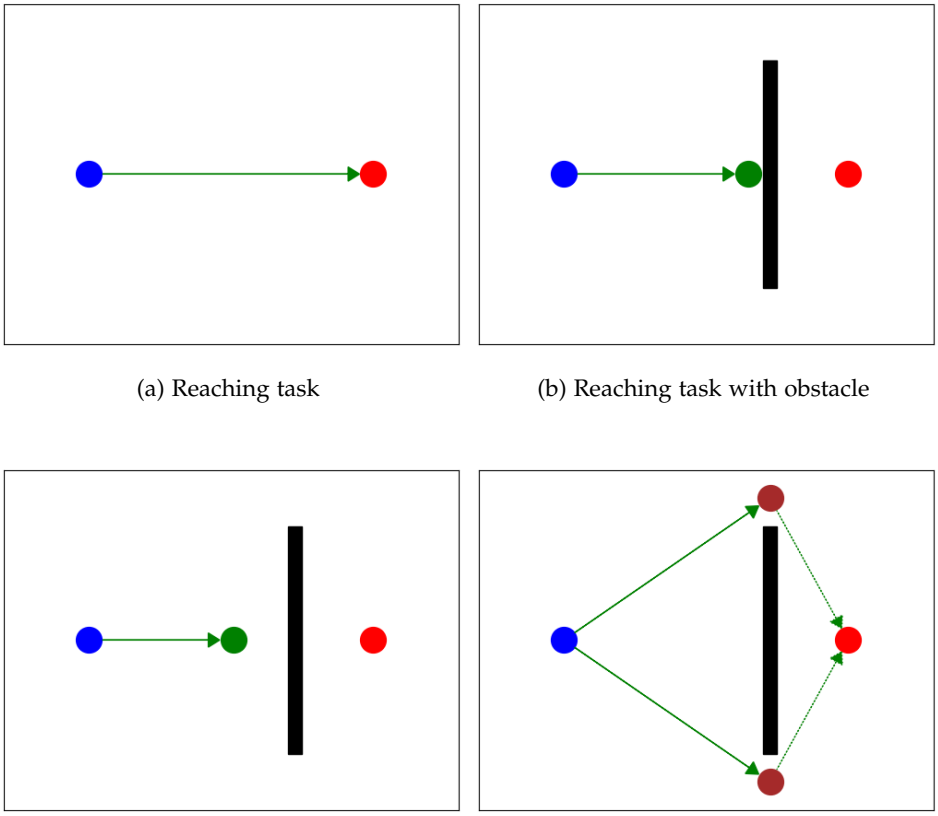
is updated accordingly. By providing a clear gradient of rewards towards each subgoal and eventually the final target, the agent is encouraged to learn a trajectory that circumvents the obstacle. This method effectively helps the agent to explore and learn the necessary detour, much like a human intuitively navigating around barriers to reach a destination. This approach combines the benefits of dense rewards with the strategic guidance needed to solve more complex spatial navigation tasks.

Despite its success, the introduction of subgoals as an artificial guide in reinforcement learning presents certain limitations. If the environment were to change or a new obstacle were introduced, the agent's ability to navigate effectively would be compromised unless new subgoals were also created. This lack of generalizability indicates that the agent does not truly understand the task: it does not learn to navigate but simply follow a set path, which is a contrast to genuine comprehension and adaptability. The reliance on explicit subgoals illustrates a shallow form of task completion that contrasts with human cognitive processes. Humans understand the goal in a holistic sense and can adapt their strategies to changing environments. Consequently, its capabilities are constrained to the limitations of the provided subgoal structure, and it lacks the ability to infer new strategies in the face of novel challenges.

Conclusion

In this example, we considered the task of an agent reaching a target. We first explored sparse verse dense reward functions, and examined how each influences the agent's learning. The introduction of an obstacle into the environment highlighted the limitations of the dense reward function, prompting the addition of an obstacle penalty to discourage collisions and subgoals to guide the agent to a longer path. However, the solution involving subgoal lacks generalizability and the placement of subgoals hinders the agent's ability to truly understand the task and environment. In practice, researchers typically perform extensive reward shaping that guides the agent towards comprehensive learning and appropriate behavior without overfitting to overly specific scenarios. To this day, the design of an ideal reward function remains a challenging and tedious task in the field of reinforcement learning.

Visualization



(a) Reaching task

(b) Reaching task with obstacle

(c) Adding collision penalty

(d) Adding subgoals

Figure 1: Visualization of reaching example.

Figure 1: (a) The agent (blue) seeks to reach the target (red). Using the dense reward function, the agent can reach the target following the optimal path (green). (b) Suppose we place an obstacle (black) between the agent and the target. The dense reward still guides the agent to minimize the distance to the target until it is no longer possible. The agent is not incentivized to circumvent the obstacle. (c) By adding a penalty for colliding with the obstacle, the agent now avoids the being too close to the obstacle. However, the agent still suffers from being stuck at local minimum, thus minimizes the distance to the target while maintains a safe distance to the obstacle. (d) We introduce subgoals (brown) to first guide the agent to reach one of the subgoals before proceeding to the target. While this can effectively guide the agent to reach the target, it is not generalizable as it does not genuinely understand the task and environment.

Learning from Demonstrations

Learning from Demonstration (LfD) is a powerful tool in the field of reinforcement learning and robotics, which enables robots and agents to acquire new skills or behaviors by observing and imitating expert demonstrations. This approach is inspired by the way humans learn many tasks, from simple actions like tying shoelaces to complex activities like driving a car, primarily through observation and mimicry. LfD leverages this intuitive strategy by allowing machines to learn from the vast and rich expertise that humans can provide through demonstration, without the need for explicit programming or exhaustive exploration of the task space. One of the primary reasons LfD is so powerful is its ability to bypass the intricate and often infeasible process of manually coding behaviors or defining reward functions for every possible scenario an agent might encounter. Instead, the agent can learn directly from examples, which is especially advantageous in tasks that are difficult to specify in a traditional programming sense. For example, teaching a robot to cook a meal or to perform a dance involves a sequence of intricate and nuanced actions that are challenging to define explicitly but can be more easily demonstrated.

Moreover, LfD can be more efficient and practical than other learning methods, such as reinforcement learning, in real-world applications. Reinforcement learning often requires a large number of trials and errors to learn a task, which can be time-consuming, costly, and even unsafe in physical environments. In contrast, LfD can significantly reduce the learning time and resources by leveraging expert knowledge to provide a more direct and focused learning process. Here, we introduce Behavior Cloning (BC) and Dataset Aggregation (DAgger), both of which provide structured ways to translate demonstrations into actionable knowledge for agents, each with its own set of advantages and challenges.

Behavior Cloning

Behavior Cloning (BC) is a straightforward and intuitive approach within the Learning from Demonstration paradigm. At its core, BC treats the task of learning from demonstrations as a supervised learning problem, drawing parallels with how we might teach a machine to recognize images or understand natural language. The fundamental idea is to create a direct mapping from the observed states or contexts to the actions that an expert demonstrator would take in those states. Consider an expert demonstrator providing a sequence of state-action pairs $(s_1, a_1), (s_2, a_2), \ldots, (s_N, a_N)$, where s_i represents the i-th state observed by the demonstrator, and a_i be the corresponding action taken by the expert in that state. Behavior cloning seeks to learn the function $f : S \to A$, where S is the space of all possible states and A is the space of all possible actions, such that for a new unseen state s_{new}, the function $f(s_{new})$ predicts and action a_{new} that the expert would have taken. In practice, this function f is often represented by a parameterized model such as a neural network, where the parameters θ are learned by minimizing a loss function. The loss function quantifies the difference

between the predicted action $f(s_i; \theta)$ and the true action a_i across all demonstrations. A common choice for the loss function is the mean squared error:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|f(s_i; l\theta) - a_i\|^2$$

Behavior cloning translates the learning problem into a supervised learning framework, in which researchers can leverage existing tools in the machine learning domain. Nevertheless, behavior cloning also struggles to generalize to unseen states.

Dataset Aggregation

Consider training a self-driving car to maintain at the center position of a lane using behavior cloning, in which we are given a dataset that captures an expert's driving behavior. During the learning process, the agent's policy π_θ cannot perform at the same level as the expert's. As a results, π_θ deviates from the center of the road, which has a compounding effect on future timesteps. However, the training dataset is limited to the expert's demonstrations, which never deviates from the center of the road and thereby does not include information on how to recover from such deviations. The distribution shift in inputs is known as *covariate shift*, which hinders the performance of the agent at deployment time.

Dataset aggregation (DAgger) addresses this issue by adopting an iterative training procedure. Instead of training the agent on a static dataset of expert demonstrations, DAgger collects training data in a series of iterations. In each iteration, the agent interacts with the environment using its current policy.The expert then provides the correct actions for these encountered states, effectively labeling them. These new state-action pairs are added to the training dataset, and the agent's policy updated based on this augmented dataset. The algorithm can be formalized in the following steps:

1. Initialize the training dataset as an empty set $\mathcal{D} = \{\}$.
2. At each iteration:
   (a) Obtain a rollout using the current policy π_θ.
   (b) Query the expert policy to label the states in the rollout with actions a.
   (c) Augment the dataset with these new state-action pairs $D \leftarrow D \cup \{(s, a)\}$.
   (d) Update the current policy $\theta \leftarrow \theta + \beta \nabla_\theta J(\theta)$, where β is the step size, and $J(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}\big[\log \pi_\theta(a|s)\big]$

By incorporating the policy's own mistakes into the training data, DAgger effectively teaches the agent how to recover from errors and handle a wider variety of states, leading to better generalization and more robust performance. While DAgger significantly improves upon the limitations of basic behavior cloning, the requirement for ongoing expert intervention to label new data can be resource-intensive, and in some scenarios, not feasible to obtain expert labels for every new state the agent encounters. Nonetheless, DAgger remains a powerful tool for refining policies in complex and dynamic environments.

Domain Randomization

In our previous discussions, we discussed learning from demonstration and algorithms like behavior cloning and DAgger, which rely on expert demonstrations to train policies. These methods can be highly effective when expert demonstrations are readily available. However, what if our training ground is limited to a simulated environment? How can we ensure that the learned policy remains effective when transitioning from the simplified rules and conditions of a simulation to the complex and often unpredictable physics of the real-world environment?

Domain randomization is a technique used to enhance the generalization of learned policies, particularly in scenarios where a model trained in simulation is deployed in the real world. This approach acknowledges the inherent discrepancy between simulated and real-world dynamics, often referred to as the *reality gap*. To bridge this gap, domain randomization varies the parameters of the simulation during the training process. This randomization could encompass a wide array of factors, including physical properties like friction and mass, visual elements such as textures and lighting, or even sensor noise. The idea behind domain randomization is to expose the learning algorithm to a broad spectrum of possible environments within the simulation. The model π_θ is trained across a distribution of varying simulation parameters instead of a fixed set of conditions. The objective is to maximize the expected performance $J(\pi_\theta)$ over the distribution of $P(z)$ of possible environment z:

$$\max_\theta \mathbb{E}_{z \sim P(z)} \left[J(\pi_\theta) \right]$$

By doing so, the policy π_θ is encouraged to learn behaviors that are not just effective in a narrow setting but are robust to variations and uncertainties that it might encounter after being deployed in the real world. This approach introduces a tradeoff between performance and robustness. In aiming for robustness, the policy may adopt more conservative strategies to ensure reliability across a range of conditions, but not achieve the optimal performance in any specific one. For example, a policy trained to navigate a robot on various surfaces, from slippery ice to rough terrain, might adopt a universally cautious gait that ensures stability across all surfaces but does not allow the robot to move as quickly as it could on a stable, high-friction surface.

The performance-robustness tradeoff implies that while domain randomization makes it more feasible to transfer models trained in simulations to real-world environments, it may not always lead to the most efficient behaviors in individual conditions. The key is to find a balance that maximizes the policy's effectiveness in the most critical or likely real-world scenarios while maintaining sufficient adaptability to handle unexpected conditions. This balance is central to the design of robust, real-world systems that can operate safely and effectively in the face of the complex, unpredictable variations they will inevitably encounter outside of the controlled training environment.

System Identification

When facing uncertainties of real-world parameters, it is essential to develop robust and versatile policies in reinforcement learning. While domain randomization improves the policy's generalizability by exposing it to a spectrum of simulated conditions, system identification finetunes this adaptability, enabling the policy to adjust to a specific condition that is closer to the real-world environment. The idea is that while the exact system parameters (such as mass, friction or drag) may be elusive, they can be inferred from the system's responses to certain inputs, allowing us to approximate these parameters in the real environment. Consider utilizing reinforcement learning in robotics, in which a robot must navigate different terrains. In simulation, the robot is trained on various ground types with randomized parameters to prepare it for diverse real-world conditions. However, once deployed, the robot must quickly discern the specific characteristics of its new environment. This process of detecting and adapting to the unique properties of an environment is system identification. It is a form of learning not just to act, but to perceive and interpret the underlying dynamics of the surroundings.

The method involves observing the outcomes of actions taken in the environment and estimating the parameters that could have generated those outcomes. For example, a policy may infer the level of friction of a surface based on the distance traveled after a short duration. By doing so, the robot can adjust its actions accordingly to maintain balance and motion efficacy, which is particularly vital when transitioning from the controlled conditions of a simulator to the unpredictable nature of real-world physics. More formally, the process can be represented s an optimization problem where a model class $\mathcal{F}$ is employed to minimize the discrepancy between the observed outcomes $h = [o_1, o_2, \ldots, o_N]$ and the predicted outcomes based on the estimated parameters $\hat{z}$:

$$f^* \in \arg\min_{f \in \mathcal{F}} \mathbb{E}_h[(z - f(h))^2]$$

where z represents the true system parameters. The optimized function f^* can then serve as a guide for the policy to perform actions that are tailored to the inferred parameters of the environment. This approach enables the policy to dynamically adjust its behavior in response to the immediate feedback from the environment, fostering a form of adaptive intelligence that is critical for the agent to operate in the diverse tapestry of the real world.