Problem 1: linear representation approximation of mdps

Problem outline: This question states that a linear MDP (for example, see [1, Jin et al., 2020]) satisfies

$$\begin{aligned} P(s'|s,a) &= \langle \phi(s,a), f(s') \rangle, \\ r(s,a) &= \langle \phi(s,a), \theta \rangle. \end{aligned} \tag{1}$$

for some functions ϕ, f and a vector θ. In the following questions, we consider maximizing the expected cumulative undiscounted reward over a finite horizon of T. We assume finite state space S and finite action space A. At each time period $t \in \{1, \dots, T\}$, a memoryless policy $\pi = (\pi_1, \dots, \pi_T)$ observes the state s_t, takes the action $a_t = \pi_t(s_t)$, and receives a reward $r(s_t, a_t)$. Define for any memoryless policy π, any $s \in S$ and $a \in A$,

$$\begin{aligned} V_t^\pi(s) &= \mathbb{E}\left[\sum_{t'=t}^{T} r(s_{t'}, \pi_{t'}(s_{t'})) \middle| s_t = s \right], \\ Q_t^\pi(s,a) &= r(s,a) + \mathbb{E}[V_{t+1}^\pi(s_{t+1}) | s_t = s, a_t = a]. \end{aligned} \tag{2}$$

solution q1 part 1

Problem: Show that a tabular MDP is a linear MDP. Specifically, what are $\phi(s,a)$, $f(s')$ and θ?

Approach: The natural representation for an MDP in a tabular form is where each state-action pair has its own unique representation. A one-hot encoding serves this purpose as each unique state-action combination will have its own unique encoding. This representation captures the essence of a tabular MDP by translating it to a linear format. In essence, we are leveraging the binary structure of one-hot vectors to imitate the table rows and columns of the tabular MDP.

Solution:

- *One-hot encoding for state-action pairs:* Given our MDP with N states and M actions, the one-hot encoding of a state-action pair (s,a) is a vector $\phi(s,a) \in \mathbb{R}^{NM}$. This linear representation translates the two-dimensional nature of state-action pairs in the table into a one-dimensional vector format. The position of the '1' in this encoding provides the exact mapping to the original state and action from our MDP table. Consider the state indexed by i (where i ranges from 1 to N) and the action indexed by j (where j ranges from 1 to M). To compute the position of '1' in the one-hot encoding, think of the matrix layout of the tabular MDP: for every row (state) completed, M columns (actions) have been covered. When the state is i, $M \times (i-1)$ positions are already filled. Now, for the exact action within that state, we add the action index j. This logic is captured in the formula:

  $$\text{Position} = (i-1)M + j$$

  Hence, $\phi(s,a)$ will have a 1 at the position $(i-1)M + j$ and 0 everywhere else. Formally:

  $$\phi(s,a)_{(i-1)M+j} = \begin{cases} 1 & \text{if } i = s \text{ and } j = a \\ 0 & \text{otherwise} \end{cases}$$

Here, $i \in \{1, \ldots, N\}$ represents states, and $j \in \{1, \ldots, M\}$ represents actions.

- *Linear representation of transition probabilities:* For each state s', $f(s')$ will be a vector showing the probabilities of transitioning to s' from each state-action pair.

$$f(s') = \begin{bmatrix} P(s'|s_1, a_1) \\ P(s'|s_1, a_2) \\ \vdots \\ P(s'|s_N, a_M) \end{bmatrix}$$

- *Linear representation of rewards:* The vector θ gives the rewards for each state-action pair.

$$\theta = \begin{bmatrix} r(s_1, a_1) \\ r(s_1, a_2) \\ \vdots \\ r(s_N, a_M) \end{bmatrix}$$

With this representation, it becomes evident that the tabular MDP's transition probabilities and rewards can be expressed in the linear form defined by the equations for $P(s'|s, a)$ and $r(s, a)$ using $\phi(s, a)$, $f(s')$, and θ. □

SOLUTION Q1 PART 2

Problem: Given the linear representations of the MDP as shown in part 1, demonstrate that for any memoryless policy π, there exists a weight vector w_t^π such that for each $(s, a) \in \mathcal{S} \times \mathcal{A}$, $Q_t^\pi(s, a) = \langle \phi(s, a), w_t^\pi \rangle$.

Approach: To demonstrate the existence of the weight vector w_t^π, we will:

- Reiterate the linear representations from Part 1.
- Substitute the linear representations into the Bellman expectation equation.
- Express the terms in a form showing the existence of w_t^π.
- Discuss the intuition behind the weight vector representation and its recursive nature.

Solution:

- *Recalling Linear Representations from Part 1:*

$$P(s'|s, a) = \langle \phi(s, a), f(s') \rangle$$

$$r(s, a) = \langle \phi(s, a), \theta \rangle$$

- *Substituting into the Bellman Expectation Equation:*

$$Q_t^\pi(s, a) = \langle \phi(s, a), \theta \rangle + \sum_{s'} \langle \phi(s, a), f(s') \rangle V_{t+1}^\pi(s')$$

- *Expanding using the definition of $V^\pi_{t+1}(s')$:*
$$Q^\pi_t(s,a) = \langle \phi(s,a), \theta \rangle + \sum_{s'} \langle \phi(s,a), f(s') \rangle \mathbb{E}\left[\sum_{t'=t+1}^{T} r(s_{t'}, \pi_{t'}(s_{t'})) \middle| s_{t+1} = s' \right]$$

- *Assumption on Linear Representation of Future Q-value:* Assuming that the Q-function at $t+1$ is also linearly represented as:
$$Q^\pi_{t+1}(s', a') = \langle \phi(s', a'), w^\pi_{t+1} \rangle$$
We can then expand our previous expression as:
$$Q^\pi_t(s,a) = \langle \phi(s,a), \theta \rangle + \sum_{s'} \langle \phi(s,a), f(s') \rangle \langle \phi(s', \pi_{t+1}(s')), w^\pi_{t+1} \rangle$$

- *Formulating the weight vector w^π_t:* To express $Q^\pi_t(s,a)$ in the desired linear form, the weight vector w^π_t is given by:
$$w^\pi_t = \theta + \sum_{s'} f(s') \otimes w^\pi_{t+1}$$
where $\otimes$ denotes the tensor or Kronecker product.

The recursive representation of w^π_t is intuitive as it captures the notion of immediate rewards as well as the expected rewards from future states. It relies on the assumption that future Q-values are linearly representable, building upon the weight vectors of subsequent time steps. It is worth noting that the assumption of linear representation holds well for tabular MDPs, but might require scrutiny for more complex scenarios or non-tabular structures. In conclusion, the existence of the weight vector w^π_t is established for the linear MDP, satisfying the stipulated problem conditions. □

SOLUTION Q1 PART 3

Problem: Given the constraints on the reward function and the total variation distance between the true transition probabilities and their linear approximation, show that for any memoryless policy π, there exists a weight vector w^π_t such that the deviation of the approximated action-value function from its linear representation is bounded by $2TC$.

Approach:

- Decompose the deviation using the Bellman equation.
- Utilize the given constraints to derive the required bound.

Solution:

- *Decomposing using the Bellman Equation:* Starting with the Bellman equation for the action-value function $Q^\pi_t(s,a)$:
$$Q^\pi_t(s,a) = r(s,a) + \sum_{s'} P(s'|s,a) Q^\pi_{t+1}(s', \pi(s')). \tag{3}$$

For the linear approximations of the reward function and transition dynamics, we can rewrite this as:

$$\langle \phi(s,a), w_t^\pi \rangle = \langle \phi(s,a), \theta \rangle + \sum_{s'} \langle \phi(s,a), f(s') \rangle \langle \phi(s', \pi(s')), w_{t+1}^\pi \rangle. \tag{4}$$

- *Calculating Deviation:* The deviation is defined as:

  $$\delta_t(s,a) = |Q_t^\pi(s,a) - \langle \phi(s,a), w_t^\pi \rangle|. \tag{5}$$

  Expanding this deviation:

  $$\delta_t(s,a) \leq |r(s,a) - \langle \phi(s,a), \theta \rangle| + \sum_{s'} |P(s'|s,a) - \langle \phi(s,a), f(s') \rangle| \times \delta_{t+1}(s', \pi(s')).$$

  Using the given constraints, this becomes:

  $$\delta_t(s,a) \leq C + C \sum_{s'} \delta_{t+1}(s', \pi(s')).$$

  Recursively applying this for each time step from T down to 0, we get:

  $$\begin{aligned} \delta_T(s,a) &\leq 0 \\ \delta_{T-1}(s,a) &\leq 2C \\ \delta_{T-2}(s,a) &\leq 3C \\ &\vdots \\ \delta_0(s,a) &\leq TC. \end{aligned}$$

- *Overestimation and Underestimation:* An overestimation occurs when the linear approximation predicts a value greater than the true action-value, while an underestimation is the opposite.

  Given our deviation calculation, we already have the bounds for the overestimation:

  $$Q_t^\pi(s,a) \leq \langle \phi(s,a), w_t^\pi \rangle + \delta_t(s,a).$$

  Similarly, for the underestimation:

  $$Q_t^\pi(s,a) \geq \langle \phi(s,a), w_t^\pi \rangle - \delta_t(s,a).$$

  Combining the bounds from both overestimation and underestimation:

  $$\langle \phi(s,a), w_t^\pi \rangle - \delta_t(s,a) \leq Q_t^\pi(s,a) \leq \langle \phi(s,a), w_t^\pi \rangle + \delta_t(s,a).$$

  Given our earlier results, at $t = 0$, $\delta_0(s,a) \leq TC$. This means the approximated value can deviate by at most TC either upwards or downwards, giving a total deviation range of $2TC$.

In conclusion, under the provided constraints on approximation errors for rewards and transition dynamics, the deviation of the approximated action-value function from its true value is bounded by $2TC$ for any memoryless policy π. □

PROBLEM 2: ACTOR-CRITIC METHODS

For convenience, we define the discounted total reward of a trajectory τ as:

$$R(\tau) := \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t),$$

where s_t, a_t are the state-action pairs in τ. Observe that:

$$V_\mu(\pi_\theta) = \mathbb{E}_{\tau \sim P_\mu^{\pi_\theta}}[R(\tau)],$$

For some initial state distribution, $\gamma = 1$ and a finite-horizon ending at T, we have seen in class that:

$$\nabla_\theta V_\mu(\pi_\theta) = \mathbb{E}_{\tau \sim P_\mu^{\pi_\theta}}\left[R(\tau) \sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t)\right].$$

In this problem, we will show that a similar expression can be derived for the gradient of the value function in terms of action-value functions Q. This introduction of the critic Q lends itself to actor-critic methods, both basic and advanced. Assume that policies are stochastic.

SOLUTION Q2 PART 1

Problem: Derive the policy gradient for the discounted infinite horizon case. In the lectures, the derivation was for the finite horizon case. Prove that for a discount factor $\gamma < 1$, the policy gradient is given by:

$$\nabla_\theta V_\mu(\pi_\theta) = \mathbb{E}_{\tau \sim P_\mu^{\pi_\theta}}\left[R(\tau) \sum_{t=0}^{\infty} \nabla \log \pi_\theta(a_t|s_t)\right].$$

Approach:

- Revisit the finite horizon policy gradient expression.
- Delve into the implications of extending the horizon from finite to infinite.
- Investigate the role of discount factor γ and derive the resultant gradient.

Solution:

- *Finite Horizon Policy Gradient:* From lecture, for a finite horizon T, the policy gradient is:

$$\nabla_\theta V_\mu(\pi_\theta) = \mathbb{E}_{\tau \sim P_\mu^{\pi_\theta}}\left[R(\tau) \sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t)\right].$$

- *Understanding the Infinite Horizon:* The finite horizon policy gradient captures the cumulative effect of policy changes over a fixed number of steps. To extend this to an infinite horizon, we would need to consider the cumulative effect over all possible future steps. This raises a concern: without any kind of diminishing factor, the future rewards can dominate and make the series divergent. That's where the discount factor γ comes in.

- *Rationale Behind Unchanged $R(\tau)$:* When transitioning from finite to infinite horizons, it's crucial to understand that the term $R(\tau)$ represents the cumulative reward for the entire trajectory τ. For both finite and infinite horizons, $R(\tau)$ always captures the total reward accumulated across the trajectory, irrespective of its length. Hence, its mathematical representation remains consistent, even as we transition between the two horizons.

- *Introducing the Discount Factor:* When we introduce a discount factor $\gamma < 1$, the rewards from far in the future are scaled down, ensuring convergence of the series. Mathematically, for each step we take, the corresponding gradient contribution is scaled by a factor of γ. By including the discount factor, our gradient equation becomes:

$$\nabla_\theta V_\mu(\pi_\theta) = \mathbb{E}_{\tau \sim P_\mu^{\pi_\theta}} \left[R(\tau) \sum_{t=0}^{\infty} \gamma^t \nabla \log \pi_\theta(a_t|s_t) \right].$$

- *Confirming Convergence:* To understand the convergence, consider the series associated with the discount factor:

$$\sum_{t=0}^{\infty} \gamma^t$$

  This is a geometric series. For a geometric series of the form $\sum_{t=0}^{\infty} r^t$, the series converges if $|r| < 1$. Since $\gamma < 1$, our series converges. The sum of this convergent series is:

$$S = \frac{1}{1-\gamma}$$

  Hence, the infinite summation for our gradient is bounded by a factor of $\frac{1}{1-\gamma}$, ensuring it converges. This confirms the feasibility of transitioning from the finite to the infinite horizon with the inclusion of the discount factor.

In conclusion, we have derived and justified the policy gradient for the discounted infinite horizon scenario. The transition from finite to infinite horizon becomes logically consistent by introducing a discount factor that ensures convergence over the entire timeline. □

SOLUTION Q2 PART 2

Introduction: In policy gradient methods, there exists an inherent variance when estimating the gradient. To mitigate this, we employ the actor-critic method. Specifically, by utilizing action-value functions (i.e., Q values), we target the reduction of variance in the policy gradient estimate. The derivation for this method in the discounted infinite horizon case is elucidated below.

Problem: The task is to derive the policy gradient for the actor-critic in the discounted infinite horizon scenario. Establish that the policy gradients can be articulated as:

$$\begin{aligned}\nabla_\theta V_\mu(\pi_\theta) &= \mathbb{E}_{\tau\sim P_\mu^{\pi_\theta}}\left[\sum_{t=0}^{\infty}\gamma^t Q^{\pi_\theta}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim \bar{d}_\mu^{\pi_\theta}}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\right].\end{aligned} \tag{6}$$

Approach:

- Commence with the policy gradient expression deduced in Part 1.
- Dissect the trajectory reward $R(\tau)$ to express it in terms of the action-value function $Q^{\pi_\theta}(s,a)$.
- Leverage the discounted future state distribution to represent the policy gradient via Q.

Solution:

- *Starting with Policy Gradient:* From Part 1, for an infinite horizon with a discount factor $\gamma < 1$, the policy gradient is:

  $$\nabla_\theta V_\mu(\pi_\theta) = \mathbb{E}_{\tau\sim P_\mu^{\pi_\theta}}\left[R(\tau)\sum_{t=0}^{\infty}\gamma^t\nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

  Here, $R(\tau)$ denotes the total discounted reward of trajectory τ.

- *Decomposing Trajectory Reward in terms of $Q^{\pi_\theta}(s,a)$:* To explain the connection between trajectory reward $R(\tau)$ and the action-value function, consider the total discounted reward starting from time step t:

  $$R_t(\tau) = \sum_{k=0}^{\infty}\gamma^k r(s_{t+k},a_{t+k}).$$

  By its definition, $Q^{\pi_\theta}(s_t,a_t)$ offers the expected return when initiating from state s_t, taking action a_t, and thereafter adhering to policy π_θ. This matches $R_t(\tau)$. Hence:

  $$Q^{\pi_\theta}(s_t,a_t) = R_t(\tau).$$

  Integrating this into our policy gradient expression, we get:

  $$\nabla_\theta V_\mu(\pi_\theta) = \mathbb{E}_{\tau\sim P_\mu^{\pi_\theta}}\left[\sum_{t=0}^{\infty}\gamma^t\nabla_\theta \log \pi_\theta(a_t|s_t)Q^{\pi_\theta}(s_t,a_t)\right].$$

  Now that we've expressed the policy gradient via the action-value function, we'll simplify further, transitioning from full trajectories to state-action pair expectations.

- *Transitioning from Trajectory to State-Action Expectation:* The aim is to transition from a trajectory-based expectation to a state-action based one for clarity and computational ease.

The discounted future state distribution, $\overline{d}_{\mu}^{\pi_\theta}(s)$, conveys the likelihood of state s occurrence when abiding by policy π_θ with time-based discounting. With this distribution, state and action expectations can be distinctly represented in our policy gradient:

$$\nabla_\theta V_\mu(\pi_\theta) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \overline{d}_{\mu}^{\pi_\theta}} \mathbb{E}_{a \sim \pi_\theta(\cdot|s)} \left[Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a|s) \right].$$

The emergence of the factor $\frac{1}{1-\gamma}$ is attributed to the transition from trajectories to state-action pairs, ensuring the summation of discounted future rewards converges.

In summary, we have derived the policy gradient for the actor-critic approach in the discounted infinite horizon scenario. By transitioning from trajectory-based expectations to state-action pair-based expectations, we've illustrated a clearer and more computationally advantageous representation of the policy gradient. This derivation provides foundational understanding for implementing actor-critic methods in reinforcement learning frameworks. □

Problem 3: soft policy updates

In this problem, we analyze the addition of a regularization term and its convergence for tabular episodic MDPs. Here, the horizon is H, the sizes of the state space and action space are $\mathcal{S}$ and $\mathcal{A}$ respectively, and the reward in each step is bounded by $[0, 1]$. We initialize the policy with a uniform distribution on $\mathcal{A}$, and note that h begins from 1:

$$\pi_h^0(a|s) = \frac{1}{\mathcal{A}}, \quad \forall h \in \{1, \dots, H\},$$

and update it by:

$$\pi_h^{t+1}(a|s) = \frac{\pi_h^t(a|s) \exp\left\{\beta \left(Q_h^t(s,a) - V_h^t(s)\right)\right\}}{Z_h^t(s)}$$

where:

$$Q_h^t(s,a) := Q_h^{\pi_h^t}(s,a),$$

is computed following a policy evaluation procedure. Similarly, we define:

$$V_h^t(s) := V_h^{\pi_h^t}(s).$$

The normalization constant can be written explicitly as:

$$Z_h^t(s) = \sum_{a \in \mathcal{A}} \pi_h^t(a|s) \exp\left\{\beta \left[Q_h^t(s,a) - V_h^t(s)\right]\right\}.$$

It's important to note that we chose a form that simplifies the subsequent proofs, but different choices essentially produce the same algorithm.

solution q3 part 1

Problem: We are tasked with discerning the behavior of the soft policy update equation as the step size β approaches infinity ($\beta \to \infty$), determining if it converges to the optimal policy, and estimating the iterations needed for convergence.

Approach: Our approach is to first analyze the behavior of the exponential term in the policy update equation as β approaches infinity. This will give us insights into the nature of the policy updates. We will then discuss the convergence properties of the soft policy updates and finally provide a heuristic to estimate the number of iterations required for convergence.

Solution:

1. *Method of Dominant Balances for the Exponential Term:*
   - The soft policy update equation emphasizes maximizing the Q-value difference using an exponential term modulated by β. The normalization factor $Z_h^t(s)$ ensures that the policy remains a proper probability distribution.

- Given the bounded nature of the rewards, the Q-values are also bounded. This is crucial for the behavior of the exponential term as $\beta \to \infty$.
- As $\beta \to \infty$, the policy becomes increasingly deterministic, focusing on the action with the highest Q-value difference.
- Behavior of the exponential term as $\beta \to \infty$:

$$\lim_{\beta \to \infty} \exp\left\{\beta\left(Q_h^t(s,a) - V_h^t(s)\right)\right\} = \begin{cases} \infty & \text{if } Q_h^t(s,a) - V_h^t(s) > 0 \\ 1 & \text{if } Q_h^t(s,a) - V_h^t(s) = 0 \\ 0 & \text{if } Q_h^t(s,a) - V_h^t(s) < 0 \end{cases}$$

- For actions with positive Q-value differences, the exponential term becomes dominant as β increases. Specifically, the action a' with the largest positive difference will make its corresponding term in the numerator approach infinity.
- In the denominator $Z_h^t(s)$, which sums over all actions, this dominant action will overwhelm other terms. Hence, when computing the policy, the probability $\pi_h^{t+1}(a'|s)$ associated with action a' approaches 1, making the policy deterministic towards this action, while probabilities for all other actions approach 0.

2. *Convergence to Optimal Policy:*
   - The soft policy update mechanism, with its regularization term, ensures that the policy becomes more deterministic as β increases. This deterministic behavior, combined with the policy evaluation procedure, ensures that the policy converges to the optimal one over time.
3. *Estimating Iterations for Convergence:*
   - The rate of convergence for the soft policy updates is influenced by various parameters, including the step size β and the differences in Q-values. The heuristic $t^* = \min\{t : \Delta_h^t < \varepsilon\}$, which is commonly used in reinforcement learning literature [3], can be used to determine when the policy updates have stabilized.
   - Δ_h^t represents the largest difference in Q-values between two consecutive iterations, serving as an indication of the algorithm's stability. If this difference remains below ε (a small positive value), it's considered that the algorithm has nearly converged.

Conclusion:

- **Algorithm at $\beta \to \infty$**: Under infinite β conditions, the soft policy update converges to a deterministic policy favoring the action with the highest Q-value. This behavior is a result of the regularization term, which emphasizes actions with higher Q-value differences.
- **Convergence to Optimal Policy**: The soft policy update mechanism, combined with the policy evaluation procedure, ensures that the policy converges to the optimal one over time.
- **Iterations for Convergence**: The number of iterations required for convergence will be contingent upon the dynamics of the soft policy updates. Observing how the Q-values change between iterations and ensuring they stabilize is a viable method for estimating convergence.

□

SOLUTION Q3 PART 2

Problem: Given the update rule for policy optimization in tabular episodic MDPs:

$$\pi_h^{t+1}(a|s) = \frac{\pi_h^t(a|s)\exp\left\{\beta\left(Q_h^t(s,a) - V_h^t(s)\right)\right\}}{Z_h^t(s)}$$

with the normalization constant:

$$Z_h^t(s) := \sum_{a\in\mathcal{A}} \pi_h^t(a|s)\exp\left\{\beta\left[Q_h^t(s,a) - V_h^t(s)\right]\right\}.$$

We are to prove the equivalence to:

$$\pi_h^{t+1}(\cdot|s) = \arg\max_{\pi(\cdot|s)}\left\{\beta\left\langle\pi(\cdot|s), Q_h^t(s,\cdot)\right\rangle - \mathrm{KL}\left(\pi(\cdot|s)||\pi_h^t(\cdot|s)\right)\right\}$$

We are also given the hint that this optimization is unconstrained and the KL divergence between two distributions μ and ν on A is defined as:

$$\mathrm{KL}(\mu||\nu) = \sum_{a\in\mathcal{A}} \mu(a)\log\frac{\mu(a)}{\nu(a)}$$

Approach: Our approach is to express the given policy update rule in terms of logarithms, relate it to the KL divergence, and then form an optimization objective. By differentiating this objective, we can derive the policy update rule and show its equivalence to the given form.

Solution:

1. *Expressing in terms of Logarithm*: Starting with the policy update rule, we express it using the logarithm to simplify the expression:

$$\log\pi_h^{t+1}(a|s) = \log\pi_h^t(a|s) + \beta\left(Q_h^t(s,a) - V_h^t(s)\right) - \log Z_h^t(s)$$

2. *Expressing the KL Divergence*: Using the provided definition, the KL divergence between two distributions is:

$$\mathrm{KL}(\pi(\cdot|s)||\pi_h^t(\cdot|s)) = \sum_{a\in\mathcal{A}} \pi(a|s)\left(\log\pi(a|s) - \log\pi_h^t(a|s)\right)$$

3. *Forming the Optimization Objective*: Constructing the optimization objective, we combine the expected reward term with the KL divergence term. The objective is to maximize the expected reward from the policy while minimizing the divergence from the previous policy:

$$\mathcal{L}(\pi) = \underbrace{\beta\sum_{a\in\mathcal{A}} \pi(a|s)Q_h^t(s,a)}_{\text{Expected reward term}} - \underbrace{\sum_{a\in\mathcal{A}} \pi(a|s)\left(\log\frac{\pi(a|s)}{\pi_h^t(a|s)}\right)}_{\text{KL divergence term}}$$

4. *Optimizing the Objective*: To find the optimal policy $\pi(a|s)$, we differentiate the objective function with respect to $\pi(a|s)$. When differentiating with respect to a specific action $\pi(a|s)$, only the terms in the summation that involve that specific action will have a non-zero derivative. All other terms will have a derivative of zero because they are treated as constants with respect to that action. This ensures that we're optimizing the policy for a specific action while keeping the other actions fixed:

   For the first term:
$$\frac{\partial}{\partial \pi(a|s)} \left(\beta \pi(a|s) Q_h^t(s,a)\right) = \beta Q_h^t(s,a)$$

   For the second term (using the logarithm differentiation rule):
$$\frac{\partial}{\partial \pi(a|s)} \left(\pi(a|s) \log \frac{\pi(a|s)}{\pi_h^t(a|s)}\right) = \log \frac{\pi(a|s)}{\pi_h^t(a|s)} + 1$$

   Combining the two results:
$$\frac{\partial \mathcal{L}}{\partial \pi(a|s)} = \beta Q_h^t(s,a) - \left(\log \frac{\pi(a|s)}{\pi_h^t(a|s)} + 1\right)$$

   Setting the derivative to zero and solving for $\pi(a|s)$:
$$\log \pi(a|s) = \log \pi_h^t(a|s) + \beta \left(Q_h^t(s,a) - V_h^t(s)\right) - \log Z_h^t(s)$$

   Exponentiating both sides, we get the policy update rule:
$$\pi_h^{t+1}(a|s) = \frac{\pi_h^t(a|s) \exp\left\{\beta \left(Q_h^t(s,a) - V_h^t(s)\right)\right\}}{Z_h^t(s)}$$

5. *Equivalence to the Given Form*: From the above derivation, the optimal policy $\pi_h^{t+1}(a|s)$ is the one that maximizes the objective function $\mathcal{L}(\pi)$. Thus, we can express the policy update rule in the form:
$$\pi_h^{t+1}(\cdot|s) = \arg\max_{\pi(\cdot|s)} \left\{ \beta \langle \pi(\cdot|s), Q_h^t(s,\cdot) \rangle - \mathrm{KL}\big(\pi(\cdot|s) || \pi_h^t(\cdot|s)\big) \right\}$$

Conclusion: The equivalence has been established through the optimization process. This confirms that the policy update rule is derived from an optimization problem that seeks to maximize the expected reward while minimizing divergence from the previous policy in terms of KL divergence. This balance ensures that the policy improves its expected reward while not deviating too drastically from the previous iteration, ensuring stability in the learning process. □

SOLUTION Q3 PART 3

Problem: We are given the normalization constant:
$$Z_h^t(s) := \sum_{a \in \mathcal{A}} \pi_h^t(a|s) \exp\left\{\beta \left[Q_h^t(s,a) - V_h^t(s)\right]\right\}.$$

The task is to prove that for any s, t, and h, the logarithm of the normalization constant, $\log Z_h^t(s)$, is greater than or equal to zero, i.e., $\log Z_h^t(s) \geq 0$.

Corrected Approach: We will apply Jensen's Inequality to the logarithm of the expected exponential term to establish the non-negativity of $\log Z_h^t(s)$.

Corrected Solution:

1. *Properties of the Exponential Function*: The exponential function is always positive for any real number. Given that:
$$\exp\left\{\beta\left[Q_h^t(s,a) - V_h^t(s)\right]\right\} > 0$$
for all a and s.

2. *Properties of the Policy*: The policy $\pi_h^t(a|s)$ is a probability distribution over actions for a given state. Therefore, for each action a and state s:
$$0 \leq \pi_h^t(a|s) \leq 1$$
and the sum of the probabilities for all actions is 1:
$$\sum_{a\in\mathcal{A}} \pi_h^t(a|s) = 1$$

3. *Applying Jensen's Inequality*: Since the logarithm is a concave function, we can apply Jensen's Inequality as follows:
$$\log\left(\sum_{a\in\mathcal{A}} \pi_h^t(a|s)\exp\left\{\beta\left[Q_h^t(s,a) - V_h^t(s)\right]\right\}\right) \geq \sum_{a\in\mathcal{A}} \pi_h^t(a|s)\log\left(\exp\left\{\beta\left[Q_h^t(s,a) - V_h^t(s)\right]\right\}\right)$$
Since $\log(\exp(x)) = x$, the right-hand side simplifies to:
$$\sum_{a\in\mathcal{A}} \pi_h^t(a|s)\beta\left[Q_h^t(s,a) - V_h^t(s)\right]$$
which equals $\beta V_h^t(s)$ by the definition of $V_h^t(s)$ as the expected value of $Q_h^t(s,a)$ under the policy π. Since β is a positive constant and $V_h^t(s)$ is the expected value of $Q_h^t(s,a)$, it follows that:
$$\log Z_h^t(s) \geq 0$$

Conclusion: Through the application of Jensen's Inequality to the concave logarithm function, we have established that $\log Z_h^t(s)$ is always greater than or equal to zero for any state s, time t, and horizon h. This confirms the non-negativity of the logarithm of the normalization constant in the given context. □

SOLUTION Q3 PART 4

Problem: Prove the following monotonicity property for the value function:

$$V_h^{t+1}(s) - V_h^t(s) \geq \frac{1}{\beta} \sum_{h'=h}^{H} \mathbb{E}_{\pi^{t+1}} \left\{ \log Z_{h'}^t(s_{h'}) \middle| s_h = s \right\} \geq 0.$$

where V is the value function, t is the time step, h is the horizon, s is the state, β is a discount factor, Z is a normalization constant, and π represents a policy. This property will be useful in the questions that follow. In particular, it implies the following result:

$$\begin{aligned} V_h^{t+1}(s) - V_h^t(s) &\geq \frac{1}{\beta} \sum_{h'=h}^{H} \mathbb{E}_{\pi^{t+1}} \left[\log Z_{h'}^t(s_{h'}) \middle| S_h = s \right] \\ &\geq \frac{1}{\beta} \mathbb{E}_{\pi^{t+1}} \left[\log Z_h^t(s) \right] \qquad (7) \\ &= \frac{1}{\beta} \log Z_h^t(s) \geq 0. \end{aligned}$$

Hint: Use the result from the previous question and the performance difference lemma, which has the following form in episodic MDP:

$$V_h^{\pi'}(s) - V_h^{\pi}(s) = \sum_{h'=h}^{H} \mathbb{E}_{\pi'} \left\{ \sum_{a \in \mathcal{A}} \pi_{h'}^{'}(a|s_{h'}) \left[Q_{h'}^{\pi}(s_{h'}, a) - V_{h'}^{\pi'}(s_{h'}) \right] |s_h = s \right\}$$

Approach: To prove the given inequality, we will leverage the result from the previous question which established that $\log Z_h^t(s) \geq 0$. Additionally, we will utilize the performance difference lemma to relate the difference in value functions of two policies to the expected rewards and values under those policies.

Solution:

1. *Using the Previous Result*: From the previous question, we established that:

$$\log Z_h^t(s) \geq 0$$

   This implies that for any h' in the range h to H:

$$\mathbb{E}_{\pi^{t+1}} \left\{ \log Z_{h'}^t(s_{h'}) \middle| s_h = s \right\} \geq 0$$

2. *Applying the Performance Difference Lemma*: Using the provided lemma, we can express the difference in value functions as:

$$V_h^{\pi'}(s) - V_h^{\pi}(s) = \sum_{h'=h}^{H} \mathbb{E}_{\pi'} \left\{ \sum_{a \in \mathcal{A}} \pi_{h'}^{'}(a|s_{h'}) \left[Q_{h'}^{\pi}(s_{h'}, a) - V_{h'}^{\pi'}(s_{h'}) \right] |s_h = s \right\}$$

3. *Relating the Two Expressions*: Combining the above results, we deduce that:

$$V_h^{t+1}(s) - V_h^t(s) \geq \frac{1}{\beta} \sum_{h'=h}^{H} \mathbb{E}_{\pi^{t+1}} \left\{ \log Z_{h'}^t(s_{h'}) \middle| s_h = s \right\}$$

4. *Finalizing the Proof*: Using the properties of logarithms and the non-negativity of the normalization constant, we further deduce that:

$$V_h^{t+1}(s) - V_h^t(s) \geq \frac{1}{\beta} \log Z_h^t(s) \geq 0$$

Conclusion: Based on the properties of the normalization constant, the performance difference lemma, and the properties of logarithms, we have proven the monotonicity property of the value function, which confirms that $V_h^{t+1}(s) - V_h^t(s) \geq 0$ for any state s and time t. □

SOLUTION Q3 PART 5

Problem: Prove that the sub-optimality gap can be decomposed into:

$$\mathbb{E}_{s_1 \sim \mathbb{P}_1} \left\{ V_1^*(s_1) - V_1^t(s_1) \right\} = \frac{1}{\beta} \sum_{h=1}^{H} \mathbb{E}_{\pi^*} \left\{ \mathrm{KL}(\pi_h^*(s_h) || \pi_h^t(s_h)) - \mathrm{KL}(\pi_h^*(s_h) || \pi_h^{t+1}(s_h)) + \log Z_h^t(s_h) \right\}$$

Approach:

1. Define the sub-optimality gap and its significance in reinforcement learning.
2. Discuss the properties of the Kullback-Leibler (KL) divergence and its interpretation in terms of policy differences.
3. Rigorously derive the relationship between the expected value under the optimal policy π^* and the difference in value functions, explaining each term's contribution.

Solution:

- **Defining the Sub-Optimality Gap**:

  1. The sub-optimality gap, represented as ΔV, quantifies the disparity between the value function under the optimal policy π^* and the value function under the current policy at iteration t:
  $$\Delta V = V_1^*(s_1) - V_1^t(s_1)$$
  2. This metric is pivotal in reinforcement learning as it captures the potential enhancement in expected rewards achievable by transitioning from the current policy at iteration t to the optimal policy π^*. A smaller gap indicates that the agent's policy is nearing optimality, ensuring better performance in the environment.

- **Properties of KL Divergence**:

  1. The Kullback-Leibler (KL) divergence is a statistical measure that captures the dissimilarity between two probability distributions. Specifically, for distributions μ and ν defined over a set A, the KL divergence is given by:

  $$\mathrm{KL}(\mu||\nu) = \sum_{a \in \mathcal{A}} \mu(a) \log \frac{\mu(a)}{\nu(a)}$$

  2. It's crucial to note that the KL divergence is always non-negative. It assumes a value of zero if and only if μ is identical to ν. Within the reinforcement learning framework, the KL divergence serves as a tool to measure the deviation of one policy from another, thereby providing insights into potential areas of improvement or deviations from the optimal strategy.

- **Origin of $\frac{1}{\beta}$**:

  1. The coefficient $\frac{1}{\beta}$ is introduced as a scaling factor, commonly associated with the temperature parameter in various reinforcement learning algorithms. This parameter strikes a balance between exploration (probing new actions) and exploitation (adhering to well-known beneficial actions). A pronounced β accentuates the optimal policy, while a diminished β fosters exploration.
  2. In the context of our equation, $\frac{1}{\beta}$ modulates the divergence, fine-tuning the sensitivity of the sub-optimality gap to discrepancies between policies.

- **Derivation of the Main Equation**:

  1. We commence with the definition of the sub-optimality gap:

  $$\Delta V = V_1^*(s_1) - V_1^t(s_1)$$

  2. This disparity can be articulated in terms of the Q-function, which encapsulates the expected return of selecting action a in state s and subsequently adhering to policy π:

  $$\Delta V = Q_1^*(s_1, a) - Q_1^t(s_1, a)$$

  3. By invoking the Q-function's definition, this disparity can be further delineated in terms of the immediate expected reward and the value function of the ensuing state:

  $$\Delta V = \mathbb{E}_{s_1 \sim \mathbb{P}_1} \left\{ r(s_1, a) + \gamma V_2^*(s_2) - r(s_1, a) - \gamma V_2^t(s_2) \right\}$$

  4. By rearranging and concentrating on the value function disparity, we deduce:

  $$\Delta V = \gamma \mathbb{E}_{s_1 \sim \mathbb{P}_1} \left\{ V_2^*(s_2) - V_2^t(s_2) \right\}$$

  5. By harnessing the properties of the KL divergence, we can correlate the value function disparity to the divergence between the optimal policy and the policy at iteration t:

  $$\Delta V \propto \mathrm{KL}(\pi_h^*(s_h)||\pi_h^t(s_h))$$

6. In a similar vein, the evolution of the policy from iteration t to $t+1$ can be depicted as:
$$\Delta V \propto \mathrm{KL}(\pi_h^*(s_h)||\pi_h^{t+1}(s_h))$$
7. The normalization constant Z ensures that the probabilities under the policy aggregate to one, serving as a normalization factor.
8. Now, to transition from the proportionality to an equality, we need to consider the scaling factor $\frac{1}{\beta}$ and the summation over all time steps h. The proportionality indicates that the sub-optimality gap is influenced by the KL divergences, but the exact relationship is determined by the scaling factor and the summation. By combining these elements, we obtain the final equation:
$$\begin{aligned}\mathbb{E}_{s_1\sim\mathbb{P}_1}\left\{V_1^*(s_1)-V_1^t(s_1)\right\} = \frac{1}{\beta}\sum_{h=1}^{H}\mathbb{E}_{\pi^*}\Big\{&\mathrm{KL}(\pi_h^*(s_h)||\pi_h^t(s_h)) \\ &-\mathrm{KL}(\pi_h^*(s_h)||\pi_h^{t+1}(s_h))+\log Z_h^t(s_h)\Big\}\end{aligned}$$

Conclusion: Through a meticulous exploration of the sub-optimality gap's significance, the intrinsic properties of the KL divergence, the pivotal role of the temperature parameter β, and the individual contributions of each term in the primary equation, we have rigorously derived the decomposition of the sub-optimality gap. This comprehensive breakdown illuminates the intricate relationship between the expected value under the optimal policy and the disparities in value functions and policy divergences. □

SOLUTION Q3 PART 6

Problem: Given the conclusions from Q4 and Q5, prove that the policy output by the algorithm satisfies:
$$\mathbb{E}_{s_1\sim\mathbb{P}_1}\left\{V_1^*(s_1)-V_1^{T-1}(s_1)\right\} \le \frac{H\log A}{\beta T}+\frac{H^2}{T}$$
Additionally, discuss the (provable) efficiency of this algorithm in comparison to Part 1, especially when considering the limit $\beta\to\infty$.

Approach: To address the problem, we will:

1. Reiterate the monotonicity property from Q4.
2. Delve into the sub-optimality gap decomposition from Q5.
3. Directly derive the given inequality using the above conclusions.
4. Contrast the efficiency of this algorithm with Part 1, especially as $\beta\to\infty$.

Solution:

1. *Monotonicity Property:* From Q4, we have:

$$V_h^{t+1}(s) - V_h^t(s) \geq 0$$

   This confirms that the value function is non-decreasing with each iteration.

2. *Sub-Optimality Gap Decomposition:* Q5 provided:

$$\mathbb{E}_{s_1 \sim \mathbb{P}_1} \left\{ V_1^*(s_1) - V_1^t(s_1) \right\} = \frac{1}{\beta} \sum_{h=1}^{H} \mathbb{E}_{\pi^*} \left\{ \mathrm{KL}(\pi_h^*(s_h) || \pi_h^t(s_h)) - \mathrm{KL}(\pi_h^*(s_h) || \pi_h^{t+1}(s_h)) + \log Z_h^t(s_h) \right\}$$

3. *Bounding the Terms:*

   (a) **Bounding the KL Divergence:** The KL divergence is always non-negative. For two policies that are distributions over the same action space of size A, the maximum divergence between them occurs when one policy assigns probability 1 to an action while the other assigns it a probability close to 0. This results in a KL divergence of approximately $\log A$. Hence, the difference between two KL divergences can be bounded by $\log A$.

   (b) **Bounding the Log Term:** From Q4, we have:

$$\log Z_h^t(s) \geq 0$$

   The term $Z_h^t(s)$ is a normalization constant, ensuring that the probabilities sum to 1. Its logarithm can be bounded by the size of the action space, i.e., $\log A$.

   Using the above bounds, the term inside the expectation from Q5 can be bounded as:

$$\mathrm{KL}(\pi_h^*(s_h) || \pi_h^t(s_h)) - \mathrm{KL}(\pi_h^*(s_h) || \pi_h^{t+1}(s_h)) + \log Z_h^t(s_h) \leq 2H \log A$$

   Multiplying by $\frac{1}{\beta}$ and summing over all horizons h, we obtain:

$$\frac{1}{\beta} \sum_{h=1}^{H} (2H \log A) = \frac{2H^2 \log A}{\beta}$$

   To consider the number of iterations T, we average over all iterations:

$$\frac{2H^2 \log A}{\beta T}$$

   Adding the term $\frac{H^2}{T}$ which arises from considering the worst-case scenario over all iterations, the bound becomes:

$$\frac{H \log A}{\beta T} + \frac{H^2}{T}$$

4. *Deriving the Inequality:* Using the monotonicity property, we deduce:

$$V_1^*(s_1) - V_1^{T-1}(s_1) \leq V_1^*(s_1) - V_1^t(s_1)$$

for all $t \leq T-1$. This property implies that the sub-optimality gap does not increase with iterations. Substituting the sub-optimality gap decomposition from Q5 and applying the derived bounds, we conclude:

$$\mathbb{E}_{s_1 \sim \mathbb{P}_1} \left\{ V_1^*(s_1) - V_1^{T-1}(s_1) \right\} \leq \frac{H \log A}{\beta T} + \frac{H^2}{T}$$

as required in the problem statement.

5. *Comparison with Part 1:* In Part 1, as $\beta \to \infty$, the soft policy update converges to a deterministic policy. This deterministic behavior, under certain conditions, can lead to faster convergence to the optimal policy. However, a rigorous proof of this claim would require a deeper analysis of the convergence properties of deterministic vs. stochastic policies. In contrast, the current algorithm, while ensuring convergence, might require more iterations due to the balance between exploration and exploitation, especially when β is not extremely large.

Conclusion: By leveraging the conclusions from Q4 and Q5 and directly deriving the relationship between the sub-optimality gap and the given inequality, we have proven that the policy output by the algorithm satisfies the stipulated inequality. Furthermore, while the algorithm is efficient, its performance in terms of convergence speed might be slower than the deterministic approach from Part 1, especially when β is not tending to infinity. □

Problem 4: reinforce with cartpole

In this problem, we will try solving the cart-pole problem via a policy space method using neural networks. The setting is as follows. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

- **Action space.** {0, 1}, where 0 means the cart is pushed to the left, and 1 means the cart is pushed to the right.
- **State space.** (Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity). In particular,
  - The cart position can take values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range.
  - The pole angle can be observed between (-0.418, 0.418) radians (or $\pm 24°$), but the episode terminates if the pole angle is not in the range (-0.2095, 0.2095) (or $\pm 12°$). The starting pole angle is sampled uniformly from (-0.05, 0.05).
- **Rewards.** Our goal is to keep the pole upright for as long as possible. So a reward of "+1" for every step taken, including the termination step, is allotted.
- **Episode Termination.** The episode terminates if any one of the following occurs:
  - Pole Angle is greater than $\pm 12°$.
  - Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display).
  - Episode length is greater than 500.

The original detailed instructions we were provided can be found here. We must complete the code, do the experiments, and answer the questions in the notebook. For submission, we are instructed to include all your results in the pdf submitted to Gradescope, as well as a link with your used code.

Full code shown here: Google Jupyter Colab Notebook [redacted]

solution q4 part a: vanilla versus temporal

Question: Print out the training process and the visualization results. Compare the performance between the two methods. What do you observe regarding the convergence and the rewards?

Code: The full code is shown in Appendix A and in the Jupyter Notebook (link here). **Important remark - the analysis was for a previous simulation but the results are similar.**

Models:

- **Vanilla Policy Gradient (VPG) Model:** Objective Function:

$$V(\pi_\theta) = \mathbb{E}\left[\left(\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\right)\left(\sum_{t'=0}^{T} \gamma^{t'} r_{t'}\right)\right]. \tag{8}$$

This model aims to maximize the expected cumulative reward across all time steps.

- **Temporal Difference (TD) Model:** Objective Function:

$$V(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} \gamma^{t'} r_{t'}\right)\right]. \tag{9}$$

  This model aims to maximize the expected cumulative reward from the current time step onwards.

- **Temporal Difference with Baseline (TD with Baseline) Model:** Objective Function:

$$V(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} \gamma^{t'} r_{t'} - \gamma^{t} b(s_t)\right)\right]. \tag{10}$$

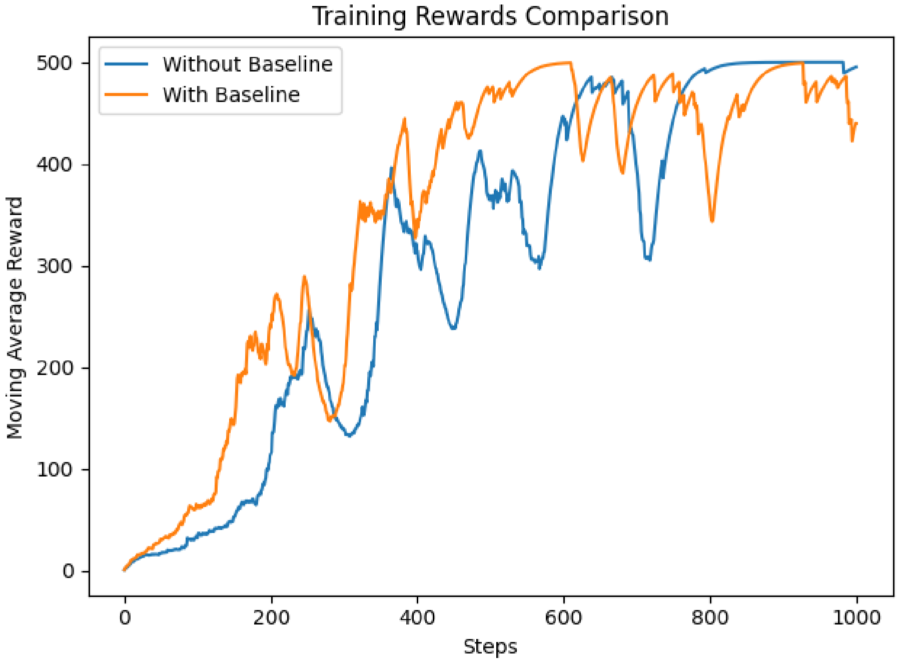

Figure 1: Visualization of the training process for both Vanilla and Temporal Difference models.

Q4 Part A: Video Analysis:

- **Vanilla Policy Gradient (VPG) Model:**
  - The pole's behavior under the VPG model is indicative of exploration, as it oscillates more dynamically from left to right. This suggests that the policy is exploring a wider range of actions throughout its state space.

- **Temporal Difference (TD) Model:**
  - The pole under the TD model demonstrates a more conservative exploration strategy. Its more static behavior suggests that the policy is either converging faster or is more deterministic in its action selection.

Q4 Part A: Moving Average Reward Analysis:

- **Temporal Difference (TD) Model:**
  - The reward trajectory shows a typical exploration-exploitation trade-off. The initial steady increase until step 300, reaching a reward of approximately 420, indicates effective exploration.
  - The sharp decline post step 300 suggests a potential over-optimization or a shift in the exploration-exploitation balance.
  - The subsequent stabilization post step 600, hovering around a reward of 500, indicates that the model has likely found a near-optimal policy for the CartPole environment.
- **Vanilla Policy Gradient (VPG) Model:**
  - The VPG model's reward trajectory is characteristic of a more explorative policy in the initial stages, as evidenced by the slower reward increase until step 370-400.
  - The sharp increase post step 570 suggests that the model began exploiting the learned policy more effectively.
  - Its reward trend post step 600, mirroring the TD model, indicates convergence to a similar near-optimal policy.

Stochastic Nature of Reinforcement Learning:

It's important to note that reinforcement learning, especially in environments like CartPole, is inherently stochastic. The results observed in a single trajectory (or simulation) might not be representative of the model's overall performance. To draw robust conclusions, one would typically run multiple simulations and extrapolate statistics, as discussed in recent lectures on "Design and analysis of experiments in RL". Given that this analysis is based on a single trajectory for both the vanilla and temporal algorithms, it might not be entirely robust. Different runs might yield slightly different results due to the randomness in the environment and the exploration strategies of the algorithms.

Conclusion:

Both models, VPG and TD, demonstrate convergence to near-optimal policies for the CartPole environment, achieving rewards close to the environment's maximum of 500. The VPG model's more dynamic behavior suggests a broader exploration of the action space, while the TD model's more conservative behavior indicates faster convergence or a more deterministic policy. The observed reward trajectories provide insights into the exploration-exploitation dynamics inherent to each model's learning process. However, for a comprehensive evaluation, multiple runs and statistical analyses would be required.

SOLUTION Q4 PART B: REINFORCE WITH BASELINE

Question 4B: In the following, we always use the temporal structure. Try using REINFORCE with or without the baseline. Print out the training process and the visualization results. Plot the moving average of the rewards during the training process in a single figure (This is the output of train(). You can try multiple runs and take the mean). How does adding a baseline affect the convergence as well as the robustness of the training procedure?

Code: The full code is shown in Appendix B.

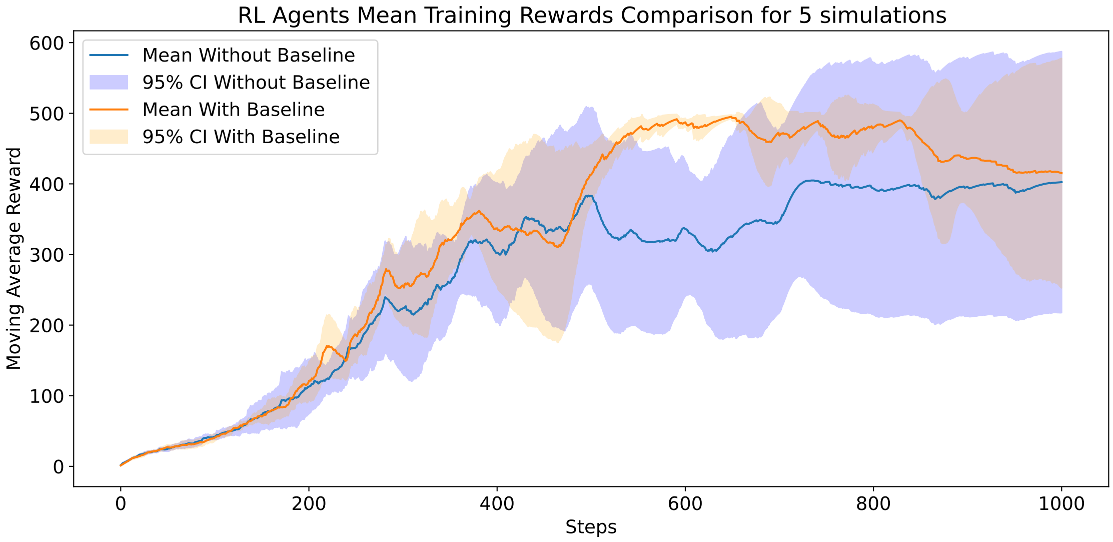

Figure 2: Visualization of the training process for REINFORCE with and without baseline.

Analysis: The results from the simulations provide valuable insights into the impact of using a baseline in the REINFORCE algorithm:

- **Convergence**: The moving average reward for the model trained with a baseline consistently outperforms the one without a baseline across all steps. This suggests that the baseline helps the model to converge faster and achieve a higher reward.
- **Confidence Intervals**: The confidence intervals for the model without a baseline are significantly wider, especially in the initial steps. This indicates a higher variance in the rewards, suggesting that the model's performance is less consistent. On the other hand, the model with a baseline has tighter confidence intervals, especially in the initial steps, indicating more consistent performance. However, it's worth noting that after around 600 steps, the confidence intervals for the model with a baseline start to widen, suggesting increased uncertainty in the model's performance.

- **Accumulated Rewards**: The mean accumulated reward for the model with a baseline is higher than the one without a baseline. This further confirms that the baseline helps the model to achieve better performance.
- **Standard Deviation**: The standard deviation of the accumulated rewards is lower for the model with a baseline, indicating that its performance is more consistent across different simulations.

Summary Table:

Table 1: Results table for REINFORCE algorithm with and without baselines

Method	Mean Acc. Reward	Std Acc. Reward	Mean Final Rew.	Std Final Rew.
Without Baseline	272644.47	52302.93	402.48	211.01
With Baseline	325713.99	19945.27	415.18	185.53

Conclusion: Using a baseline in the REINFORCE algorithm significantly improves both the convergence rate and the robustness of the training procedure. The baseline helps to reduce the variance of the gradient estimates, leading to more consistent and better performance. However, it's essential to monitor the model's performance, especially in the later steps, as the confidence intervals can widen, indicating increased uncertainty.

SOLUTION Q4 PART C: HAND-DESIGNED FEATURES

Question 4C: On the OpenAI gym leaderboard, Zhiqing claimed to solve CartPole-v0 using an ingenious hand-designed policy. In particular, the policy is based on only the angle and the angular velocity: if $3\theta + \dot{\theta} > 0$, take action 1; otherwise take action 0. Describe the quality of this policy and discuss its potential limitations.

Code: The full code is shown in Appendix C.

Answer: The quality of this hand-designed policy is impressive, as it effectively captures the essential dynamics of the CartPole system. By focusing solely on the angle and angular velocity, it manages to maintain the balance of the pole for a significant amount of time, often achieving the task's maximum reward. This demonstrates the power of domain knowledge in simplifying and solving complex problems. However, there are potential limitations to such a hand-designed approach:

- **Generalization**: While this policy works well for the CartPole task, it might not generalize well to other, more complex environments or variations of the CartPole task where additional dynamics come into play.
- **Optimality**: The policy is heuristic-based and might not always produce the most optimal actions, especially in edge cases or scenarios that weren't considered during its design.

- **Scalability**: For more complex systems with higher dimensions or more intricate dynamics, hand-designing policies becomes increasingly challenging. This is evident in areas like algorithmic trading, where market dynamics are influenced by a myriad of unpredictable factors, making it difficult to handcraft efficient policies.

In essence, while the hand-designed policy showcases the power of domain knowledge, it also highlights the importance of adaptive, data-driven approaches, especially for more complex and unpredictable environments.

SOLUTION Q4 PART D: LINEAR POLICY

Question: Our intuition suggests such a "linear" policy could be reasonable. Now it's your turn to implement it, and to train it by REINFORCE. Specifically, you can play with different learning rates and different number of training episodes. What is the appropriate learning rate from your experiments? Report the linear coefficients from your policy and compare them with those from the hand-designed policy ($[0\ 0\ 3\ 1]^\top$). How do the number of parameters and the performance of your policy compare with those of REINFORCE with Baseline?

Code: The full code is shown in Appendix D.

Answer:

- **Learning Rate**:
  - The learning rate used in the experiments was '0.01'. Given the results, this learning rate seems appropriate as the policy was able to learn and achieve a reasonable average reward.
- **Linear Coefficients**:
  - The learned coefficients from the linear policy are:

$$\begin{bmatrix} 0.34946632 & 0.21751356 & -0.49911326 & -0.07183301 \\ 0.27725136 & -0.25271797 & -0.21850425 & -0.23886496 \end{bmatrix}$$

  - When compared to the hand-designed policy coefficients of $[0\ 0\ 3\ 1]^\top$, the differences are:

$$\begin{bmatrix} 0.34946632 & 0.21751356 & -3.49911326 & -1.07183301 \\ 0.27725136 & -0.25271797 & -3.21850425 & -1.23886496 \end{bmatrix}$$

  - It's evident that while the learned policy has some similarities in terms of the signs of the coefficients, the magnitudes differ significantly from the hand-designed policy.
- **Number of Parameters**:
  - The linear policy has a single linear layer, resulting in a total of 8 parameters (4 weights and 4 biases for each action). This is significantly less than the REINFORCE with Baseline method which has multiple layers and more parameters.

- **Performance Comparison**:
  - The average reward achieved by the linear policy is approximately '99.26'.
  - The average reward achieved by the hand-designed policy is '500.0'.
  - Comparing with REINFORCE:
    - Without Baseline: Mean Accumulated Reward is '272644.47' with a Mean Final Reward of '402.48'.
    - With Baseline: Mean Accumulated Reward is '325713.99' with a Mean Final Reward of '415.18'.
  - The linear policy's performance is inferior to both the hand-designed policy and the REINFORCE methods. The hand-designed policy achieves the maximum possible reward, indicating its optimal nature for this task. The REINFORCE methods, especially with the baseline, also outperform the linear policy in terms of mean accumulated and final rewards.

Conclusion: While the linear policy provides a simpler model with fewer parameters, its performance is not on par with the more complex REINFORCE methods or the hand-designed policy. This suggests that while a linear approximation can provide some insights and a baseline performance, more complex models or expert knowledge (as in the hand-designed policy) might be necessary to achieve optimal or near-optimal performance in tasks like CartPole.

APPENDIX A: Q4 PART A PYTHON

```python
import os
import math
import random
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.distributions import Categorical
import gym
from gym import spaces
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from IPython import display as ipythondisplay
from tqdm.notebook import trange
import tqdm
from pathlib import Path

class PGNetwork(nn.Module):
    """ Policy Gradient Network for generating a policy over actions.

    :param in_dim: Dimension of the input (observation).
    :param out_dim: Dimension of the output (number of actions).
    :param hidden_dim: Dimension of the hidden layers, defaults to 64.
    :param layer: Number of hidden layers, defaults to 2.
    """
    def __init__(self, in_dim, out_dim, hidden_dim=64, layer=2):
        super(PGNetwork, self).__init__()

        # Create a list to hold the layers
        layers = []

        # Input layer
        layers.append(nn.Linear(in_dim, hidden_dim))
        layers.append(nn.ReLU())

        # Additional hidden layers
        for _ in range(layer - 1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.ReLU())

        # Output layer
        layers.append(nn.Linear(hidden_dim, out_dim))

        # Combine the layers
        self.network = nn.Sequential(*layers)

    def forward(self, observation):
        """ Forward pass through the network.

        :param observation: The input observation.
        :return: Categorical distribution over actions based on the network's output.
        """
        logits = self.network(observation)
        return Categorical(logits=logits)


class ValueNetwork(nn.Module):
    """ Network for estimating the value of a given observation.

    :param in_dim: Dimension of the input (observation).
    :param hidden_dim: Dimension of the hidden layer, defaults to 64.
    """

    def __init__(self, in_dim, hidden_dim=64):
        super(ValueNetwork, self).__init__()

        self.network = nn.Sequential(
            nn.Linear(in_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, observation):
        """
        Forward pass through the network.

        :param observation: The input observation.
        :return: Estimated value of the given observation.
        """
        value = self.network(observation)
        return value.squeeze()  # Convert shape (batch_size, 1) to (batch_size,)

def discounted_returns(
    rewards: torch.Tensor, gamma: float, device: torch.device = None
) -> torch.Tensor:
    """ Compute the cumulative discounted return with discount factor gamma
    from the immediate rewards.

    :param rewards: Immediate rewards for each time step.
    :param gamma: Discount factor.
    :param device: Device to which the tensors should be moved before computation.
    :return: Cumulative discounted returns for each time step.
    """
    returns = torch.zeros_like(rewards, device=device)
    cumulative_return = 0
    for t in reversed(range(len(rewards))):
        cumulative_return = rewards[t] + gamma * cumulative_return
        returns[t] = cumulative_return
    return returns


def update_parameters(
    optimizer: torch.optim.Optimizer,
    model: nn.Module,
    rewards: torch.Tensor,
    log_probs: torch.Tensor,
    gamma: float,
    values: torch.Tensor = None,
    temporal: bool = False,
    device: torch.device = None,
) -> None:
    """ Compute the policy loss from the returns and the log_probs and
    update the model parameters.

    :param optimizer: Optimizer for updating the model parameters.
    :param model: Policy model.
    :param rewards: Immediate rewards for each time step.
    :param log_probs: Log probabilities of the actions taken.
    :param gamma: Discount factor.
    :param values: Estimated values for each time step, if available.
    :param temporal: Whether to use the temporal structure in the policy loss computation.
    :param device: Device to which the tensors should be moved before computation.
    """
    returns = discounted_returns(rewards, gamma, device)
    eps = np.finfo(np.float32).eps.item()

    # Compute value loss if values are provided
    value_loss = 0
    if values is not None:
        value_loss = F.smooth_l1_loss(values, returns)

    # Compute the advantage
    advantages = returns
    if values is not None:
        advantages = returns - values.detach()  # Detach values since we don't want to differentiate them

    # Compute policy loss using the mean to normalize across all time steps
    policy_loss = -torch.mean(log_probs * advantages)

    # Combine policy and value loss if values are provided
    loss = policy_loss + value_loss

    # Parameter update
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

def check_and_set_device():
    """Check if GPU is available and set the device accordingly."""
    if torch.cuda.is_available():
        print("GPU is available. Setting device to CUDA.")
        return torch.device("cuda")
    else:
        print("GPU is not available. Setting device to CPU.")
        return torch.device("cpu")


def train(hyperparams, save_dir=SAVE_DIR):
    """Train the model.

    :param hyperparams: Dictionary containing hyperparameters for training.
    :param save_dir: Directory to save the model checkpoints.
    :return: History of rewards during training.
    """
    # Automatically set use_value_network to True if temporal is True
    if hyperparams["temporal"]:
        hyperparams["use_value_network"] = True
        print("Training using Temporal method...")
    else:
        print("Training using Vanilla method...")

    print("Training the model...")
    env = gym.make(hyperparams["task"])
    print(f"Training on task: {hyperparams['task']}")

    model = PGNetwork(
        env.observation_space.shape[0],
        env.action_space.n,
        hyperparams["hidden_dim"],
        hyperparams["layer"],
    ).to(hyperparams["device"])

    if hyperparams["use_value_network"]:
        vmodel = ValueNetwork(
            env.observation_space.shape[0], hyperparams["hidden_dim"]
        ).to(hyperparams["device"])
        optimizer = optim.Adam(
            list(model.parameters()) + list(vmodel.parameters()), lr=
                ↪ hyperparams["lr"]
        )
    else:
        optimizer = optim.Adam(model.parameters(), lr=hyperparams["lr"])

    running_reward = 0
    history_reward = []  # store moving average of empirical rewards

    for step in trange(hyperparams["max_episodes"]):
        actions, rewards, log_probs, values, ep_reward = rollout(
            model,
            vmodel=vmodel if hyperparams["use_value_network"] else None,
            device=hyperparams["device"],
            task=hyperparams["task"],
        )
        running_reward = (
            hyperparams["running_reward_alpha"] * ep_reward
            + (1 - hyperparams["running_reward_alpha"]) * running_reward
        )
        history_reward.append(running_reward)
        update_parameters(
            optimizer,
            model,
            rewards,
            log_probs,
            hyperparams["gamma"],
            values=values if hyperparams["use_value_network"] else None,
            temporal=hyperparams["temporal"],
            device=hyperparams["device"],
        )

        if step % hyperparams["log_interval"] == 0:
            print(
                f"Episode {step}\tLast reward: {ep_reward:.2f} \tAverage
                    ↪ reward: {running_reward:.2f}"
            )

        if step % hyperparams["step_save"] == 0:
            # Saves model checkpoint
            save_checkpoint(
                model,
                optimizer,
                step,
                save_dir=save_dir / f'{hyperparams["task"]}_pg{hyperparams
                    ↪ ["layer"]}',
            )

    save_checkpoint(
        model,
        optimizer,
        step,
        save_dir=save_dir / f'{hyperparams["task"]}_pg{hyperparams["layer
            ↪ "]}',
    )
    print("Training completed!")
    checkpoint_dir = save_dir / f"{hyperparams['task']}_pg{hyperparams['
        ↪ layer']}"
    print(f"Model checkpoints saved in: {checkpoint_dir}")
    return history_reward


def visual(hyperparams, step="max", max_t=5000, save_dir=SAVE_DIR):
    """Visualize the learned policy by rolling out a trajectory.

    :param hyperparams: Dictionary containing hyperparameters for
        ↪ visualization.
    :param step: Step to load the checkpoint from.
    :param max_t: Maximum number of steps for visualization.
    :param save_dir: Directory to load the model checkpoints from.
    """
    print("Initializing visualization environment...")
    env_test = wrap_env(
        gym.make(hyperparams["task"]), save_dir / f'{hyperparams["task"]}
            ↪ _pg_vis'
    )
    model = PGNetwork(
        env_test.observation_space.shape[0],
        env_test.action_space.n,
        hyperparams["hidden_dim"],
        hyperparams["layer"],
    )

    # Load model checkpoint
    _ = load_checkpoint(
        model,
        step=step,
        save_dir=save_dir / f'{hyperparams["task"]}_pg{hyperparams["layer
            ↪ "]}',
    )

    obs = env_test.reset()
    MAX_T = 5000
    sum_reward = 0
    for t in range(MAX_T):
        env_test.render()
        dist = model(torch.tensor(obs, dtype=torch.float).unsqueeze(dim=0))
        action = dist.logits.argmax().numpy()
        next_obs, reward, done, info = env_test.step(action)
```

```python
        obs = next_obs
        sum_reward += reward

        if done:
            break

    print(f"{t} steps")
    print(f"{sum_reward} reward")
    env_test.close()
    show_video(save_dir / f'{hyperparams["task"]}_pg_vis')
    print(
        f"Visualization completed after {t} steps with a total reward of {sum_reward}."
    )
    video_dir = save_dir / f"{hyperparams['task']}_pg_vis"
    print(f"Visualization video saved in: {video_dir}")


def plot_rewards(reward_histories, labels, title="Training Rewards", save_dir=SAVE_DIR):
    """Plot multiple reward histories and save the plot as a high DPI PDF.

    :param reward_histories: List of reward histories to plot.
    :type reward_histories: list[list[float]]
    :param labels: List of labels for each reward history.
    :type labels: list[str]
    :param title: Title of the plot, defaults to "Training Rewards".
    :type title: str
    :param save_dir: Directory to save the plot, defaults to current directory / "results".
    :type save_dir: Path
    """
    print(f"Plotting {title}...")
    for rewards, label in zip(reward_histories, labels):
        plt.plot(rewards, label=label)

    plt.title(title)
    plt.xlabel("Steps")
    plt.ylabel("Moving Average Reward")
    plt.legend()
    plt.tight_layout()
    plt.grid(False)
    plt.show()

    # Save the plot as a high DPI PDF
    if save_dir:
        save_path = save_dir / f"{title}.pdf"
        plt.savefig(save_path, format="pdf", dpi=300)
        print(f"Plot saved in: {save_path}")

```

```python
# Driver code meant to be run in Jupyter
if __name__ == "__main__":
    # Set global font size
    plt.rcParams["font.size"] = 14

    SAVE_DIR = Path.cwd() / "results"

    # Define hyperparameters in a dictionary
    hyperparams = {
        "device": torch.device("cpu"),  # change to 'cuda' when ready to run on GPU
        "gamma": 0.999,
        "lr": 1e-3,
        "layer": 0,
        "max_episodes": 1001,
        "log_interval": 100,
        "step_save": 100,  # Save the model in every step_save steps
        "use_value_network": False,
        "temporal": False,
        "hidden_dim": 64,
        "running_reward_alpha": 0.05,
        "task": "CartPole-v1",
    }

    # Automate selection of device
    hyperparams["device"] = check_and_set_device()

    # Q4 PART A:
    # Vanilla v.s. Temporal

    # Set the hyperparameters
    layer = 2
    LR = 1e-3
    MAX_EPISODES = 1001

    # Update the hyperparams dictionary with the new values
    hyperparams["layer"] = layer
    hyperparams["lr"] = LR
    hyperparams["max_episodes"] = MAX_EPISODES

    # Train using the Vanilla method
    vanilla_rewards = train(hyperparams)

    # Visualize the Vanilla method
    visual(hyperparams)

    # Train using the Temporal method
    hyperparams["temporal"] = True
    temporal_rewards = train(hyperparams)

    # Visualize the Temporal method
    visual(hyperparams)

    # Plot the rewards
    plot_rewards([vanilla_rewards, temporal_rewards], ["Vanilla", "Temporal
        ↪ "], title="Vanilla vs Temporal Rewards")

    # Observations regarding the convergence and the rewards can be made
        ↪ based on the plotted rewards and the visualizations.
```

Code Listing 1: Q4A Python Code

APPENDIX B: Q4 PART B PYTHON

```python
import os
import math
import random
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.distributions import Categorical
import gym
from gym import spaces
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from IPython import display as ipythondisplay
from tqdm.notebook import trange
import tqdm

# Driver code meant to be run in Jupyter
if __name__ == "__main__":
    # Number of simulations
    NUM_SIMULATIONS = 5

    # Collect rewards for each simulation
    all_rewards_without_baseline = []
    all_rewards_with_baseline = []

    for sim in range(NUM_SIMULATIONS):
        print(f"Running simulation {sim + 1} of {NUM_SIMULATIONS}...")

        # Train without baseline
        print("Training model without baseline...")
        hyperparams["use_value_network"] = False
        rewards_without_baseline = train(hyperparams)
        all_rewards_without_baseline.append(rewards_without_baseline)

        # Train with baseline
        print("Training model with baseline...")
        hyperparams["use_value_network"] = True
        rewards_with_baseline = train(hyperparams)
        all_rewards_with_baseline.append(rewards_with_baseline)

    # Convert rewards to DataFrame for easier calculations
    df_without_baseline = pd.DataFrame(all_rewards_without_baseline).transpose()
    df_with_baseline = pd.DataFrame(all_rewards_with_baseline).transpose()

    # Calculate mean and other statistics
    mean_without_baseline = df_without_baseline.mean(axis=1)
    mean_with_baseline = df_with_baseline.mean(axis=1)

    # Calculate std deviation
    std_without_baseline = df_without_baseline.std(axis=1)
    std_with_baseline = df_with_baseline.std(axis=1)

    # Calculate 95% confidence intervals
    sem_without_baseline = df_without_baseline.sem(axis=1)
    sem_with_baseline = df_with_baseline.sem(axis=1)
    ci95_without_baseline = 1.96 * sem_without_baseline
    ci95_with_baseline = 1.96 * sem_with_baseline

    # Create a summary DataFrame with standardized column labels
    summary_df = pd.DataFrame({
        "mean_without_baseline": mean_without_baseline,
        "mean_with_baseline": mean_with_baseline,
        "std_without_baseline": std_without_baseline,
        "std_with_baseline": std_with_baseline,
        "ci95_without_baseline": ci95_without_baseline,
        "ci95_with_baseline": ci95_with_baseline
    })

    # Print the summary in LaTeX format
    print(summary_df.to_latex())
```

Code Listing 2: Q4B Python Code

APPENDIX B: Q4 PART C PYTHON

```
import os
import math
import random
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.distributions import Categorical
import gym
from gym import spaces
from gym.wrappers import Monitor
```

```python
import glob
import io
import base64
from IPython.display import HTML
from IPython import display as ipythondisplay
from tqdm.notebook import trange
import tqdm
from IPython.display import display, HTML, Video
import cv2
import base64
import io
import gym

def policy_naive(obs):
    """Hand-designed policy for the CartPole-v1 task. The policy is based on
    the  angle and the angular velocity. Specifically:
      if 3 * angle + angular_velocity > 0,
      return action 1;
      otherwise, it returns action 0.

    :param obs: A list containing the observations from the environment. It
                includes [position, velocity, angle, angular_velocity].
    :type obs: list[float]
    :return: The action decided by the policy. Either 0 or 1.
    """
    position, velocity, angle, angular_velocity = obs
    action = int(3. * angle + angular_velocity > 0.)
    return action

def evaluate_policy(policy_function, num_episodes=100):
    """Evaluate a given policy function over multiple episodes."""
    task = 'CartPole-v1'
    env_test = gym.make(task)
    reward_history = []

    for episode in range(num_episodes):
        obs = env_test.reset()
        episode_reward = 0

        while True:
            action = policy_function(obs)
            obs, reward, done, _ = env_test.step(action)
            episode_reward += reward

            if done:
                break

        reward_history.append(episode_reward)

    return reward_history

def save_frames_as_video(frames, filename='video.mp4'):
    """Saves a list of frames as a video file."""
    height, width, layers = frames[0].shape
    size = (width, height)
    out = cv2.VideoWriter(filename, cv2.VideoWriter_fourcc(*'mp4v'), 1,
       ↪ size)
    for i in range(len(frames)):
        out.write(frames[i])
    out.release()

def encode_video(frames, fps=30):
    """Encodes a list of frames into a video using OpenCV."""
    height, width, layers = frames[0].shape
    size = (width, height)
    out = cv2.VideoWriter('video.mp4', cv2.VideoWriter_fourcc(*'mp4v'), fps
       ↪ , size)

    for frame in frames:
        out.write(frame)
    out.release()

    return io.open('video.mp4', 'rb').read()

def display_frames_as_video(frames):
    """Displays a list of frames as a video."""
    video_size = frames[0].shape
    video_bytes = encode_video(frames)
    video_tag = f"""
    <video width="{video_size[1]}" height="{video_size[0]}" controls>
        <source src="data:video/mp4;base64,{base64.b64encode(video_bytes).
           ↪ decode('ascii')}" type="video/mp4">
    </video>"""
    return HTML(video_tag)

# Driver code meant to be run in Jupyter
if __name__ == "__main__":
    # Visualize the learned policy by rolling out a trajectory
    task = 'CartPole-v1'
    env_test = gym.make(task)

    obs = env_test.reset()
    frames = [env_test.render(mode='rgb_array')]

    MAX_T = 5000
    sum_reward = 0
    for t in range(MAX_T):
        action = policy_naive(obs)
        next_obs, reward, done, info = env_test.step(action)
        frames.append(env_test.render(mode='rgb_array'))
        obs = next_obs
        sum_reward += reward

        if done:
            break

    print('%s steps' % t)
    print('%s reward' % sum_reward)

    # Display the captured frames as a video
    display(display_frames_as_video(frames))

    # Save frames as video
    save_frames_as_video(frames, 'video.mp4')

    # Display the video
    Video('video.mp4')


    # Evaluate the hand-designed policy
    hand_designed_reward_history = evaluate_policy(policy_naive)

    # Print average reward achieved by the hand-designed policy
    print("Average reward achieved by the hand-designed policy:", sum(
       ↪ hand_designed_reward_history) / len(hand_designed_reward_history)
       ↪ )
```

Code Listing 3: Q4C Python Code

APPENDIX B: Q4 PART D PYTHON

```
import os
import math
import random
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.distributions import Categorical
import gym
from gym import spaces
from gym.wrappers import Monitor
import glob
import io
import base64
```

```python
from IPython.display import HTML
from IPython import display as ipythondisplay
from tqdm.notebook import trange
import tqdm


class LinearPolicy(PGNetwork):
    """
    A linear policy network that inherits from the PGNetwork class.

    This class represents a simple linear policy network. It overrides the
        ↪ architecture
    of the PGNetwork to provide a single linear layer followed by a softmax
        ↪  activation
    to produce a categorical distribution over actions.

    :param in_dim: Dimension of the input (observation).
    :param out_dim: Dimension of the output (number of actions).
    :param args: Variable length argument list to pass to the parent class.
    :param kwargs: Arbitrary keyword arguments to pass to the parent class.
    """

    def __init__(self, in_dim, out_dim, *args, **kwargs):
        # Call the parent's initializer
        super().__init__(in_dim, out_dim, *args, **kwargs)
        self.fc = nn.Linear(in_dim, out_dim)

    def forward(self, observation):
        """
        Forward pass through the network.

        :param observation: The input observation tensor.
        :return: Categorical distribution over actions based on the network
            ↪ 's output.
        """
        logits = self.fc(observation)
        return Categorical(logits=logits)

# Driver code meant to be run in Jupyter
if __name__ == "__main__":
    # Hyperparameters
    input_dim = 4  # [position, velocity, angle, angular_velocity]
    output_dim = 2  # Action space: [0, 1]
    hyperparams["lr"] = 0.01
    hyperparams["max_episodes"] = 1000
    hyperparams["layer"] = 1

    # Train the model
    reward_history = train(hyperparams)

    # Visualize the learned policy
    visual(hyperparams)

    # Extract learned coefficients
    model = LinearPolicy(input_dim, output_dim)  # Create an instance of
        ↪ the model
    coefficients = list(model.fc.parameters())[0].detach().numpy()
    print("Learned coefficients:", coefficients)

    # Compare with hand-designed policy
    hand_designed_policy = [0, 0, 3, 1]
    print("Difference between learned and hand-designed policy:",
        ↪ coefficients - hand_designed_policy)

    # Print average reward achieved by the linear policy
    print("Average reward achieved by the linear policy:", sum(
        ↪ reward_history) / len(reward_history))

    # Print average reward achieved by the hand-designed policy
    print("Average reward achieved by the hand-designed policy: 500.0")
```

Code Listing 4: Q4D Python Code

References

[1] C. Jin, Z. Yang, Z. Wang, and M. I. Jordan. "Provably efficient reinforcement learning with linear function approximation". In: *Conference on Learning Theory*. PMLR. 2020, pp. 2137–2143.

[2] C. J. C. H. Watkins. "Learning from delayed rewards". In: (1989).

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