Untitled 4

```
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

from stable_baselines3 import DQN
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.callbacks import BaseCallback

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.float32)

        self.action_space = spaces.Discrete(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_index):
        if self.current_task_data is None:
            return np.array([0, 0, 0], dtype=np.float32), 0, True, False, {"task_dropped": False, "task_id": "N/A"}
```

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

delay_action_idx, compress_action_idx, radar_choice = self._decode_action(action_index)

# Map action indices to actual values
# These are the same as before, assuming 0,1,2 for delay/compress
delay = delay_action_idx
compress = compress_action_idx

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 SHAPING IMPROVEMENT ---
reward = 0.0 # Initialize reward as float

# Base reward for success/failure
if not task_dropped:
    reward += 1.0 # Successfully scheduled
else:
    reward -= 1.0 # Task dropped

# 1. Penalize nearing deadlines
if not task_dropped: # Only apply if not already dropped by deadline
    time_to_deadline = task_to_process['Deadline'] - tns
    if time_to_deadline < 2: # Within 2 units of deadline
        reward -= 0.1 # Small penalty for cutting it close
    elif time_to_deadline < 0: # This case shouldn't happen if not task_dropped, but as safeguard
        reward -= 0.2
# If task is dropped due to deadline, it already gets -1.0, maybe a slight additional penalty for large overshoot
elif tns > task_to_process['Deadline'] + 5: # If deadline is grossly exceeded
     reward -= 0.2 # Additional penalty for very bad time management

# 2. Penalize nearing max power
if not task_dropped: # Only apply if not already dropped by power
    power_margin = task_to_process['Max_Power'] - Pn
    if power_margin < 2: # Within 2 units of max power
        reward -= 0.1 # Small penalty for using almost max power
    elif power_margin < 0: # Should not happen if not task_dropped
        reward -= 0.2
# If task is dropped due to power, similar large overshoot penalty
elif Pn > task_to_process['Max_Power'] + 5:
    reward -= 0.2

# 3. Reward for using minimal delay if feasible
# If delay_action was 0 and task was successfully scheduled (not dropped)
if delay == 0 and not task_dropped:
    reward += 0.05 # Small bonus for optimal delay

# 4. Reward for using minimal compression if feasible
# If compress_action was 0 and task was successfully scheduled (not dropped by power)
if compress == 0 and not task_dropped:
    reward += 0.05 # Small bonus for optimal power usage (minimal compression)

# 5. Optional: Slightly higher reward for high priority tasks
# This is very subtle and needs careful tuning to not upset overall goal
# priority_factor = task_to_process['Priority'] / self.tasks_df['Priority'].max()
# if not task_dropped:
#     reward += (priority_factor * 0.01) # Small bonus
# else:
#     reward -= (priority_factor * 0.02) # Slightly larger penalty

# --- END REWARD SHAPING IMPROVEMENT ---

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.float32)
        else:
            self.current_task_data = None
            return np.array([0, 0, 0], dtype=np.float32)

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

    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 > 0 else ((255,255,0) if self._last_reward == 0 else (255, 0, 0))
            self._draw_text(f"Reward: {self._last_reward:.2f}", reward_color, y_offset) # Display reward with 2 decimal places
            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 CustomCallback(BaseCallback):
    def __init__(self, verbose: int = 0, env=None):
        super().__init__(verbose)
        self.env = env
        self.episode_rewards = []
        self.episode_task_drops = []
        self._current_episode_reward = 0
        self._current_episode_task_drops = 0
        self._last_episode_start_step = 0

    def _on_training_start(self) -> None:
        print("\n--- Starting DQN Training ---")
        self._current_episode_reward = 0
        self._current_episode_task_drops = 0
        self._last_episode_start_step = self.num_timesteps

    def _on_step(self) -> bool:
        if self.env.render_mode == "human":
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    print("\nPygame window closed. Stopping training prematurely.")
                    return False

        if self.locals['infos'] and len(self.locals['infos']) > 0:
            info = self.locals['infos'][0]
            self._current_episode_reward += self.locals['rewards'][0]
```

```
            if info.get('task_dropped', False):
                self._current_episode_task_drops += 1

        if self.locals['dones'][0]:
            self.episode_rewards.append(self._current_episode_reward)
            self.episode_task_drops.append(self._current_episode_task_drops)

            episode_num = len(self.episode_rewards)
            print(f"Episode {episode_num} completed - Timesteps: {self.num_timesteps - self._last_episode_start_step} - Total
Reward: {self._current_episode_reward:.2f} - Tasks Dropped: {self._current_episode_task_drops}")

            self._current_episode_reward = 0
            self._current_episode_task_drops = 0
            self._last_episode_start_step = self.num_timesteps

        return True

    def _on_training_end(self) -> None:
        print("\n--- DQN Training Complete ---")

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)

    try:
        check_env(env, warn=True)
        print("Environment check passed!")
    except Exception as e:
        print(f"Environment check failed: {e}")
        sys.exit(1)

    model = DQN("MlpPolicy", env,
                learning_rate=1e-3,
                buffer_size=10000,
                learning_starts=500,
                batch_size=32,
                tau=1.0,
                gamma=0.99,
                train_freq=(1, "step"),
                target_update_interval=100,
                exploration_fraction=0.5,
                exploration_initial_eps=1.0,
                exploration_final_eps=0.05,
                verbose=0,
                device="auto"
                )

    callback = CustomCallback(env=env)

    total_timesteps = 100000
    print(f"Starting DQN training for {total_timesteps} timesteps...")

    try:
        model.learn(total_timesteps=total_timesteps, callback=callback, log_interval=10)
    except KeyboardInterrupt:
        print("\nTraining interrupted by user.")
    except Exception as e:
        print(f"\nAn error occurred during training: {e}")
    finally:
        env.close()
        print("Environment closed.")

    reward_log = callback.episode_rewards
    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 (DQN)', fontsize=14)
        plt.legend(fontsize=10)
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()

    task_drop_log = callback.episode_task_drops
    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 (DQN)', 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:
        print("The last episode might have been interrupted or did not complete all tasks.")
        print("To see a full schedule, ensure the training completes at least one full episode.")
```