Artificial Intelligence's Pacman Project: Reinforcement Learning

Deborah Dore

December 2019

1 Introduction to Reinforcement Learning

Reinforcement Learning concerns about how software agents acts and take actions in the environment in order to maximise notions of cumulative reward.
The environment is typically started in the form of a **Markov Decision Process**. This is constituted by 4 elements:

1. A set of environment and agent states, *S*.
2. A set of actions, *A*, of the agent (sometimes restrict).
3. $P_a(s,s')$ is the probability if transition from a state S to a state S'under action a.
4. $R_a(s,s')$ is the reward for the transition from S to S' with action a.
5. Rules that describe what the agent observes.

Rules are often stochastic which means that are randomly decided. The agent is assumed to observe the current environmental state in a way called full observability. If not, the agent has partial observability.

There are three mechanisms that operate: *A, B, C*

1. *A* choses the output based on the inputs mechanisms.
2. *B* evaluates the effective of the output based on a precise parameter.
3. *C* changes *A* based on *B* evaluation in order to maximise it.

This three mechanisms collaborates:

1. if *A* takes a choice, *B* sends a reward based on *A*'s efficiency or a penalty based on its inefficiency.
2. *C* tries to modify *A*'s mathematical function based on *B*'s decisions.

Reinforcement learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including robot control, elevator scheduling, telecommunications, backgammon, checkers and Go (AlphaGo).

Policy. A policy is a stochastic rule that gives the action to take from a given state. The agent acts according to its policy. A common way to characterise a policy is by its **value function** that quantifies how good it is to be in a state.

$$V^{\pi}(s) = R(s, \pi(s)) + E[\sum_{t=1}^{\infty} \gamma^t R(s_{\mathrm{t}}, \pi(s_{\mathrm{t}}))] \tag{1}$$

State-Value Function. The expectation operator averages over the stochastic transition model, which leads to the following recursion:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi(s)) V^{\pi}(s') \tag{2}$$

Extracting a policy π from a value function V is done with:

$$\pi(s) = argmax_{a \in A}[R(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) V(s')] \tag{3}$$

Solving this system of equation for each state s yields the optimal value function, and an optimal policy π.

2 Value Iteration

The first question of the project requires to write an algorithm for the Value Iteration. This is a method of computing an optimal MDP policy and its value.

Inputs:

1. P is the state transition function specifying P(s' | a,s).
2. s is a reward function R(s, a, s').
3. 0 is a threshold greater than zero.

Returns:

1. π [s] is approximately an optimal policy.
2. V[s] is a value function.

Data Structures:

1. V_k [s] is a sequence of value function

Pseudo-algorithm:

begin
  for $k = 1 : \infty$
    for each state s
      $V_k[s] = \max_a \sum_{s'} P(s'|a, s)(R(s, a, s') + \gamma V_{k-1}[s'])$
    if $\forall s \, |V_k(s) - V_{k-1}(s)| < \theta$
      for each state s
        $\pi(s) = \arg\max_a \sum_{s'} P(s'|a, s)(R(s, a, s') + \gamma V_{k-1}[s'])$
      return π, V_k
end

As explained, this algorithm computes the value function for each state and the best policy given all states.

In the Pacman Project, we have many function available.
First of all, a set of all states is given by the "self.mdp.getStates()" and a set of all actions that the agent can choose with a given state is computed with "self.mdp.getPossibleActions(state)".
The state transition function based on a state and the action taken is given by "self.mdp.getTransitionStatesAndProbs(state, action)" and the reward for it is computed by "self.mdp.getReward(state, action, nextState)". At the beginning of the code, the discount variable is initialised with the value of 0.9. We can also utilise a dict with default 0.
The algorithm starts with an iteration, like the pseudo-algorithm, but, in our case, it only arrives to 100 iteration as initialised in the first part.
For every state that is not terminal, a value function is computed. This part is calculated by the "compute Q Value from values" method. At the beginning, it was difficult to understand how to calculate the value function. However in the end, the value function resulted to be the summation of every probability * (reward + discount + the value of the next states) as the pseudo-code suggested.

The last method to be implemented was the "compute action from values". It was required to calculate as a policy the best action in a given state according to the values stored in self.values. For every possible action, the Q value is calculated and if that was better than the current best Q value, assigned to the best value and the action corresponding to the best action.

3 Bridge Crossing Analysis

BridgeGrid is a grid world map with the a low-reward terminal state and high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. The discount factor determines the importance of future rewards. A factor of 0 will make the agent short-sighted, meaning that it will only considering current rewards. While a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the action values may diverge. Adding noise, on the other part, can lead to more unpredictable exploration. Changing only one parameter was allowed:

Changing the noise in a range that goes from 0.1 to 0.9 does not make the agent cross the bridge. Instead, it makes it go back to the low-reward. At noise 1, the agents jumps off the bridge and with noise 2, it makes

it get stuck. With the noise equal to 0 and the discount equal to 0.9, the agent succeeds and cross the bridge.

Changing the discount in a range that goes from 0.1 to infinity makes the agent go back to the low-reward. At the value of 0, it makes it jump off the bridge.

The only configuration possible seems the one with a discount equal to 0.9 and a noise equal to 0.

4 Policies

The third part of the project was similar to the second with an addition: the living reward. The DiscountGrid layout has two terminal states with positive payoff, a close exit with payoff +1 and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff; each state in this "cliff" region has payoff -10. We distinguish between two types of paths: paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. Paths that "avoid the cliff" and travel along the top edge of the grid.

1. First question was to find the correct values for noise, discount and living reward in order for the agent to prefer the close exit risking the cliff. Using a discount and a noise of of 0.2 and a living reward of -7 the agent prefers the nearest terminal state risking the cliff.

2. Second question was to find the correct values for noise, discount and living reward in order for the agent to prefer the close exit avoiding the cliff. I've noticed that changing the living reward from an higher value to a lower value would make the agent avoid the cliff. Choosing -1 as a value for the living reward would make the agent also choose close exit.

3. Third question was to find the correct values for noise, discount and living reward in order for the agent to prefer the distant exit risking the cliff. The noise in this case was 0, living reward -1 and discount 0.9. In this case, I've noticed that a discount in a range between from 0 to 0.5 makes the agent choose the closest exit and the other values makes it choose the distant exit.

4. Fourth question was to find the correct values for noise, discount and living reward in order for the agent to prefer the distant exit avoiding the cliff. The strategy was only to lower the living reward to 0.2 and adding a + 0.1 to the noise.

5. Last question was to find the correct values for noise, discount and living reward in order for the agent to avoid both the exit and the cliff. For this question is was sufficient to choose as a noise , discount and living reward, 0.

5 Q-Learning

Last part of the project was about Q-Learning. An agent ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. It simply follows the precomputed policy. In this part of the project, the choice of the action was left to the user. The objective was to observe how the agent learn through experience.
First of all, in the "init" section, I had to initialise the qValue which is a dict with default 0. Four methods were implemented:

1. In the computeValueFromQValues's method it was required to return the max action of Q(state, value) where the max was over legal action. If the state was terminal, a value of 0.0 was required to be returned. In this case, a state is terminal if there aren't no legal action to take. Then, for every legal action the q value was computed and the max between them was returned.

2. In the computeActionFromQValues's method, a similar approach was used for choosing the best action. The q value between all legal action was computed but, in this case, the action corresponding to the best value was returned.

3. In the getAction's method, it was required to choose an action for the state. With the probability self.epsilon, it was required to take a random action and the best policy action otherwise using the "util.flipCoin()" method as "self.epsilon" as parameter.

4. In the update's method, the only thing was to recalculate the Q value for a given state and action with the new values.