Untitled 1

```
import gymnasium
from gymnasium import spaces
import numpy as np
import pandas as pd
import heapq
import pygame
import matplotlib.pyplot as plt
from collections import defaultdict
import sys

class RadarTaskSchedulerEnv(gymnasium.Env):
    metadata = {"render_modes": ["human"], "render_fps": 1}

    def __init__(self, task_file="dataset2.csv", render_mode=None):
        super(RadarTaskSchedulerEnv, self).__init__()

        self.tasks_df = pd.read_csv(task_file)
        self.num_tasks = len(self.tasks_df)

        self.priority_queue = []
        self.entry_count = 0

        self.current_task_data = None

        self.scheduled_tasks_log = []

        self.observation_space = spaces.Box(low=0, high=100, shape=(3,), dtype=np.int64)

        self.action_space = spaces.MultiDiscrete([3, 3, 3])

        self.render_mode = render_mode
        self.screen = None
        self.font = None
        self.clock = None
        self.width = 800
        self.height = 600

        self._last_info = {}
        self._last_reward = 0

    def reset(self,*, seed=None, options=None):
        super().reset(seed=seed)

        self.priority_queue = []
        self.entry_count = 0

        for index, row in self.tasks_df.iterrows():
            negative_priority = -row['Priority']
            heapq.heappush(self.priority_queue, (negative_priority, self.entry_count, row.to_dict()))
            self.entry_count += 1

        self.scheduled_tasks_log = []
        self._last_info = {}
        self._last_reward = 0

        obs = self._get_observation()

        if self.render_mode == "human":
            self._init_render()
            self.render()

        return obs, {}

    def step(self, action):
        if self.current_task_data is None:
            return np.array([0, 0, 0], dtype=np.int64), 0, True, False, {"task_dropped": False, "task_id": "N/A"}

        _priority, _entry_count, task_to_process = heapq.heappop(self.priority_queue)

        delay, compress, radar_choice = action
```

```
        tns = task_to_process['Request_Time'] + delay
        Pn = task_to_process['Init_Power'] + compress

        task_dropped = (tns > task_to_process['Deadline']) or \
                       (Pn > task_to_process['Max_Power'])

        reward = 1 if not task_dropped else -1

        info = {
            "task_id": task_to_process["Task_ID"],
            "task_dropped": task_dropped,
            "delay_action": delay,
            "compress_action": compress,
            "radar_choice_action": radar_choice,
            "calculated_tns": tns,
            "calculated_Pn": Pn,
            "original_deadline": task_to_process['Deadline'],
            "original_max_power": task_to_process['Max_Power']
        }
        self._last_info = info
        self._last_reward = reward

        self.scheduled_tasks_log.append({
            'Task_ID': task_to_process['Task_ID'],
            'Priority': task_to_process['Priority'],
            'Action_Delay': delay,
            'Action_Compress': compress,
            'Action_Radar': radar_choice,
            'Dropped': task_dropped,
            'Final_TNS': tns,
            'Final_Pn': Pn,
            'Original_Deadline': task_to_process['Deadline'],
            'Original_Max_Power': task_to_process['Max_Power']
        })

        done = not self.priority_queue

        next_obs = self._get_observation()

        if self.render_mode == "human":
            self.render()

        return next_obs, reward, done, False, info

    def _get_observation(self):
        if self.priority_queue:
            _priority, _entry_count, task_dict = self.priority_queue[0]
            self.current_task_data = task_dict
            return np.array([int(task_dict['Duration']), int(task_dict['Deadline']), int(task_dict['Init_Power'])],
dtype=np.int64)
        else:
            self.current_task_data = None
            return np.array([0, 0, 0], dtype=np.int64)

    def _init_render(self):
        if self.screen is None:
            pygame.init()
            pygame.display.set_caption("Radar Task Scheduler")
            self.screen = pygame.display.set_mode((self.width, self.height))
            self.font = pygame.font.Font(None, 30)
            self.clock = pygame.time.Clock()

    def render(self):
        if self.render_mode != "human":
            return

        self._init_render()

        self.screen.fill((0, 0, 0))

        y_offset = 20
        self._draw_text(f"Tasks Remaining: {len(self.priority_queue)}", (255, 255, 255), y_offset)
        y_offset += 40

        if self._last_info:
```

```
            self._draw_text("--- Last Task Processed ---", (255, 255, 0), y_offset)
            y_offset += 30
            task_id = self._last_info.get('task_id')
            priority_val = 'N/A'
            if task_id and 'Task_ID' in self.tasks_df.columns and not self.tasks_df[self.tasks_df['Task_ID'] ==
task_id].empty:
                priority_val = self.tasks_df[self.tasks_df['Task_ID'] == task_id]['Priority'].iloc[0]

            self._draw_text(f"Task ID: {task_id} (Prio: {priority_val})", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Action: Delay {self._last_info.get('delay_action')}, Compress
{self._last_info.get('compress_action')}, Radar {self._last_info.get('radar_choice_action')}", (150, 250, 150), y_offset)
            y_offset += 25
            self._draw_text(f"TNS: {self._last_info.get('calculated_tns')} (Deadline:
{self._last_info.get('original_deadline')})", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Pn: {self._last_info.get('calculated_Pn')} (Max: {self._last_info.get('original_max_power')})",
(200, 200, 200), y_offset)
            y_offset += 25

            task_dropped_color = (255, 0, 0) if self._last_info.get('task_dropped', False) else (0, 255, 0)
            self._draw_text(f"Dropped: {self._last_info.get('task_dropped')}", task_dropped_color, y_offset)
            y_offset += 25

            reward_color = (0, 255, 0) if self._last_reward == 1 else (255, 0, 0)
            self._draw_text(f"Reward: {self._last_reward}", reward_color, y_offset)
            y_offset += 40

        if self.current_task_data:
            self._draw_text("--- Next Task to Process (Highest Priority) ---", (0, 255, 255), y_offset)
            y_offset += 30
            self._draw_text(f"Task ID: {self.current_task_data['Task_ID']}", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Priority: {self.current_task_data['Priority']}", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Duration: {self.current_task_data['Duration']}", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Request Time: {self.current_task_data['Request_Time']}", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Deadline: {self.current_task_data['Deadline']}", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Init Power: {self.current_task_data['Init_Power']}", (200, 200, 200), y_offset)
            y_offset += 25
            self._draw_text(f"Max Power: {self.current_task_data['Max_Power']}", (200, 200, 200), y_offset)
            y_offset += 25
        else:
            self._draw_text("--- All tasks processed ---", (0, 255, 0), y_offset)

        pygame.display.flip()
        self.clock.tick(self.metadata["render_fps"])

    def _draw_text(self, text, color, y):
        text_surface = self.font.render(text, True, color)
        text_rect = text_surface.get_rect(center=(self.width // 2, y))
        self.screen.blit(text_surface, text_rect)

    def close(self):
        if self.screen is not None:
            pygame.quit()
            self.screen = None
            self.font = None
            self.clock = None

class QLearningAgent:
    def __init__(self, state_size, action_size, alpha, gamma, epsilon, epsilon_decay, final_epsilon):
        self.q_table = defaultdict(lambda: np.zeros(action_size))
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.initial_epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon
        self.action_size = action_size
        self.state_size = state_size

    def choose_action(self, state):
```

```
        state_key = str(state)
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.action_size)
        return np.argmax(self.q_table[state_key])

    def update_q_value(self, state, action, reward, next_state):
        state_key = str(state)
        next_state_key = str(next_state)

        best_next_action = np.argmax(self.q_table[next_state_key])

        td_target = reward + self.gamma * self.q_table[next_state_key][best_next_action]

        td_error = td_target - self.q_table[state_key][action]

        self.q_table[state_key][action] += self.alpha * td_error

    def decay_epsilon(self):
        self.epsilon = max(self.epsilon * self.epsilon_decay, self.final_epsilon)

if __name__ == "__main__":
    enable_rendering = input("Enable Pygame rendering during training? (Y/N): ").strip().lower()
    if enable_rendering == 'y':
        render_mode_choice = "human"
        print("Pygame rendering ENABLED. Close the Pygame window to stop training prematurely.")
    else:
        render_mode_choice = None
        print("Pygame rendering DISABLED. Training will run silently.")

    env = RadarTaskSchedulerEnv(task_file="dataset2.csv", render_mode=render_mode_choice)

    agent = QLearningAgent(
        state_size=3,
        action_size=27,
        alpha=0.15,
        gamma=0.95,
        epsilon=0.9,
        epsilon_decay=0.99,
        final_epsilon=0.05
    )

    def decode_action(index):
        delay = index // 9
        compress = (index % 9) // 3
        radar = index % 3
        return [delay, compress, radar]

    episodes = 1000
    reward_log = []
    task_drop_log = []

    print("Starting Q-Learning training...")

    for episode in range(episodes):
        state, info = env.reset()
        done = False
        total_reward = 0
        task_drops = 0
        episode_step_count = 0

        while not done:
            if env.render_mode == "human":
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        done = True
                        break
                if done:
                    break

            action_index = agent.choose_action(state)
            action = decode_action(action_index)

            next_state, reward, done, _, info = env.step(action)

            agent.update_q_value(state, action_index, reward, next_state)
```

```
            state = next_state
            total_reward += reward
            episode_step_count += 1

            if info.get('task_dropped', False):
                task_drops += 1

        reward_log.append(total_reward)
        task_drop_log.append(task_drops)

        agent.decay_epsilon()

        if (episode + 1) % 50 == 0 or episode == episodes - 1:
            print(f"Episode {episode+1}/{episodes} - Tasks Processed: {episode_step_count} - Total Reward: {total_reward} -
Tasks Dropped: {task_drops} - Epsilon: {agent.epsilon:.3f}")

        if done and episode_step_count < env.num_tasks and env.render_mode == "human":
            print("\nPygame window closed. Stopping training prematurely.")
            break

    print("\nTraining complete!")

    window_size = 50

    if not reward_log:
        print("\nReward log is empty. No reward plot will be generated.")
    else:
        moving_avg = []
        x_moving_avg = []
        if len(reward_log) >= window_size:
            moving_avg = np.convolve(reward_log, np.ones(window_size)/window_size, mode='valid')
            x_moving_avg = range(window_size - 1, len(reward_log))

        plt.figure(figsize=(14,7))
        plt.plot(reward_log, label='Episode Rewards', color='orange', alpha=0.7, linewidth=1)

        if moving_avg.size > 0:
            plt.plot(x_moving_avg, moving_avg, label=f'Moving Average ({window_size} episodes)', color='blue', linewidth=2)

        plt.xlabel('Episode', fontsize=12)
        plt.ylabel('Total Reward', fontsize=12)
        plt.title('Total Reward over Episodes (Q-Learning)', fontsize=14)
        plt.legend(fontsize=10)
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()

    if not task_drop_log:
        print("\nTask drop log is empty. No task drop plot will be generated.")
    else:
        moving_avg_drops = []
        x_moving_avg_drops = []
        if len(task_drop_log) >= window_size:
            moving_avg_drops = np.convolve(task_drop_log, np.ones(window_size)/window_size, mode='valid')
            x_moving_avg_drops = range(window_size - 1, len(task_drop_log))

        plt.figure(figsize=(14,7))
        plt.plot(task_drop_log, label='Tasks Dropped per Episode', color='red', alpha=0.8, linewidth=1)

        if moving_avg_drops.size > 0:
            plt.plot(x_moving_avg_drops, moving_avg_drops, label=f'Moving Average ({window_size} episodes)', color='darkred',
linewidth=2)

        plt.xlabel('Episode', fontsize=12)
        plt.ylabel('Tasks Dropped', fontsize=12)
        plt.title('Number of Tasks Dropped per Episode', fontsize=14)
        plt.legend(fontsize=10)
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()

    print("\n--- Final Task Schedule (from the last completed training episode) ---")
    if env.scheduled_tasks_log and len(env.scheduled_tasks_log) == env.num_tasks:
        schedule_df = pd.DataFrame(env.scheduled_tasks_log)
        print(schedule_df.to_string(index=False))
    else:
```