# Untitled 2

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

- These imports are for **Stable Baselines 3**, a popular library to train RL agents.
- `DQN` is the Deep Q-Network algorithm.
- `check_env` validates if your environment is properly set up.
- `BaseCallback` allows you to customize actions during training (like logging or early stopping).

---

## Defining the Environment Class

```
class RadarTaskSchedulerEnv(gymnasium.Env):
```

- Defines a new environment named `RadarTaskSchedulerEnv` inheriting from Gymnasium's base Env class.

```
metadata = {"render_modes": ["human"], "render_fps": 1}
```

- Metadata tells Gym this environment supports "human" rendering mode and should update at 1 frame per second.

---

## Initialization ( `__init__` method)

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

- Constructor runs when you create an environment instance.
- `task_file` is the CSV file with task data.
- `render_mode` decides if it should show graphics.
- `super()` calls the parent class initializer.

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

- Loads the CSV file into a pandas DataFrame.
- Stores the total number of tasks.

```
self.priority_queue = []
self.entry_count = 0
```

- Initializes an empty priority queue for tasks.
- `entry_count` is a counter to keep track of task order (helps with tie-breaking).

```
self.current_task_data = None
self.scheduled_tasks_log = []
```

- `current_task_data` will hold data for the task currently processed.
- `scheduled_tasks_log` will store info about all tasks after scheduling decisions.

```
self.observation_space = spaces.Box(low=0, high=100, shape=(3,), dtype=np.float32)
```

- Defines what the agent observes: a vector of 3 floats (e.g., duration, deadline, initial power), each between 0 and 100.

```
self.action_space = spaces.Discrete(3 * 3 * 3)
```

- Defines possible actions as one of 27 discrete choices. (3 options each for delay, compression, and radar choice; total 3×3×3=27.)

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

- Sets rendering options and window size (used only if rendering enabled).

```
self._last_info = {}
self._last_reward = 0
```

- Keeps info about the last task processed and reward given.

---

## Reset Method

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

- Resets the environment state for a new episode.
- Calls parent class reset (seed for randomness).

```
self.priority_queue = []
self.entry_count = 0
```

- Empties the priority queue and resets counter.

```
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
```

- Loads all tasks into the priority queue.
- Uses negative priority because `heapq` is a min-heap but we want highest priority first.
- `entry_count` breaks ties to maintain insertion order.

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

- Resets logs and last info.

```
obs = self._get_observation()
```

- Gets the observation of the current highest priority task.

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

- If rendering is enabled, initialize the window and draw the initial screen.

```
return obs, {}
```

- Returns initial observation and empty info dictionary.

---

## Step Method (one action execution)

```
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"}
```

- If no current task, return zero observation, zero reward, done=True, and info showing no task.

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

- Pop highest priority task from the queue.

```
delay, compress, radar_choice = self._decode_action(action_index)
```

- Decode the action index (0-26) into 3 separate decisions: delay, compression, radar choice.

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

- Calculate new task start time and power after actions.

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

- Task is dropped if start time exceeds deadline or power exceeds max allowed.

```
reward = 1 if not task_dropped else -1
```

- Reward agent for completing (or punishing for dropping) the task.

```
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
```

- Store all useful info about the step to show in rendering or for logging.

```
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']
})
```

- Log all task info and actions after scheduling.

```
done = not self.priority_queue
```

- Episode ends when no more tasks left.

```
next_obs = self._get_observation()
```

- Get next task's observation.

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

- Update graphics.

```
return next_obs, reward, done, False, info
```

- Return new state, reward, done flag, no truncation, and info dictionary.

---

## Helper functions

```
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)
```

- Returns an observation vector of the next task's duration, deadline, and initial power.
- If no tasks left, returns zeros.

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

- Converts a single action number (0 to 26) to a triple of discrete actions.

---

## Rendering methods (visual display)

```
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()
```

- Initializes pygame display window and sets up fonts and timing.

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

- Does nothing if rendering is off.

```
self._init_render()
self.screen.fill((0, 0, 0))  # Clear screen with black background
```

- Initializes display and clears the screen.

```
# Then draws info texts for remaining tasks, last processed task, and next task with colors and formatting
```

- The method draws text info about the environment state and last action, such as:
  - Number of tasks remaining
  - Last task processed info (ID, priority, actions taken, whether dropped, reward)
  - Next highest priority task info

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

- Updates the display to show the drawings and caps FPS to 1 frame per second.

```
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)
```

- Helper function to draw centered text in the window.

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

- Closes pygame window and clears resources.

---

## Custom Callback Class

```
class CustomCallback(BaseCallback):
```

- A callback to hook into training to monitor and print episode progress.

```
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
```

- Stores environment and tracks rewards and dropped tasks per episode.

```
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
```

- Called when training starts.

```
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
```

- Checks if the pygame window is closed to stop training early.

```
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
```

- Adds up rewards and task drops for this episode.

```
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} - 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
```

- When an episode finishes, print stats and reset counters.

```
return True
```

- Continue training unless told otherwise.

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

- Prints a message when training finishes.

---

## Main Program

```
if __name__ == "__main__":
```

- Runs only if the script is run directly, not if imported.

```
enable_rendering = input("Enable Pygame rendering during training? (Y/N): ").strip().lower()
```

- Asks user if they want visual rendering during training.

```
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.")
```

- Sets rendering mode accordingly.

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

- Creates the environment instance.

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

- Validates environment compatibility with Stable Baselines.

---

## Setting up and training DQN agent

```
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"
            )
```

- Creates the Deep Q-Network agent with specified hyperparameters:
  - `MlpPolicy` : neural network with fully connected layers
  - `learning_rate` : step size for training
  - `buffer_size` : replay memory size
  - `batch_size` : samples per training update
  - `gamma` : discount factor
  - `exploration_*` : control epsilon-greedy exploration schedule
  - `device` : use GPU if available

```
callback = CustomCallback(env=env)
```

- Creates the callback instance to monitor training.

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

- Defines total training steps.

```
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.")
```

- Starts training and handles user interrupt or errors.
- Ensures environment is closed properly after training.

---

## Plotting results

- Plots episode rewards and tasks dropped over time with moving averages using matplotlib for visualization.

---

## Final task schedule print

```
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.")
```

- Prints the full final schedule of tasks with actions taken if the last episode completed successfully.