CS747 - Foundations of Intelligent and Learning Agents

CS747 Autumn 2023

Chinmay Makarand Pimpalkhare
200100115

Assignment 2 | Markov Decision Planning Methods for Half-field offense Football

1 Task 1

Default algorithm used is Value Iteration, due to fastest convergence even on the large MDP

1.1 Value Iteration

1.1.1 Bellman Optimality Operator

Implementation:

```
def Bellman_optimality_operation(num_states, num_actions, mdp_type, is_mdp_episodic, \
    these_are_episodic, transition_matrix, reward_matrix, \
    gamma, value_function
    ):
    #Steps: For all states, max over actions (sum of (T*(R + gamma*V)))
    new_value_function = np.zeros(num_states)
    optimal_action = np.zeros(num_states)

    for i in range(num_states):
        if these_are_episodic[i] == 0:
            new_value_function[i] = np.max(np.sum(np.multiply(transition_matrix[i], \
                                                reward_matrix[i] +\
                                                    gamma*value_function), axis = 1))
            optimal_action[i] = np.argmax(np.sum(np.multiply(transition_matrix[i], \
                                                reward_matrix[i] +\
                                                    gamma*value_function), axis = 1))

    return new_value_function, optimal_action
```

Details:

- The new value function $\boldsymbol{V}$ has been initialized to $\mathbf{0}$ before applying the operator to it
- The value functions of only non-terminal states are updated. Thus, it is guaranteed that terminal states will always have a value function which is identically $\mathbf{0}$

1.1.2 Value Iteration

Implementation

```
    def value_iteration(num_states, num_actions, mdp_type, is_mdp_episodic, \
    these_are_episodic, transition_matrix, reward_matrix, \
    gamma):
    epsilon = 1e-12
    V_old = np.zeros(num_states)
    V_new, pi = Bellman_optimality_operation(num_states, num_actions, mdp_type,
    is_mdp_episodic, \
```

```
        these_are_episodic, transition_matrix, reward_matrix, \
        gamma, V_old)
    V_copy = V_old
    while np.linalg.norm(V_new - V_copy, 1) >= epsilon:
        V_copy = V_old
        V_new, pi = Bellman_optimality_operation(num_states, num_actions, mdp_type,
    is_mdp_episodic, \
        these_are_episodic, transition_matrix, reward_matrix, \
        gamma, V_old)
        V_old = V_new
    for i in range(num_states):
        print(V_new[i],int(pi[i]))
```

Details:

- The initial value function at the start of solving the MDP, $\boldsymbol{V}_0$ has been initialized to $\mathbf{0}$
- The Bellman optimality operator is then repeatedly applied to the value function to get the updated value function.
- The termination criteria used is that the $\mathscr{L}_1$ norm of $V^t - V^{t+1}$ should be less than or equal to $\varepsilon = 10^{-12}$

1.2 Linear Programming

Implementation

```
    def linear_programming(num_states, num_actions, mdp_type, is_mdp_episodic, \
    these_are_episodic, transition_matrix, reward_matrix, \
    gamma):
    prob = LpProblem("Linear_Programming_Value", LpMaximize)
    set_S = range(0, num_states)
    set_A = range(0, num_actions)
    val_func = LpVariable.dicts("Value_Function", set_S, cat='Continuous')

    #objective function for the primal
    prob += lpSum([-val_func[i] for i in set_S])

    #nk constraints
    for i in set_S:
        for j in set_A:
            prob += val_func[i] >= lpSum([transition_matrix[i][j][k]*\
                                (reward_matrix[i][j][k] + \
                                gamma*val_func[k]) for k in set_S])


    #solve
    prob.solve(PULP_CBC_CMD(msg=0))
    V_star = np.array([val_func[i].varValue for i in set_S])
    Q_values = np.zeros((num_states, num_actions))
    for i in set_S:
        for j in set_A:
            Q_values[i][j] = np.sum(np.multiply(transition_matrix[i][j], \
                                            reward_matrix[i][j] + \
                                                gamma*V_star))
    pi_star = np.argmax(Q_values, axis = 1)
    for i in set_S:
        print(V_star[i], int(pi_star[i]))
```

Details:

- Objective function used is (as given in slides)

$$\min -\left(\sum_{s \in S} V(s)\right)$$

- nk Constraints used are also as given in the slides

- The primal problem is solved and to compute the optimal policy, we compute Q-values and pick the action with maximum Q-value for a given state
- Ties are broken internally by np.argmax

1.3 Howard's Policy Iteration

Implementation

```
    def howard_policy_iteration(num_states, num_actions, mdp_type, is_mdp_episodic, \
            these_are_episodic, transition_matrix, reward_matrix, \
            gamma):

    #initialize arrays for improvable states and improvable actions
    #corresponding to the improvable states
    improvable_states = np.ones(num_states)
    improving_actions = np.zeros((num_states, num_actions))
    pi_old = np.random.randint(num_actions, size = num_states)
    while(np.sum(improvable_states) > 0):
        V_old = policy_evaluation_no_print(num_states, num_actions, mdp_type,
    is_mdp_episodic, \
                these_are_episodic, transition_matrix, reward_matrix, \
                gamma, pi_old)
        Q_pi = compute_action_values(num_states, num_actions, mdp_type, is_mdp_episodic, \
                these_are_episodic, transition_matrix, reward_matrix, \
                gamma, V_old)
        improvable_states.fill(0)
        improving_actions.fill(0)
        for i in range(0,num_states):
            for j in range(0,num_actions):
                if Q_pi[i][j] > V_old[i] and j != pi_old[i] :
                    improvable_states[i] = 1
                    improving_actions[i][j] = 1
        for i in range(0,num_states):
            if improvable_states[i] == 1:
                pi_old[i] = int(np.random.choice(num_actions,\
                    1, p = improving_actions[i]/np.sum(improving_actions[i])))

    for i in range(0, num_states):
        print(V_old[i],int(pi_old[i]))
```

Details

- The arrays pertaining to improvable states and actions are initialized by 1 and 0 respectively
- The initial policy π_0 is randomly chosen
- Termination occurs when $IS(s) = \phi \ \forall s \in S$
- Policy evaluation is used for computing the value function of the given policy. It uses PuLP in the background to solve a system of n **linear equations.** (First method shown in class)
- Inside the loop, all states with improvable actions are marked by comparing action values with value functions. The improvable actions are stored
- **For all improvable states**, we switch randomly to a new action using the np.choice function

1.4 Policy Evaluation

Implementation

```
# A modification of policy evaluation which only returns the value functions
def policy_evaluation_no_print(num_states, num_actions, mdp_type, is_mdp_episodic, \
    these_are_episodic, transition_matrix, reward_matrix, \
    gamma, policy):
    #We shall be using pulp
    #We can define a dummy objective function
```

```
    #We have the following 2 constraints for each state:
    #   V(s) >= \sum_s' T*(R + V(s'))
    #   V(s) <= \sum_s' T*(R + V(s'))
    prob = LpProblem("Policy_Evaluation")
    set_S = range(0, num_states)
    val_func = LpVariable.dicts("Value_Function", set_S, cat='Continuous')
    for i in set_S:
        prob += val_func[i] == lpSum([ transition_matrix[i][int(policy[i])][j]*\
                                      (reward_matrix[i][int(policy[i])][j] + \
                                       gamma*val_func[j]) for j in set_S])
    prob.solve(PULP_CBC_CMD(msg=0))
    V_pi = np.array([val_func[i].varValue for i in set_S])
    return V_pi
```

Details

- Define n equations corresponding to the n states of the MDP. Solves them using PuLP to get the value function for a given **deterministic** policy

1.5 Summary of Algorithms

- On given test cases for task 1, HPI and LP were faster than VI. However, they were very slow for the football problem. Hence, I found that on smaller MDPs it was convenient to use HPI and LP
- VI was very slow on test case 4, possibly because the value functions were large in magnitude

2 Task 2: 2v1 Football

Default algorithm used is Value Iteration, due to best performance.

2.1 MDP Formulation

States There are in all 8194 states. 8192 states are non-terminal and correspond to the $16 \times 16 \times 16 \times 2$ string configurations possible as given in the policy files. The last 2 states are terminal. The 8193^{th} state corresponds to termination due to losing the game with a reward 1, while the 8194^{th} state corresponds to scoring a goal and the game ending with a reward equal to 1.

```
    def state_encoder(state_string):
    if state_string[0] == '0':
        B1_state = int(state_string[1])
    elif state_string[0] == '1':
        B1_state = int(state_string[0:2])
    if state_string[2] == '0':
        B2_state = int(state_string[3])
    elif state_string[2] == '1':
        B2_state = int(state_string[2:4])
    if state_string[4] == '0':
        R_state = int(state_string[5])
    elif state_string[4] == '1':
        R_state = int(state_string[4:6])
    poss = int(state_string[-1])
    state_encoding = int((B1_state - 1)*512 + (B2_state - 1)*32 + (R_state - 1)*2 + poss -
    1)
    return state_encoding
```

State encoding is done for convenience as shown in the code above. It will be useful for the actions.

Actions We use the actions as given in the question. For checking if a given action is legitimate (meaning that the players do not go out of bounds), we define left, right, up and down operators which are functions of the string description of a state and a number which tells us which player has moved)

```python
def left_operator(state_string, move_whom): #Move_whom is 1 for B1, 2 for B2 and 3 for R
    B1 = state_string[0:2]
    B2 = state_string[2:4]
    R  = state_string[4:6]
    poss = state_string[-1]
    if move_whom == 1:
        if B1 == '02':
            new_state = '01' + B2 + R + poss
            return new_state
        elif B1 == '03':
            new_state = '02' + B2 + R + poss
            return new_state
        elif B1 == '04':
            new_state = '03' + B2 + R + poss
            return new_state
```

The first few lines of the "left_operator" function have been given. The logic is the same for other 3 movements.

Tackling, Shooting and Passing Appropriate conditions have been implemented in the code to check for all these operations (refer to code)

Transitions The transitions from state s to s' following action a occur with probability

$$\begin{aligned} T(s,a,s') &= \sum_{a_0 \in L,R,U,D} T(s^*,a,s')\mathbb{P}(a_0) \\ &= \sum_{a_0 \in L,R,U,D} T(s^*,a,s')\pi_0 \end{aligned}$$

where we are conditioning on the action that the opponent takes, and thus the policy of the opponent

Rewards If a goal is scored, a reward of 1 is obtained. All other transitions have reward equal to 0.

Discount Factor We assume a discount factor equal to 1, since this is an episodic MDP

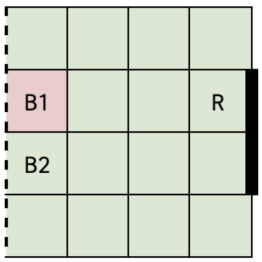


Fig 11: Starting state

Figure 1: Starting state for the MDP as per the evaluation metrics

Computing Expected Number of Goals

- First, we must note that the expected number of goals starting from a given state is nothing but the value function of that state. This is because the reward is 1 only when a goal is scored, thus if we examine the equation

  $$\mathbb{E}[\text{number of goals scored starting at s}] = \mathbb{P}(\text{event that we end in a goal-scoring terminal state})$$

- Further,

$$\mathbb{E}[\text{number of goals scored starting at s}] = \mathbb{E}[(R_0 + R_1 + R_2 + \cdots + R_T | s_0 = s)]$$

  where $R_0, R_1, R_2, \ldots R_T$ are the rewards obtained (goals scored) at timesteps given by the subscripts
- Note that as per our MDP formulation, only scoring a goal gives us a goal. Thus $\forall t < T, R_t = 0$
- Hence, we have

$$\mathbb{E}[\text{number of goals scored starting at s}] = \mathbb{E}[(R_T | s_0 = s)]$$

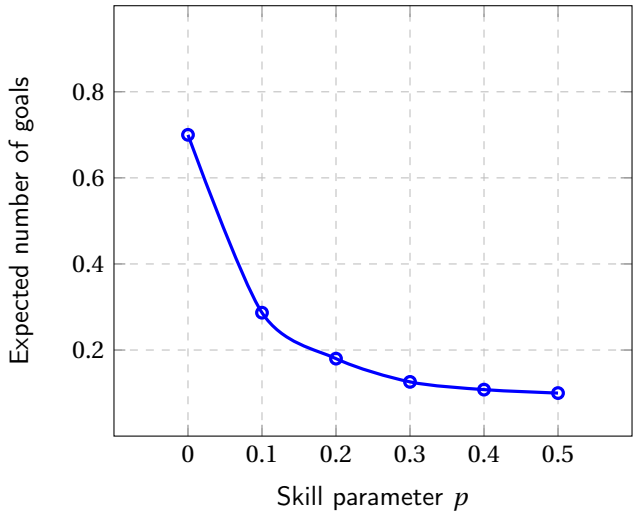
- But note that $\mathbb{E}[(R_0 + R_1 + R_2 + \cdots + R_T | s_0 = s)]$ is nothing but the value function of state s given policy π assuming $\gamma = 1$
- Thus we have

$$\mathbb{E}[\text{number of goals scored starting at s}] = V^{\pi}(s)$$

- Thus, the task reduces to computing $V^{\pi}(s)$

2.2 Graph 1: Variation of expected number of goals with p, keeping q constant against greedy defense (policy 1)

Variation when $q = 0.7$ is constant and opponent follows greedy defense



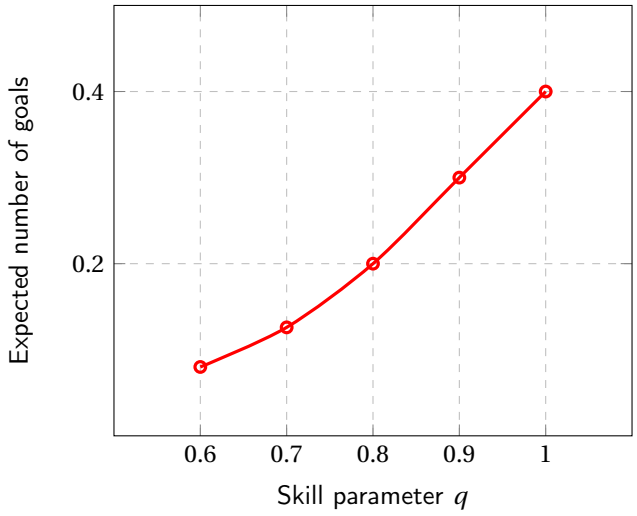
Observations

- From the graph, we can see that $V^{\pi}(s)$ reduces as p increases. This can be seen from the following statement:
  - Probability of losing the game by unsuccessful movement is proportional to p, with the constant of proportionality depending on who has possession (with ball or without ball).
- If $p = 0$, then almost all movements (except the tackles) are successful. This case has a very high value function
- The policy which the opponent is following is greedy defense. Thus, he is trying to move towards the player with the ball. This increases the chances of there being tackles. Hence, parameter p is of great importance as it mainly controls movement.

- The rate of change of reduction in value function gets smaller as we increase p. As p increases, movements start becoming more and more unsucessful so we are more likely to lose possession.
- If we analyse the **optimal** actions being taken at this state, we see the following things
  - The optimal actions are 1, 1, 1, 5, 5, 9 corresponding to the six p values as given.
  - Thus, if p is very low, the player with the ball B_1 moves. The strategy might be to allow the defender to come towards him and make a pass to the adjacent player, since now the defender has been pulled out from the goal, which increases the probability of a successful strike to goal. Also, the player B_2 can sneak into the goal.
  - Also, R is in front of B_1 and there is high possibility of tackle after some iterations of the game. At $p = 0.3$ onwards, the optimal action changes and B_2 starts to move. This is because $2p$ has become very large so the player with the ball moving becomes very risky.
  - As p increases to 0.5, movement of B_1 will always lead to failure and even movement of B_2 will fail with probability 0.5, hence the best choice is to shoot. For this case, most other states also have the optimal action to be shoot

2.3 Graph 2: Variation of expected number of goals with q, keeping p constant against greedy defense (policy 1)

Variation when $p = 0.3$ is constant and opponent follows greedy defense



Observations

- From the graph, we can see that $V^{\pi}(s)$ increases as q increases. This can be seen from the following statements:
  - The probability is a function of the distance from the teammate for passing the ball. If player B1 is on (x1, y1) and B2 is on (x2, y2) then the probability of a successful pass is $q - 0.1 * max(|x1 - x2|, |y1 - y2|)$
  - For shooting towards the goal, the probability is a function of the distance of the x coordinate of the shooter. If B1 has the ball and is on (x1, y1) the probability of a goal is $q - 0.2 * (3 - x1)$.
- Thus q directly affects the chance that a pass or a goal is successful and with increase in q, the success rate increases.
- Again the intuition is that movement is risky in case of greedy defense since the defender is directly moving towards the player who has the ball, and passing is more likely to make the defender switch directions. Hence, success rate of passing is important.

- We can also analyse the **optimal actions** taken for each value of q
  - The optimal actions taken at the state are 1, 5, 9, 9, 9 for the five values of q given
  - This is because there is always a trade-off between movement and passing/shooting. Passing and shooting is safer as compared to movement at larger values of q and hence as q increases the tendency to shoot increases.
  - The rate of increase in expected number of goals increases as q increases. This is because the optimal action gets fixed after a certain threshold of q and thus we can relate the expected number of goals directly to the shooting accuracy. In the initial phase of the curve, since the optimal action is movement, p is the defining parameter.