Assignment -2

SARSA and Q-Learning Algorithm

```
# ----------------------------
# SARSA implementation
# ----------------------------
def train_sarsa(env, num_episodes, alpha, gamma, epsilon=None, tau=None,
                max_steps=100, seed=None, record_every=1, wandb_run=None):
    """Train SARSA on given env. Caller should create a fresh env per call
for reproducibility."""
    if seed is not None:
        random.seed(seed)
        np.random.seed(seed)

    n_actions = env.num_actions
    Q = defaultdict(lambda: np.zeros(n_actions, dtype=np.float64))

    rewards_per_episode = []
    steps_per_episode = []
    state_visit_counts = defaultdict(int)

    for ep in range(num_episodes):
        # ensure environment start/goal reset behavior occurs
        start_seq = env.reset()
        state_key = state_to_key(start_seq, env)
        state_visit_counts[state_key] += 1

        # Choose initial action
        if epsilon is not None:
            a = epsilon_greedy_action(Q, state_key, n_actions, epsilon)
        elif tau is not None:
            a = softmax_action(Q, state_key, n_actions, tau)
        else:
            a = random.randrange(n_actions)

        total_reward = 0.0
        steps = 0
```

```
        done = False

        while not done and steps < max_steps:
            next_state, rew = env.step(state_key, a)
            # env.step in assignment returns (next_state, reward) — handle
both orders if necessary
            # If type indicates the first is reward (unlikely with
provided env), try to detect:
            if isinstance(next_state, (int, np.integer)) and
(isinstance(rew, (float, np.floating, np.ndarray))):
                # expected ordering (next_state, reward)
                pass
            # ensure reward is scalar float
            rew = float(np.array(rew).reshape(-1)[0])
            next_key = state_to_key(next_state, env)
            state_visit_counts[next_key] += 1

            # Choose next action (on-policy)
            if epsilon is not None:
                a_next = epsilon_greedy_action(Q, next_key, n_actions,
epsilon)
            elif tau is not None:
                a_next = softmax_action(Q, next_key, n_actions, tau)
            else:
                a_next = random.randrange(n_actions)

            # SARSA update
            Q[state_key][a] += alpha * (rew + gamma * Q[next_key][a_next]
- Q[state_key][a])

            state_key = next_key
            a = a_next
            total_reward += rew
            steps += 1

            # terminal check
            if is_goal_state(env, state_key) or steps >= max_steps:
                done = True

        rewards_per_episode.append(total_reward)
```

```
        steps_per_episode.append(steps)

        # optional logging to wandb
        if wandb_run is not None and ((ep + 1) % record_every == 0):
            wandb_run.log({"episode": ep + 1,
                           "reward": total_reward,
                           "steps": steps,
                           "algorithm": "sarsa"})

    return {
        'Q': Q,
        'rewards_per_episode': rewards_per_episode,
        'steps_per_episode': steps_per_episode,
        'state_visit_counts': state_visit_counts
    }


# ----------------------------
# Q-Learning implementation
# ----------------------------
def train_q_learning(env, num_episodes, alpha, gamma, epsilon=None,
tau=None,
                     max_steps=100, seed=None, record_every=1,
wandb_run=None):
    """Train Q-learning on given env. Caller should create a fresh env per
call for reproducibility."""
    if seed is not None:
        random.seed(seed)
        np.random.seed(seed)

    n_actions = env.num_actions
    Q = defaultdict(lambda: np.zeros(n_actions, dtype=np.float64))
    rewards_per_episode = []
    steps_per_episode = []
    state_visit_counts = defaultdict(int)

    for ep in range(num_episodes):
        start_seq = env.reset()
        state_key = state_to_key(start_seq, env)
        state_visit_counts[state_key] += 1
```

```
        total_reward = 0.0
        steps = 0
        done = False

        while not done and steps < max_steps:
            # Choose action (off-policy)
            if epsilon is not None:
                a = epsilon_greedy_action(Q, state_key, n_actions,
epsilon)
            elif tau is not None:
                a = softmax_action(Q, state_key, n_actions, tau)
            else:
                a = random.randrange(n_actions)

            next_state, rew = env.step(state_key, a)
            rew = float(np.array(rew).reshape(-1)[0])
            next_key = state_to_key(next_state, env)
            state_visit_counts[next_key] += 1

            # Q-learning update (off-policy)
            Q[state_key][a] += alpha * (rew + gamma * np.max(Q[next_key])
- Q[state_key][a])

            state_key = next_key
            total_reward += rew
            steps += 1

            if is_goal_state(env, state_key) or steps >= max_steps:
                done = True

        rewards_per_episode.append(total_reward)
        steps_per_episode.append(steps)

        if wandb_run is not None and ((ep + 1) % record_every == 0):
            wandb_run.log({"episode": ep + 1,
                           "reward": total_reward,
                           "steps": steps,
                           "algorithm": "qlearning"})
```

```
    return {
        'Q': Q,
        'rewards_per_episode': rewards_per_episode,
        'steps_per_episode': steps_per_episode,
        'state_visit_counts': state_visit_counts
    }
```

SARSA Vs. Q-Learning Algorithm

1. **Core Idea**

Algorithm	Type	Description
SARSA	*On-policy*	Learn the value of the policy it actually follows — including exploration.
Q-learning	*Off-policy*	Learn the value of the optimal policy while potentially following a different (exploratory) policy.

---

2. **Update Rule**

SARSA (On-policy TD control)

It updates the Q-value using the *action actually taken next* (a').

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

Where:

- **s** = current state
- **a** = current action
- **r** = reward received
- **s′** = next state
- **a′** = next action chosen according to the current policy (ε-greedy or softmax)
- **γ** = discount factor
- **α** = learning rate

SARSA uses the next action chosen by the policy, not the max one.

Q-Learning (Off-policy TD control)

It updates the Q-value using the *best possible action* in the next state *(greedy action)*:

$$\mathbf{Q(s, a) \leftarrow Q(s, a) + \alpha\,[\, r + \gamma \max_{a'} Q(s', a') - Q(s, a)\,]}$$

Where:

- **s** = current state
- **a** = current action
- **r** = reward received
- **s′** = next state
- **a′** = next possible action in state *s′*
- **γ** = discount factor
- **α** = learning rate

Q-learning assumes the next action will be optimal, even if the agent actually explores during training.

---

3. **Key Behavioral Difference**

Aspect	SARSA	Q-learning
Policy type	On-policy (learns about the policy it uses)	Off-policy (learns about the optimal policy)
Exploration awareness	Learns safely; aware of ε-greedy exploration	Learns aggressively; assumes greedy future actions

Convergence	Converges to the policy being followed	Converges to optimal policy under sufficient exploration
Performance in risky environments	More conservative	More optimistic / risk-taking

---

4. Example: Cliff Walking

Imagine a gridworld where the shortest path from start to goal runs along a cliff — stepping off the cliff gives a large negative reward (−100).

- Q-learning learns the *optimal policy* assuming perfect future actions → it takes the *risky shortest path*, because it assumes it will never fall.
- SARSA learns based on *actual actions* taken under ε-greedy exploration → since it sometimes falls off due to exploration, it learns to take a *safer but longer path*.

Result:

- Q-learning → optimal but risky path
- SARSA → safer, more realistic path given ε exploration

---

5. Summary Table

Feature	SARSA	Q-Learning
Learning type	On-policy	Off-policy
Next action in update	Actual next action a'	Greedy next action argmax Q(s', a')
Stability	More stable in stochastic or risky environments	Faster convergence but riskier behavior
Sensitivity to ε	High	Low

Common use	Safe learning or when exploration affects outcome	Pure performance or known safe environments

---

6. Intuition

- **SARSA** learns *what happens when I act this way*.
- **Q-learning** learns *what would happen if I always acted optimally*.

Experimentation and Hyperparameter tuning

Configurations:

10×10 Grid World
1. Q-learning:– transition prob = 0.7 or 1.0
– start state = (0,4) or (3,6)
– exploration strategy = ϵ-greedy or Softmax
– wind = False
Total Configuration: 8

2. SARSA:
– wind = True or False
– start state = (0,4) or (3,6)
– exploration strategy = ϵ-greedy or Softmax
– transition prob = 1.0
Total Configuration: 8

Four Room Grid World
1. Q-learning:
– goal change = True or False
– Keep other configurations as default configurations as defined in the source code.
Total Configuration: 2

2. SARSA:– goal change = True or False
– Keep other configurations as default configurations as defined in the source code.
Total Configuration: 2

Hyperparameters:
Learning Rate (α): $\alpha \in \{0.001, 0.01, 0.1, 1.0\}$
Discount Factor (γ): $\gamma \in \{0.7, 0.8, 0.9, 1.0\}$
Exploration Strategy: Use one of the following:
• ϵ-greedy: $\epsilon \in \{0.001, 0.01, 0.05, 0.1\}$
• Softmax: $\tau \in \{0.01, 0.1, 1, 2\}$

1. Training Curves:
• Plot the average reward per episode over time.
• Plot the average number of steps to reach the goal per episode.
2. State Visit Heatmap:
• Generate a heatmap of the grid showing the average number of visits to each state, aggregated over all runs during training.
3. Q-Value Heatmap and Optimal Policy: • After training, visualize a heatmap showing the average maximum Q-value for each state across runs.
• Overlay the optimal action (policy) for each state, derived from the learned Q values.

Please find the report for the run below:

Run_Report_20_configurations