Reinforcement Learning (CS60077)
Term-Project (Phase-1)
Project Code: MTV1

Yashaswini
20EE38020

Task 1 - Defining the environment

Defined the `GridWorldEnv` class for a grid world scenario where a runner tries to survive against taggers in a grid. The environment manages the state of the runner and taggers, handles actions, and calculates rewards.

1. **`__init__(self, n=15, k=2)`**:
    - **Purpose**: Initializes the grid world environment.
    - **Parameters**:
        - n: The size of the grid (default is 15x15).
        - k: The number of taggers (default is 2).
    - **Attributes**:
        - `self.n`: Stores the grid size.
        - `self.k`: Stores the number of taggers.
        - `self.action_space`: Number of possible actions for the runner (8 directions).
        - `self.action_labels`: Labels for each of the 8 possible movement directions.
        - `self.runner_pos`: Stores the current position of the runner.
        - `self.tagger_positions`: Stores the positions of the taggers.
        - `self.steps`: Tracks the number of steps taken in an episode.
        - `self.max_steps`: Maximum steps allowed per episode (200).
        - `self._last_reward`: Stores the reward from the last step.
2. **`reset(self)`**:
    - **Purpose**: Resets the environment to the initial state at the start of each episode.
    - **Returns**: The current state (runner and tagger positions).
    - **Functionality**:
        - Randomly initializes the runner's position within the grid.
        - Randomly initializes positions for all the taggers within the grid.
        - Resets the step counter to zero.
3. **`step(self, action)`**:
    - **Purpose**: Executes a single step in the environment based on the runner's action.
    - **Parameters**:
        - `action`: An integer representing the direction in which the runner will move (0 to 7 for 8 directions).

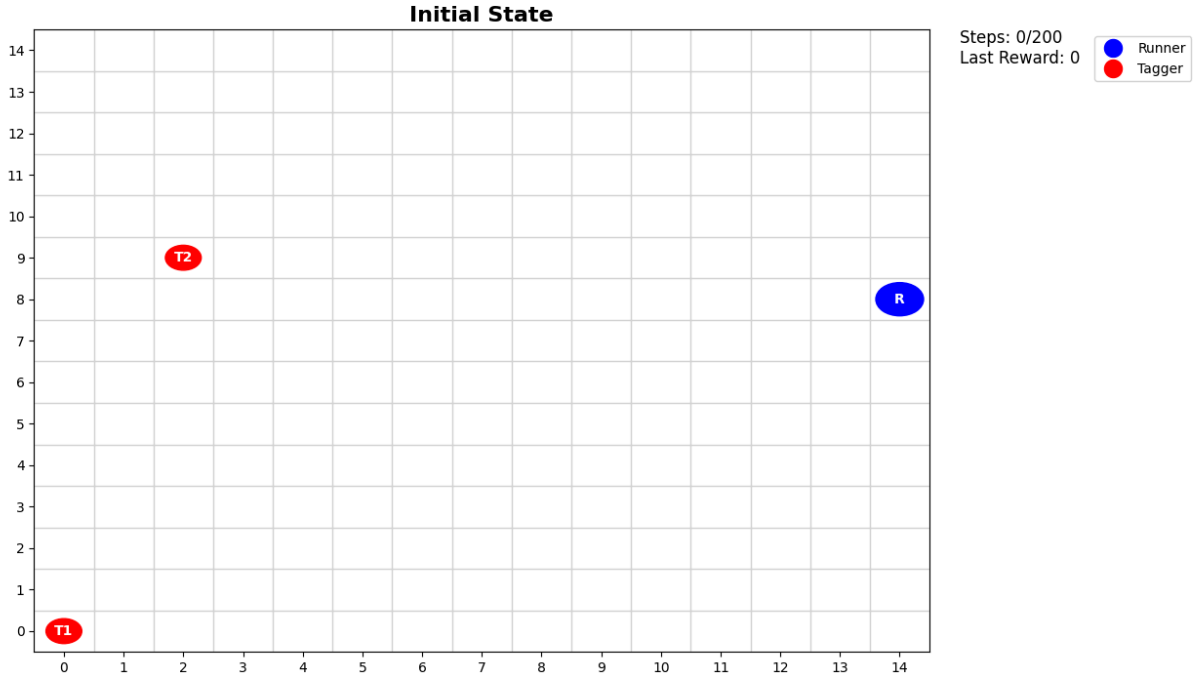
- **Returns**:
  - The updated observation (runner and tagger positions), reward, and whether the episode is done (`done`).
- **Functionality**:
  - Increases the step counter.
  - Moves the runner in the specified direction.
  - Randomly moves the taggers.
  - Checks if the runner has been caught by any tagger or if the maximum number of steps has been reached.
  - Assigns a reward based on whether the runner is caught (`-100`), survives the maximum steps (`100`), or survives the current step (`1`).
  - Sets the `done` flag to `True` if the runner is caught or the maximum steps are exceeded.

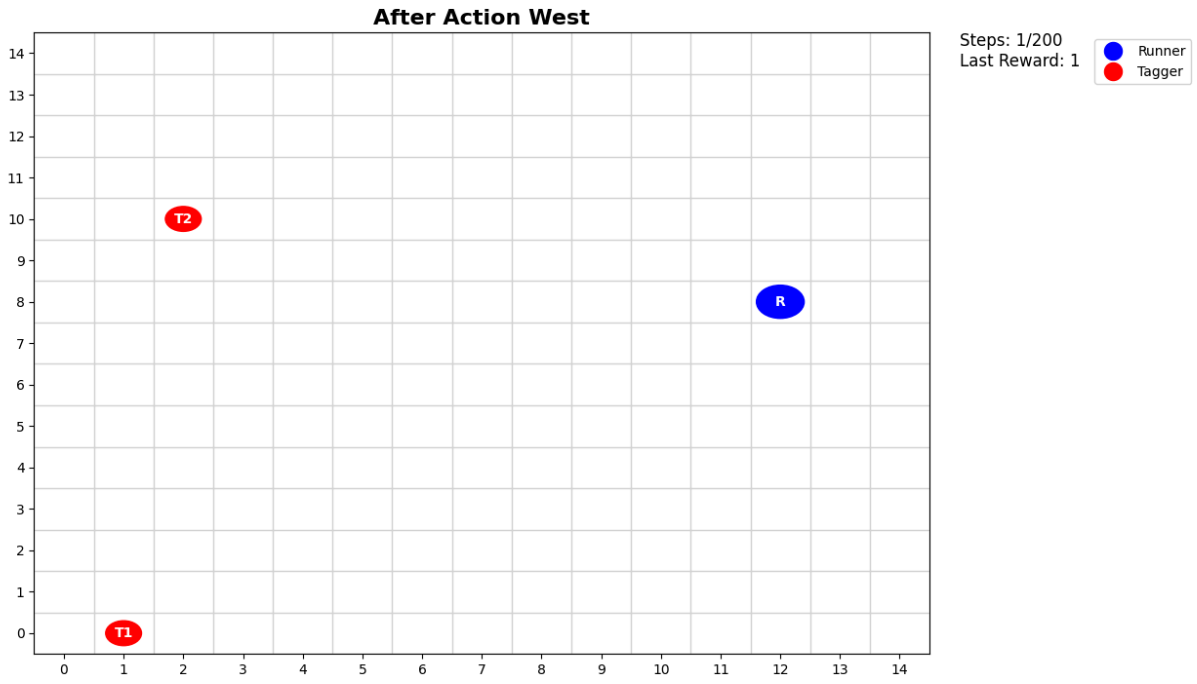
4. **`_get_obs(self)`**:
   - **Purpose**: Provides the current observation, which includes the positions of the runner and taggers.
   - **Returns**: A concatenated array of the runner's and taggers' positions.
5. **`_move_runner(self, action)`**:
   - **Purpose**: Moves the runner by two steps in one of the 8 possible directions based on the input `action`.
   - **Parameters**:
     - `action`: An integer indicating the direction of movement (0 to 7).
   - **Functionality**:
     - Updates the runner's position using pre-defined movement vectors for each direction.
     - Ensures the runner stays within the bounds of the grid
6. **`_move_taggers(self)`**:
   - **Purpose**: Randomly moves each tagger one step in one of the four cardinal directions (up, down, left, right).
   - **Functionality**:
     - Each tagger's position is updated using randomly selected movement vectors.
     - Ensures taggers stay within the grid bounds.
7. **`sample_action(self)`**:
   - **Purpose**: Generates a random action for the runner, which can be used for simulating the environment.
   - **Returns**: A tuple containing the action (integer) and its corresponding label (direction string).
8. **`get_action_space(self)`**:
   - **Purpose**: Returns the size of the action space, which is 8 in this case, corresponding to the runner's 8 possible movement directions.
   - **Returns**: The size of the action space.

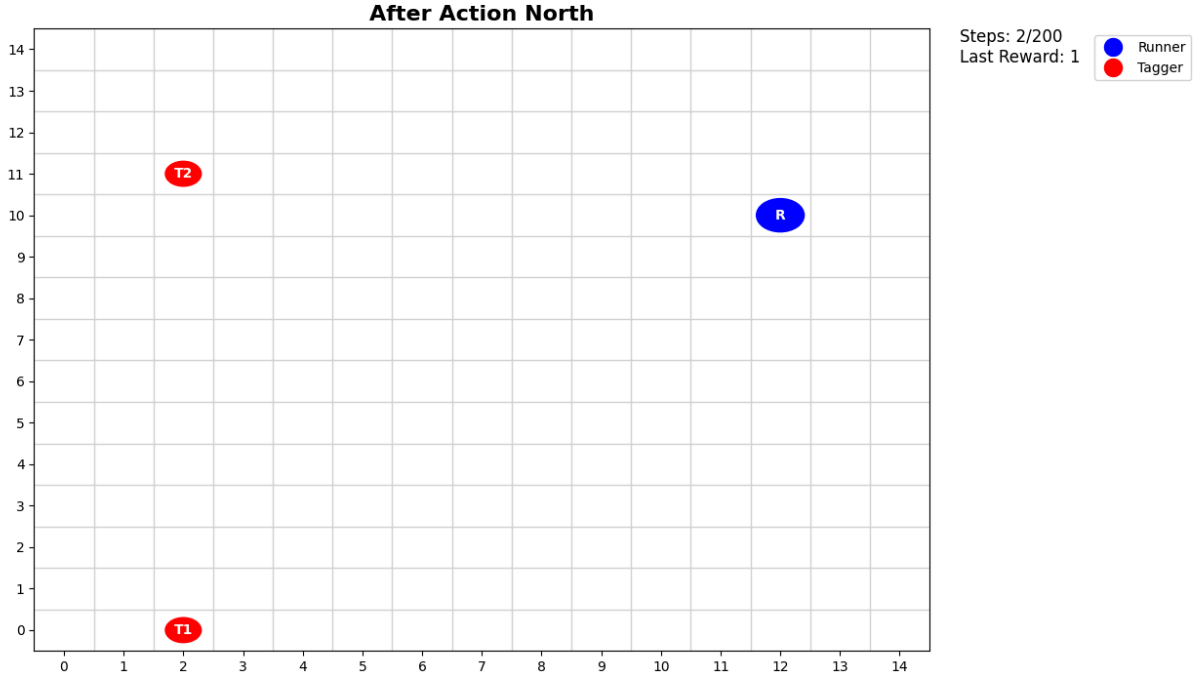
Task-2 - Visualizing the environment

For this task, a function `visualize_grid_world(env, title=None)` is defined that creates a visual representation of the current state of the grid world environment defined in the `GridWorldEnv` class.

1. **Setting Up the Plot**:
    - A square plot is created with a grid of size `n x n`, where n is the size of the environment.
    - The grid lines are drawn using light gray lines to form the structure of the grid.
2. **Visualizing the Runner**:
    - The runner's position is retrieved from the environment and represented as a blue circle placed at the center of the corresponding grid cell.
3. **Visualizing the Taggers**:
    - The taggers' positions are retrieved from the environment, and each tagger is represented as a red circle on the grid.
    - Labels such as "T1", "T2", etc., are added to the taggers to differentiate between them.
4. **Displaying Title and Information**:
    - If provided, a title is displayed at the top of the plot.
    - Additional information about the environment, such as the number of steps taken and the last reward, is displayed in the corner of the grid.
5. **Rendering the Plot**:
    - The function finalizes the layout and displays the plot using `matplotlib`, offering a clear visual representation of the current state of the environment.




After Action West
T2
R
T1
Steps: 1/200
Last Reward: 1
Runner
Tagger

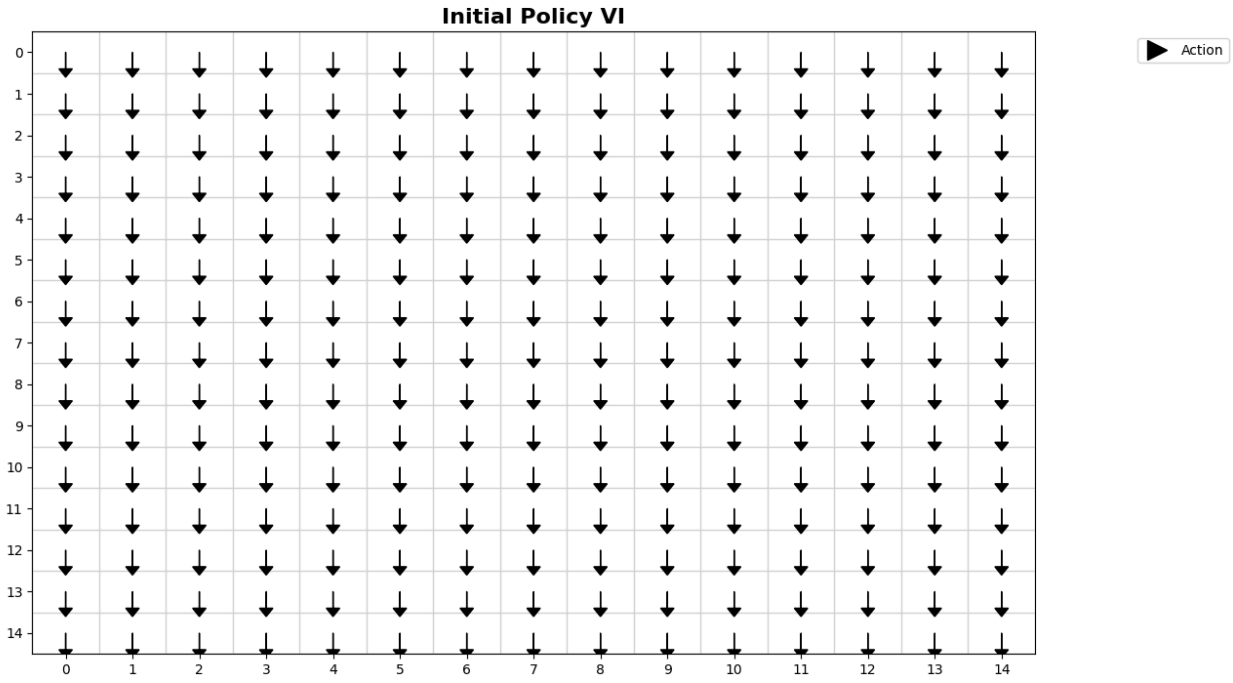

After Action North
T2
R
T1
Steps: 2/200
Last Reward: 1
Runner
Tagger

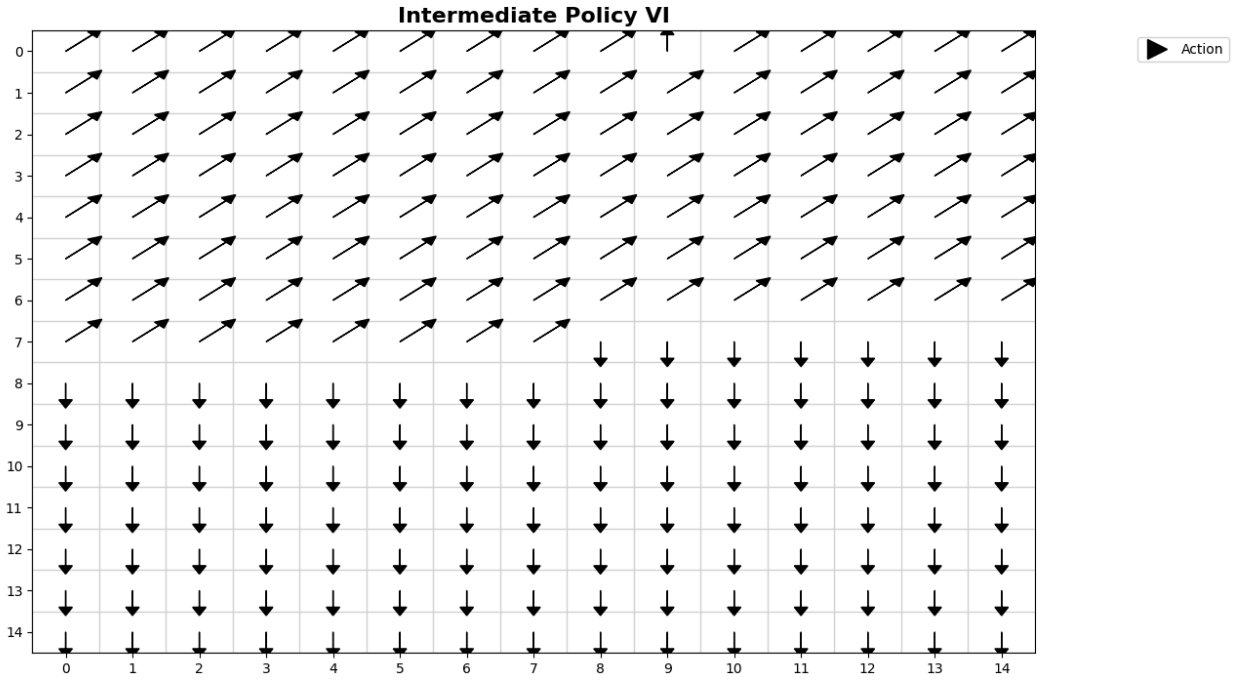
Task 3 - a) Demonstrating Value Iteration Algorithm (VI) on the GridWorldEnv

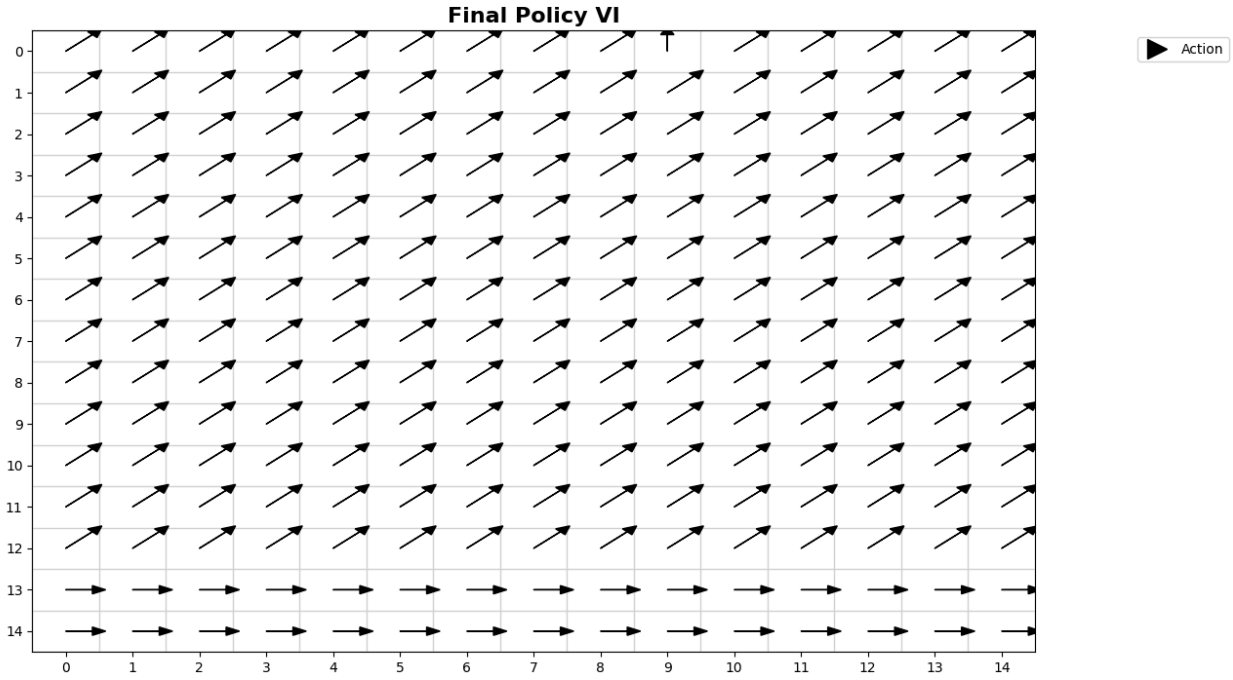
Implemented the value iteration algorithm to compute the optimal policy for the runner in the `GridWorldEnv` environment. It aims to maximize the expected cumulative reward over time by iterating over state values and determining the best actions.

1. Initialization:
    - A value matrix V is initialized, where each cell represents the estimated value of the corresponding state in the grid.
    - A small threshold (`theta`) is set to determine when the value iteration has converged. The discount factor (`gamma`) controls how future rewards are weighted.
2. Value Iteration Loop:
    - The value of each state in the grid is updated iteratively based on the Bellman optimality equation.
    - For each state (represented by a position in the grid), the algorithm evaluates all possible actions the runner can take.
    - After selecting an action, the environment's state is updated, and the algorithm computes the expected value of the next state based on the reward and the discounted value of future states.
    - The state value is updated with the maximum value across all possible actions, ensuring that the best action is selected.
    - The difference between the previous and new value (`delta`) is tracked to check if the updates are significant. Once `delta` falls below `theta`, the loop terminates, indicating convergence.
3. Stopping Criteria:
    - The value iteration loop continues until the maximum change in state values (`delta`) across all grid cells is less than the predefined threshold (`theta`), indicating that the values have stabilized and the optimal values have been found.
4. Policy Extraction:
    - Once the value matrix has converged, a policy is derived by selecting the action that maximizes the value for each state.
    - The policy matrix is initialized, and for each grid cell, the action with the highest expected value is stored.
    - Intermediate steps of the policy are captured to illustrate how the policy evolves, specifically highlighting the central position of the grid.
5. Outputs:
    - The function returns the final value matrix, the initial empty policy, an intermediate policy (when the state is the center of the grid), and the final optimal policy.

Initial, Intermediate and Final Policies of the Value Iteration Algorithm






Final Policy VI
Action
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Task 3 - b) Demonstrating Temporal Difference Learning (TD) on the GridWorldEnv
Implemented TD learning with eligibility traces to estimate the state-value function V(s) and derive an optimal policy for the `GridWorldEnv` environment

.
Key Parameters:

- episodes: Number of episodes for learning.
- alpha: Learning rate for updating the value function.
- gamma: Discount factor for future rewards.
- epsilon: Probability for exploration in the epsilon-greedy policy.
- lambd: Decay rate for the eligibility trace.

Eligibility Traces:

- Tracks the frequency of state visits during an episode, helping to distribute the TD error across visited states.

State Exploration:

- The agent uses an epsilon-greedy policy to balance exploration and exploitation when choosing actions.

TD Error Calculation:

- The TD error δ is computed as the difference between the predicted and actual rewards, which is used to update the value function.

Value Function Update:

- The value function V(s) is updated using the TD error and eligibility trace for each state, helping the agent learn over time.

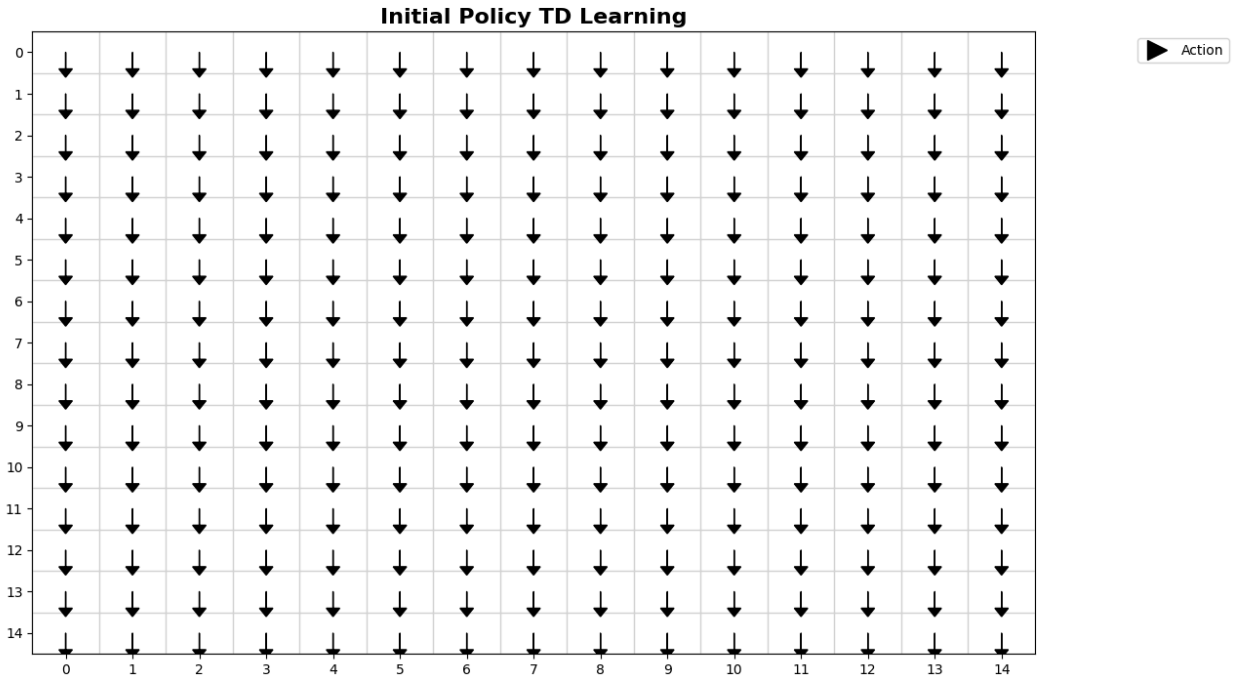
Eligibility Trace Decay:

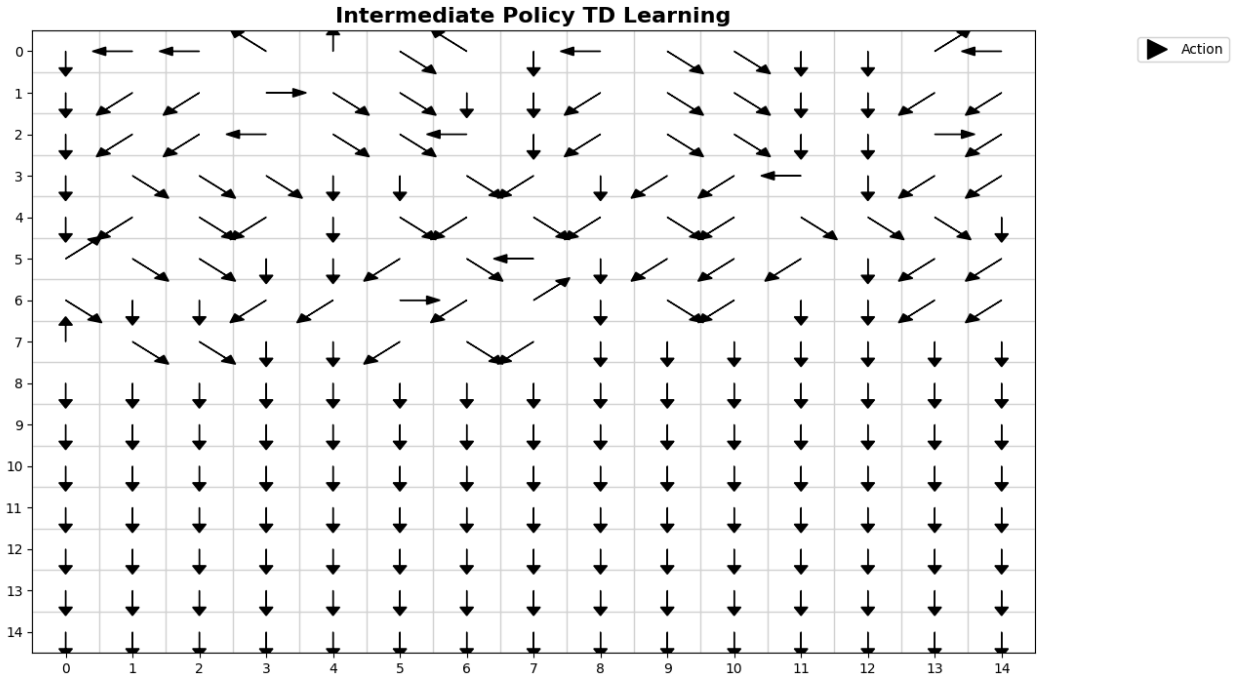
- After each update, the eligibility trace decays by a factor of λ\lambdaλ, reducing the impact of earlier state visits.

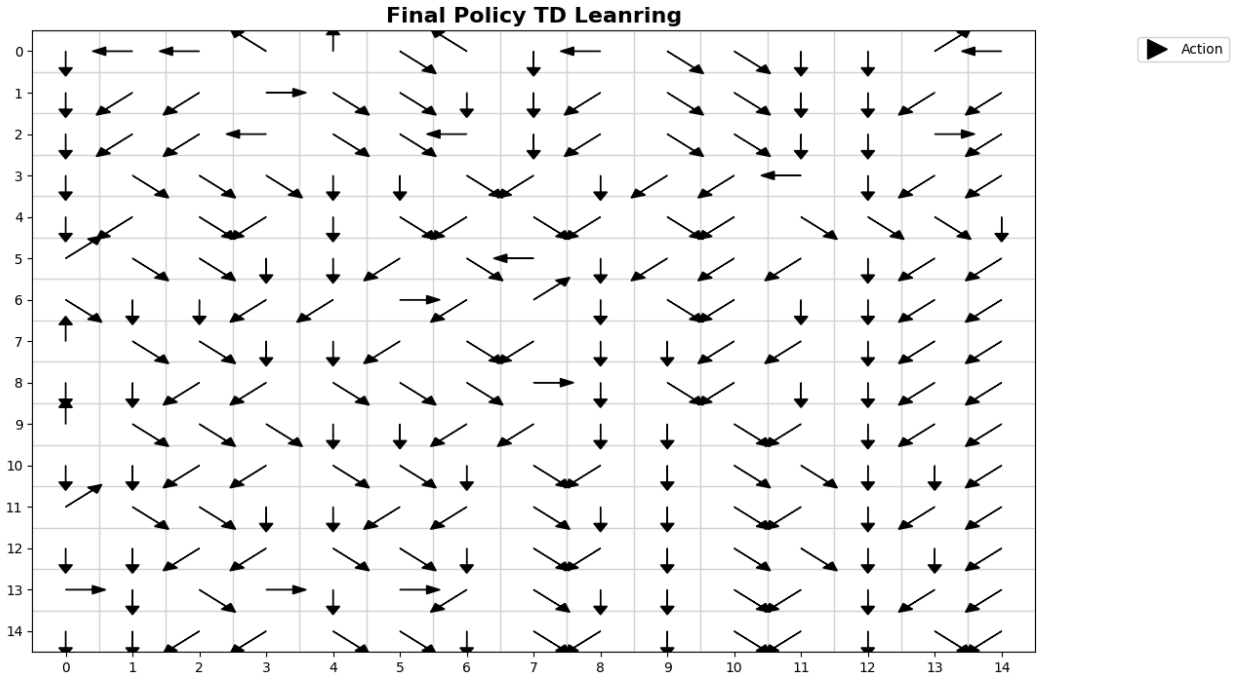
Policy Extraction:

- The optimal policy is derived by selecting actions that maximize future rewards based on the learned value function.
- The function returns the final value function and the initial, intermediate, and final policies.

Initial, Intermediate and Final Policies of the Temporal Difference Learning Algorithm





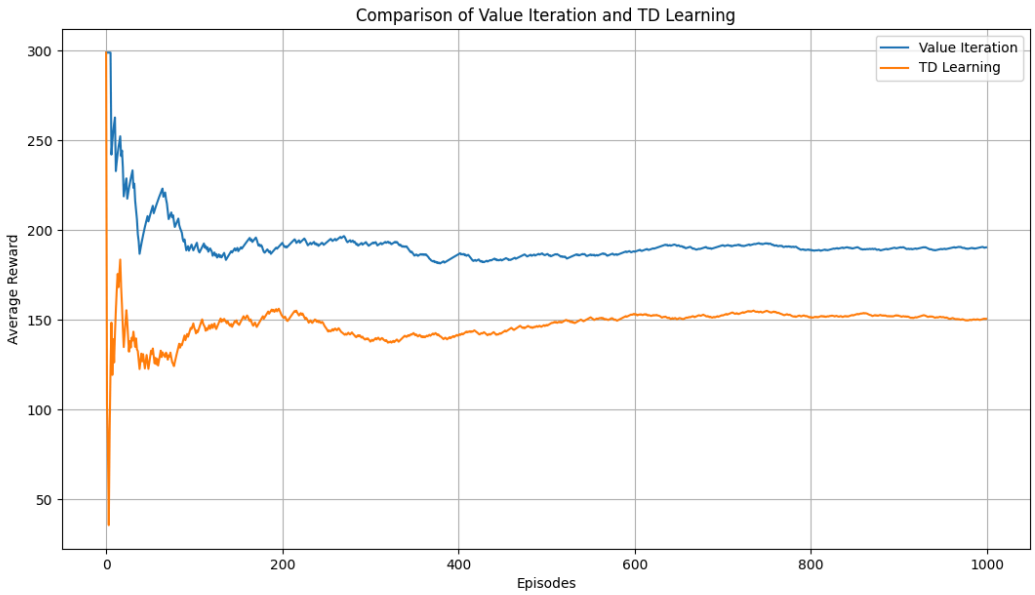


Task 3 - c) Run the environment with VI and TD algorithms from github repo

A few changes in names of variables were made to adapt the algorithms to the custom defined environment.

After making the final changes - the notebooks “**Value_Iteration_Solution.ipynb”** and “**SARSA_Solution.ipynb”** are added to the code files from the github repository.

Task 4 - Compare the results of the above algorithms



1. **Policy Analysis (Value Iteration vs TD learning):**
    - In the TD Learning policy, the arrows indicate that the agent tends to move downward or in various diagonal directions, possibly aiming to survive by keeping a random yet evasive strategy. However, the policy looks noisy, with no clear directional bias, suggesting that the agent might not have fully learned an optimal escape strategy.
    - In contrast, the Value Iteration policy shows a much clearer pattern. Most arrows point towards a specific diagonal direction or to the right, suggesting that the agent consistently moves toward a specific direction to evade the taggers. This indicates that value iteration has found a more stable and deterministic policy for survival.
2. **Plot Analysis:**
    - In the reward plot, we see that Value Iteration (blue curve) stabilizes at a higher reward compared to TD Learning (orange curve). This aligns with the final policies, as Value Iteration has a more stable escape strategy, while TD Learning seems to fluctuate more, possibly due to incomplete learning.
    - The lower reward in TD Learning could be attributed to its slower convergence compared to Value Iteration, which is more robust and precise.

Conclusion:

- Since TD Learning is an online learning method, the runner's agent might take longer to learn the optimal policy, explaining the noisier behavior. It learns through experience, so with fewer episodes or noisier exploration, it might still be in the process of refining the policy.
- Value Iteration, being a model-based method, computes the optimal policy more reliably from the start, hence it learned the more optimal and consistent policy for evading taggers.

Bonus - Maximum N and k for which the algorithms are feasible

To compare feasibility, we used a 5-minute time limit for running the algorithms.

Value Iteration algorithm

Grid size **n = 30** and Number of taggers **k = 4** takes approximately 5 minutes to run

Temporal Difference Learning algorithm

Grid size **n = 80** and Number of taggers **k = 4** takes approximately 5 minutes to run