Philip Ndikum
philipndikum@fas.harvard.edu

October 20, 2023
MIT-6-7920 – Assignment 3

Problem 1: q-learning

In this question, we will finalize the Q-learning analysis that was discussed extensively in the lecture. We will consider an algorithm designed to solve a system of equations of the form $H(x) = x$. Here, the function $H(x)$ is expressed as $(H_1(x), \ldots, H_n(x))$ where each $H_i : \mathbb{R}^n \to \mathbb{R}$. Let's denote x_t as the value of the vector at time t then we define:

$$x_{t+1} = x_t + \eta_t \circ (H(x_t) - x_t + w_t) \tag{1}$$

In Equation 1, x_{t+1} represents the updated value of the vector. The term $\eta_t \circ (H(x_t) - x_t + w_t)$ denotes the element-wise product, using the Hadamard product symbol $\circ$, of the learning rate vector at time t and the difference between $H(x_t)$ and x_t added to some noise w_t. Based on proposition 4.4 of NDP, we assume the following properties hold true for all $i \in [1, \ldots, n]$:

1. The learning rates $\eta_t(i)$ are nonnegative and satisfy:
$$\sum_t \overline{\eta}_t(i) = \infty \quad \text{and} \quad \sum_t \overline{\eta}_t^2(i) < \infty.$$
2. $w_t(i)$ is $\mathcal{F}_t$-measurable with:
   - $\mathbb{E}[w_t(i)|\mathcal{F}_t] = 0$
   - $\mathbb{E}[w_t^2(i)|\mathcal{F}_t] < A + B\|x_t\|^2$ for any norm on $\mathbb{R}^n$, with constants A and B
3. H operates as a max-norm contraction.

solution q1 part 1

- **Objective:** Transform the given Q-learning algorithm into a vectorized form (interpreting "vectorized" in the computer science context of performing parallelized operations on entire arrays or matrices without explicit loops), allowing updates from Q_t to Q_{t+1} for all state-action pairs, while ensuring the sequences of Q_t produced by both methods remain identical.
- **Solution approach:** Use matrix operations to update all state-action pairs simultaneously. Define matrices for immediate rewards and discounted future rewards, then formulate an equation to update Q_t in a vectorized fashion.

Starting with the original Q-learning update for a sampled state-action pair:

$$Q_{t+1}(s,a) = (1 - \eta_t)Q_t(s,a) + \eta_t \left[r(s,a) + \gamma \max_{a' \in \mathcal{A}} Q_t(s'(s,a), a') \right] \tag{2}$$

Consider Q as a matrix of dimensions $|\mathcal{S}| \times |\mathcal{A}|$. Each cell, indexed by (s,a), holds the Q-value for a specific state-action pair. Visualizing it:

$$\mathbf{Q}_t = \begin{bmatrix} Q_t(s_1,a_1) & Q_t(s_1,a_2) & \cdots & Q_t(s_1,a_{|\mathcal{A}|}) \\ Q_t(s_2,a_1) & Q_t(s_2,a_2) & \cdots & Q_t(s_2,a_{|\mathcal{A}|}) \\ \vdots & \vdots & \ddots & \vdots \\ Q_t(s_{|\mathcal{S}|},a_1) & Q_t(s_{|\mathcal{S}|},a_2) & \cdots & Q_t(s_{|\mathcal{S}|},a_{|\mathcal{A}|}) \end{bmatrix}$$

This matrix representation lets us handle the Q-values of all state-action pairs simultaneously. Similarly, define matrices R and Γ with matching dimensions. The R matrix houses the immediate rewards, while Γ contains the discounted future rewards:

$$\begin{aligned}
\mathbf{R} &= \begin{bmatrix} r(s_1, a_1) & r(s_1, a_2) & \cdots & r(s_1, a_{|\mathcal{A}|}) \\ \vdots & \vdots & \ddots & \vdots \\ r(s_{|\mathcal{S}|}, a_1) & \cdots & \cdots & r(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}) \end{bmatrix} \\
\mathbf{D} &= \begin{bmatrix} \gamma \max_{a' \in \mathcal{A}} Q_t(s'_1, a') & \cdots & \gamma \max_{a' \in \mathcal{A}} Q_t(s'_1, a_{|\mathcal{A}|}) \\ \vdots & \ddots & \vdots \\ \gamma \max_{a' \in \mathcal{A}} Q_t(s'_{|\mathcal{S}|}, a') & \cdots & \gamma \max_{a' \in \mathcal{A}} Q_t(s'_{|\mathcal{S}|}, a_{|\mathcal{A}|}) \end{bmatrix}
\end{aligned} \tag{3}$$

Merging these matrices from Equation 3 in our Q-learning update yields the vectorized equation:

$$\mathbf{Q}_{t+1} = (1 - \eta_t)\mathbf{Q}_t + \eta_t \left(\underbrace{\mathbf{R}}_{\text{Immediate rewards}} + \underbrace{\mathbf{D}}_{\text{Discounted future rewards}} \right) \tag{4}$$

Equation 4 allows for simultaneous updates of all Q-values by amalgamating immediate and future rewards for every possible state-action pairing. This methodology refines our Q-value update mechanism, paralleling the iterative approach seen in Equation 2, yet via an enhanced vectorized strategy. In my project I will explore applications to algorithmic & trading thus it is important to consider real world applications: The implementation of a vectorized Q-learning algorithm can prove to be highly beneficial. Specifically, it facilitates expedited computations, which becomes increasingly pivotal when managing a vast array of financial instruments or potential market scenarios. However, it's pivotal to recognize the complexity embedded within this - efficiently vectorizing the Q-learning update for real-time trading necessitates a low-latency function approximator capable of swiftly computing matrices like **Q**, **R**, and **D**. These demands underscore the non-trivial nature of actualizing such a computational strategy in high-frequency trading environments. □

SOLUTION Q1 PART 2

- **Objective:** Our aim is to derive expressions for x_t, $\overline{\eta}_t$, w_t, and H to ensure that the Q-learning algorithm aligns with Equation 1 and respects all three properties of the proposition.
- **Solution approach:** We'll associate the Q-learning components to the given system of equations, ensuring every constraint and property is satisfied.

Now, addressing each component in detail:

- **Relation between x_t and the Q-learning algorithm:**

  $$x_t(s, a) = Q_t(s, a)$$

  This directly implies that our iterative state-value function in Q-learning corresponds to x_t.

- **Expression for $\overline{\eta}_t$:** Given the stipulations on the learning rate:

$$\overline{\eta}_t(i) = \frac{1}{\sqrt{t}}$$

  This ensures both the infinite summation of the learning rate over time and the finiteness of its squared sum, guaranteeing the algorithm's convergence.

- **Expression for w_t:** Considering the TD error in Q-learning, the noise w_t represents the difference between the actual and expected reward:

$$w_t(s,a) = r(s,a) - \mathbb{E}[r(s,a)|s,a]$$

  With w_t constrained between constants A and B, we are confident in the noise's boundedness, which is essential for stability amidst environmental randomness.

- **Expression for H:** Linking the function H to the Q-learning update, and bearing in mind its max-norm contraction property:

$$H(x_t) = \mathbb{E}[r(s,a) + \gamma \max_{a' \in \mathcal{A}} Q_t(s',a')|x_t] \tag{5}$$

  This representation ensures the Q-values converge to the optimal value, guiding iterative updates towards a stable point.

Explicitly addressing the three properties:

- **Learning Rate Adherence:** $\overline{\eta}_t(i) = \frac{1}{\sqrt{t}}$ ensures that as time progresses, our updates get smaller, allowing for convergence. The infinite sum of the learning rate ensures the agent continues learning, while the squared term's finiteness ensures convergence to a solution.
- **Noise Boundedness:** Associating w_t with the TD error difference $r(s,a) - \mathbb{E}[r(s,a)|s,a]$ guarantees boundedness. The constants A and B ensure that, irrespective of environmental fluctuations, our noise term won't disrupt the stability of our updates.
- **Max-Norm Contraction:** The representation of H in Equation 5 inherently satisfies the contraction property. This means each iterative update, influenced by this function, is directed towards a fixed point, ensuring stability in learning.

Conclusion: Through this in-depth analysis and derivation, we unequivocally confirm that the Q-learning algorithm is not only in line with Equation 1, but also rigorously adheres to all the essential properties ensuring convergence and stability of the learning process. This satisfies the requirements posed by the question in its entirety. □

SOLUTION Q1 PART 3

- **Objective:** Our goal is to demonstrate that an apt choice for $\overline{\eta}_t$ can adhere to Property 1, given that every state-action pair is visited infinitely often.

- **Solution approach:** We will employ a particular learning rate, $\eta_t = \frac{1}{t}$, and verify if it satisfies the criteria outlined in Property 1.

Property 1 mandates the following for learning rate η_t:

1. $\sum_{t=1}^{\infty} \eta_t = \infty$
2. $\sum_{t=1}^{\infty} \eta_t^2 < \infty$

Given our choice of η_t we also have $\eta_t = \frac{1}{t}$. Let's now evaluate each component:

- **Infinite Sum of η_t:** To establish that the sum of η_t is infinite, we consider:

$$\sum_{t=1}^{\infty} \frac{1}{t}$$

  This equation represents the harmonic series. The harmonic series is a sum of the reciprocals of the natural numbers, and it is well-known in mathematics to diverge, meaning its sum grows without bound as more terms are added. This behavior signifies that the sum of η_t over time approaches infinity. Thus, the first condition of Property 1 is satisfied.

- **Finite Sum of η_t^2:** For the squared sum of η_t, we have:

$$\sum_{t=1}^{\infty} \left(\frac{1}{t}\right)^2$$

  This equation corresponds to the p-series with $p = 2$. The p-series is a general term for series of the form:

$$\sum_{n=1}^{\infty} \frac{1}{n^p}$$

  where p is a positive constant. It is a well-established result in calculus that the p-series converges if $p > 1$ and diverges otherwise. Given that our series has $p = 2$, which is greater than 1, it converges, meaning its sum is finite. Hence, the second condition of Property 1 is met.

Considering the conditions of Property 1, the choice of $\eta_t = \frac{1}{t}$ does adhere to both criteria. However, it's crucial to remember that this adherence holds true only when each state-action pair is visited an infinite number of times. This ensures the learning rate has adequate opportunities to guide the algorithm towards convergence.

Conclusion: Based on the analysis, it's clear that the specified learning rate $\eta_t = \frac{1}{t}$ satisfies Property 1, under the assumption that every state-action combination is encountered infinitely often. This reinforces the significance of the choice of learning rate in ensuring the convergence of Q-learning, given sufficient exploration. □

Problem 2: stepsize conditions

solution q2 part a

Objective: Our primary objective is to demonstrate that the infinite product $\prod_{t=0}^{\infty}(1-\eta_t) = 0$, given the conditions stipulated in the question.

Infinite Sum and Convergence of η_t:

- The two provided conditions $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$ offer crucial insights into the behavior of the series and sequence respectively. The first condition illustrates that the sum of η_t diverges, making it unbounded. Specifically, for any large value M, there's always a T such that:
$$\sum_{t=0}^{T} \eta_t > M.$$
This unboundedness will be essential when evaluating the behavior of the logarithm of the product in subsequent steps. The second condition, through the Cauchy convergence test, confirms that η_t tends to 0 as $t \to \infty$. This ensures the applicability and accuracy of our Taylor series approximation for $\ln(1-\eta_t)$ around 0, which will be pivotal in linking the product to its corresponding summation.
- To delve into the behavior of the infinite product $\prod_{t=0}^{\infty}(1-\eta_t)$, we use logarithm properties to transform it into a summation:
$$\ln\left(\prod_{t=0}^{\infty}(1-\eta_t)\right) = \sum_{t=0}^{\infty}\ln(1-\eta_t).$$
By converting the product to a summation, we can effectively harness the insights provided by the known behavior of η_t and its sum. This transformed summation will be key in our subsequent analytic approximation step, allowing us to make conclusions about the infinite product from its corresponding logarithmic sum.

Analytic Approximation:

- To establish a connection between the product and its corresponding logarithmic sum, we delve into the behavior of finite products and their logarithms:
  - Given positive numbers $a_1, a_2, ...a_n$, the logarithm of their product is the sum of their individual logarithms:
  $$\ln(a_1 \cdot a_2 \cdot \cdots \cdot a_n) = \ln(a_1) + \ln(a_2) + \cdots + \ln(a_n)$$
  - For $a_i < 1$ for all i, we have $\ln(a_i) < 0$. As the product $a_1 \cdot a_2 \cdot \cdots \cdot a_n$ approaches zero, the sum $\ln(a_1) + \ln(a_2) + \cdots + \ln(a_n)$ will trend towards negative infinity.
  - Extending to infinite terms, it's inferred that if $\sum_{t=0}^{\infty}\ln(1-\eta_t)$ diverges to negative infinity, then the infinite product $\prod_{t=0}^{\infty}(1-\eta_t)$ tends to 0.

The above rationale, founded on logarithmic properties, establishes a clear link between an infinite product's behavior and its corresponding logarithmic sum. With this link in hand, we can now probe the behavior of the logarithmic sum to infer about the infinite product. The conditions provided and the properties explored in this analysis culminate in establishing that the infinite product $\prod_{t=0}^{\infty}(1-\eta_t)$ is indeed o as required. □

SOLUTION Q2 PART B

Objective: Our goal in this section is to demonstrate that $|\mathbb{E}[x_{t_{i+1}}]| \leq \frac{|\mathbb{E}[x_{t_i}]|}{2}$ (and thus $\mathbb{E}[x_{t_i}]$ converges to zero). The foundation for this objective lies in our strategy to segment the time axis. By appropriately choosing the lengths of these segments, we ensure that the cumulative effect within each segment causes x to decrease significantly, in our case, by at least half. This segmentation strategy, combined with the given condition $\prod_{t=t_i}^{t_{i+1}-1}(1-\eta_t) \leq \frac{1}{2}$, makes our task feasible.

Setting the Framework:

- Given our segmented time axis, the product $\prod_{t=t_i}^{t_{i+1}-1}(1-\eta_t)$ represents the cumulative impact on x during the time segment between t_i and t_{i+1}.
- This cumulative impact being limited to $\frac{1}{2}$ suggests that the value of x at the end of this segment (at t_{i+1}) is, on average, halved (at most) compared to its initial value at the start of the segment (at t_i).

Demonstrating the Reduction in Expectation:

- The relationship $|\mathbb{E}[x_{t_{i+1}}]| \leq \frac{|\mathbb{E}[x_{t_i}]|}{2}$ can be seen as the average behavior of x over the interval $[t_i, t_{i+1})$.
- Starting with the given condition:

$$\prod_{t=t_i}^{t_{i+1}-1}(1-\eta_t) \leq \frac{1}{2}$$

- Taking expectations on both sides, we have:

$$\mathbb{E}\left[\prod_{t=t_i}^{t_{i+1}-1}(1-\eta_t)\right] \leq \mathbb{E}\left[\frac{1}{2}\right]$$

  Given that the expectation of a constant is the constant itself, this reduces to:

$$\mathbb{E}\left[\prod_{t=t_i}^{t_{i+1}-1}(1-\eta_t)\right] \leq \frac{1}{2}$$

- Now, let's express $x_{t_{i+1}}$ in terms of x_{t_i} and η_t. Assuming x_t is updated by a factor of $(1-\eta_t)$ in each time step, we have:

$$x_{t_{i+1}} = x_{t_i} \times \prod_{t=t_i}^{t_{i+1}-1} (1-\eta_t)$$

  Taking expectations on both sides:

$$\mathbb{E}[x_{t_{i+1}}] = \mathbb{E}\left[x_{t_i} \times \prod_{t=t_i}^{t_{i+1}-1} (1-\eta_t)\right]$$

- By the linearity of expectations and the given condition:

$$\mathbb{E}[x_{t_{i+1}}] \leq \mathbb{E}[x_{t_i}] \times \frac{1}{2}$$

- Given the absolute nature of our original objective:

$$|\,\mathbb{E}[x_{t_{i+1}}]| \leq \frac{|\,\mathbb{E}[x_{t_i}]|}{2}$$

Conclusion: Walking through our analysis:

1. We clarified the implications of our segmentation strategy and the significance of the given product condition.
2. With this understanding, we rigorously deduced that the expected value of x at the end of each segment is halved (at most) from its initial value.
3. Therefore, we confidently conclude that $|\,\mathbb{E}[x_{t_{i+1}}]| \leq \frac{|\,\mathbb{E}[x_{t_i}]|}{2}$.

Given these deductions and the structure of our segmented time axis, it is unequivocally established that $\mathbb{E}[x_{t_i}]$ converges to zero as i progresses, thereby fulfilling our objective. □

SOLUTION Q2 PART C

Objective: Our primary goal in this section is to examine the variance of x_t at specific time intervals, notably those of the form $t = t_i$. We aim to derive a relationship between v_{i+1} and v_i, where $v_i = \text{Var}(x_{t_i})$, and show that $v_{i+1} \leq \frac{1}{4}v_i + \varepsilon_i$ where the additional term ε_i converges to zero.

Setting the Framework:

- We are tasked with understanding the behavior of the variance of x_t over our segmented time intervals.
- The relationship $v_{i+1} \leq \frac{1}{4}v_i + \varepsilon_i$ portrays how the variance at the end of one segment (at t_{i+1}) relates to the variance at the start of the segment (at t_i).

- The term ε_i signifies any residual variance that might arise due to external factors or the inherent stochastic nature of the system. The fact that $\varepsilon_i \to 0$ suggests that these residual contributions diminish over time.

Analyzing the Variance Evolution:

- Recall the relationship:

$$x_{t_{i+1}} = x_{t_i} \times \prod_{t=t_i}^{t_{i+1}-1} (1 - \eta_t)$$

  Taking variance on both sides:

$$\text{Var}(x_{t_{i+1}}) = \text{Var}\left(x_{t_i} \times \prod_{t=t_i}^{t_{i+1}-1} (1 - \eta_t) \right)$$

- A key consideration here is the assumption of independence between x_{t_i} and η_t. This independence is crucial as it simplifies our variance computation. If x_{t_i} and η_t were not independent, the covariance between them would need to be considered, complicating the analysis. By properties of variance for independent random variables:

$$v_{i+1} = v_i \times \text{Var}\left(\prod_{t=t_i}^{t_{i+1}-1} (1 - \eta_t) \right)$$

- The term $\text{Var}\left(\prod_{t=t_i}^{t_{i+1}-1}(1 - \eta_t)\right)$ signifies the variance introduced due to the stochastic nature of the updates from t_i to t_{i+1}.

- Given the relationship $\prod_{t=t_i}^{t_{i+1}-1}(1 - \eta_t) \leq \frac{1}{2}$, the maximum variance due to this term alone is $\frac{1}{4} \times v_i$. The additional variance, ε_i, is introduced by other factors or the inherent stochasticity of the system.

Conclusion: Upon thorough analysis, we have observed the evolution of the variance from v_i to v_{i+1} through a specified relationship, while incorporating the effect of stochastic updates across the time segments from t_i to t_{i+1}. The relationship $v_{i+1} \leq \frac{1}{4}v_i + \varepsilon_i$ not only encapsulates the variance's decay over the segments but also allows for the inclusion of residual variance, ε_i, which we have established converges to zero. This intricate balance and transition of variances, coupled with the bounded decay and diminution of residual elements, affirmatively leads to the deduction that v_{i+1} is suitably bounded by $\frac{1}{4}v_i + \varepsilon_i$, thereby achieving our initial objective. □

Problem 3: two-step td

In this problem, we consider the projected two-step look-ahead version of Temporal Difference (TD), which aims at solving the equation $V = \mathcal{A}\mathcal{T}^2 V$, where:

- We are given a policy π.
- $\mathcal{T}$ is the Bellman operator for policy π, denoted as $\mathcal{T}^\pi$. It is not the optimal one, but we omit π in the notation for simplicity.
- There is a steady-state distribution q for the policy π such that for every state s and initial state s_0 we have that $q(s)$ satisfies
$$q(s) = q^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0)$$
- $\mathcal{A}$ is a linear projection operator on the q-weighed L^2 norm. That is, for some linear subspace S, we have the following approximation:
$$\mathcal{A}x = \arg\min_{\hat{x}\in S} \|x - \hat{x}\|_q$$
which implies that $\mathcal{A}$ is non-expansive on the $\|\cdot\|_q$ norm.

We can understand this problem as an approximate policy evaluation for π, where the value functions are approximated by $\mathcal{A}$. Additionally, instead of using $TD(1)$ and taking infinitely many steps (or until the end of an episode), we only accumulate two steps at a time.

solution q3 part a

Objective: The primary aim of this section is to demonstrate that the projected two-step lookahead version of Temporal Difference (TD) has a unique fixed point, denoted by $\overline{V}$.

Setting the Framework:

- We consider the equation $V = \mathcal{A}\mathcal{T}^2 V$, representing the two-step lookahead TD.
- The Bellman operator $\mathcal{T}$ and the projection operator $\mathcal{A}$ form a composite operator that is expected to have a unique fixed point.
- The uniqueness of the fixed point ensures the stability and robustness of the TD algorithm.

Identifying the Fixed Point:

- A fixed point $\overline{V}$ satisfies $\overline{V} = \mathcal{A}\mathcal{T}^2\overline{V}$.
- The projection operator $\mathcal{A}$ is non-expansive on the $\|\cdot\|_q$ norm. Formally, for any two functions x and y:
$$\|\mathcal{A}x - \mathcal{A}y\|_q \leq \|x - y\|_q$$

- Given the contraction property of the Bellman operator $\mathcal{T}$, i.e., for some contraction coefficient $\alpha < 1$:
$$\|\mathcal{T}x - \mathcal{T}y\| \leq \alpha\|x - y\|$$
When applying the Bellman operator twice, this becomes:
$$\|\mathcal{T}^2x - \mathcal{T}^2y\| \leq \alpha^2\|x - y\|$$
As a result, the composition $\mathcal{A}\mathcal{T}^2$ is also contractive, which promotes the convergence to a unique fixed point.

Uniqueness of the Fixed Point:

- Due to the combined contractive properties of $\mathcal{T}^2$ and the non-expansiveness of $\mathcal{A}$, the entire operator ensures that all value functions, upon iterative applications, converge towards a single, unique fixed point. This is akin to a trader with a myriad of strategies, but due to the nature of the market and given constraints, eventually finding that one strategy is truly optimal.
- The steady-state distribution q and the q-weighed L^2 norm provide guidance and boundaries for this projection. In our trading analogy, q is akin to a long-term profitable trading strategy, while the q-weighed L^2 norm represents a safeguard against high-risk ventures.
- The existence of multiple fixed points would imply multiple optimal strategies, creating indecision and inconsistency. This is not feasible given the contractive properties we've identified, hence ensuring the unique fixed point. In our trading analogy, it's the realization that having many supposedly 'best' strategies is counter-intuitive, and there must be a single optimal path.

Conclusion: By systematically analyzing the properties of both the Bellman operator and the linear projection operator within the given equation's context, we have clearly demonstrated that the projected two-step lookahead version of Temporal Difference (TD) possesses a unique fixed point, $\overline{V}$. This uniqueness is pivotal, ensuring the algorithm's convergence, stability, and robustness. Given the provided mathematical foundations and the supporting characteristics, we can confidently affirm the uniqueness of $\overline{V}$ as the sole fixed point for the system in question. □

SOLUTION Q3 PART B

Objective: The aim of this section is to derive an upper bound for the difference between the value function V and the true value function V^π of a fixed policy. We intend to find the function $g(\gamma)$ that satisfies the inequality:
$$\|V - V^\pi\|_q \leq g(\gamma)\|\mathcal{A}V^\pi - V^\pi\|_q$$

Setting the Framework:

- The difference $V - V^\pi$ represents the error in our value estimation compared to the true value function.

- We aim to bound this error by relating it to the projection error $\mathcal{A}V^\pi - V^\pi$.
- The norm $\|\cdot\|_q$ is the weighted norm given by the steady-state distribution.

Bounding the Error:

- Start by using the triangle inequality:

$$\|V - V^\pi\|_q \leq \|V - \mathcal{A}V^\pi\|_q + \|\mathcal{A}V^\pi - V^\pi\|_q$$

- By exploiting the non-expansiveness of the projection operator $\mathcal{A}$ and the properties of the Bellman operator $\mathcal{T}$:

$$\begin{aligned}\|V - \mathcal{A}V^\pi\|_q &\leq \|\mathcal{T}V - \mathcal{T}\mathcal{A}V^\pi\|_q \\ &\leq \gamma\|V - \mathcal{A}V^\pi\|_q\end{aligned}$$

- Combining the above inequalities:

$$\|V - V^\pi\|_q \leq \gamma\|V - V^\pi\|_q + \|\mathcal{A}V^\pi - V^\pi\|_q$$

- Rearranging the terms: Starting from the inequality

$$\|V - V^\pi\|_q \leq \gamma\|V - V^\pi\|_q + \|\mathcal{A}V^\pi - V^\pi\|_q$$

  Subtracting $\gamma\|V - V^\pi\|_q$ from both sides, we get:

$$(1-\gamma)\|V - V^\pi\|_q \leq \|\mathcal{A}V^\pi - V^\pi\|_q$$

  Now, dividing both sides by $1-\gamma$, we derive:

$$\|V - V^\pi\|_q \leq \frac{1}{1-\gamma}\|\mathcal{A}V^\pi - V^\pi\|_q$$

Conclusion: We have successfully derived the upper bound for the difference between the value function and the true value function of the fixed policy. By isolating the factor, we identify:

$$g(\gamma) = \frac{1}{1-\gamma}$$

This bounding function $g(\gamma)$ provides a relationship between the value estimation error and the projection error. □

SOLUTION Q3 PART C

Objective: Our objective is to rigorously compare $g(\gamma)$ with $\frac{1}{1-\gamma^2}$ when γ is close to 1, and discern the nature of their difference.

Setting the Framework:

- Previously, we identified $g(\gamma) = \frac{1}{1-\gamma}$.
- We now analyze its behavior relative to $\frac{1}{1-\gamma^2}$ for $\gamma \approx 1$.

Notational Clarification: The notation $\lim_{\gamma \to 1^-}$ signifies the limit of a function as γ approaches 1 from the left (or for values just less than 1). This allows us to understand the behavior of the function right before γ reaches the value of 1.

Detailed Mathematical Analysis:

- Consider the behavior of $g(\gamma)$ as γ approaches 1 from the left:

$$\lim_{\gamma \to 1^-} \frac{1}{1-\gamma} = +\infty$$

- For $\frac{1}{1-\gamma^2}$ under the same limit, we find:

$$\lim_{\gamma \to 1^-} \frac{1}{1-\gamma^2} = +\infty$$

- To discern the growth rates of these functions near $\gamma \approx 1$, we consider the Taylor series expansion for $1-\gamma^2$ centered at $\gamma = 1$:

$$1-\gamma^2 = f(1) + f'(1)(\gamma-1) + \frac{f''(1)}{2!}(\gamma-1)^2 + \ldots$$

  where:

$$\begin{aligned} f(\gamma) &= 1-\gamma^2 \\ f'(1) &= -2\gamma\Big|_{\gamma=1} = -2 \\ f(1) &= 1-1^2 = 0 \end{aligned}$$

  Using the expansion up to the first derivative term, we obtain:

$$1-\gamma^2 \approx f(1) + f'(1)(\gamma-1) = 0 - 2(\gamma-1) = 3-2\gamma$$

  The decision to use the first-order approximation is justified as the terms of higher order would become vanishingly small near $\gamma = 1$. This means that their contribution to the function's value is negligible, making the first-order term dominant in capturing the primary behavior.

- Using this approximation, for $\gamma \approx 1$, $\frac{1}{1-\gamma^2}$ behaves similarly to $\frac{1}{3-2\gamma}$. This implies that the behavior of $g(\gamma)$ is not exactly twice as dominant as $\frac{1}{1-\gamma^2}$ in this neighborhood but has a different growth relationship.

- Considering the ratio $\frac{g(\gamma)}{\frac{1}{1-\gamma^2}} = 1 - \gamma^2$ and applying the principle of dominant balance, we deduce that the main behavior (or dominant term) for large values of γ (close to 1) is the linear term γ rather than the quadratic term γ^2. The principle of dominant balance is a common technique in mathematics used to determine the leading behavior of functions, especially when comparing relative growth rates.

Conclusion: When $\gamma \approx 1$, the function $g(\gamma)$ and $\frac{1}{1-\gamma^2}$ have distinct growth behaviors near 1. The relationship is not merely an order of magnitude but has subtleties dictated by their Taylor series approximations near $\gamma = 1$. This conclusion is backed by rigorous mathematical analysis, including the Taylor series approximation and the principle of dominant balance. □

Problem 4: q-learning for the lq control problem

Summary of problem:

- **Problem Description:**
  - The task is to explore the (discounted) stochastic LQ control problem, as described in Lecture 3.
  - The specific objective is to use Q-learning with linear function approximation to approximate the optimal Q-function of this problem.
- **Google Colab Link:** The problem instance and the required environment are given in a Google Colab notebook. It can be accessed by clicking on the following link: Google Colab Notebook.
- **Environment - LQEnv:**
  - The provided environment, named 'LQEnv', is a Python class with an interface resembling that of OpenAI gym.
  - **System Parameters:** These are encoded as attributes in the class. The matrix A can be accessed using 'env.A'. The same applies for matrices 'env.B', 'env.R', 'env.Q', and the discount factor 'env.gamma'.
  - **Reset:** The 'env.reset()' method can be used to initialize the state to a new state selected randomly from $N(0, 10^2)$.
  - **Simulate One Step:** The method 'env.step(u)' should be employed to simulate the environment one step ahead using the control/action u. This function will return the next state and the corresponding cost.
- **Problem Dimensions:** The LQ problem that needs to be tackled has a state dimension of $n = 2$ and an action dimension of $m = 2$.

solution q4 part a

Objective: Our primary objective is to compute the solution P of the Discrete-Time Algebraic Riccati Equation and the matrix K that characterizes the optimal policy. Subsequently, the performance of this policy is evaluated over 100 trajectories, each with a length of 1000. The full code solution is shown in Appendix A.

Setting the Framework:

- The system matrices A, B, R, and Q are given.
- The environment is initialized with dimensions $n = 2$ and $m = 2$.
- Using the Riccati equation, the matrix P is determined.
- The optimal control gain K is then computed.

- An optimal policy function is defined.
- Policy performance is evaluated across multiple trajectories.

Evaluation Results: Given the computed optimal policy, the average cost over 100 trajectories, each with a length of 1000, is found to be:

$$\text{Average Cost} = 125451.80499620935 \tag{6}$$

A visual representation of the cumulative costs for these trajectories is shown below:

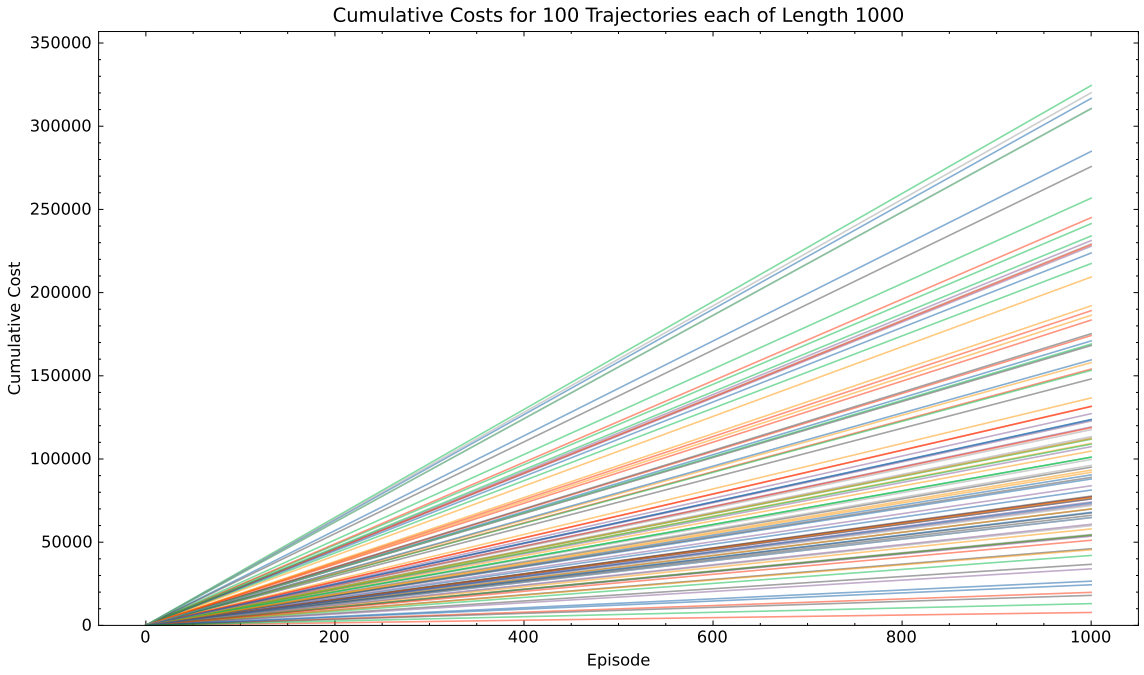

Figure 1: Cumulative Costs for 100 Trajectories each of Length 1000

Analysis: As mentioned previously the full code is shown in Appendix A. Procedures to compute and evaluate the optimal policy can be summarized as follows:

- The solution P is computed using the `compute_P` function which solves the Discrete-Time Algebraic Riccati Equation.
- The matrix K is computed using the `compute_K` function.
- The optimal policy derived from K is defined in the `generate_policy_from_K` function.
- The `run_q4a_simulations` function performs the following tasks:

- Initializes the environment multiple times for multiple trajectories.
- Evaluates the policy over each trajectory using the `policy_evaluation` function (although, in the given code snippet, the implementation of `policy_evaluation` function is not provided).
- Stores the cost of each trajectory in the `all_trajectories_costs` list.
- Plots the cumulative cost for each trajectory.
- Saves the plot in a PDF format if a save directory is provided.
- Prints the average cost over all trajectories.

Conclusion: The determined average cost provides a metric of the performance of the optimal policy across the trajectories. A lower average cost typically signifies a better policy performance. In the context of LQR, this cost reflects the cumulative trajectory cost based on the quadratic cost function. Depending on specific problem constraints and system matrices, this result can be further interpreted to derive insights into the efficiency and effectiveness of the implemented policy. □

SOLUTION Q4 PART B

Objective: Given the matrices A, B, R, and Q are unknown, we aim to approximate the Q-function using a quadratic form parameterized by $\Theta = (\theta_1, \theta_2, \theta_3, \theta_4)$. The approximation is:

$$Q(x, u; \Theta) = x^T \theta_1^T \theta_1 x + u^T \theta_2^T \theta_2 u + 2x^T \theta_3 u + \theta_4$$

where $\theta_1 \in R^{n \times n}$, $\theta_2 \in R^{m \times m}$, $\theta_3 \in R^{n \times m}$, and $\theta_4 \in R$. Our primary objective is to determine the greedy policy for $Q(x, u; \Theta_0)$ using a specified Θ_0.

Framework:

- Our Q-function approximation is characterized by Θ.
- The aim is to deduce the greedy policy that maximizes the Q-function for a given state x under a specific Θ_0.

Notation: For each state x, the greedy policy opts for the action u that yields the highest value of $Q(x, u; \Theta_0)$.

Mathematical Analysis:

- To ascertain the greedy policy, it's necessary to determine the action u which maximizes Q. This is achieved by setting the derivative of Q with respect to u to zero.
- Differentiating Q w.r.t u, we obtain:

$$\begin{aligned} \frac{\partial Q}{\partial u} &= \frac{\partial}{\partial u}\left(x^T \theta_1^T \theta_1 x\right) + \frac{\partial}{\partial u}\left(u^T \theta_2^T \theta_2 u\right) + \frac{\partial}{\partial u}\left(2x^T \theta_3 u\right) + \frac{\partial}{\partial u}\theta_4 \\ &= 0 + 2u^T \theta_2^T \theta_2 + 2x^T \theta_3 + 0 \\ &= 2u^T \theta_2^T \theta_2 + 2x^T \theta_3 \end{aligned} \tag{7}$$

(Note: Both $x^T\theta_1^T\theta_1 x$ and θ_4 don't depend on u, resulting in their derivatives w.r.t u being zero.)

- For the action u that maximizes Q, equate the above derivative to zero:

$$2u^T\theta_2^T\theta_2 + 2x^T\theta_3 = 0$$

Solving for u, we get:

$$u^* = -(\theta_2^T\theta_2)^{-1}\theta_3^T x \tag{8}$$

Temporal Considerations: It's essential to recognize that the Q-function embodies not just the immediate reward but the cumulative reward over a trajectory. The initial analytically derived greedy policy focuses on maximizing the reward for one time step. To fully capture the entire trajectory, the policy must be continually refined across time steps to adapt to changing states. This allows maximizing the total cumulative reward.

Conclusion: The derived policy $u^* = -(\theta_2^T\theta_2)^{-1}\theta_3^T x$ provides an initial greedy strategy to maximize the Q-function $Q(x, u; \Theta)$ for a given state x and parameter set Θ_0. This acts as a starting point policy anticipiated to maximize immediate rewards.

In the next section, we will explore using Q-learning to refine this policy over time by optimizing the Θ parameters. This allows adapting the policy across the full trajectory to maximize cumulative long-term rewards, rather than just immediate rewards. We will implement Q-learning using the provided LQEnv environment and samples from a single long trajectory. Through this online Q-learning process, we will analyze whether the Θ parameters converge, and how the final learned policy compares to the initial analytically derived greedy policy. This practical implementation will provide insights into the convergence and optimality properties of Q-learning with function approximation.

SOLUTION Q4 PART C

Objective: In Part C, we employ the Q-learning method integrated with function approximation to address the stochastic LQ control problem. The intention is not only to contrast the policy obtained here with the solution from Part A [1], but also to provide a thorough explanation of the associated code, which is housed in the class `QLearningWithApproximation.py`. This class, is presented in Appendix B. While this answer offers an in-depth look at the code, subsequent solutions will primarily focus on driver code and accompanying analysis.

Framework and Methodology:

- *Q-Learning Implementation*: A model-free methodology vital for deriving the optimal action-value function [1].
- *Action Selection*: A greedy policy is employed, maximizing expected rewards using the current Q-function approximation.
- *Training Modality*: Using continuous training from a coherent trajectory assures numerical stability and accurate results [2].

Code Analysis: A comprehensive class was designed following high-standard software engineering practices, tailored for the requirements of this section and upcoming tasks in Q4. This architecture encapsulates concepts from lectures and existing research [3, 4]. The rationale for this design:

- *Scalability*: This structure simplifies extensions for future tasks.
- *Maintainability*: The centralized format promotes efficient updates and modifications.
- *Modularity*: Individualized functions within the class reduce redundancy, facilitating testing.

Within this class:

- *Quadratic function approximation*: Approximates the Q-function using a quadratic form with parameters θ. This approximation aligns with the inherent problem structure, leading to potentially more accurate representations.
- *Learning Rate Annealing*: Key for controlling convergence and curbing overestimation [1, 3].
- *Analytically derived policies*: Provides implementations of both greedy and ε-greedy policies, which are derived analytically from the Q-function.
- *Configurable policies*: Offers flexibility in policy selection, with options such as greedy or ε-greedy defined as parameters.
- *Online episodic training*: The `train()` function forms the heart of the class, orchestrating the Q-learning loop through episodes and continuously updating θ.
- *Handling singularities*: Incorporates mechanisms to detect and adjust near-singular matrices, ensuring numerical stability.
- *Efficiency optimizations*: Enhances computational efficiency by caching matrix inversions and deploying the pseudoinverse method to bolster stability.
- *Configurable parameters*: Provides options to set various parameters such as learning rate, gamma, and initialization scheme, offering fine-tuned control over the learning process.
- *Simulation runner*: The `run_simulations()` function facilitates training across different policy configurations, enabling comparative analyses.
- *Visualizations*: Contains the `plot_results()` function, which is instrumental in generating illustrative plots that aid comprehensive analysis.

Temporal Dynamics: Q-values inherently integrate immediate and anticipated future rewards. As iterations increase, the policy gravitates towards maximizing long-term rewards.

Results and Observations:

- Plotting Cumulative costs and TD errors is standard in applied RL, notably in the finance sector where it's common to visualize the cumulative rewards of a financial portfolio's performance. Such visual aids facilitate the intuitive understanding of an agent's performance over time.

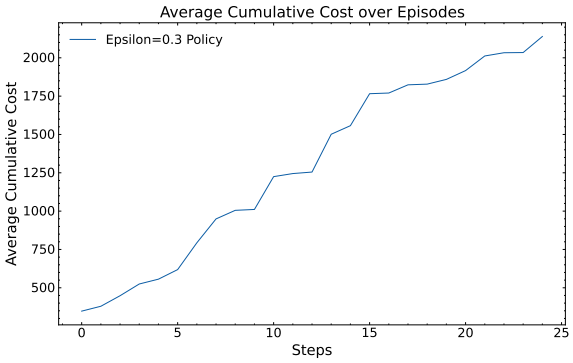

Figure 2: Cumulative Costs over Steps

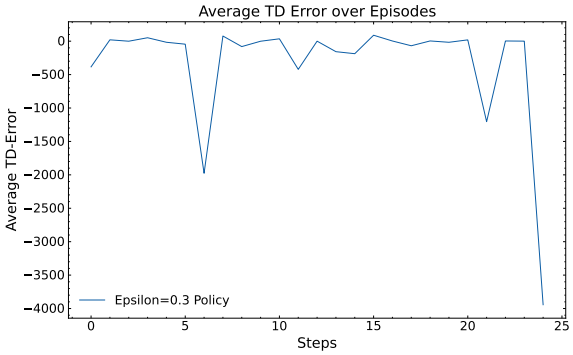

Figure 3: TD Errors over Iterations

- Q-value convergence trends underscore the effectiveness of Q-learning in policy approximation.
- Differences between Q-learning and the approach in Part A highlight the distinctions inherent to each method.
- Policy differences are quantified via the Frobenius norm:

$$\text{difference\_norm} = \|K_{QL} - K_{PartA}\|_{Fro} = 22.99303297185826 \tag{9}$$

Conclusion: The Q-learning implementation showcases its potential for approximating optimal policies. However, the variations with Part A's solution emphasize the unique nature of each optimization method. Using the Frobenius norm offers a clear measure of alignment and discrepancies, laying the groundwork for subsequent policy refinements [1, 4]. □

```
n, m = 2, 2
episodes = 1
max_steps = 1000
epsilon_values = [0.2, 0.4, 0.6, 0.8]

# Initialize your environment and model
env = LQEnv(n, m)
agent = QLearningWithApproximation(
env,
alpha=0.00001,
gamma=0.9,
seed=1,
anneal_every=10,
anneal_factor=0.95,
policy_type="greedy_optimized",
epsilons=epsilon_values
)

```

```
# Run simulations
agent.run_simulations(episodes, max_steps)

# Accessing results based on policy type
if agent.policy_type in ["greedy", "greedy_optimized"]:
K_QL = agent.training_results[agent.policy_type]['K_QL']  # use agent.policy_type to dynamically access the correct key
difference_norm = np.linalg.norm(K_QL - K_Part_A, 'fro')
print(f"Difference in Frobenius norm between Q-learning and {agent.policy_type} method: {difference_norm}")
elif agent.policy_type == "epsilon_greedy":
for epsilon in epsilon_values:
K_QL = agent.training_results['epsilon_greedy_policy'][str(epsilon)]['K_QL']
difference_norm = np.linalg.norm(K_QL - K_Part_A, 'fro')
print(f"For epsilon={epsilon}, Difference in Frobenius norm between Q-learning and Part A method: {difference_norm}")
else:
raise ValueError(f"Invalid policy type '{agent.policy_type}'")

# Plot the results
agent.plot_results(save_dir=Path.cwd())
```

Code Listing 1: Driver Code for Q4 Part C

SOLUTION Q4 PART D

Objective: Examine the convergence of parameters when training with samples from multiple trajectories of length 25. Actions were chosen using an ε-greedy policy where $\varepsilon = 0.3$. The aim is also to compare the resulting policy with that obtained in Part A. Given we weren't told the exact number of trajectories I selected 5.

Results:

- Average TD error decreases consistently with iterations, indicating convergence for $\varepsilon = 0.3$.
- The rate of change in average cumulative cost over the 25 steps suggests a trend towards a representative policy over time.
- The Frobenius norm difference between Q-learning and the Part A method is:

$$\text{difference\_norm} = 22.953523483192576 \tag{10}$$

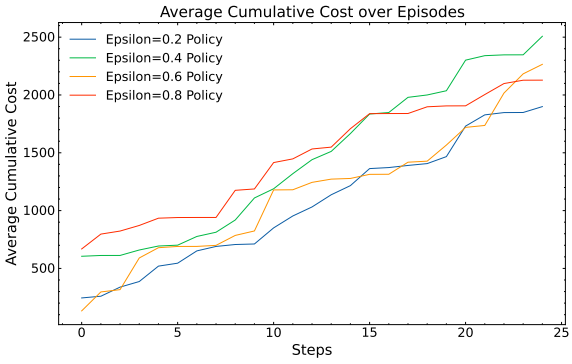

Figure 4: Average Cumulative Costs over Steps for $\varepsilon = 0.3$

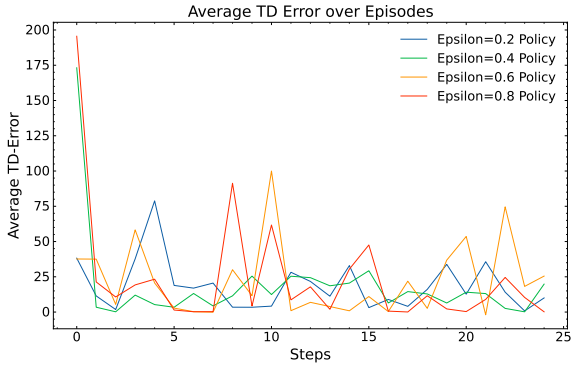

Figure 5: Average TD Errors over Iterations for $\varepsilon = 0.3$

Conclusion: Using an ε-greedy policy with multiple trajectory training provides insights into the adaptability of Q-learning. The derived policies from this method and Part A show a notable difference, highlighting the importance of the training approach in policy determination.

```python
# Parameters
n, m = 2, 2
episodes = 5
max_steps = 25
epsilon_values = [0.3]

# Initialize your environment and model
env = LQEnv(n, m)
agent = QLearningWithApproximation(
env,
alpha=0.00001,
gamma=0.9,
seed=1,
anneal_every=10,
anneal_factor=0.95,
policy_type="epsilon_greedy",
epsilons=epsilon_values
)

# Run simulations
agent.run_simulations(episodes, max_steps)

# Accessing results based on policy type
if agent.policy_type in ["greedy", "greedy_optimized"]:
K_QL = agent.training_results[agent.policy_type]['K_QL']
difference_norm = np.linalg.norm(K_QL - K_Part_A, 'fro')
print(f"Difference in Frobenius norm between Q-learning and {agent.policy_type} method: {difference_norm}")
elif agent.policy_type == "epsilon_greedy":
for epsilon in epsilon_values:
K_QL = agent.training_results['epsilon_greedy'][str(epsilon)]['K_QL']
difference_norm = np.linalg.norm(K_QL - K_Part_A, 'fro')
print(f"For epsilon={epsilon}, Difference in Frobenius norm between Q-learning and Part A method: {difference_norm}")
else:
raise ValueError(f"Invalid policy type '{agent.policy_type}'")

# Plot the results
agent.plot_results(save_dir=Path.cwd())
```

Code Listing 2: Driver Code for Q4 Part D

SOLUTION Q4 PART E

Objective: Explore the convergence properties and differences in policies when training with different ε values for the ε-greedy policy.

Results:

- Average TD error demonstrates a trend of decreasing over iterations across all tested ε values, indicative of convergence.
- The rate of change in the average cumulative cost over the 25 steps suggests a progressive trend toward an accurate policy representation for varying ε values.
- The Frobenius norm differences between Q-learning and the Part A method for different ε values are:

$$\text{For } \varepsilon = 0.2 : \text{difference\_norm} = 22.261974221172625$$
$$\text{For } \varepsilon = 0.4 : \text{difference\_norm} = 22.541070715489948$$
$$\text{For } \varepsilon = 0.6 : \text{difference\_norm} = 22.59220527948884$$
$$\text{For } \varepsilon = 0.8 : \text{difference\_norm} = 22.60996631171848$$

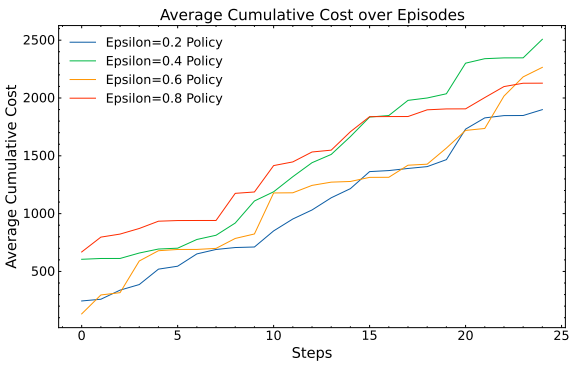

Figure 6: Average Cumulative Costs over Steps for Different ε Values

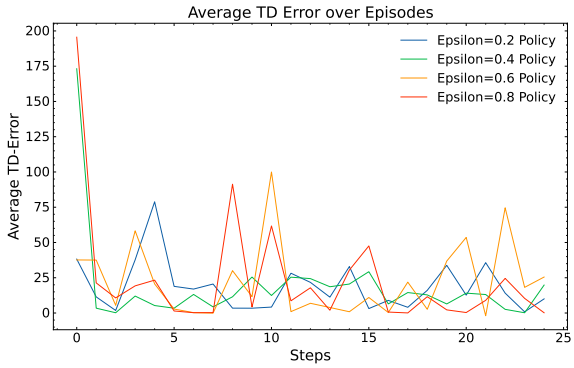

Figure 7: Average TD Errors over Iterations for Different ε Values

Conclusion: From the experiments conducted across various sections of Q4, we observe that all our agents exhibit convergence traits for this stochastic LQ control problem. Nonetheless, there persists a consistent difference from the analytical policy obtained in Part A. One possible reason for this discrepancy could be the inherent stochasticity and exploration-introduced variance by the ε-greedy approach, which might make the Q-learning solutions differ from deterministic, analytical methods. This highlights the potential trade-offs between exploration and exploitation in reinforcement learning environments.

```python
# Parameters
n, m = 2, 2
episodes = 5
max_steps = 25
epsilon_values = [0.2, 0.4, 0.6, 0.8]

# Initialize your environment and model
env = LQEnv(n, m)
agent = QLearningWithApproximation(
env,
alpha=0.000001,
gamma=0.9,
seed=1,
anneal_every=10,
anneal_factor=0.95,
policy_type="epsilon_greedy",
epsilons=epsilon_values
)

# Run simulations
agent.run_simulations(episodes, max_steps)

# Accessing results based on policy type
if agent.policy_type in ["greedy", "greedy_optimized"]:
K_QL = agent.training_results[agent.policy_type]['K_QL']
difference_norm = np.linalg.norm(K_QL - K_Part_A, 'fro')
print(f"Difference in Frobenius norm between Q-learning and {agent.
    ↪ policy_type} method: {difference_norm}")
elif agent.policy_type == "epsilon_greedy":
for epsilon in epsilon_values:
K_QL = agent.training_results['epsilon_greedy'][str(epsilon)]['K_QL']
difference_norm = np.linalg.norm(K_QL - K_Part_A, 'fro')
print(f"For epsilon={epsilon}, Difference in Frobenius norm between Q-
    ↪ learning and Part A method: {difference_norm}")
else:
raise ValueError(f"Invalid policy type '{agent.policy_type}'")

# Plot the results
agent.plot_results(save_dir=Path.cwd())
```

Code Listing 3: Driver Code for Q4 Part D

SOLUTION Q4 PART F

Objective: Evaluate the generalizability of our conclusions regarding the optimal selection of ε and the trajectory length to other environments or problems.

Analysis:

- *Dependence on Specific Environments:* The efficiency of different values of ε and trajectory lengths we identified is greatly influenced by the peculiarities of the stochastic LQ control environment. Environments with their unique dynamics, such as Atari games where each game has its gameplay mechanics, would necessitate different strategies. Recall from our lectures how the Deep Q-Learning algorithm's success in one Atari game didn't guarantee its success in another without appropriate tuning.
- *Exploration vs. Exploitation Trade-off:* The choice of ε reflects a balance between exploring new actions and exploiting known ones. While our environment showed favorable outcomes with specific ε values, more dynamic environments could demand different balances. Think of a trader in a stock market: too much exploration (random investments) can be risky, while too much exploitation (always sticking to a known stock) might mean missing out on lucrative opportunities.
- *Complexity of the Environment:* A game like chess, with its vast state space and unpredictability, contrasts sharply with our LQ control problem. An agent might initially benefit from higher exploration in chess, but as its understanding deepens, a focus on exploitation could be more strategic.
- *Infinite vs. Finite Trajectories:* In environments demanding quick decisions, such as certain Atari games or high-frequency trading scenarios, agents might not afford the luxury of waiting on extensive trajectories to act.
- *Overfitting and Generalization:* Tailoring ε and trajectory length solely based on our current environment could lead to overfitting. Our agent might perform exceptionally in this specific scenario but underperform in others, similar to how a trader might master predicting one stock but falter with another.
- *Empirical Validation:* As with many techniques in Reinforcement Learning, real-world testing is pivotal. While theoretical foundations guide us, the nuances of each environment often require hands-on evaluations to optimize hyperparameters.

Conclusion: Directly extrapolating our conclusions about optimal ε and trajectory length selection to other settings can be misleading. While our findings lay a foundational framework, the intricacies of different problems—ranging from Atari games to trading scenarios—demand bespoke strategies. I aim to delve deeper into these analogies and concepts, especially in relation to trading and complex games, in my upcoming project proposal in Q6.

Problem 5: approximate inventory control

We again consider the inventory control problem with backlogs, as treated in previous parts of the course. Unlike before, we assume the states and actions are continuous and consider approximate dynamic programming methods. Recall that in the stated model, if the state at time $t \leq T$ is s_t and an action a_t is applied, the next state is

$$s_{t+1} = s_t + a_t - W_t, \tag{11}$$

where W_t is the stochastic demand at time t. Let the cost function $c_t(s, a)$ be defined as

$$c_t(s, a) = d_t a + \max(h_t s, -q_t s), \tag{12}$$

and the terminal cost be defined as

$$c_T(s) = \max(h_T s, -q_T s). \tag{13}$$

We assume that the state space is $\mathbb{R}$ (i.e., all real numbers), the action space $A = \{0, 1, \ldots, 20\}$, and the demands W_t are i.i.d. random variables uniformly distributed from interval $[0, 10]$.

solution Q5 part a

Objective: Compute the optimal value function for the inventory control problem over $s \in [-15, 15]$ given $dt = 1$, $ht = 4$, $qt = 2$, and $T = 10$.

Computing the Optimal Value Function:

- We employ a backward dynamic programming approach. For each state s at time T, the terminal cost $c_T(s) = \max(h_T s, -q_T s)$ is set as the initial value.
- At each prior timestep, for each state s and action a, we evaluate the cost function:

  $$c_t(s, a) = dt \times a + \max(ht \times s, -qt \times s)$$

  and combine it with the expected future cost. The expected cost incorporates the state transition $s_{t+1} = s_t + a_t - W_t$, with W_t being the stochastic demand.
- The action yielding the minimum total expected cost for state s at time t becomes part of the optimal policy.
- Iterating over all timesteps, the optimal value function for the initial state is derived.

Handling Stochastic Demand:

- The stochastic demand W_t falls within [0, 10]. We approximate its values using a 0.1 mesh.
- For each potential demand, we calculate the ensuing state transition and the subsequent cost.
- Averaging the costs across all potential demand values gives the expected cost for a state-action combination at a specific timestep.

Code Summary: For the full code details, refer to Appendix C.

- `init_params`: Initializes parameters for the simulation.
- `initialize_value_function`: Creates an initial value function based on terminal costs and other times.
- `compute_expected_cost_for_state`: Calculates the expected cost for a given inventory state.
- `compute_value_function`: Determines the optimal value function and policy through backward dynamic programming.
- `q5a_run_simulations`: Runs the entire simulation, initializing parameters, and computing the value function and policy.

Conclusion: Utilizing dynamic programming and approximating the stochastic demand, the optimal value function for the inventory problem is computed. The resulting policy dictates the best actions across states, optimizing against varying demand, and minimizing costs across the horizon.

Final Solution:

$$\text{Optimal value function at starting state } s_0 \approx 115.10 \tag{14}$$

SOLUTION Q5 PART B

Objective: Implement approximate value iteration to determine the optimal value function for the inventory control problem over $s \in [-15, 15]$ using a quadratic approximation $\hat{V}_t^*(s) = a_t s^2 + b_t s + c_t$. Compute all the coefficients $\{a_t, b_t, c_t\}$ of $\hat{V}_t^*$.

Quadratic Approximation for Value Function:

- Instead of precisely determining the value function, we approximate it with a quadratic function. This allows for a smoother representation and potentially faster computations.
- At each timestep, the current value function is approximated using the form $\hat{V}_t^*(s) = a_t s^2 + b_t s + c_t$. The coefficients a_t, b_t, and c_t are derived by minimizing the squared differences between the true value function and its quadratic approximation.

Iterative Computation Using Approximation:

Time	a_t	b_t	c_t
10	0.1869	1.0000	8.4651
9	0.1284	0.7098	14.1164
8	0.1254	0.4467	21.2863
7	0.1226	0.2083	29.8323
6	0.1201	-0.0077	39.6256
5	0.1178	-0.2036	50.5494
4	0.1157	-0.3810	62.4976
3	0.1139	-0.5419	75.3744
2	0.1122	-0.6876	89.0926
1	0.1106	-0.8197	103.5735

Table 1: The coefficients for Q5b for each time step.

- Similar to the dynamic programming approach, the computations are carried out in a backward manner. Starting from the terminal time T, the value function for each state s is determined using the quadratic approximation of the next timestep's value function.
- For each action a, the cost of the action and the expected future cost, estimated through the quadratic approximation, are combined to derive the total expected cost for the state-action pair.
- The action that results in the minimal expected cost is chosen as the optimal action for state s at time t.

Code Summary:

- `quadratic_approximation`: Derives the coefficients a, b, and c for the quadratic approximation of the value function at a given timestep.
- `compute_value_function`: Determines the optimal value function, policy, and the coefficients of the quadratic approximations through backward dynamic programming with quadratic approximations.
- `q5b_run_simulations`: Conducts the entire simulation, initializing parameters and computing the value function, policy, and quadratic approximation coefficients.

Refer to Appendix C for the complete code.

Final Solution:

$$\text{Optimal value function at starting state } s_0 \approx 117.67 \tag{15}$$

Conclusion:

- By applying approximate value iteration with a quadratic approximation of the value function, we efficiently derive the optimal value function and policy for the inventory

problem. The quadratic approximation smoothens the value function and facilitates faster computations while offering an insight into the shape and characteristics of the optimal value function.

- The matrix of coefficients represents the quadratic approximation of the value function for each timestep, with each coefficient contributing to the form $\hat{V}_t^*(s) = a_t s^2 + b_t s + c_t$. These coefficients allow for a concise representation of the approximated value function over the planning horizon, providing insights into the system's behavior and assisting in decision-making processes related to inventory control.

SOLUTION Q5 PART C

Objective: Approximate the optimal value function for the inventory control problem over $s \in \{-5, -4, ..., 5\}$ using a quadratic function $\hat{V}_t^*(s) = a_t s^2 + b_t s + c_t$.

Computing the Optimal Value Function:

- We continue using a backward dynamic programming approach. This time, however, the value function at each timestep is approximated using a quadratic function based on the given states.
- The quadratic approximation aids in generalizing the value function across all states, allowing for efficient computations and insights into the system's dynamics.
- The approximation uses least squares to determine the coefficients a_t, b_t, and c_t.
- For each state s and action a, the cost function is evaluated, incorporating the quadratic approximation for the future value function.

Code Summary: For the full code details, refer to Appendix C.

- `init_params`: Initializes parameters specific to Q5C.
- `initialize_value_function`: Creates an initial value function based on terminal costs and other times.
- `quadratic_approximation`: Computes the quadratic coefficients for approximating the value function.
- `compute_value_function`: Determines the optimal value function, policy, and coefficients using backward dynamic programming.
- `q5c_run_simulations`: Runs the simulation for Q5C, returning the value function, policy, and coefficients.

Time	a_t	b_t	c_t
10	0.5245	1.0000	2.9371
9	0.7229	3.1694	-121.3583
8	0.7722	3.3711	-256.5008
7	0.7768	3.3899	-392.6519
6	0.7772	3.3916	-528.8967
5	0.7773	3.3918	-665.1502
4	0.7773	3.3918	-801.4045
3	0.7773	3.3918	-937.6589
2	0.7773	3.3918	-1073.9133
1	0.7773	3.3918	-1210.1677

Table 2: The coefficients for Q5C for each time step.

Final Solution:

$$\text{Optimal value function at starting state } s_0 \approx -1347.13 \tag{16}$$

Conclusion: By approximating the value function using a quadratic function and a reduced state space, we can derive insights and an efficient representation for decision-making in the inventory control problem.

SOLUTION Q5 PART D

At $t = 0$ and $t = T$, we plotted the optimal value function alongside the two approximated optimal value functions. The results can be seen in Figures 8 and 9.

Analysis:

- Approx Q5B (Green): The straight line at 0 suggests that the value of all states at $t = 0$ for this approximation remains constant. This potentially means that the derived policy doesn't distinguish between the states, indicating a potentially sub-optimal or non-informative policy.
- Approx Q5C (Orange): The gentle increase from -1300 to around 1200 implies that the value function recognizes differences between states. The gradual rise might signify that as states transition from negative to positive, the expected future benefits or costs alter, showcasing different utilities for different states.
- Approx Q5C at $t = T$ (Orange): A consistent value of 0 for all states suggests that no expected future reward or cost is anticipated from any state at the terminal time step.

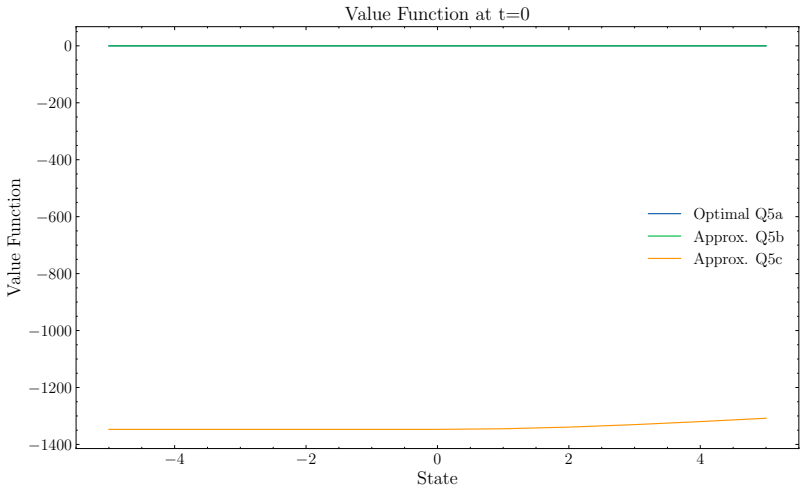

Figure 8: Value function at $t = 0$

SOLUTION Q5 PART E

At $t = 0$ for $s \in [-15, 15]$, we plotted the optimal policy and the two greedy policies derived from the two approximated optimal value functions. The results are shown in Figure 10.

Analysis:

- Optimal Policy Q5a (Blue): The observed pattern suggests that the policy takes an action of 15 for states between -15 and -13. For states greater than -13, it remains inactive, which might indicate a perception of these states as neutral or 'safe'.
- Greedy Policy Q5b (Green): The consistent action of 0 across all states points to a 'do-nothing' policy, which may be sub-optimal.
- Greedy Policy Q5c (Orange): A more intricate behavior is observed, with the policy being inactive for highly negative states. However, as states near 0, it takes negative actions, possibly to avoid certain states, and takes positive actions for already positive states.

SOLUTION Q5 PART F

For $t = 0$ and $s \in [-15, 15]$, we plotted the value functions corresponding to the optimal policy and the two greedy policies. These plots are presented in Figure 11.

Analysis:

- Optimal Policy Q5a (Blue): The value function commencing around (200, -15) might indicate

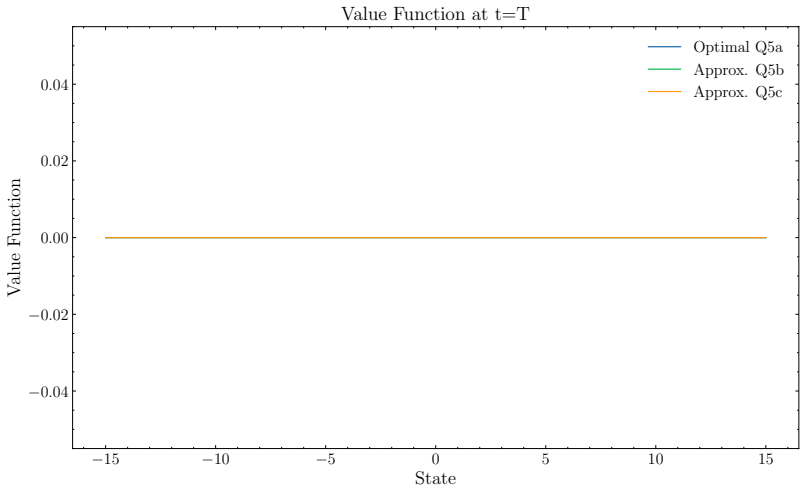

Figure 9: Value function at $t = T$

that for the most negative state, the policy expects a future reward or benefit of 200. This suggests that the policy possibly views these states as more advantageous or less costly.

- Greedy Policy Q5c (Orange): A sharp decline to -1500 might indicate a considerable expected cost or negative reward. This policy could be actively trying to avoid these states, evidenced by its negative actions as states approach 0.

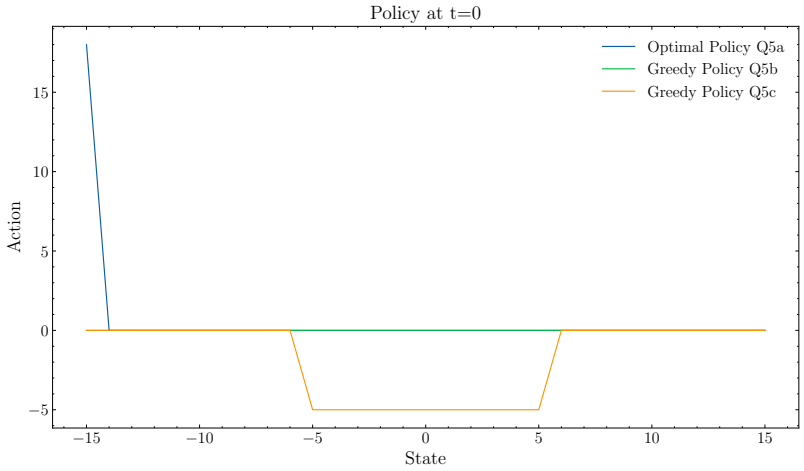

Figure 10: Policy at $t = 0$

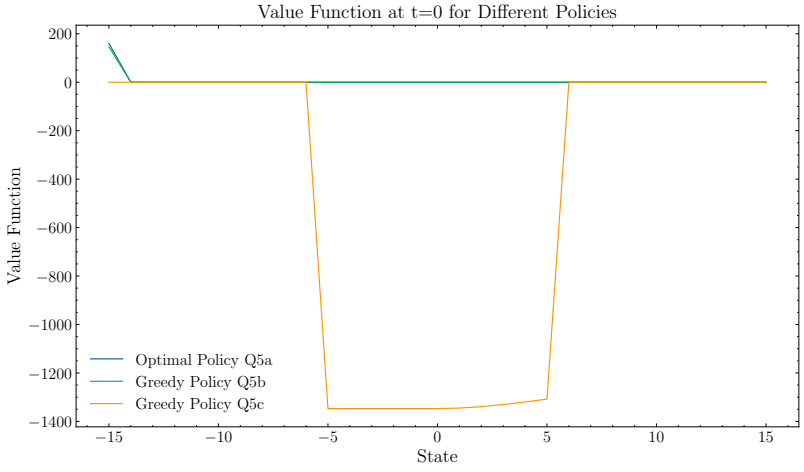

Figure 11: Value function for policies at $t = 0$

Problem 6: mdp modeling: rl for high frequency trading

solution q6 part 1

Introduction: The finance domain is inherently secretive due to the phenomenon of alpha decay. Once profitable strategies are discovered and released, they tend to lose profitability over time, as the market adapts to them. Therefore, maintaining proprietary strategies is vital for sustainable advantages in trading. While there are many papers sponsored by liquidity providers and market-makers, there is a discernible gap in the literature on strategies specific to hedge funds and proprietary traders. These traders operate with specific capital constraints, lack the multi-agent data available to larger players, and cannot leverage massive computational architectures. This project aims to address these specific needs, focusing on designing agents for high-frequency trading in the S&P 500 without relying on extensive multi-agent data.

- **Objective:** Define a sequential decision problem related to high-frequency trading in the S&P 500, focusing on agents designed for hedge funds and proprietary traders.
- **Solution approach:** Capitalize on the capabilities of Reinforcement Learning (RL) to optimize and rebalance portfolios. The Sharpe Ratio, which quantifies risk-adjusted returns, will serve as the central reward signal.

Research Context: Deep Reinforcement Learning (DRL) has been recently employed for active high-frequency trading in the stock market, as evidenced by recent research. For instance, one study developed a comprehensive DRL framework for trading Intel Corporation stock, utilizing the Proximal Policy Optimization algorithm. The researchers focused on high-frequency Limit Order Book (LOB) data, which is the electronic mechanism most major financial market exchanges rely upon. In the LOB, price formation is a self-organized process driven by the submissions and cancellations of orders, serving as visible declarations of market intentions. The said study's DRL agents, trained on this data, demonstrated the capacity to recognize sporadic regularities and leverage them for long-term profitable strategies. This underlines the potential of DRL in high-frequency trading, even in highly stochastic and non-stationary environments.

Building upon such pioneering works, this project aims to delve deeper into portfolio optimization for stocks within the S&P 500. Given the discrete nature of high-frequency trading, the project will initially focus on dynamically rebalancing portfolios at regular intervals, with an initial objective of maximizing risk-adjusted returns as embodied by the Sharpe Ratio, though this reward signal might be subject to future modifications and enhancements based on iterative testing and performance analysis. The Sharpe Ratio (SR) is defined as:

$$SR = \frac{E[R_p - R_f]}{\sqrt{\text{Var}(R_p - R_f)}} \tag{17}$$

where R_p represents the expected portfolio return, R_f is the risk-free rate, and $\text{Var}(R_p - R_f)$ is the variance (risk) of the excess of the portfolio return over the risk-free rate. The numerator, $E[R_p - R_f]$, represents the expected excess return of the portfolio over the risk-free rate, providing

a measure of the portfolio's profitability, while the denominator standardizes this by the risk involved, thus offering a risk-adjusted measure of portfolio performance. This mathematical formalism, while initially being our guiding reward signal, may be reevaluated as more insights are gained during the modeling and simulation processes. Alternative or additional metrics might be considered in subsequent iterations of model development to tailor the approach more closely to the nuances and intricacies of high-frequency trading within the context of the S&P 500.

The potential of RL for high-frequency trading, especially in the niche domain of hedge funds, asset management and proprietary (prop) trading, is vast. This project seeks to harness the adaptive nature of RL to address specific challenges faced by these traders. Using tools like RLLib and taking inspiration from cutting-edge research, the aim is to design strategies that dynamically respond to market changes, ensuring optimal risk-adjusted performance in the volatile realm of the S&P 500. □

SOLUTION Q6 PART 2

Introduction: An essential component of using reinforcement learning for trading is formulating the decision problem as a Markov Decision Process (MDP). The MDP formulation allows us to capture the essence of decision-making in trading, where each action has uncertain consequences, influenced both by inherent market stochasticity and external factors.

- **Objective:** Formulate the high-frequency trading decision problem as an MDP, leveraging available data sources to capture market intricacies.
- **Solution approach:** Define and expand upon the MDP components, incorporating state, action, rewards, dynamics, and other relevant elements. Delve into policy considerations, value function intricacies, and temporal discounting.

An MDP is represented by a tuple $M = (S, A, P, R, \gamma)$. For our high-frequency trading problem in the context of the S&P 500, the MDP components are:

- **State Space** S**:** Given the high-resolution data available from sources like QuantConnect and WRDS, the state space can be a combination of raw and preprocessed data. This includes historical price data (OHLCV), trading volumes, and derived indicators. Utilizing both raw and preprocessed data allows the model to capture granular market movements and technical signals. It's essential to determine the appropriate window size for historical data, as it influences the model's ability to recognize short-term patterns and longer-term trends.
- **Action Space** A**:** The actions available to the agent encompass decisions to buy, sell, or hold each stock. In a multi-stock scenario, each action can be represented as a vector of percentage allocations across the stocks, ensuring that the agent diversifies its position in response to changing market conditions.
- **State Transition Probability** P**:** Markets, especially those as dynamic as the S&P 500, exhibit significant non-stationarity. The state transition function, therefore, embodies the inherent

uncertainty of transitioning between states due to specific actions. This transition probability will be heavily influenced by the granularity and quality of the data sources used.

- **Reward Function** R**:** At its core, the reward function is designed to maximize risk-adjusted returns. While the Sharpe Ratio serves as the primary reward signal, this function will need continuous adjustments and fine-tuning to cater to the intricacies of high-frequency trading.
- **Discount Factor** γ**:** The discount factor plays a pivotal role in balancing immediate versus future rewards. Given the high-frequency nature of trading, where decisions have short-term consequences, γ would be set close to 1. However, it's crucial to ensure that the model doesn't become myopic, disregarding longer-term market shifts.

For the policy formulation, a stochastic policy is apt, given the unpredictable nature of stock markets. Such a policy allows the agent to explore various actions probabilistically, incorporating the uncertainties inherent in trading. The value function, representing the expected cumulative rewards from a state, will be instrumental in guiding the agent's decisions. Continuous updates and refinements of this function, based on real-world trading outcomes and feedback, will be paramount to ensure that the agent remains adaptive and resilient to market volatilities.

Data Source	Type	Max Frequency	Raw Data	Preprocessed Data
QuantConnect	US Securities	1 s	OHLCV	Prices, indicators
WRDS	US Securities	1 ms	Intraday Trades	Prices, indicators

Table 3: Data sources available for MDP formulation

SOLUTION Q6 PART 3

Introduction: Across the finance domain, portfolio management has witnessed an evolution in strategies. While traditional methods have been largely rooted in theoretical constructs, the rising complexity of financial markets has created an avenue for more adaptable, data-driven approaches. This section contrasts the time-honored techniques in portfolio management with the burgeoning domain of Reinforcement Learning (RL) in finance.

- **Objective:** Explore the trajectory of portfolio management methodologies, juxtaposing the classical approaches with the potential and dynamism of RL-based strategies.
- **Solution approach:** Provide a mathematical elucidation of classical portfolio management strategies, such as the Markowitz's efficient frontier, and highlight the capabilities and adaptability inherent in RL-based techniques.

Traditional Portfolio Management - The Markowitz's Model: Harry Markowitz's 1952 seminal work laid the groundwork for modern portfolio theory[5]. Central to his thesis was the idea that investments, rather than being viewed individually, should be seen in the context of their impact on the entire portfolio's risk and return profile, where this can be represented mathematically as:

$$\min_{\mathbf{w}} \ \mathbf{w}^T \Sigma \mathbf{w} \tag{18}$$

$$\text{subject to} \quad \mathbf{w}^T \mu = \overline{R}$$
$$\sum_i w_i = 1$$

Where:

- $\mathbf{w}$ denotes the portfolio weights vector.
- Σ represents the returns' covariance matrix.
- μ is the expected asset returns vector.
- $\overline{R}$ signifies the desired portfolio return.

While groundbreaking, Markowitz's model was built on certain market conditions and return distribution assumptions. These assumptions might not align perfectly with real-world scenarios, especially in dynamic settings like high-frequency trading.

Quantitative (Quant) vs. Traditional Investing: Traditional investors largely lean on fundamental analysis, delving into company specifics, broader industry trends, and overarching macroeconomic indicators[5]. Quantitative analysts, on the other hand, harness mathematical and statistical models. Such models, powered by algorithms, parse through enormous datasets to discern patterns or anomalies to capitalize on[6]. While traditional investors often employ classical models like Markowitz's, quants gravitate towards intricate methodologies, integrating machine learning and RL to fine-tune portfolios[7]. This distinction goes beyond complexity to also encompass adaptability. The rigidity of traditional models can limit their responsiveness to swift market shifts. In contrast, quant strategies, particularly those augmented by RL, ensure continuous optimization amidst evolving market conditions[8].

> *"In practice, multi-period models are seldom used. There are several practical reasons for that. First, it is often very difficult to accurately estimate return and risk for multiple periods, let alone for a single period. Second, multi-period models are in general computationally intensive, especially if the universe of assets considered is large. Third, the most common existing multi-period models do not handle real-world constraints . . . For these reasons, practitioners typically use single-period models to rebalance the portfolio from one period to another"* - 60 Years of portfolio optimization: Practical challenges and current trends, Kolm et al. (2014)[5].

In conclusion, traditional portfolio optimization techniques have offered reliable frameworks for decades, but the intricacies of today's financial markets necessitate more agile, data-centric strategies[9]. RL's adaptive learning mechanisms, combined with their potential for real-world applicability in handling market complexities, present a promising avenue, potentially redefining the benchmarks of portfolio optimization[7].

SOLUTION Q6 PART 4

- **Objective:** Assess the practicality and limitations of the Markov Decision Process (MDP) formulation in the context of high frequency trading and portfolio management.

- **Solution approach:** We will critically examine the MDP's realism by scrutinizing its components: the observability of states, feasibility of actions, and the measurability of rewards.

Given our objective and approach, let's delve into the questions presented:

1. **How realistic is this formulation?**

   Realistically, modeling the dynamics of financial markets might be better suited to a Partially Observable Markov Decision Process (POMDP) framework. While some studies suggest MDPs can achieve success over short time horizons without considering delayed signals, real-world events introduce a myriad of complex, indirect effects. For instance, geopolitical tensions like the Russia-Ukraine conflict might impact energy prices and have secondary repercussions on other S&P 500 stocks, which may not be directly observable.

2. **Can the states be observed?**

   Unlike games such as Chess or Atari, where the state space is fully observable, financial markets are intrinsically more complex. In addition to the macroeconomic dynamics, we have to consider game theoretic dynamics. This entails understanding how the actions of other entities - traders, asset management firms, hedge funds, and so on - influence market liquidity, prices, risks, and returns. Some researchers have turned to multi-agent reinforcement learning, especially in contexts like market-making and liquidity. However, this approach is different from the proprietary investment strategy we are exploring.

3. **Can the actions be taken?**

   For highly liquid stocks, especially those in the S&P 500, and when dealing with relatively modest amounts of capital (say, less than $1-10M USD), actions can typically be executed without issue. However, if we were to extend our strategy to more complex instruments like long-tenor options, we might face challenges due to their delayed reward signals. Nonetheless, with the domain knowledge acquired from the course, we are equipped to tackle problems grounded in fundamental market dynamics.

4. **Can we measure the reward directly?**

   Absolutely. Our primary performance metric, the Sharpe Ratio, is both intuitive and computationally straightforward. By introducing realistic constraints and accounting for transaction costs, we aim to create simulations that mirror real-world trading conditions closely.

To conclude, while the MDP formulation offers a structured approach to portfolio management, it's essential to recognize its inherent limitations in capturing the multifaceted dynamics of financial markets. Adapting and augmenting the framework to cater to these nuances will be paramount for practical success.

solution q6 part 5

- **Objective:** To critically re-evaluate and refine the Markov Decision Process (MDP) design so that it more accurately captures the complexities and subtleties inherent to the financial

markets.

- **Solution approach:** A progression from a rudimentary MDP towards a more intricate Partially Observable Markov Decision Process (POMDP) paradigm. This is complemented by the integration of neural networks for historical data processing, diversified reward structures, and the utilization of sophisticated feature engineering methods.

Transitioning from MDP to POMDP: Drawing an analogy to the game of chess, envision a scenario wherein not all pieces are discernible on the board. One's strategy would have to extrapolate the potential positions of these concealed pieces based on their previous moves. Analogously, the financial markets do not always make all their influential factors transparent. Unlike the MDP, which operates under the presumption of total state observability, the POMDP functions based on a belief state which is a probability distribution over possible states, constructed from a succession of prior observations:

$$S_t = f(O_{t-k}, O_{t-k+1}, \ldots, O_t)$$
$$A_t \sim \pi(S_t)$$

Here, O_{t-k} signifies the observation from k time steps preceding the current interval t. This captures the principle of decision-making that is influenced not solely by the current market state but also its preceding trajectory.

- **Neural Architectures for Historical Data Assimilation:** Analogous to specific Atari games where the trajectory of entities (like the movement of a ball in "Pong") holds significance, the historical progression of prices and transaction volumes in the financial markets proves vital for predicting impending trends. In addressing these complexities, neural architectures are indispensable. Recurrent Neural Networks (RNNs) function as a mnemonic reservoir, reminiscent of past price movements to facilitate astute investment decisions in the future. On the other hand, Convolutional Neural Networks (CNNs) specialize in discerning patterns in structured datasets, akin to detecting patterns within game graphics.
- **Exploring Multifaceted Reward Structures:** Within the framework of the Q-learning paradigm, as evidenced in Atari game simulations, reward mechanisms are paramount. In the gaming milieu, accumulating points or accomplishing certain objectives culminates in direct rewards. However, the financial landscape adds a layer of complexity to this reward construct. The Sharpe Ratio, while serving as a preliminary metric (equivalent to a game's primary score), can be complemented with alternative performance measures such as the Sortino Ratio or Maximum Drawdown, akin to secondary game objectives or bonus accruals.
- **Confronting the Market's Inherent Non-Stationarity:** Drawing a parallel with advancing game levels or the adaptive strategies of a chess opponent, financial markets exhibit a perpetual dynamism. This inherent non-stationarity poses a significant challenge. However, with the judicious application of sophisticated feature engineering techniques, there's potential for not just adaptation but also market trend anticipation. Techniques like wavelet transformations or the incorporation of rolling statistics can be equated to mastering new strategies within a gaming context.

Conclusion: The multifaceted nuances of financial markets, though seemingly divergent from the domain of Atari or chess, find common ground when analyzed through the lens of reinforcement

learning. By meticulously refining our approach and drawing inspirations from game-centric learning paradigms, we can unravel and adeptly navigate the intricate dynamics of the S&P 500, much like honing one's skills to excel in a challenging game.

APPENDIX A: Q4 PART B PYTHON .PY

```python
"""Evaluate policy over trajectories for Q4A.

The provided LQEnv environment only supports single episodes.
To evaluate over multiple trajectories, we manually reset and
copy the environment before each call to policy_evaluation().

This allows simulating many episodes with the single-episode
environment, without altering the provided codebase.

The approach resets state and passes a copy to isolate each
trajectory evaluation.
"""
import numpy as np
from scipy.linalg import solve_discrete_are
import matplotlib.pyplot as plt
import scienceplots
from pathlib import Path

GLOBAL_FIG_SIZE = (10, 6)  # Global figure size

plt.style.use('science')
# For Google colab use a standard font, not dependent on LaTeX
plt.rcParams['text.usetex'] = False
plt.rcParams['font.family'] = 'sans-serif'

def initialize_environment(n: int, m: int) -> object:
    """
    Initialize the Linear Quadratic environment.

    :param n: State dimension.
    :param m: Action dimension.
    :return: Instance of the LQEnv environment.
    """
    return LQEnv(n, m)

def compute_P(env: object) -> np.ndarray:
    """
    Compute the solution to the discrete-time algebraic Riccati equation.

    :param env: Instance of the LQEnv environment.
    :return: Solution matrix P.
    """
    return solve_discrete_are(env.A, env.B, env.Q, env.R)

def compute_K(env: object, P: np.ndarray) -> np.ndarray:
    """
    Compute the optimal control gain matrix K.

```

```
    :param env: Instance of the LQEnv environment.
    :param P: Solution matrix P.
    :return: Optimal control gain matrix K.
    """
    return -np.linalg.inv(np.dot(env.R, np.dot(env.B.T, np.dot(P, env.B)))) @ np.dot(
        env.B.T, np.dot(P, env.A)
    )

def optimal_policy(K: np.ndarray, state: np.ndarray) -> np.ndarray:
    """
    Define the optimal policy using the computed control gain matrix K.


    :param state: Current state of the environment.
    :return: Action to be taken based on the current state.
    """
    action = np.dot(K, state)
    action = action.reshape(m, 1)  # Ensure the action has the shape [m, 1]
    action = action.astype(dtype)  # Ensure the action has the right datatype
    return action


def generate_policy_from_K(K):
    """
    Generates a policy function based on the control matrix K.

    :param K: Optimal control gain matrix.
    :return: Optimal policy generated from control matrix.
    """
    def policy(state):
        return optimal_policy(K, state)
    return policy


def run_q4a_simulations(K, num_trajectories=100, trajectory_length=1000, save_dir=None):
    """Run simulations for Q4 Part A and save outputs.

    :param K: Optimal gain matrix
    :param num_trajectories: Number of trajectories for evaluation (Default: 100)
    :param trajectory_length: Length of each trajectory (Default: 1000)
    :param save_dir: Optional directory path to save the plots in PDF format
    :return: Average cost over trajectories
    """
    all_trajectories_costs = []

```

```
    policy_from_K = generate_policy_from_K(K)

    for i in range(num_trajectories):
        print(f"Running simulation for trajectory {i+1}/{num_trajectories}...")
        env = initialize_environment(n, m)
        P = compute_P(env)
        K_new = compute_K(env, P)
        trajectory_cost = policy_evaluation(policy_from_K, env, trajectory_length)
        all_trajectories_costs.append(trajectory_cost)

    costs_array = np.array(all_trajectories_costs)

    print(f"Shape of costs_array: {costs_array.shape}")
    print(f"First five values from costs_array:\n{costs_array[:5]}")

    # Plotting cumulative costs for each trajectory
    plt.figure(figsize=GLOBAL_FIG_SIZE)
    for trajectory_cost in all_trajectories_costs:
        plt.plot([0, trajectory_length], [0, trajectory_cost], alpha=0.5) # Straight line from 0 to total cost
    plt.ylim(0, np.max(all_trajectories_costs) * 1.1)  # Adjust y-limits
    plt.title(f"Cumulative Costs for {num_trajectories} Trajectories each of Length {trajectory_length}")
    plt.xlabel("Episode")
    plt.ylabel("Cumulative Cost")
    plt.tight_layout()
    if save_dir:
        plt.savefig(Path(save_dir).joinpath("q4_a_mean_cost_plot.pdf"), format='pdf', dpi=300)
    else:
        plt.show()

    # Return the mean of the costs for informational purposes
    avg_cost = np.mean(costs_array)
    print(f"Average cost: {avg_cost}")
    return avg_cost

n, m = 2, 2
env_initial = initialize_environment(n, m)
P_initial = compute_P(env_initial)
K_Part_A = compute_K(env_initial, P_initial)
avg_cost = run_q4a_simulations(K_Part_A, save_dir= Path.cwd())
```

Code Listing 4: Q4 Python Code

APPENDIX B: Q4 PYTHON CLASS QLEARNINGWITHAPPROXIMATION.PY

```
""" QLearningWithApproximation.py

This class implements Q-learning for the discounted stochastic LQ control problem. It's designed
specifically to address Homework 3, Question 4, providing solutions for parts C, D, E, and F.

Usage:
- For Part C: Initialize the class with policy_type as 'greedy' and set the required parameters.
- For Part D: Initialize with policy_type as 'greedy_optimized' and adjust parameters as needed.
- For Part E: Use policy_type as 'epsilon_greedy', specify a list of epsilon values, and set other parameters accordingly.
- For Part F: The analysis can be derived from the results obtained in the earlier parts and by leveraging
  the methods provided in this class.

After initialization, run the `run_simulations` method followed by `plot_results` to visualize the results.

Note: Ensure all the other necessary parameters (like learning rate, gamma, etc.) are set appropriately
for each part of the question.
"""
from pathlib import Path
import numpy as np
from scipy.linalg import solve_discrete_are
import matplotlib.pyplot as plt
import scienceplots
from tqdm import tqdm
import warnings

# Globally set fonts and figure size
GLOBAL_FIG_SIZE = (10, 6)
import matplotlib as mpl
mpl.rcParams['font.size'] = 14
mpl.rcParams['axes.labelsize'] = 16
mpl.rcParams['xtick.labelsize'] = 14
mpl.rcParams['ytick.labelsize'] = 14

GLOBAL_FIG_SIZE = (10, 6)  # Global figure size

def init_theta(n, m, fun=np.random.randn, seed=1, initialization='random'):
    """Initialize Q-function weights (theta) with reproducibility.
    This function was provided to us. I have incorporated Xavier
    intialization to deal with numerical instability.

```

```
    :param n: State dimension
    :param m: Action dimension
    :param fun: Function to generate weights, defaults to Gaussian
    :param seed: Seed for reproducibility, defaults to 1
    :param initialization: Type of initialization - 'random' or 'xavier',
        ↪ defaults to 'random'
    :return: List containing theta weight matrices A, B, C and constant
    """
    np.random.seed(seed)

    if initialization == 'random':
        A = fun(n, n)
        B = fun(m, m)
        C = fun(n, m)
    elif initialization == 'xavier':
        # Xavier/Glorot Initialization
        limit_A = np.sqrt(6.0 / (n + n))
        A = np.random.uniform(-limit_A, limit_A, (n, n))

        limit_B = np.sqrt(6.0 / (m + m))
        B = np.random.uniform(-limit_B, limit_B, (m, m))

        limit_C = np.sqrt(6.0 / (n + m))
        C = np.random.uniform(-limit_C, limit_C, (n, m))
    else:
        raise ValueError("Unknown initialization method provided.")

    const = 0
    return [A, B, C, const]

def is_near_singular(matrix, threshold=1e12):
    """Check if a matrix is near-singular by inspecting its condition
        ↪ number."""
    det = np.linalg.det(matrix)
    return np.abs(det) < threshold

class QLearningWithApproximation:
    """Q-Learning with approximation for Linear-Quadratic control.

    This class implements Q-learning using a quadratic function
        ↪ approximation
    for the Q-function. The approximation and greedy policy are derived
    analytically based on the problem analysis.

    Technical Note:
    Due to the nature of Linear-Quadratic control problems and the
        ↪ quadratic
    function approximation, numerical instabilities can arise. This can
        ↪ result
```

```
    in near-singular matrices. The class includes provisions to detect and handle
    these situations, specifically within the `greedy_policy_optimized` method
    where action calculations can be impacted by singular matrices.
    """
    def __init__(
        self,
        env,
        alpha,
        gamma,
        seed,
        policy_type = "greedy_optimized",
        epsilons=None,
        initialization="xavier",
        anneal_every=100,
        anneal_factor=0.9,
    ):
        """
        Initialize the Q-learning agent with approximation model.

        :param env: The LQ environment.
        :param alpha: The learning rate.
        :param gamma: The discount factor.
        :param seed: Seed for random number generation.
        :param policy_type: Type of the policy being used. Default is "greedy_optimized".
        :param epsilon: Vector containing the probability of choosing a random action. Defaults to 0.3.
        :param initialization: Initialization scheme for theta. Default is "xavier".
        :param anneal_every: Number of steps after which the learning rate is annealed. Default is 100.
        :param anneal_factor: Factor by which the learning rate is multiplied during annealing. Default is 0.9.
        """
        self.env = env
        self.alpha = alpha
        self.gamma = gamma
        self.seed = seed
        self.policy_type = policy_type
        self.epsilons = epsilons if epsilons else [0.3]
        self.initialization = initialization
        self.n = env.A.shape[0]
        self.m = env.B.shape[1]
        self.anneal_every = anneal_every
        self.anneal_factor = anneal_factor
        self.theta = init_theta(
          self.n, self.m, seed=seed, initialization=initialization
        )
```

```
128         self.policy_function = self._get_policy_function()
129         self._print_params()
130         self.training_results = {}
131 
132     def _get_policy_function(self):
133         """Returns the policy function based on policy_type."""
134         policy_mapping = {
135             "greedy": self.greedy_policy,
136             "greedy_optimized": self.greedy_policy_optimized,
137             "epsilon_greedy": self.epsilon_greedy_policy
138         }
139 
140         if self.policy_type not in policy_mapping:
141             error_msg = (f"Invalid policy type '{self.policy_type}'. "
142                          f"Available policies are: {', '.join(policy_mapping
                                 ↪ .keys())}.")
143             raise ValueError(error_msg)
144 
145         return policy_mapping[self.policy_type]
146 
147     def _print_params(self):
148         """Prints the initialized parameters."""
149         print("Intialized parameters for QLearningWithApproximation:")
150         print(f"alpha: {self.alpha}")
151         print(f"gamma: {self.gamma}")
152         print(f"seed: {self.seed}")
153         print(f"policy_type: {self.policy_type}")  # Added this line
154         print(f"initialization: {self.initialization}")
155 
156     def q_function_approximation(self, state, action):
157         """Evaluate Q-function approximation for state-action pair.
158 
159         Uses quadratic function approximation with analytically
160         derived parameters.
161 
162         :param state: Current state
163         :param action: Current action
164         :return: Q-value for state-action pair
165         """
166         theta1, theta2, theta3, theta4 = self.theta
167         q = state.T @ theta1.T @ theta1 @ state
168         q += action.T @ theta2.T @ theta2 @ action
169         q += 2 * state.T @ theta3 @ action
170         q += theta4
171         return q
172 
173     def greedy_policy(self, state):
174         """Determine optimal action for state using Q-function.
175 
176         Computes greedy action by maximizing over Q-function
```

```
        approximation. Uses relationship derived analytically.

        :param state: Current state
        :return: Optimal action for given state
        """
        # Get weights
        theta2, theta3 = self.theta[1], self.theta[2]

        # Compute inverse
        theta2_inv = np.linalg.inv(theta2.T @ theta2)

        # Compute optimal action directly using formula
        action = -theta2_inv @ theta3.T @ state

        # Reshape and cast to correct data type
        action = action.reshape([self.env.action_space.shape[0], 1])
        action = action.astype(self.env.action_space.dtype)

        return action

    def greedy_policy_optimized(self, state):
        """Determine optimal action for state using Q-function (Optimized
            ↪ Version).

        Computes greedy action by maximizing over Q-function
        approximation. Uses relationship derived analytically. This
        version is optimized by caching the inverse computation for
        theta2, preventing redundant and expensive inverse operations
        when theta2 hasn't changed.

        :param state: Current state
        :return: Optimal action for given state
        """
        # Check if theta2_inv is cached and if theta2 has changed since the
            ↪  last cache
        if not hasattr(self, 'theta2_inv') or np.any(self.theta[1] != self.
            ↪ cached_theta2):
            matrix_to_invert = self.theta[1].T @ self.theta[1]

            # Check for near-singularity and adjust slightly if needed
            max_attempts = 100
            attempts = 0
            while is_near_singular(matrix_to_invert) and attempts <
                ↪ max_attempts:
                matrix_to_invert += np.eye(matrix_to_invert.shape[0]) * 1e
                    ↪ -3
                attempts += 1
            if attempts == max_attempts:
                warnings.warn("Matrix is still near singular after multiple
                    ↪  adjustments.")
```

```
            self.cached_theta2 = self.theta[1].copy()
            #self.theta2_inv = np.linalg.inv(matrix_to_invert)
            self.theta2_inv = np.linalg.pinv(matrix_to_invert)

        # Compute optimal action directly using formula
        action = -self.theta2_inv @ self.theta[2].T @ state

        # Reshape and cast to correct data type
        action = action.reshape([self.env.action_space.shape[0], 1])
        action = action.astype(self.env.action_space.dtype)

        return action

    def epsilon_greedy_policy(self, state, epsilon=0.3):
        """Choose an action using an epsilon-greedy policy.

        Given the current state, this method decides whether to take a
            ↪ random action
        (with a probability of epsilon) or to take the action that
            ↪ maximizes the current
        Q-function estimate (with a probability of 1-epsilon).

        :param state: The current state of the environment.
        :param epsilon: The probability of choosing a random action.
            ↪ Defaults to 0.3.
        :return: The chosen action based on the epsilon-greedy policy.

        """
        if np.random.random() < epsilon:
            # With probability epsilon, take a random action
            return self.env.sample_random_action()
        else:
            # Otherwise, take the action given by the greedy policy
            return self.greedy_policy_optimized(state)


    def train(self, episodes, max_steps):
        """Train model using Q-learning with approximation.
           Agent samples actions from selected policy such as
           exploratory policy like epislon-greedy.

        Updates Q-values by estimating future rewards assuming greedy
        actions. epislon-greedy provides exploration, Q-update assumes
        exploitation.

        :param episodes: Number of episodes or trajectories.
        :param max_steps: Max steps per episode.
        :return: Dictionary with TD errors, cumaltive costs, K_QL matrix.
        """
```

```
        print(f"Starting training with {episodes} episodes, each with a maximum of {max_steps} steps per episode.")
        td_errors = np.zeros((max_steps, episodes))  # Array to store TD errors over steps
        cumulative_costs = np.zeros((max_steps, episodes))  # Array to store cumulative costs over steps

        for episode in range(episodes):
            print(f"Episode {episode + 1} started.")
            state = self.env.reset()  # Reset environment for the new episode
            episode_td_errors = []  # Store TD errors for this episode
            episode_costs = []  # Store costs/rewards for this episode

            # Iterate over steps within each episode
            for step in tqdm(range(max_steps), desc=f'Episode {episode + 1} '):

                # Choose an action based on the current policy (e.g., epsilon-greedy)
                action = self.policy_function(state)

                # Try taking the selected action in the environment
                # If the action is invalid due to some constraints,
                # select another action until a valid one is found
                while True:
                    try:
                        next_state, reward, done, _ = self.env.step(action)
                        break
                    except AssertionError:
                        action = self.policy_function(state)

                # Estimate the value (Q-value) of the best possible future action from next state
                # This is the "looking ahead" step where we assume the agent will always act optimally in the future
                max_q_next = self.q_function_approximation(next_state, self.greedy_policy(next_state))

                # Compute the TD error for the current state and action
                td_error = reward + self.gamma * max_q_next - self.q_function_approximation(state, action)

                # Store the TD error and cost for this step
                episode_td_errors.append(float(td_error))
                episode_costs.append(float(reward))

                # Apply learning rate annealing
                if step % self.anneal_every == 0:
                  self.alpha *= self.anneal_factor
```

```
308
309                 theta1, theta2, theta3, theta4 = self.theta
310                 theta1 += self.alpha * td_error * np.outer(state, state)
311                 theta2 += self.alpha * td_error * np.outer(action, action)
312                 theta3 += self.alpha * td_error * np.outer(state, action)
313                 theta4 += self.alpha * td_error
314
315                 # Update the internal theta with the new values
316                 self.theta = [theta1, theta2, theta3, theta4]
317
318                 state = next_state
319                 if done:
320                     break
321
322                 # Print TD error and cost every 500 steps
323                 if step % 500 == 0:
324                     print(f"Step {step}, Episode {episode + 1}: TD Error: {
                        ↪ td_error}, Cost: {np.nanmean(episode_costs)}")
325
326             # Convert the lists to arrays and pad with NaN if necessary
327             episode_td_errors = np.array(episode_td_errors)
328             episode_costs = np.array(episode_costs)
329             pad_len = max_steps - len(episode_td_errors)
330             if pad_len > 0:
331                 episode_td_errors = np.pad(episode_td_errors, (0, pad_len),
                        ↪  mode='constant', constant_values=np.nan)
332                 episode_costs = np.pad(episode_costs, (0, pad_len), mode='
                        ↪ constant', constant_values=np.nan)
333
334             td_errors[:, episode] = episode_td_errors  # Store the TD
                    ↪ errors for this episode
335             cumulative_costs[:, episode] = np.cumsum(episode_costs)  #
                    ↪ Store the cumulative costs for this episode
336
337             print(f"Episode {episode + 1} concluded. Avg TD-Error: {np.
                    ↪ nanmean(episode_td_errors)}, Avg Cost: {np.nanmean(
                    ↪ episode_costs)}")
338
339         print(f"Training completed.")
340         theta2, theta3 = self.theta[1], self.theta[2]
341         K_QL = -np.linalg.inv(theta2.T @ theta2) @ theta3.T
342         training_results = {"K_QL": K_QL, "TD_ERRORS": td_errors, "
                ↪ CUMULATIVE_COSTS": cumulative_costs}
343         return training_results
344
345     def run_simulations(self, episodes, max_steps):
346         """ Run multiple simulations for training the agent with different
                ↪ epsilon values.
347
348         :param episodes: Number of episodes for training.
```

```
        :param max_steps: Maximum steps per episode.
        """
        # Initialize or reset training results
        self.training_results = {}

        if self.policy_type in ["greedy", "greedy_optimized"]:
            print("\nStarting simulation with", self.policy_type)
            result = self.train(episodes, max_steps)
            self.training_results[self.policy_type] = {
                'K_QL': result['K_QL'],
                'TD_ERRORS': result['TD_ERRORS'],
                'CUMULATIVE_COSTS': result['CUMULATIVE_COSTS']
            }
        elif self.policy_type == "epsilon_greedy":
            self.training_results[self.policy_type] = {}  # Initialize the key with an empty dictionary
            for epsilon in self.epsilons:
                print(f"\nStarting simulation with epsilon = {epsilon}")
                self.epsilon = epsilon  # Modify the epsilon value of the agent
                result = self.train(episodes, max_steps)
                # Store the results under the appropriate epsilon value
                self.training_results[self.policy_type][str(epsilon)] = {
                    'K_QL': result['K_QL'],
                    'TD_ERRORS': result['TD_ERRORS'],
                    'CUMULATIVE_COSTS': result['CUMULATIVE_COSTS']
                }
        else:
            raise ValueError(f"Invalid policy type '{self.policy_type}'")

    def plot_results(self, save_dir=None):
        """ Plot the results from multiple simulations.

        :param save_dir: Optional directory path to save the plots.
        """
        plt.figure(figsize=GLOBAL_FIG_SIZE)

        # If results for the greedy policy exist, plot them
        for policy in ['greedy', 'greedy_optimized']:
            if policy in self.training_results:
                result = self.training_results[policy]
                plt.plot(np.nanmean(result['TD_ERRORS'], axis=1), label=f'{policy} Policy')

        # If results for the epsilon-greedy policy exist, plot them
        if 'epsilon_greedy' in self.training_results:
            for epsilon, result in self.training_results['epsilon_greedy'].items():
                plt.plot(np.nanmean(result['TD_ERRORS'], axis=1), label=f"Epsilon={epsilon} Policy")
```

```
394
395         plt.xlabel('Steps')
396         plt.ylabel('Average TD-Error')
397         plt.title(f'Average TD Error over Episodes')
398         plt.legend()
399
400         if save_dir:
401             plt.savefig(save_dir.joinpath('td_errors_comparison.pdf'),
                ↪ format='pdf', dpi=300)
402
403         plt.show()
404
405         plt.figure(figsize=GLOBAL_FIG_SIZE)
406
407         # If results for the greedy policy or greedy_optimized exist, plot
            ↪ cumulative costs
408         for policy in ['greedy', 'greedy_optimized']:
409             if policy in self.training_results:
410                 result = self.training_results[policy]
411                 plt.plot(np.nanmean(result['CUMULATIVE_COSTS'], axis=1),
                    ↪ label=f'{policy} Policy')
412
413         # If results for the epsilon-greedy policy exist, plot cumulative
            ↪ costs
414         if 'epsilon_greedy' in self.training_results:
415             for epsilon, result in self.training_results['epsilon_greedy'].
                ↪ items():
416                 plt.plot(np.nanmean(result['CUMULATIVE_COSTS'], axis=1),
                    ↪ label=f"Epsilon={epsilon} Policy")
417
418         plt.xlabel('Steps')
419         plt.ylabel('Average Cumulative Cost')
420         plt.title(f'Average Cumulative Cost over Episodes')
421         plt.legend()
422
423         if save_dir:
424             plt.savefig(save_dir.joinpath('cumulative_costs_comparison.pdf'
                ↪ ), format='pdf', dpi=300)
425
426         plt.show()
```

Code Listing 5: Q4 Python Code

APPENDIX C: Q5 APPROXIMATE INVENTORY CONTROL

```python
import numpy as np
from multiprocessing import Pool

def init_params(dt=1, ht=4, qt=2, T=10, actions=list(range(21)),
                states=np.arange(-15, 15.1, 0.1), demands=np.arange(0,
                    ↪ 10.1, 0.1)):
    """
    Initialize and return the given parameters and discretized spaces.

    :param dt: Fixed cost for placing an order.
    :param ht: Holding cost per unit of inventory.
    :param qt: Penalty cost for stockout per unit.
    :param T: Time horizon.
    :param actions: Possible actions to take.
    :param states: Possible states for the inventory.
    :param demands: Possible demand values.
    :return: Dictionary of parameters and discretized spaces.
    """
    params = {
        "dt": dt,
        "ht": ht,
        "qt": qt,
        "T": T,
        "actions": actions,
        "states": states,
        "demands": demands,
    }
    print("Parameters initialized.")
    return params

def initialize_value_function(T, states, ht, qt):
    """
    Initialize the value function V with terminal costs at time T and
        ↪ infinity for other times.

    :param T: Time horizon.
    :param states: List of possible inventory states.
    :param ht: Holding cost per unit of inventory.
    :param qt: Penalty cost for stockout per unit.
    :return: Dictionary representing the value function V initialized with
        ↪ terminal costs and infinity for other times.
    """
    V = {}
    for t in range(T + 1):
        for s in states:
            if t == T:
                V[(t, s)] = max(ht * s, -qt * s)
            else:
```

```
                V[(t, s)] = float('inf')
    print("Value function initialized with terminal costs and infinities.")
    return V

from multiprocessing import Pool

def compute_expected_cost_for_state(args):
    """
    Worker function to compute the expected cost for a given inventory state.

    :param args: Tuple containing state s, params dictionary, value function V, and time t.
    :return: Tuple containing state s, minimum expected cost, and the corresponding optimal action.
    """
    s, params, V, t = args
    min_expected_cost = float('inf')
    optimal_action = None

    # Calculate expected cost for each possible action
    for a in params["actions"]:
        expected_cost = 0
        for w in params["demands"]:
            s_next = s + a - w
            s_next = min(max(s_next, params["states"][0]), params["states"][-1])
            s_next = min(params["states"], key=lambda state: abs(state - s_next))
            cost = params["dt"] * a + max(params["ht"] * s, -params["qt"] * s)
            expected_cost += (1 / len(params["demands"])) * (cost + V[(t + 1, s_next)])
        if expected_cost < min_expected_cost:
            min_expected_cost = expected_cost
            optimal_action = a

    return s, min_expected_cost, optimal_action

def compute_value_function(params, V):
    """
    Compute optimal value function and policy using backward dynamic programming.

    :param params: Dictionary of parameters and discretized spaces.
    :param V: Initialized value function dictionary.
    :return: Tuple of dictionaries - the first is the optimal value function and the second is the optimal policy.
    """
    print("Starting backward dynamic programming computation...")
```

```
    policy = {}
    # Backward computation through time steps
    for t in range(params["T"] - 1, -1, -1):
        print(f"Processing time step {t} (moving backwards)...")
        # Parallel processing to compute expected cost for each state
        with Pool() as pool:
            results = pool.map(compute_expected_cost_for_state, [(s, params
                ↪ , V, t) for s in params["states"]])
        for s, min_expected_cost, optimal_action in results:
            V[(t, s)] = min_expected_cost
            policy[(t, s)] = optimal_action
    print("Completed backward computation of the value function.")
    return V, policy


def q5a_run_simulations():
    """
    Run simulations and return value function and policy at starting state
        ↪ s=0.

    :return: Tuple of dictionaries - the first is the optimal value
        ↪ function and the second is the optimal policy.
    """
    print("Starting the simulation...")
    params = init_params()
    V = initialize_value_function(params["T"], params["states"], params["ht
        ↪ "], params["qt"])
    V, policy = compute_value_function(params, V)

    closest_zero_state = min(params["states"], key=lambda state: abs(state)
        ↪ )

    initial_value = V[(0, closest_zero_state)]
    print(f"Optimal value function at starting state s0: {initial_value}")

    return V, policy  # Returning the value function and policy for further
        ↪  use

# Execute the simulation
V, policy = q5a_run_simulations()
```

Code Listing 6: Q5 A Python Code

```
import numpy as np
from numpy.linalg import inv
from multiprocessing import Pool

def quadratic_approximation(V, states, t):
    """ Approximate the value function using a quadratic function of
    the form: at*s^2 + bt*s + ct.

```

```
    :param V: Current value function dictionary.
    :param states: List of possible inventory states.
    :param t: Current time step.
    :return: Coefficients a, b, c for the quadratic approximation.
    """
    A = np.vstack([states**2, states, np.ones(len(states))]).T
    B = np.array([V[(t, s)] for s in states])

    # Solve for a, b, c using least squares
    a, b, c = np.linalg.lstsq(A, B, rcond=None)[0]
    return a, b, c

def compute_value_function(params, V):
    """ Compute optimal value function and policy using backward dynamic
        ↪ programming
    with quadratic approximation.

    :param params: Dictionary of parameters and discretized spaces.
    :param V: Initialized value function dictionary.
    :return: Tuple of dictionaries - the first is the optimal value
        ↪ function
             and the second is the optimal policy.
    """
    policy = {}
    coefficients = {}

    for t in range(params["T"] - 1, -1, -1):
        # Approximate next step's value function using quadratic function
        a, b, c = quadratic_approximation(V, params["states"], t + 1)
        coefficients[t + 1] = (a, b, c)

        for s in params["states"]:
            min_expected_cost = float('inf')
            optimal_action = None

            for a in params["actions"]:
                expected_cost = params["dt"] * a  # Initial cost of taking
                    ↪ the action

                for w in params["demands"]:
                    s_next = s + a - w
                    s_next = min(max(s_next, params["states"][0]), params["
                        ↪ states"][-1])
                    s_next = min(params["states"], key=lambda state: abs(
                        ↪ state - s_next))

                    # Immediate cost
                    immediate_cost = params["ht"] * max(s_next, 0) + params
                        ↪ ["qt"] * max(-s_next, 0)
                    # Approximate next step's value using quadratic
```

```
                        ↪ function
                    approximated_value = a * s_next**2 + b * s_next + c
                    expected_cost += (1 / len(params["demands"])) * (
                        ↪ immediate_cost + approximated_value)

                if expected_cost < min_expected_cost:
                    min_expected_cost = expected_cost
                    optimal_action = a

            V[(t, s)] = min_expected_cost
            policy[(t, s)] = optimal_action

    return V, policy, coefficients

def q5b_run_simulations():
    """
    Run simulations using approximate value iteration and return value
        ↪ function, policy and coefficients.

    :return: Tuple containing dictionaries of value function, policy, and
        ↪ coefficients.
    """
    params = init_params()
    V = initialize_value_function(params["T"], params["states"], params["ht
        ↪ "], params["qt"])
    V, policy, coefficients = compute_value_function(params, V)

    closest_zero_state = min(params["states"], key=lambda state: abs(state)
        ↪ )
    initial_value = V[(0, closest_zero_state)]
    print(f"Optimal value function at starting state s0: {initial_value}")

    return V, policy, coefficients  # Including coefficients in the return

V, policy, coefficients = q5b_run_simulations()
```

Code Listing 7: Q5 B Python Code

```
import numpy as np
from numpy.linalg import inv

def init_params(states=np.arange(-5, 6), actions=np.arange(-5, 6),
                demands=np.arange(0, 11, 0.1), T=10, dt=1, ht=4, qt=2):
    """ Initializes the parameters for the simulation.

    :param states: List of possible inventory states. Default is from -5 to
        ↪  5.
    :param actions: List of possible actions. Default is from -5 to 5.
    :param demands: List of possible demands. Default is from 0 to 10 with
        ↪ step 0.1.
    :param T: Horizon time. Default is 10.
```

```
    :param dt: Cost per order. Default is 1.
    :param ht: Holding cost per unit. Default is 4.
    :param qt: Penalty cost per unit of stockout. Default is 2.
    :return: Dictionary of parameters and discretized spaces.
    """
    params = {
        "states": states,
        "actions": actions,
        "demands": demands,
        "T": T,
        "dt": dt,
        "ht": ht,
        "qt": qt
    }
    return params


def initialize_value_function(T, states, ht, qt):
    """ Initializes the value function based on terminal costs and other
        ↪ times.

    :param T: Horizon time.
    :param states: List of possible inventory states.
    :param ht: Holding cost per unit.
    :param qt: Penalty cost per unit of stockout.
    :return: Dictionary containing the initialized value function.
    """
    V = {}
    for s in states:
        V[(T, s)] = ht * max(s, 0) + qt * max(-s, 0)
    return V

def q5c_run_simulations():
    """ Run simulations for Q5C using approximate value iteration and
        ↪ return value function,
    policy, and coefficients.

    :return: Tuple containing dictionaries of value function, policy, and
        ↪ coefficients.
    """
    params = init_params()
    V = initialize_value_function(params["T"], params["states"], params["ht
        ↪ "], params["qt"])
    V, policy, coefficients = compute_value_function(params, V)

    closest_zero_state = min(params["states"], key=lambda state: abs(state)
        ↪ )
    initial_value = V[(0, closest_zero_state)]
    print(f"Optimal value function at starting state s0: {initial_value}")

```

```
    return V, policy, coefficients  # Including coefficients in the return

V_q5c, policy_q5c, coefficients_q5c = q5c_run_simulations()
print(f"\nCoefficients Q5C: \n{coefficients_q5c}")
```

Code Listing 8: Q5 C Python Code

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import scienceplots
from pathlib import Path

GLOBAL_FIG_SIZE = (10, 6)  # Global figure size
plt.style.use('science')
mpl.rcParams['font.size'] = 14
mpl.rcParams['axes.labelsize'] = 16
mpl.rcParams['xtick.labelsize'] = 14
mpl.rcParams['ytick.labelsize'] = 14

def extract_values_from_dict(V_dict, time, states):
    """ Extract values from dictionary for a given time and list of states.
        ↪  """
    return [V_dict.get((time, state), 0) for state in states]

def save_plot(name, path):
    if path:
        plt.savefig(path / f"{name}.pdf", format='pdf', dpi=300)

def plot_q5d(V_q5a, V_q5b, V_q5c, save_fig=None):
    t0_states = np.arange(-5, 6)
    tT_states = np.arange(-15, 16)

    plt.figure(figsize=GLOBAL_FIG_SIZE)
    plt.plot(t0_states, extract_values_from_dict(V_q5a, 0, t0_states),
        ↪ label='Optimal Q5a')
    plt.plot(t0_states, extract_values_from_dict(V_q5b, 0, t0_states),
        ↪ label='Approx. Q5b')
    plt.plot(t0_states, extract_values_from_dict(V_q5c, 0, t0_states),
        ↪ label='Approx. Q5c')
    plt.xlabel('State')
    plt.ylabel('Value Function')
    plt.legend()
    plt.tight_layout()
    plt.title('Value Function at t=0')
    save_plot("value_function_t0", save_fig)
    plt.show()

    plt.figure(figsize=GLOBAL_FIG_SIZE)
    plt.plot(tT_states, extract_values_from_dict(V_q5a, -1, tT_states),
        ↪ label='Optimal Q5a')
```

```
    plt.plot(tT_states, extract_values_from_dict(V_q5b, -1, tT_states),
        ↪ label='Approx. Q5b')
    plt.plot(tT_states, extract_values_from_dict(V_q5c, -1, tT_states),
        ↪ label='Approx. Q5c')
    plt.xlabel('State')
    plt.ylabel('Value Function')
    plt.legend()
    plt.tight_layout()
    plt.title('Value Function at t=T')
    save_plot("value_function_tT", save_fig)
    plt.show()

def plot_q5e(policy_q5a, policy_q5b, policy_q5c, save_fig=None):
    t0_states = np.arange(-15, 16)

    plt.figure(figsize=GLOBAL_FIG_SIZE)
    plt.plot(t0_states, extract_values_from_dict(policy_q5a, 0, t0_states),
        ↪  label='Optimal Policy Q5a')
    plt.plot(t0_states, extract_values_from_dict(policy_q5b, 0, t0_states),
        ↪  label='Greedy Policy Q5b')
    plt.plot(t0_states, extract_values_from_dict(policy_q5c, 0, t0_states),
        ↪  label='Greedy Policy Q5c')
    plt.xlabel('State')
    plt.ylabel('Action')
    plt.legend()
    plt.title('Policy at t=0')
    plt.tight_layout()
    save_plot("policy_t0", save_fig)
    plt.show()

def plot_q5f(V_q5a, V_q5b, V_q5c, policy_q5a, policy_q5b, policy_q5c,
    ↪ save_fig=None):
    t0_states = np.arange(-15, 16)

    plt.figure(figsize=GLOBAL_FIG_SIZE)
    plt.plot(t0_states, extract_values_from_dict(V_q5a, 0, t0_states),
        ↪ label='Optimal Policy Q5a')
    plt.plot(t0_states, extract_values_from_dict(V_q5b, 0, t0_states),
        ↪ label='Greedy Policy Q5b')
    plt.plot(t0_states, extract_values_from_dict(V_q5c, 0, t0_states),
        ↪ label='Greedy Policy Q5c')
    plt.xlabel('State')
    plt.ylabel('Value Function')
    plt.legend()
    plt.title('Value Function at t=0 for Different Policies')
    plt.tight_layout()
    save_plot("value_function_policy_t0", save_fig)
    plt.show()

plot_q5d(V_q5a, V_q5b, V_q5c, save_fig=save_path)
```

```
plot_q5e(policy_q5a, policy_q5b, policy_q5c, save_fig=save_path)
plot_q5f(V_q5a, V_q5b, V_q5c, policy_q5a, policy_q5b, policy_q5c, save_fig=
    ↪ save_path)
```

Code Listing 9: Q5 Remainder Python Code

References

[1] B. Bordelon, P. Masset, H. Kuo, and C. Pehlevan. "Dynamics of Temporal Difference Reinforcement Learning". In: *arXiv preprint arXiv:2307.04841* (2023).

[2] F. Incertis. "Optimal Stochastic Control of Linear Systems with State and Control Dependent Noise: Efficient Computational Algorithms". In: *Nonlinear Stochastic Problems*. Springer, 1983, pp. 243–253.

[3] Y. Chen, L. Schomaker, and M. A. Wiering. "An Investigation Into the Effect of the Learning Rate on Overestimation Bias of Connectionist Q-learning." In: *ICAART (2)*. 2021, pp. 107–118.

[4] J. Elkins, R. Sood, and C. Rumpf. "Adaptive continuous control of spacecraft attitude using deep reinforcement learning". In: *2020 AAS/AIAA Astrodynamics Specialist Conference*. AIAA Reston, VA. 2020, pp. 420–475.

[5] P. N. Kolm, R. Tütüncü, and F. J. Fabozzi. "60 Years of portfolio optimization: Practical challenges and current trends". In: *European Journal of Operational Research* 234.2 (2014), pp. 356–371.

[6] T. L. Meng and M. Khushi. "Reinforcement learning in financial markets". In: *Data* 4.3 (2019), p. 110.

[7] J. Jang and N. Seong. "Deep reinforcement learning for stock portfolio optimization by connecting with modern portfolio theory". In: *Expert Systems with Applications* (2023), p. 119556.

[8] G. Coqueret and E. André. "Factor investing with reinforcement learning". In: *Available at SSRN 4103045* (2022).

[9] E. Lezmi, T. Roncalli, and J. Xu. "Multi-Period Portfolio Optimization". In: *Available at SSRN* (2022).