Programming Assignment 1

Tuesday, 14 February 2023 1:57 AM

Frozen-Lake with value iteration

(1) Approach

For solving the frozen-lake problem, we are using the value iteration algorithm.

In our case, we are using the 4x4 grid of frozen-lake.

For value iteration, we can write the bellman equations as following.

$$q(s,a) = \text{Immediate reward} + \text{Expected future reward}$$

$$= r(s,a) + E[\gamma v'(s)]$$

$$= r(S,a) + \gamma \sum p(s'/s,a)\ v'(s)$$

Using this, we can write

$$v(s) = r(s,a) + \max_a \left[\gamma \sum p(s'|s,a)\ v'(s)\right]$$

Here we are computing the optimal state value function by iteratively updating the estimate $v(s)$

Hence, we can define the value iteration algorithm as follows.

for a small threshold $\theta > 0$
Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in S$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum p(s', r/s,a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \operatorname{argmax}_a \sum (p', r | s, a)\ [r + \gamma V(s')]$$

(2) Results

Choosing a threshold value of 1×10^{-10},
we can see that the value iteration is converging at iteration 877.

Using this optimal value function, we find the optimal policy. which is,

[0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]

where each of the number defines the optimal action at each state.

Actions are defined as
0 → left   2 → Right
1 → down   3 → Up

On computing the average reward for 1000 iterations, we can see that the reward is in range 0.844, which is quite close to the optimal reward of 1.

Hence, after value iteration, we can say that it is very likely to take the next best step

(3) Discussions

While playing around with different hyperparameters, these were some interesting results.

tried different thresholds (θ) of 1×10^{-10}, 10^{-15} & 10^{-20}.

The model converged faster with higher θ value i.e. at 877, 1333 & 1373 iterations respectively.

Although the model converged at different iterations, the optimal policy was the same.

This could be due to as the current problem is relatively simple. But for more complex tasks, it becomes imp to choose the right threshold.

Similarly, while experimenting with different maximum iterations, if we have lower number, the model may not converge fully. So it is best to have higher number of maximum iterations and break the loop when the change is less than the defined threshold.

Another hyperparameter is gamma

For gamma = 1 & threshold = 10^{-10}

the average reward is around $\approx$ 0.82

But changing the
gamma = 0.5 & threshold = 10^{-10}
the average reward dips to $\approx$ 0.45

& for gamma = 0.9 & threshold = 10^{-10}
the average reward $\approx$ 0.77.

Hence, it becomes important to choose the best possible combination of hyperparameter gamma as well.

Lastly, we solved the question using value iteration, but could have done the same with a different approach of policy iteration.

In [23]:
```
import gym
import numpy as np
```

In [45]:
```
# make the frozen lake environment using OpenAI's Gym
env = gym.make("FrozenLake-v1") # or the latest version
```

In [85]:
```
# explore the environment
observation_space = env.observation_space.n
action_space = env.action_space.n

print(observation_space)
print(action_space)
```

```
16
4
```

In [204…
```
def value_iteration(env, gamma = 1.0):
    """
    Inputs:
        - env: the frozen lake environment.
        - gamma: discount factor

    Returns:
        - value_table: state value function
        - Q_value: state-action value function (Q function).
    """

    observation_space = env.observation_space.n
    value_table = np.zeros(observation_space)

    threshold = 1e-10

    for i in range(10000):

        prev_value_table = np.copy(value_table)

        for state in range(observation_space):
            q_value = []
            for action in range(action_space):
                next_states_rewards = []
                for next_state_reward in env.P[state][action]:
                    transition_probability, next_state, reward, done = next_sta
                    next_states_rewards.append((transition_probability * (rewa

                q_value.append(np.sum(next_states_rewards))

            value_table[state] = max(q_value)

        # check for convergence
        if (np.sum(np.fabs(prev_value_table - value_table)) <= threshold):
             print ('Value-iteration converged at iteration# %d.' %(i+1))
             break

    return value_table, q_value
```

In [205…
```
def extract_policy(value_table, gamma = 1.0):
    """
    Inputs:
        - value_table: state value function
        - gamma: discount factor

    Returns:
        - policy: the optimal policy.
    """
    policy = np.zeros(observation_space)

    for state in range(observation_space):
        q_table = np.zeros(action_space)

        for action in range(action_space):
            for next_state_reward in env.P[state][action]:
                transition_probability, next_state, reward, done = next_state_
                q_table[action] += (transition_probability * (reward + gamma *

        # getting argmax
        policy[state] = np.argmax(q_table)

    return policy
```

In [230…
```
optimal_value_function, q_value = value_iteration(env=env, gamma=1.0)
```

```
Value-iteration converged at iteration# 877.
```

In [231…
```
optimal_policy = extract_policy(optimal_value_function, gamma=1.0)
```

In [232…
```
print(optimal_policy)
```

```
[0. 3. 3. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]
```

In [235…
```
all_rewards=[]
for _ in range(1000):
    obs=env.reset()[0]
    total_reward = 0
    while True:
        action = optimal_policy[obs]
        obs,reward,done,info,_ = env.step(action)
        if done:
            all_rewards.append(reward)
            break

print("Average Reward: ", np.mean(all_rewards))
```

```
Average Reward:  0.838
```

In [ ]: