Problem 1: MDPs and Dynamic Programming (18 points)

A mobile robot is moving around on a rechargeable battery. There are three battery level states: *high*, *low*, and *off*. In the first two states, the robot may move *fast* or *slow*, while in *off*, the robot may only *recharge*. Transitions are stochastic; moving *fast* is guaranteed to lower the robot's battery level, while moving *slow* may sometimes do so. A transition and reward function are defined for this robot as follows.

s	a	s'	$T(s,a,s')$	$R(s,a,s')$
high	*fast*	*low*	1.0	+3
high	*slow*	*high*	0.5	+2
high	*slow*	*low*	0.5	+2
low	*fast*	*off*	1.0	+2
low	*slow*	*low*	0.75	+2
low	*slow*	*off*	0.25	+1
off	*recharge*	*high*	1.0	−2

1. (4 pts) Consider the policy π in which the robot goes *fast* in both the *high* and *low* states and *recharges* in the *off* state. Write down the system of linear equations describing the value function V^π in terms of γ, and then solve for the values using $\gamma = 0.5$. (You don't need to do the last step by hand, but please state if you are using any programs to help with it.)

$$V^\pi(s) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]$$

$$V^\pi(high) = T(high, fast, low)[R(high, fast, low) + \gamma V^\pi(low)] = 1.0 * [3 + \gamma V^\pi(low)] = 3 + \gamma V^\pi(low)$$

$$V^\pi(low) = T(low, fast, off)[R(low, fast, off) + \gamma V^\pi(off)] = 1.0 * [2 + \gamma V^\pi(off)] = 2 + \gamma V^\pi(off)$$

$$V^\pi(off) = T(off, recharge, high)[R(off, recharge, high) + \gamma V^\pi(high)] = 1.0 * [-2 + \gamma V^\pi(high)]$$
$$= -2 + \gamma V^\pi(high)$$

$$\gamma = 0.5$$

$$V^\pi(high) = 3 + 0.5V^\pi(low) = 3 + 0.5 * (2 + 0.5V^\pi(off)) = 3 + 0.5 * (2 + 0.5(-2 + 0.5V^\pi(high)))$$

$$V^\pi(high) = 3.5 + 1/8 * V^\pi(high)$$

$$\Rightarrow V^\pi(high) = 4, V^\pi(low) = 2, V^\pi(off) = 0$$

2. (6 pts) Suppose the values that you obtained above are the time-limited values V_i in iteration i of value iteration, which is being used here to find the optimal policy π^*. Show the computations done in the next iteration, and find the next set of time-limited values V_{i+1}.

$$V^{\pi}_{i+1}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{i}(s')]$$

$$V^{\pi}_{i+1}(high, a = fast) = T(high, fast, low)[R(high, fast, low) + \gamma V^{\pi}_{i}(low)] = 1.0 * [3 + \gamma V^{\pi}_{i}(low)]$$
$$= 3 + \gamma V^{\pi}_{i}(low)$$
$$V^{\pi}_{i+1}(high,\ a = slow) = T(high, slow, high)[R(high, slow, high) + \gamma V^{\pi}_{i}(low)]$$
$$+ T(high, slow, low)[R(high, slow, low) + \gamma V^{\pi}_{i}(low)]$$
$$= 0.5 * [2 + \gamma V^{\pi}_{i}(low)] + 0.5 * [2 + \gamma V^{\pi}_{i}(low)] = 2 + \gamma V^{\pi}_{i}(low)$$
$$V^{\pi}_{i+1}(high) = max(3 + \gamma V^{\pi}_{i}(low), 2 + \gamma V^{\pi}_{i}(low)) = max(4, 3) = 4$$

$$V^{\pi}_{i+1}(low,\ a = fast) = T(low, fast, off)[R(low, fast, off) + \gamma V^{\pi}_{i}(off)]$$
$$= 1.0 * [2 + \gamma V^{\pi}_{i}(off)] = 2 + \gamma V^{\pi}_{i}(off)$$
$$V^{\pi}_{i+1}(low,\ a = slow) = T(low, slow, low)[R(low, slow, low) + \gamma V^{\pi}_{i}(low)]$$
$$+ T(high, slow, off)[R(high, slow, off) + \gamma V^{\pi}_{i}(off)]$$
$$= 0.75 * [2 + \gamma V^{\pi}_{i}(low)] + 0.25 * [1 + \gamma V^{\pi}_{i}(off)]$$
$$= 1.75 + 0.75\gamma V^{\pi}_{i}(low) + 0.25\gamma V^{\pi}_{i}(off)]$$
$$V^{\pi}_{i+1}(low) = max(2 + \gamma V^{\pi}_{i}(off), 1.75 + 0.75\gamma V^{\pi}_{i}(low) + 0.25\gamma V^{\pi}_{i}(off)] = max(2, 2.5) = 2.5$$

$$V^{\pi}_{i+1}(off) = T(off, recharge, high)[R(off, recharge, high) + \gamma V^{\pi}_{i+1}(high)] = 1.0 * [-2 + \gamma V^{\pi}_{i}(high)]$$
$$=- 2 + \gamma V^{\pi}_{i}(high) = 0$$

Next set of time limit values:

$V^{\pi}_{i+1}(high) = 4,\ V^{\pi}_{i+1}(low) = 2.5,\ V^{\pi}_{i+1}(off) = 0$

3. (5 pts) We find that the optimal values are $V^*(high) = 4.42, V^*(low) = 2.84, V^*(off) = 0.21$ at convergence. Show the computations done by policy extraction to find π^*.

$V^*(off) =- 2 + \gamma V^*(high) = 0.21 \Rightarrow 0.5 * V^*(high) = 2.21 \Rightarrow V^*(high) = 4.42$

$\pi^*(off) = recharge$

$V^\pi(low,\ a = fast) = 2 + \gamma V^\pi(off) = 2 + 0.5 * 0.21 = 2.105$

$V^\pi(low,\ a = slow) = 1.75 + 0.75\gamma V^\pi(low) + 0.25\gamma V^\pi(off)]$

$= 1.75 + 0.75 * 0.5 * 2.84 + 0.25 * 0.5 * 0.21 = 2.84125$

$\pi^*(low) = slow$

$V^\pi(high, a = fast) = 3 + \gamma V^\pi(low) = 3 + 0.5 * 2.84 = 4.42$

$V^\pi(high,\ a = slow) = 2 + \gamma V_i^\pi(low) = 2 + 0.5 * 2.84 = 3.42$

$\pi^*(high) = fast$

$\pi^*(high) = fast,\ \pi^*(low) = slow,\ \pi^*(off) = recharge$

4. (3 pts) Now suppose $\gamma = 0$; find the optimal policy for this scenario. (You should be able to do so without resorting to dynamic programming.) Briefly explain how changing γ to 0 also changes any optimal actions from those you found in part 3 above with $\gamma = 0.5$.

$V^*(off) = 1.0 * (-2) =- 2$

$V^\pi(low,\ a = fast) = 1.0 * (+2) = 2$

$V^\pi(low,\ a = slow) = 0.75 * (+2) + 0.25 * (+1) = 1.75$

$V^\pi(high, a = fast\ ) = 1.0 * (+3) = 3$

$V^\pi(high,\ a = slow) = 0.5 * (+2) + 0.5 * (+2) = 2$

$optimal\, policy$: $\pi^*(high) = fast,\ \pi^*(low) = fast,\ \pi^*(off) = recharge$

Explanation:

$V^\pi(s) \ = \ \sum_{s'} T(s, a, s')[R(s, a, s') \ + \ \gamma V^\pi(s')]$

$\gamma = 0, V^\pi(s)\ simplifies\ to\ V^\pi(s) \ = \ \sum_{s'} T(s, a, s')R(s, a, s')$

optimal policy refers to the policy of s that leads to $V^\pi(s) \ = \ max_a \sum_{s'} T(s, a, s')R(s, a, s')$

The optimal policy is different from part 3 because $\gamma = 0$ does not consider any value of future states but rather only focusing on immediate rewards.

Problem 2: Reinforcement Learning (22 points)

After having been in operation for a long time, we find that the robot's performance has degraded, and the original model no longer appears to be valid. In particular, the robot has lost the ability to recharge itself in the *off* state, so we will treat it as a terminal state (we can still manually recharge it ourselves). Suppose we observe the following two episodes of state and reward sequences following the policy π of going *slow* in both states.

- Episode 1: $high, 2, high, 1, low, 1, low, 1, low, 0, off$
- Episode 2: $high, 2, high, 2, high, 1, low, 0, off$

1. (6 pts) We use first-visit Monte Carlo to perform prediction for the policy π. Again using $\gamma = 0.5$, show the calculations for finding the individual state return values G in each episode. Then compute the estimated state values $V^\pi(high)$ and $V^\pi(low)$.

$$G = \sum_{j=t+1}^{T} \gamma^{j-(t+1)} r_j$$

Episode 1:

$$V^{\pi}(high) = G(high) = 2 + 0.5 * 1 + 0.5^2 * 1 + 0.5^3 * 1 + 0.5^4 * 0 = 2.875$$

$$V^{\pi}(low) = G(low) = 1 + 0.5 * 1 + 0.5^2 * 0 = 1.5$$

Episode 2:

$$N(s_t) = 1$$

$$G(high) = 2 + 0.5 * 2 + 0.5^2 * 1 + 0.5^3 * 0 = 3.25$$

$$G(low) = 0$$

$$V^{\pi}(high) = \frac{1}{2}(V^{\pi}(high) + G(high)) = \frac{1}{2}(2.875+3.25) = 3.0625$$

$$V^{\pi}(low) = \frac{1}{2}(V^{\pi}(low) + G(low)) = \frac{1}{2}(1.5 + 0) = 0.75$$

2. (4 pts) Suppose that we apply different weights to the returns in each episode when computing the average values $V^\pi(high)$ and $V^\pi(low)$. Compute the values obtained by applying a weight $\alpha = 0.8$ to the returns in episode 2 (and correspondingly, $1 - \alpha = 0.2$ in episode 1). Briefly describe a scenario in which this weighting scheme may give more accurate value estimates.

$V^\pi(high) = 0.8 * 3.25 + 0.2 * 2.875 = 3.175$

$V^\pi(low) = 0.8 * 0 + 0.2 * 1.5 = 0.3$

For example, a robot may learn and adapt its policy over time. In the early stage of this process, the robot may not perform optimally. By exploring the state space and learning the outcome of different actions, the robot improves its policy later on. Giving a larger weigh for the returns from later episode will better reflects current policy (which is improved over time), rather than earlier suboptimal policy. Therefore the weighting scheme may produce more accurate value estimates.

3. (4 pts) We want the robot to try different actions to learn a better policy. It has the following Q-values so far: $Q(high, fast) = 2, Q(high, slow) = 0, Q(low, fast) = -1, Q(low, slow) = 1$. What is its current greedy policy? If we know that all possible future rewards are nonnegative, is it possible for the robot to learn a different policy if it always acts greedily and never explores? Explain your answer.

Current greedy policy:

$\pi^*(high) = fast, \pi^*(low) = slow.$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

It is **not** possible for the robot to learn a different policy if it always acts greedily and never explores. If all future rewards are non-negative, then r is non-negative. It has no opportunity to find out if there are other actions that could lead to higher rewards in the long term. Acting greedily can lead the robot to be stuck in a suboptimal policy if it never explores actions that might lead to higher long-term rewards.

Q-values so far: $Q(high, fast) = 2, Q(high, slow) = 0, Q(low, fast) = -1, Q(low, slow) = 1$.

4. (4 pts) We send the robot off to do some active reinforcement learning. Starting off in the *high* state, it takes the greedy action (according to its Q-values from part 3 above), receives a reward of +2, and lands in the *low* state. It then takes the exploratory action, receives a reward of +1, and stays in the *low* state. Show the resultant Q-value updates performed by the Q-learning algorithm, using $\gamma = 0.5$ and $\alpha = 0.8$.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

$greedy\ action:\ high,\ fast$

$exploratory\ action:\ low,\ fast$

$Generate\ sequence:\ high, fast, +\ 2, low, fast, +\ 1, low$

$Q(high, fast) = 2 + 0.8 * (2 + 0.5 * 1 - 2) = 2.4$

$Q(low, fast) = -1 + 0.8 * (1 + 0.5 * 1 + 1) = 1.0$

5. (4 pts) Recompute the first Q-value update using SARSA instead of Q-learning. Briefly explain how the SARSA update results in a different Q-value, making reference to how the robot “interprets” its second transition.

$Q(high, fast) = 2 + 0.8 * (2 + 0.5 * (-1) - 2) = 1.6$

SARSA updates results based on actual action a robot takes in the next state. This allows the robot to learn from less optimal exploratory moves, rather than current optimal option.

Part 3: Analysis (6 points)

1. Let the `DP_agent` run for at least 200 steps. What is the robot's 100-step average velocity? How does this change when you a) increase `gamma` to at least 0.9, and b) decrease it below 0.7 (give it time to settle after each change)? Describe how the discount factor affects the robot's performance.

100-step average velocity: 1.73-1.75

a) gamma = 0.919, 100-step average velocity: 3.32-3.34
b) gamma = 0.667, 100-step average velocity: 0.00

The discounting factor gamma influence the 100-step average velocity. Higher gamma results in higher average velocity, while lower gamma results in lower velocity, potentially dropping to 0.

A higher gamma means that the robot values future rewards more and has benefits in the long term, while a lower gamma values immediate reward more and can therefore be short-sighted.

2. Let the **`RL_agent`** train until it crosses the screen at least once so that it has learned an optimal or near-optimal policy. Describe how its performance changes when you a) increase and b) decrease **`epsilon`** by at least 0.25 in each direction.

100-step velocity: roughly 0.6-1.3

a) Increase epsilon: (epsilon = 0.8)

100-step average velocity: roughly 0-0.7

b) Decrease epsilon: (epsilon = 0.2)

100-step average velocity: roughly 1.1-1.8

This is because
With probability `epsilon`, choose a random action.
With probability `1 - epsilon` , choose the action according the Q-values.

Therefore increase epsilon increases randomness and more explorations. This can lead to a lower average velocity if the agent explores non-optimal actions. On the other hand, decreasing epsilon increases performance by taking good actions according to the Q-values.

3. Let the `RL_agent` train until it crosses the screen at least once, and note approximately how many steps it took to do so. Then start a new run and decrease **`alpha`** to about 0.1 at the very beginning. Describe the effect of the learning rate on the robot's training time.

(alpha = 0.8): Approximately 1900 steps
After decreasing alpha by 0.1 (alpha=0.667): Approximately 2800 steps

Decreasing learning rate increases training time, as shown by the number of steps taken to cross the screen at least once.