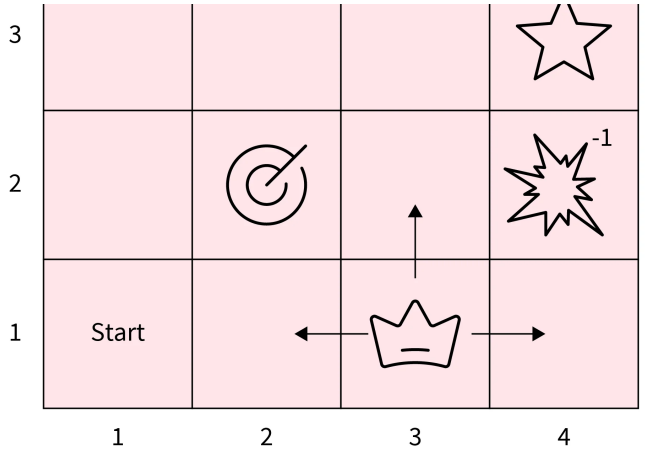


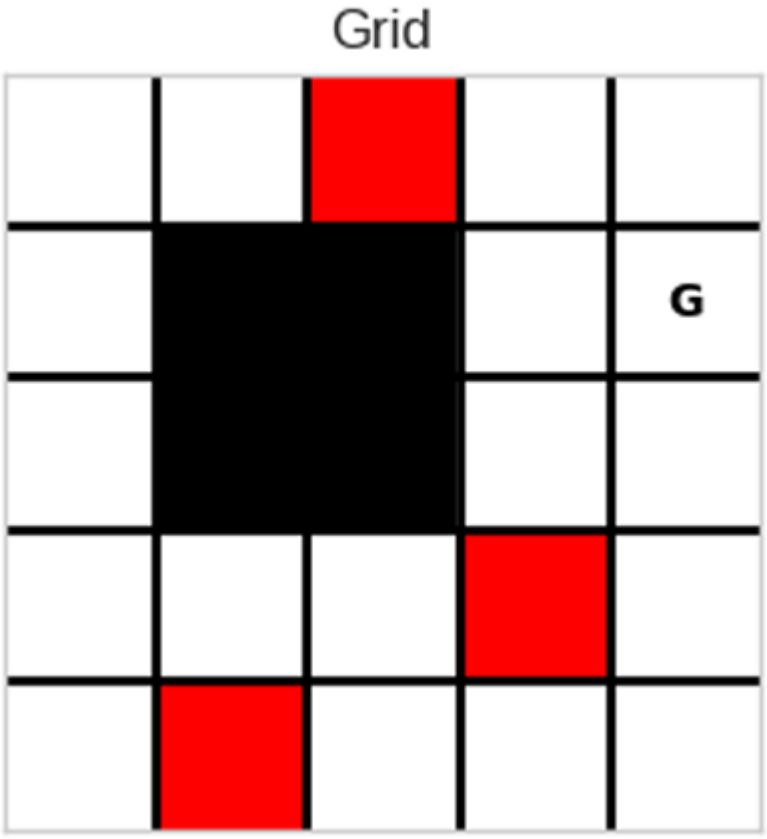
SCALER Topics

Assignment 1 Report: Implementing and Analyzing a Basic RL Algorithm

Solving an MDP Problem with a linear solver and a Dynamic Programming Algorithm

Introduction

We are working on an environment that can be described as a simple MDP for this assignment.



Components of the Markov Decision Process (MDP):

1. ***States (S):*** Each cell in the grid represents a state. The agent can be in any cell except for the blocked cells. Let's denote the states by grid coordinates (i, j).
2. ***Actions (A):*** The agent can move in four directions:
   - North (N)
   - South (S)
   - East (E)
   - West (W)
3. ***Transition Probability (P):*** The agent moves in the chosen direction with a probability of 1−noise. With probability noise, it may move in a different direction. For simplicity, let's assume the noise is evenly distributed among the other three directions.

4. ***Reward Function (R):*** The reward structure is as follows:
    - Reward of 0 for moving into any non-terminal and non-blocked cell.
    - Reward of -5 for moving into a fire cell (red cell).
    - Reward of +5 for moving into the goal cell (G).
5. ***Discount Factor (γ):*** The discount factors to be considered are 0.95 and 0.75.
6. ***Termination:*** The episode ends when the agent reaches the goal cell (G) or a fire cell (red cell).

The goal of this effort:

It's to find a model or agent that can calculate values `grid_values` at each state in the environment then follow these values to reach the goal state and obtain the maximum reward.

This goal can be reached in many different ways but here we are using two methods: a linear solver and a dynamic programming algorithm. Both algorithms will work in the same manner by using the ***Bellman Equation*** to find the states' values.

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

Fig. The Value Function

However, one difference to note here, is that we will include an additional factor when trying the dynamic programming approach. The factor is the Noise. This means that when the agent decides to take an action `a`, it might end in the expected state with a probability equals to `1-noise`. It also might not reach that state with a probability equals to `noise`.

Implementation

1. Simple Grid World

The simple grid world implementation (using the provided snippet) typically represents the cells as a `5x5` matrix with zeros as the initial state values. And stating the values of the special cases to be `danger=-0.5`, `goal=5`, and `blocked=0`.

Initializing a 5×5 grid of zeroes

```
        self._grid_values = [0 for _ in range(height * width)
```

Getting a list of all valid states in the environment aka. not a blocked cell, and inside the grid boundaries:

```
    def get_states(self):
        """
        Gets all non-terminal states in the environment
        """
        return [s for s in range(self._height * self._width)\
             if s not in self._blocked_cells]
```

Simple Grid World Result

Following is the grid output in the console in its initial state:

```
0.00    0.00   -5.00    0.00    0.00
0.00    ----    ----    0.00    0.00
0.00    ----    ----    0.00    GOAL
0.00    0.00    0.00   -5.00    0.00
0.00   -5.00    0.00    0.00    0.00
```

2. Linear Solver

A linear solver is a system that applies a linear function to find a function value based on an input value. The system at hand can be solved using the formula

`Ax=b`. We are assuming that the *x* in the formula refers to the state value we are looking for. So, what happens here is that we calculate the values of *A* and *b* at each state, form a matrix with all available states and plug all these values to the equation. Lastly, we use the linear algebra solver from Numpy module `np.linalg.solve` to find the values of *x*. And x here is the matrix saving all the states.

Following is the Python code implementation for the solver function:

```
    def solve_linear_system(self, discount_factor=1.0):
        """
        Solve the gridworld using a system of linear equation
        :param discount_factor: The discount factor for futur
        """
        # To solve the linear system of equations
        #   find the values of metrices A and B

        # Initialize metrices
        num_states = self._width * self._height
        A = np.zeros((num_states, num_states))
        b = np.zeros(num_states)

        # Assign values to A and B
        for state in range(num_states):
            # When the state is at an obstacle or terminal do
            #   an s(v) it's always the same as reward
            if state in self._blocked_cells or self.is_termin
                A[state, state] = 1.0
                b[state] = self.get_reward(state)
            else:
              # If it's not terminal or obstacle, calculate i
              A[state, state] = 1.0
              for action in self.get_actions(state):
                  # v(s) = P(s'|s,a)(r+)
                  n_actions = len(self.get_actions(state))
                  next_state = self._state_from_action(state,
                  reward = self.get_reward(next_state)
                  A[state, next_state] -= discount_factor / n
                  b[state] += reward / n_actions
```

```
        # Solve the equation
        V = np.round(np.linalg.solve(A, b), 2)

        for state in range(num_states):
            self.set_value(state, V[state])

        return V.reshape((self._height, self._width))
```

Linear Solver Results

Given the state values found by the linear solver and an agent starting at state 1, we can note that this method doesn't lead to a smart agent. If the agent chooses its actions according to the policy found here, it won't reach the state goal. We can conclude that directly solving the equation (Bellman equation) based on the initial values given (set of zeroes for non-terminal states) is not the best solution for our current problem. We can call this the dummy solver.

```
[[-9.97 -9.99 -5.   -3.33  0.  ]
 [-9.96  0.    0.    0.    3.33]
 [-9.94  0.    0.    0.    5.  ]
 [-9.93 -9.88 -9.71 -5.   -1.6 ]
 [-9.96 -5.   -9.24 -8.02 -4.81]]
```

3. Value Iterative Algorithm (DP)

The second method to solve this problem is the **Value Iteration Algorithm** which employs a dynamic programming technique. This time instead of just finding a state value and saving it, we will iterate multiple times over all the states until reaching stability within the value. The stability step is called convergence. It's calculated by finding that the value of the change between the states in every two consecutive iterations is less than a predefined

threshold *delta*. Given that inside each iteration again the Bellman equation is used to find state values.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Fig. Pseudo Algorithm (Value Iteration)

A new factor is added for this implementation that we didn't consider previously which is the *noise* value. The environment in this case is stochastic and has a probability equal to the noise that the current model won't reach the planning state after each decision. Unlike the deterministic case, here the agent can decide to take an action that should lead to a specific state but doesn't end up as expected. Supposing we have a noise equal to 0.2 and we are at state 1. If the agent decides to take the action right to reach state 2, there is a probability of %20 that it will not be at state 2 after moving to the right.

Following is the Python implementation for the value iteration algorithm:

```
def value_iteration(gw, discount, tolerance=0.1):
  """Solve the gridworld using a dynamic programming method (
    gw.reset()
    loops = 0
    while True:
        loops += 1
        delta = 0
        gw.create_next_values()
        for state in gw.get_states():
            if gw.is_terminal(state):
```

```
                continue
            action_values = []
            for action in gw.get_actions(state):
                action_value = 0
                for transition in gw.get_transitions(state, a
                    next_state = transition['state']
                    prob = transition['prob']
                    reward = gw.get_reward(next_state)
                    action_value += prob * (reward + discount
                action_values.append(action_value)
            best_value = max(action_values)
            delta = max(delta, abs(best_value - gw.get_value(
            gw.set_value(state, best_value)
        gw.set_next_values()
        if delta < tolerance:
            break
    return loops
```

The code implementation for the calculation of the available transition:

```
def get_transitions(self, state, action):
        """
        Get a list of transitions as a result of attempting t
        current state.
        Each item in the list is a dictionary, containing the
        reaching that state and the state itself.
        """
        # When the state is terminal then only one probable a
        if self.is_terminal(state):
            return [{'prob': 1.0, 'state': state}]
        transitions = []
        primary_state = self._state_from_action(state, action
        # Probability that the agent wants to move to state s
        #     actually moved there
        transitions.append({'prob': 1 - self._noise, 'state':
        if self._noise > 0:
            # from the actions list find valid states that ar
            other_actions = [a for a in self.get_actions(stat
```

```
            noise_prob = self._noise / len(other_actions)
            for other_action in other_actions:
                secondary_state = self._state_from_action(sta
                # Given the noise: the probability that the a
                #     to the direction in the decision
                transitions.append({'prob': noise_prob, 'stat
        return transitions
```

Value Iterative Algorithm Results

With a noise value equal to zero, the model could find the following state values after 12 trials

3.15	2.99	-5.00	4.51	4.75
3.32	----	----	4.75	5.00
3.49	----	----	5.00	GOAL
3.68	3.87	4.07	-5.00	5.00
3.49	-5.00	4.29	4.51	4.75

Result Analysis

The question to answer based on the reached results is the following: “Given that the model starts at state 1 as its initial state, did it reach a policy which will lead to the goal state?” The is the question we will try to answer through many trials and hyperparameters tweaking.

a. Linear Solver:

   i. Solving the linear system leaded the search to find the actual values of each state.

   ii. Values reached by this solver seemed to improve by changes in the discount factor (gamma).

   iii. Not in any of the trials did the solver reach a policy that will lead the agent to the goal sate. Given that the initial state was state 1.

iv. For the gamma = 0.75, the solver reached values closer to zero around the goal state and more negatives faraway emphasizing on its importance. Check the following figure.

```
[[-1.37 -3.91 -5.    -2.69  0.  ]
 [-0.66  0.    0.     0.    2.69]
 [-0.84  0.    0.     0.    5.  ]
 [-2.14 -4.29 -4.78 -5.    -0.3 ]
 [-4.14 -5.   -4.74 -4.09 -1.32]]
```

b. Value Iteration:

The hyperparameters in this algorithm are the values of noise, discount factor (gamma) and the tolerance.

In the figure we can see the effect on the number of iterations using different values for all the hyperparameters.

i. The deterministic environment or noise equal to zero the number of iterations is fixated to 12 no matter the change in other values and resulted in a policy that will lead to state goal at any starting point. This behavior is understandable because when you are sure about the next action's value and result you always take the same action.

ii. The stochastic environment or noise equals to 0.2, the effect of tolerance value and the discount factor seemed clearer. However, the agent won't always reach the goal state, this will only happen in some cases and might vary based on different starting states.

iii. For discount factor equals 1 in a stochastic version, the number of iterations seemed greater than 0.95 and 0.75 unlike the expected. %100 sure that the next reward is granted should have resulted in faster convergence which didn't happen.

#Trial	Noise	Discount	Tolerance	Iterations
1	0	1	0.1	12
2	0	0.95	0.1	12
3	0	0.75	0.1	12
4	0.2	1	0.1	18
5	0.2	0.95	0.1	15
6	0.2	0.75	0.1	7
7	0.2	0.75	0.001	13

Reference

https://www.scaler.com/topics/artificial-intelligence-tutorial/markov-decision-process/

https://github.com/clumsyhandyman/mad-from-scratch

https://medium.com/@ngao7/markov-decision-process-basics-3da5144d3348

https://medium.com/@ngao7/markov-decision-process-value-iteration-2d161d50a6ff

https://www.analyticsvidhya.com/blog/2018/09/reinforcement-learning-model-based-planning-dynamic-programming/