Research Project

Author
M. Rafael MEUNIER

March 8, 2024

Contents

Chapter 1

Custom Environment Implementation

In this section, we describe the implementation details of the custom environment designed to simulate the growth of hydroponic tomatoes. The environment is crucial for training and evaluating reinforcement learning agents tasked with controlling temperature and light to maximize the Leaf Area Index (LAI) of the plants.

1.1 Horticulture Environment Class

The custom environment is encapsulated within the `HorticultureEnvironment` class. This class initializes the state space, action space, and various parameters necessary for simulating tomato growth.

1.1.1 State Space

The state space consists of a four-dimensional array representing the following:

- **Temperature**: The current temperature in Celsius.
- **Light Intensity**: The current intensity of light in Photosynthetically Active Radiation (PAR) units.
- **Leaf Area Index (LAI)**: An indicator of the leaf density or canopy cover of the tomato plants.
- **Day of Simulation**: An integer representing the current day of the simulation.

The state space can be represented as follows:

$$\text{state} = [\text{temperature}, \text{light intensity}, \text{LAI}, \text{day}]$$

1.1.2 Action Space

The action space consists of nine discrete actions, allowing the agent to adjust temperature and light settings. It is structured as follows:

- **Temperature Actions**: The agent can choose one of three actions to adjust the temperature: decrease by a certain value, maintain the current temperature, or increase by a certain value.
- **Light Actions**: Similarly, the agent can choose one of three actions to adjust the light intensity: decrease by a certain value, maintain the current light intensity, or increase by a certain value.

The combination of these temperature and light actions results in a total of nine possible actions, calculated as $3 \times 3 = 9$.

1.2 Plant Growth Simulation (hort_syst)

The `hort_syst` function simulates plant growth over a 24-hour period in the hydroponic environment. Below is the pseudocode outlining the algorithm used:

Algorithm 1 Plant Growth Simulation (hort_syst)

```
procedure HORT_SYST(state, action)
    Input:
    state: Current environment state (temp, light, LAI, day).
    action: Chosen action by the agent (adjustments to temp and light).
    Procedure:
    update_environment(action)
    for each hour in a 24-hour period do
        update_plant_growth()
    end for
    Return updated state
end procedure
```

1.3 Initialization and Resetting

Upon initialization, the environment randomly samples initial states within predefined ranges for temperature and light. The LAI is initialized to a small value, representing the initial state of the plant. Additionally, milestones for the reward function are set to `True`.

The `reset` method is responsible for resetting the environment to its initial state at the beginning of each episode. It resets the state, hour, dry matter production (DMP), plant temperature index (PTI), and other relevant variables.

1.4 Environment Step Function

The `step` function advances the environment by one time step, updates the state, calculates the reward, and determines if the episode is complete. Below is the pseudocode outlining the algorithm used:

Algorithm 2 Step Function

```
function STEP(action_index)
    action ← get_action_from_index(action_index)
    hort_syst(state, action)                    ▷ Update the environment for each day
    reward ← calculate_reward(state)            ▷ Calculate reward based on changes
    day ← day + 1                               ▷ Increment the day
    hour ← 0                                    ▷ Reset hour to 0 for the next day
    if day ≥ num_days then                      ▷ Check if simulation has ended
        done ← True
    end if
    return state, reward, done, {}
end function
```

1.5 Reward Calculation

In reinforcement learning, defining an appropriate reward function is crucial for guiding the learning process. In our hydroponic tomato cultivation environment, we employ a sparse reward scheme aimed at encouraging the agent to maximize the Leaf Area Index (LAI) of the plants while minimizing deviations from a target LAI.

The reward function is defined as follows:

$$\text{reward} = \begin{cases} 1, & \text{if } |\text{current\_LAI} - \text{target\_LAI}| \leq \text{tolerance} \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

where:

- current_LAI is the current Leaf Area Index of the plants.
- target_LAI is the desired target Leaf Area Index.
- tolerance is the permissible deviation from the target LAI.

This reward scheme provides a sparse reward signal to the agent. The agent receives a reward of 1 only when the current LAI is within the specified tolerance range of the target

LAI, indicating successful optimization of the plant growth. Otherwise, the agent receives a reward of 0, discouraging actions that lead to significant deviations from the target LAI.

The sparse reward system encourages the agent to explore and learn actions that lead to desired outcomes while avoiding unnecessary changes that may negatively impact plant growth. By optimizing the LAI while adhering to the defined tolerance, the agent learns to effectively control temperature and light settings to promote healthy plant growth in the hydroponic environment.

1.6 Action Interpretation

Actions chosen by the agent are interpreted within the `get_action_from_index`, `temperature_action`, and `light_action` methods. These methods map action indices to corresponding changes in temperature and light intensity.

1.7 Auxiliary Functions

Several auxiliary functions are implemented to calculate thermal time, PAR (Photosynthetically Active Radiation), ETc (Crop Evapotranspiration), DMP (Dry Matter Production), and nutrient uptake.

Overall, the custom environment provides a realistic simulation platform for training reinforcement learning agents to optimize hydroponic tomato cultivation by controlling environmental variables.

Chapter 2

Agent implementation with Q-learning

2.1 Initialization

The `QLearningAgent` class is initialized with the following parameters:

- **Number of Actions** (`num_actions`): The total number of possible actions that the agent can take in the environment.
- **Number of States** (`num_states`): The total number of possible states in the environment.
- **Exploration Method** (`exploration_method`): The method used for exploration, which can be one of the following: epsilon-greedy, Boltzmann, or Upper Confidence Bound (UCB).
- **Temperature** (`temperature`): A parameter used in Boltzmann exploration to control the level of exploration.

The Q-learning agent initializes its Q-table (`q_table`) with zeros, where the rows correspond to states and columns correspond to actions. Additionally, it sets the following Q-learning parameters:

- **Learning Rate** (`alpha`): Denoted as α, it controls the weight given to new information when updating the Q-table.
- **Discount Factor** (`gamma`): Denoted as γ, it determines the importance of future rewards when updating the Q-table.
- **Exploration Rate** (`epsilon`): Denoted as ϵ, it controls the balance between exploration and exploitation in epsilon-greedy policy.

Furthermore, the agent defines bins for discretizing continuous variables such as temperature, light intensity, and LAI. These bins are used to map continuous states to indices in the Q-table.

2.2 Selecting Actions

The Q-learning agent provides methods to select actions based on different exploration strategies. Three exploration methods are supported:

1. **Epsilon-Greedy**: In this method, the agent chooses a random action with probability ϵ (exploration rate) and selects the action with the highest Q-value with probability $1 - \epsilon$ (exploitation).
2. **Boltzmann Exploration**: This method selects actions probabilistically based on the Boltzmann distribution, where the probability of selecting an action is proportional to its Q-value exponentiated by a temperature parameter.
3. **Upper Confidence Bound (UCB)**: UCB is a strategy that balances exploration and exploitation by selecting actions that have a high estimated value but also have not been explored much.

Each method has a corresponding function (`select_action_epsilon_greedy`, `select_action_bolt` and `select_action_ucb`) to choose actions based on the defined exploration strategy.

The selection of actions is crucial for the agent's learning process, as it determines how the agent explores and exploits the environment to maximize rewards.

2.3 Updating Q-Table

The Q-table is updated using the Q-learning update rule, which incorporates the observed reward and estimates of future rewards. The update equation for a state-action pair (s, a) is given by:

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a')\right)$$

Where:

- $Q(s, a)$ is the Q-value of taking action a in state s.
- α is the learning rate, controlling the weight given to new information ($0 \leq \alpha \leq 1$).
- r is the observed reward for taking action a in state s.
- γ is the discount factor, determining the importance of future rewards ($0 \leq \gamma \leq 1$).
- s' is the next state after taking action a.

The Q-table is updated iteratively as the agent interacts with the environment. This process allows the agent to learn the optimal action-value function, which maximizes cumulative rewards over time.

2.4 State to Index Mapping

To facilitate Q-table updates, the continuous state space is discretized and mapped to indices in the Q-table. This mapping allows the agent to associate Q-values with specific states and actions.

The `state_to_index` function converts a state tuple (temperature, light intensity, LAI, day) into an index in the Q-table. The process involves discretizing each continuous variable (temperature, light intensity, LAI) into bins and calculating a unique index based on these discretized values.

For example, if there are N bins for each continuous variable, the total number of possible states is $N^3 \times \text{num\_days}$. Each combination of discretized values corresponds to a unique index in the Q-table, allowing the agent to update Q-values efficiently during learning.

2.5 Discretization Function

The discretization function (`discretize`) is responsible for converting continuous values into bin indices. It takes a continuous value and a list of bin edges as input and returns the index of the bin to which the value belongs.

The function iterates over the bin edges and checks if the value falls within the range of each bin. Once the appropriate bin is found, the corresponding index is returned.

If the value falls outside the defined bins, the function returns the index of the last bin. This ensures that all values are mapped to a valid bin index within the specified range.

The discretization process enables the Q-learning agent to handle continuous state variables by representing them in a discrete form suitable for Q-table indexing and updates.

Chapter 3

Agent using a Deep-Q-Network

3.1 Initialization

The DQN agent is initialized with the following parameters:

- **State Shape**: The shape of the state space, representing the input dimensions to the neural network model.
- **Action Shape**: The shape of the action space, indicating the number of possible actions.
- **Learning Rate**: The rate at which the neural network model adjusts its weights during training.
- **Discount Factor**: The discount factor used to discount future rewards in the Q-learning update equation.
- **Main Model and Target Model**: Two neural network models are initialized - the main model and the target model. The target model is initially set to have the same weights as the main model.
- **Replay Memory**: A replay memory buffer is initialized to store experiences for training the agent.

3.2 Action Selection

The DQN agent selects actions using an epsilon-greedy policy.

- With probability ϵ, the agent selects a random action to explore the environment.
- With probability $1 - \epsilon$, the agent exploits its current knowledge by selecting the action with the highest Q-value predicted by the neural network model.

3.3 Training Process

The agent is trained using experiences stored in the replay memory buffer.

- If the replay memory contains enough experiences, a mini-batch is sampled from the memory.
- The main steps of the training process include:
  - Calculating target Q-values based on the Bellman equation.
  - Updating the Q-values of the current state-action pair in the mini-batch.
  - Training the neural network model using the current states and target Q-values.

3.4 Memory Management

Experiences (state, action, reward, new state, done) are stored in the replay memory buffer.

- The memory buffer ensures that experiences are randomly sampled during training to break the correlation between consecutive experiences.
- Once the replay memory reaches a certain size, the agent begins training using experiences from the buffer.

3.5 Training Loop

Algorithm 3 Training Loop

```
Initialize agent and environment
Initialize parameters
for each episode do
    Reset environment
    Set done to False
    Set total_reward to 0
    while not done do
        Select action
        Take step in environment
        Remember experience
        Train agent
    end while
    Update target network
end for
```