AIL 7022: Assignment 1
Semester I, 2025-26

Due 4th September 2025 12:00 PM

Instructions:

- You should submit all your code (including any pre-processing scripts written by you) and any graphs that you might plot.
- Include **a write-up (pdf) file**, which includes a brief description for each question explaining what you did. Include any observations and/or plots required by the question in this single write-up file.
- Please use Python for implementation using only standard python libraries (e.g. numpy). Do not use any third party libraries/implementations for algorithms.
- Your code should have appropriate documentation for readability.
- You will be graded based on what you have submitted as well as your ability to explain your code.
- This assignment is supposed to be done individually. You should carry out all the implementation by yourself.
- We plan to run Moss on the submissions. We will also include submissions from previous years since some of the questions may be repeated. Any cheating will result in a zero on the assignment, a penalty of -10 points and possibly much stricter penalties (including a **fail grade** and/or a **DISCO**).
- Starter code is available at this link.
- A total of 5 buffer days are allowed across all assignments. After the exhaustion of the 5 buffer days 10% penalty will be levied for each day late.

1 Policy Iteration and Value Iteration(60 Marks)

In this section, you will implement the standard policy iteration and value iteration for the custom(gym environment) environment provided. Use the discount factor ($\gamma = 0.95$) and the convergence criteria to be used is based on the value function's stability. The algorithms will iterate until the maximum absolute change in the value function across all states between successive iterations falls below a small threshold, θ. For this implementation, the threshold is set to $\theta = 10^{-6}$. It is important to keep the convergence same for all parts for the sake of comparision.

This convergence condition can be formally expressed as:

$$\max_{s \in \mathcal{S}} |V_{k+1}(s) - V_k(s)| \leq \theta$$

where $V_k(s)$ is the value of state s at iteration k. For Policy Iteration, this criterion applies to the inner Policy Evaluation step, while the outer loop terminates when the policy becomes stable.

1.1 Environment Description

The environment you will be working on is the Football Skills Environment(a custom gymnasium environment), which is implicitly divided into a grid of size 20x20. The agent, representing a football player, always starts at the top-center of the pitch at a fixed position of `(0, 10)`. The challenge is designed around a specific training scenario where the agent must dribble and score against a wall(of cones and cutouts). The environment's dynamics, rewards, and states are described below.

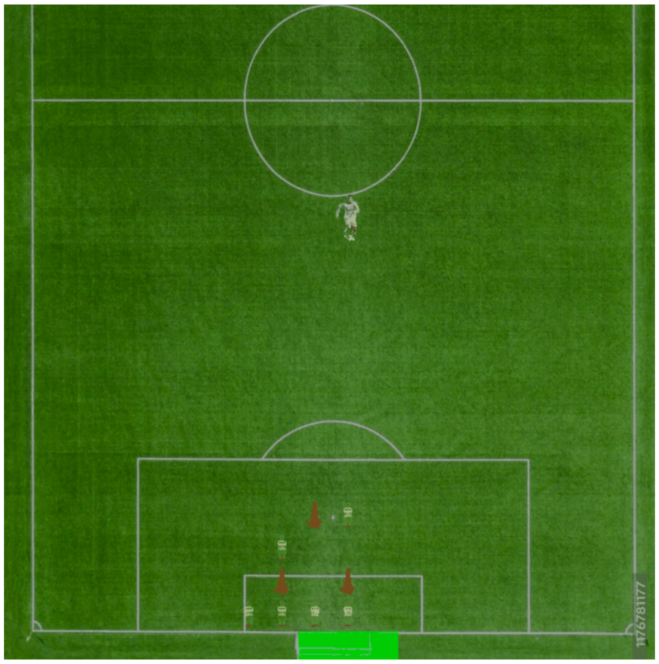

Figure 1: Snapshot of the environment

1.1.1 State Space

The state of the environment is defined by a tuple `(row, column, has_shot)`, where:

- **row** and **column** represent the player's position on the 20x20 grid. The top-left corner is `(0, 0)`.
- **has_shot** is a binary flag (0 or 1) that indicates whether the player has taken a shot. Once a shot is taken, the episode ends.

The total number of states is $20 \times 20 \times 2 = 800$.

1.1.2 Action Space

The agent has a discrete set of 7 actions available, which are divided into two categories:

- **Movement Actions (Deterministic):** These actions move the player one square in the specified direction.
  - 0: Move Up
  - 1: Move Down
  - 2: Move Left
  - 3: Move Right
- **Shooting Actions (Stochastic):** These actions end the episode. The ball travels a set distance, and its final position determines the outcome.
  - **4: Short Straight Shot:** A high-accuracy shot intended to travel approximately **2 rows forward**. It has a high probability of going straight.
  - **5: Long Straight Shot:** A lower-accuracy shot intended to travel approximately **4 rows forward**. It is more powerful but has a higher chance of missing the target.
  - **6: Long Right Curl Shot:** A skillful shot that travels approximately **4 rows forward** while also **curving to the right**. This is useful for bending the ball around obstacles. (Note the absebce of the curl left shot, it is difficult for a player to bend the ball both ways!)

1.1.3 Transition Dynamics

The environment can operate in two modes: a standard stationary mode, where transition matrix is fixed throughout the episode, and a non-stationary "degraded pitch" mode, where transition matrix changes with each timestep. To handle both cases correctly, you must retrieve the transition probabilities for a given state-action pair by using the provided helper method.

You **must** use `env.get_transitions_at_time(state, action, time_step)` to get the transition dynamics. **Do not pass anything for time_step when working with the stationary transition matrix.** Do **not** access the `env.T` attribute directly in both the cases. This method returns a list of possible outcomes, where each outcome is a tuple of the form `(probability, next_state)`.

1.1.4 Reward Structure

The reward function is designed to encourage scoring goals efficiently while avoiding mistakes:

- **Goal:** A large positive reward is given for scoring. This reward includes a bonus based on the distance from which the shot was taken, incentivizing skillful long shots. Note that the goal can be shot if the distance between the goal row and the player row is either 2 or 4.
- **Missed Shot:** A significant negative reward of -30 is given if the ball does not land in one of the goal positions.
- **Obstacle Collision:** Colliding with a cone results in a -10 penalty, while hitting a cutout defender is a -20 penalty.
- **Movement Cost:** Every movement action incurs a small penalty of -1 to encourage the agent to find the most direct path to a good shooting position.

1.1.5 Terminating Conditions

An episode concludes under the following conditions:

- The primary terminating condition is when the agent performs any of the three shooting actions (4, 5, or 6). Once a shot is taken, the episode ends, and the final reward is calculated based on the ball's landing position.

1.2 Stationary Environment Analysis

In this section, you will implement and compare the standard Value Iteration and Policy Iteration algorithms on the stationary version of the Football Skills Environment. Your goal is to analyze their computational properties and the impact of the discount factor on the resulting policies. Hint: All algorithms will converge in under 10s on i9-12900, 32 GB RAM.

1. Implement the standard Policy Iteration and Value Iteration algorithms. You should create the stationary environment instance using `env_normal = FootballSkillsEnv(render_mode='gif')`. Use a discount factor of $\gamma = 0.95$ and a convergence threshold of $\theta = 10^{-6}$.
2. Report the count of total number of calls made to the `env.get_transitions_at_time(state, action)` function. Report this count for both the algorithms.
3. Compare the final policies obtained from both algorithms. Are they identical? Explain your observation based on the theoretical guarantees of these methods.
4. Evaluate the performance of each policy by running 20 episodes with different seeds. For each evaluation run, you can set the seed using `obs, _ = env.reset(seed=your_seed)`. Report the mean reward and the standard deviation of the reward for both the Value Iteration and Policy Iteration policies.
5. Repeat the entire analysis from questions 1, 2, and 3 for two different discount factors: $\gamma = 0.3$ and $\gamma = 0.5$. Summarize how the change in γ affects the number of transition calls, the final policy, and the evaluated reward.
6. For the policy obtained using $\gamma = 0.95$, create the GIF of one episode using both the algorithms. You may use the helper function `env.get_gif(policy)`

1.3 Non-Stationary Environment Analysis

In this section, you will analyze the environment in its non-stationary "degraded pitch" mode. Because the transition dynamics evolve with each time step, the optimal policy will also be a function of both state and time, i.e., $\pi(s,t)$. You will implement a time-dependent version of Value Iteration. Hint: All algorithms will converge in under 10s on i9-12900, 32 GB RAM.

1. Implement a variation of the earlier Value Iteration algorithm by incorporating time as a part of the state. You will run this algorithm for a **maximum horizon length of 40**. The non-stationary environment can be created using `env_degraded = FootballSkillsEnv(render_mode='gif', degrade_pitch=True)`. Use a discount factor of $\gamma = 0.95$.
2. Report the total number of calls made to the

   `env.get_transitions_at_time( state, action, time_step)` function.
3. For comparison implement your usual Value Iteration(without time as a state), this can be done by making calls to get_transition_at_time function without specifying the time step and maintaing a Value function that depends only on the state. What do you observe?
4. Evaluate the performance of each of the two resulting policies by running 20 episodes with different seeds. For each evaluation run, you can set the seed using `obs, _ = env.reset(seed=your_seed)`. Report the mean and standard deviation of the reward for both policies.
5. Create a GIF of one episode for each of the two policies. Since the policy is time-dependent, you may use a helper function `get_gif(policy, is_time_dependent=True)`.

1.4 Can we do something better?

Value Iteration can be quite inefficient in many case. Consider a case when the successors of a particular state do not undergo any change in value function. Is it useful to backup the value function for that particular state? In this section you are required to modify the Value Iteration algorithm such that states that are more likely to benefit from the Bellman backup are prioritized over states that are already close to convergence.

1. Implement the modified Value Iteration algorithm and run it for the Stationary Environment. Justify your design choices.
2. Compare the total number of calls made to the `env.get_transitions_at_time(state, action)` function with VI and PI. Did your algorithm reduce the number of call made to the Transition function?

2 Dynamic Programming (40 Marks)

2.1 Environment setup

- Install gym, using pip install gym.

2.2 Online Knapsack Problem

Use Value iteration to solve the Online Knapsack problem. The objective is to collect items with a given weight and value such that value of items collected in a knapsack is maximized within a given weight limit of the knapsack. At each time step the agent is presented an option to accept or reject the item given the current weight of the knapsack, the item index, weight of the item presented and the value of the item. If the agent accepts the item, it is added to the knapsack i.e the weight and value of the knapsack increase by item weight and value. If the agent rejects the item, a new item will be presented. The number of steps in each episode is limited and each item may be presented more than once during the episode (with replacement), however there is a limit to the number of items of each item index that can be added to the knapsack. This limit is chosen randomly from between 1 and 10 when the environment is initialized. Item weights and values are also initialized randomly when the environment is reset.

2.2.1 Problem Formulation

For A set of n items, where item j has value v_j and weight w_j and Knapsack capacity: B, each item can only be added a maximum of c_j times in the knapsack. The objective is :

$$\begin{aligned}
\text{Maximize:} \quad & \sum_{j=1}^{n} v_j x_j \\
\text{Subject to:} \quad & \sum_{j=1}^{n} w_j x_j \leq B \\
& x_j \in \{0,1\}, \quad \forall j \\
& \sum_{j} x_j < c_j, \quad \forall j
\end{aligned}$$

1. Implement the value iteration policy to solve the environment STARTER CODE. The number of items is 200 and maximum weight of knapsack is 200. The environment runs for 50 time steps per episode. Report the value of the knapsack after training for five different seeds. Plot the value of the knapsack as the items are presented during evaluation for the five seeds. For a seed plot the heatmap of the final value function with value of each state as the intensity such that Current weight of knapsack is on the y-axis. On the x-axis plot weight, value and ratio of weight to value. Sort the values on the x-axis in increasing order before plotting.

2. Implement the value iteration policy to solve the environment. The number of items is 200 and maximum weight of knapsack is 200. The environment runs for 50 time steps per episode. Report the value of the knapsack after training for five different seeds. Plot the value of the knapsack as the items are presented during evaluation for the five seeds. For a seed plot the heatmap of the final value function with value of each state as the intensity such that Current weight of knapsack is on the y-axis. On the x-axis plot weight, value and ratio of weight to value. Sort the values on the x-axis in increasing order before plotting.

3. Repeat the value iteration with $\{10, 50, 500\}$ steps per episode for a single seed. Plot the heatmap of the final value function with value of each state as the intensity such that Current weight of knapsack is on the y-axis. On the x-axis plot weight, value and ratio of weight to value. Sort the values on the x-axis in increasing order before plotting.

Note : You are not allowed to tweak the code of the environment files.
Hint : On Ryzen7 5800h, policy iteration with step limit of 50 took 1510 seconds. During development and testing you may chose smaller step limit to ensure correctness.

2.3 Portfolio Optimization

A commodity trader is wants to compare his trading decisions for an asset "Unobtainium" with optimal decisions he should have taken to maximize wealth. In this task you will use dynamic programming to find the optimal decisions. Maximize the total wealth of a portfolio of cash and an asset Unobtanium. An initial amount of cash is given at the start of the simulation. At each time step you can buy or sell the asset by paying a transaction cost You may also choose to forgo the buying and selling and keep the status quo. The prices of the asset change during each time step. Cash value remains constant i.e. it does not automatically decay or accrue interest. The wealth of the portfolio is calculated when the episode ends by adding the values of cash and total value of each asset in the portfolio. There is a limit on the quantity of each asset that can be bought/sold at each step. The episode ends when a fixed number of time steps have been executed or if the amount of any given asset held becomes negative. Your solution should not make any explicit assumptions about the change in asset prices over the episode. For evaluation you may be asked to re-train the model for a hidden price history.

Details of the environment are given below :
Number of assets besides cash is 1 (Unobtainium). Initial cash is 20. Step limit is 10. Each asset can be bought in a lot size of 1 or 2. When selling, the lot size is represented as -1,-2. For keeping the status quo the lot size is 0. Buy/sell costs 1 per transaction. Asset mean represents the mean price of the asset over the episode. The price variance can be set by providing a keyword argument "variance". For problems 1 and 2, use the following price sequences :

- 1,3,5,5,4,3,2,3,5,8

- 2,2,2,4,2,2,4,2,2,2
- 4,1,4,1,4,4,4,1,1,4

For part 3. use the default sequence.

2.3.1 Problem Formulation

- T: investment horizon (number of periods), here $T = 10$.
- N: number of risky assets (excluding cash), here $N = 1$.
- $p_i(t)$: price of asset i at time t, $\quad i = 1, \ldots, N$.
- b_i: buy transaction cost for asset i.
- s_i: sell transaction cost for asset i.
- $H_i^{\max}$: maximum allowable holdings of asset i.
- C_0: initial cash available at $t = 0$.

For each period $t = 0, \ldots, T-1$ and asset $i = 1, \ldots, N$:

- $x_i(t) \in \{-2, -1, 0, 1, 2\}$: number of shares of asset i bought ($x_i(t) > 0$) or sold ($x_i(t) < 0$) in period t.
- $h_i(t) \geq 0$: number of shares of asset i held at the end of period t.
- $C(t) \geq 0$: amount of cash held at the end of period t.

$$W(t) = C(t) + \sum_{i=1}^{N} p_i(t) \cdot h_i(t), \quad t = 0, \ldots, T.$$

For each $t = 0, \ldots, T-1$:

$$h_i(t+1) = h_i(t) + x_i(t), \quad \forall i$$

$$C(t+1) = C(t) - \sum_{i=1}^{N} x_i^+(t) \cdot (p_i(t) + b_i) + \sum_{i=1}^{N} x_i^-(t) \cdot (p_i(t) - s_i)$$

where

$$x_i^+(t) = \max\{x_i(t), 0\}, \quad x_i^-(t) = -\min\{x_i(t), 0\}.$$

$$h_i(t) \geq 0, \quad \forall i,\ t = 0, \ldots, T \tag{1}$$
$$h_i(t) \leq H_i^{\max}, \quad \forall i,\ t = 0, \ldots, T \tag{2}$$
$$C(t) \geq 0, \quad t = 0, \ldots, T \tag{3}$$
$$x_i(t) \in \{-2, -1, 0, 1, 2\}, \quad \forall i,\ t = 0, \ldots, T-1 \tag{4}$$

$$\max_{\{x_i(t)\}} \quad W(T) = C(T) + \sum_{i=1}^{N} p_i(T) \cdot h_i(T)$$

1. Implement the value iteration algorithm to solve the environmentSTARTER CODE. For discount factor $\gamma \in \{0.999, 1.0\}$: Plot the total portfolio wealth at the end of each episode as the training proceeds. After training, plot the evolution of total wealth, cash and the number of units of asset held across time steps for an episode. Also report the execution time of the code.
2. Implement the policy iteration algorithm to solve the environment. For discount factor $\gamma \in \{0.999, 1.0\}$: Plot the total portfolio wealth at the end of each episode as the training proceeds. After training, plot the evolution of total wealth, cash and number of units of asset held across time steps for an episode. Also report the execution time of the code.
3. Change the variance of price to 1.0 by providing keyword argument "variance" set to 1.0 during environment initialization. Does the policy iteration algorithm converge to the optimal solution in this setting? Plot the maximum value difference vs iteration count for policy iteration. Assume convergence threshold of 10^{-2} for this problem. You may limit the maximum iterations to 1000 for reporting this result.

Note : You will not be evaluated on execution time as this may be machine dependent. Transaction cost may be changed during evaluation to check the correctness of the code. You may be asked to re-run the training with a different cost to ensure correctness of the implementation.
Hint : Try to limit the size of the state space to be explored to ensure the code runs within reasonable compute resources. Running time for 100 policy iteration updates on Ryzen 7 5800h was approximately 5400 seconds. Value iteration with variance set to 0.0 should converge much faster. Or-gym environment is adapted from https://github.com/hubbs5/or-gym.