AI: Principles & Techniques
Machine Programming Task 4; Machine Learning

Jordy Aaldering, s1004292
Elianne Heuer, s4320514

8 January 2021

1 Introduction

One of the challenges of creating good AI is making them able to make the most ideal choices in the situations they are placed in. They do this by using a method called reinforcement learning. Every time the AI makes a good choice from it's current state it gets a reward (a reinforcement signal). This problem is known as a Markov Decision Process (MDP). From this feedback system the AI can learn the ideal behaviour.

In our case we have an agent in a 2D world. The agent has to take steps (up, down, left, or right) to get to a reward, after which it is done walking. This reward can either be negative or positive. We want the agent to reach the positive reward and stay clear of the negative reward. Of course, we also want the agent to reach the end goal in the most efficient way possible. There are two ways we will achieve this: by using a value iteration algorithm and a Q-learning algorithm.

We will begin by taking a look at the value iteration algorithm. With this algorithm the agent knows two things beforehand. Firstly, it knows the probability of ending up in new states when taking certain actions. So in our case that means that the agent, for example, knows the probability of ending up in one square up when it says it will move up and it also knows the probability it will actually end up one square to the left, right, down or at the same place. This is called the transition function. Secondly, it knows the immediate reward of taking one step in a certain direction. This is called the reward function. Based on these two functions the agent calculates the most optimal route it can take to get a positive reward. Furthermore, a discount factor (gamma) can be used in calculating the most optimal route. This discount factor represents how quickly the agent wants to find a reward. If gamma is small, the agent will look mostly at the most immediate rewards and base its choices on this. If gamma is big, the agent will look more at the more distant rewards and base its choices on this.

Finally, we take a look at the Q-Learning algorithm. In this algorithm the agent does not know the transition function nor does it know the reward function. The agent chooses its route by trial and error. It creates a Q-table (Q[s,a]; s = state, a = action) to keep track of the best choices (Q-values) in each time step. The agent will choose the best action at every step, taking a random best action if multiple actions are equally beneficial. After a set number of steps, an iteration will stop. These iterations are repeated many times to optimize the Q-table.

2 Methods

2.1 Value-Iteration

In our implementation of the value-iteration algorithm we use a grid class containing a 1d array representing the grid as a list of Fields. A Field is another class that can either represent a wall, a negative reward, a positive reward or an empty space.

After initialising the grid, we call upon a function called iterate (see the code snippet below). This function iterates through all states with their four possible actions while taking in account their randomness. It keeps doing this as long as their Q-values keep changing more than a certain threshold (delta). In the final for-loop it decides what the best action is for each state.

```
def iterate(self):
    converged = False
    while not converged:
        max_delta = 0
        for state in self.state_space:
            if self.grid.is_end_state(state):
                self.V[state] = 0
                continue

            old_v = self.V[state]
            new_vs = [0, 0, 0, 0]

            for i, action in enumerate(Action.as_list()):
                new_state, reward = self.Q.get((state, action))
                new_vs[i] += self.grid.p_perform *
                    (reward + self.grid.gamma * self.V[new_state])

                new_state, reward = self.Q.get((state, action.next_action()))
                new_vs[i] += self.grid.p_sidestep / 2 *
                    (reward + self.grid.gamma * self.V[new_state])

                new_state, reward = self.Q.get((state, action.prev_action()))
                new_vs[i] += self.grid.p_sidestep / 2 *
                    (reward + self.grid.gamma * self.V[new_state])

                new_state, reward = self.Q.get((state, action.back_action()))
                new_vs[i] += self.grid.p_backstep *
                    (reward + self.grid.gamma * self.V[new_state])

            self.V[state] = max(new_vs)
            max_delta = max(max_delta, np.abs(old_v - self.V[state]))
            converged = max_delta < self.delta

    for state in self.state_space:
        new_vs = [0, 0, 0, 0]
        for i, action in enumerate(Action.as_list()):
            new_state, reward = self.Q.get((state, action))
            new_vs[i] = reward + self.grid.gamma * self.V[new_state]

        i = new_vs.index(max(new_vs))
        self.policy[state] = Action(i)
```

2.2 Q-Learning

The implementation of Q-Learning also starts of with initialising the grid and then calling another iterate function (see the code snippet below). This function repeats the process of walking the bot through the grid many times.

Walking the bot is done within a set number of maximum steps, though usually the bot will reach an end state before this happens, breaking from the loop early. For each step the bot first chooses which action it wants to take. This is done by looking at the currently known rewards for each action, which will initially be 0 for every action. If multiple actions are equally beneficial it will choose a random action of that list. Then the bot tries that action, possibly ending up in a different state then expected, depending on the randomness of the state transitions. Using the reward it got from this action the Q-table will be updated.

```
def learn(self, old_state, action, new_state, reward):
    old_q_val = self.q_table[old_state][action]
    new_q_vals = self.q_table[new_state]
    max_q_val = max(new_q_vals.values())

    self.q_table[old_state][action] = ((1 - self.alpha) * old_q_val
        + self.alpha * (reward + self.grid.gamma * max_q_val))

def iterate(self, trials=500, max_steps=1000):
    for _trial in range(trials):
        self.grid.reset_pos()

        for _step in range(max_steps):
            old_state = self.grid.pos
            action = self.choose_action()
            reward = self.grid.try_step(action)
            new_state = self.grid.pos

            self.learn(old_state, action, new_state, reward)

            if self.grid.is_end_state(new_state):
                break
```

3 Results

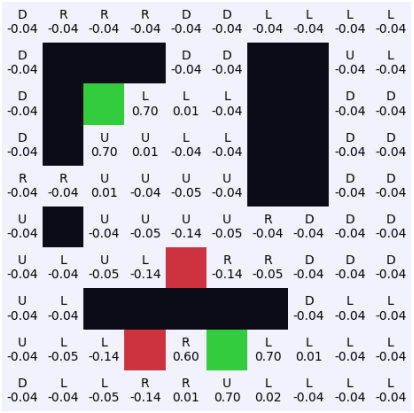
We will now look at the effects of changing different variables. Firstly, we will look at the effect of differently sized worlds. This does not change the results much, but as expected it does spread out the V-values more evenly. Obviously larger worlds would also take longer to compute.

Changing the discount factor (gamma) is more interesting. Increasing the gamma produces more risky plays, while decreasing the gamma produces slower, but safer decisions. For instance, with a high gamma the policy might prefer walking right past a negative space, while using a low gamma would result in a policy that avoids this negative space more. These results can be seen in figures 1a and 1b.

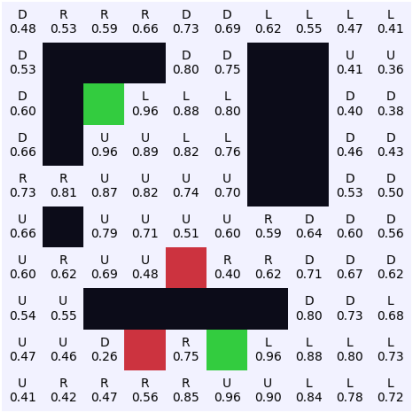
Changing the state penalty (no_reward) to 0 makes all V-values equal, or very close, to 0. This also means that the policy can avoid negative spaces much more as there is no negative reward for taking longer to reach the goal. Increasing the state penalty means that the policy much more prefers reaching an end state quickly. In the case where the state penalty is very high it might even prefer moving into a negative reward state to avoid having to walk too much, reducing the reward by even more.

Increasing the randomness of transitions (p_perform, p_sidestep, and p_backstep) makes for more safe choices. For instance, when the positive reward is up but there is a negative reward to the right, the policy would prefer moving left to reduce the chance of randomly moving into the negative reward state.

Our implementation of Q-Learning produces very similar results to our value iteration. Though in the policy produced by Q-Learning some more randomness can be seen, like a sudden move in a different direction. This is likely due to the randomness in the state transitions.



(a) Results with a gamma of 0.1



(b) Results with a gamma of 1.0

4 Discussion

Our implementation of Value Iteration produces good results, is efficient, and uses our own version of the MDP. We verified this by trying different grid sizes and testing multiple values for our variables.

In our examples the value iteration program seems to produce better results compared to the Q-Learning program, since the Q-Learning program has more randomness in its results. We think Q-Learning might work better for bigger problems since it is more efficient.

Our suggestions for further work would be to compare the efficiency and results of using either value iteration or Q-Learning for bigger problems. Furthermore, it would be interesting to see these algorithms applied to more different problems.