Applying Reinforcement learning algorithms to solve Gridworld problems.

Hardik Prabhu

April 2021

1 Introduction

In a gridworld problem, an agent is placed on a MXN rectangular array. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: UP, Down, Left, and Right. The objective of the agent is to learn a policy with which it stochastically or deterministically chooses an action in a particular state. The enviroment responds by taking the agent to the next state and giving the agent an immediate reward. For example, if the agent is situated at (i, j), ith row, jth column. By choosing the action "Left", the enviroment may respond by taking the the agent to the $(i, j-1)$ cell. Our objective is to train the agent to travel from an arbitrary state to the terminal state such that it learns to maximize the long term gains. In order to implement the reinforment learning technqiues, the agent-enviroment interactions are set up as a Markov decision process.

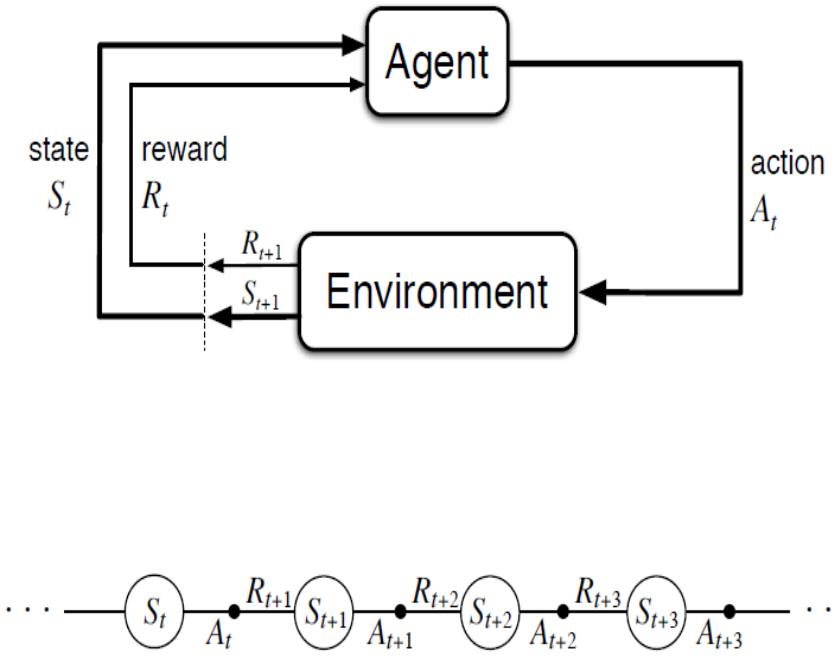

Figure 1: The agent environment interaction in reinforcement learning.

At each time step t, the agent is at one of the environment's state, $S_t \in S$, where S is the set of possible states, and on that basis selects an action, $A_t \in A$, where A is the set of actions. One time step later, as a consequence of its action, the agent receives a reward, R_{t+1}, and is taken in a new state, S_{t+1}. The probability of reaching a new state

S_{t+1} and getting a reward R_{t+1} is independent of the past transitions provided the state S_t and action A_t is known (Markov property).

2 Problem Statement

Consider a $6X6$ grid with states (i, j), $1 \leq i; j \leq 6$. Suppose states $(4, 3)$ and $(5, 3)$ are removed (you can think of these as holes). At each square you are allowed the following actions attempt left, attempt right, attempt up and attempt down. When you attempt an action, you can succeed with probability 0.8. With probability 0.1 you stay where you are. With probability 0.05 you move in a direction $+90°$ (clockwise) to the direction attempted, and $-90°$ (anti-clockwise) to the direction attempted. If ever an attempted move takes you out of the grid or takes you into a hole, you remain where you were. Also there is restricted cell $(6, 3)$ which the agent should learn to avoid. $(6, 6)$ is a terminal state and once you visit that state the episode ends.

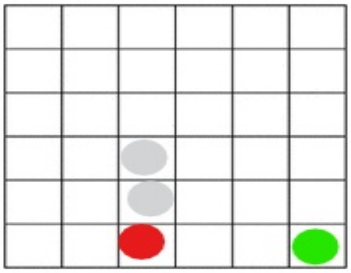

Figure 2: The grid world setup.The green dot reperesnts the terminal state. The red represents the state that is to be avoided and grey dots are the holes.

2.1 Setting up the rewards

If the agent's actions result in visiting $(6, 3)$ it get a reward of -15 and if the action results in visiting $(6, 6)$ it get reward $+15$. Rest of the remaining transitions return zero rewards.

2.2 Dynamic programming approach

In dynamic programming, we assume an infinitely long episode with the terminal state being an absorbing state. Once reached, no matter what actions are chosen the agent remains in terminal state with no subsequent rewards.

Let us first look at the problem of prediction. Let us start with an uniform policy, which gives equal probability $(p = 1/4)$ for choosing an action at each state. The total discounted reward from time step t is given as,

$$G_t = R_{t+1} + \gamma G_{t+1}$$

And the value of a state s for a given policy π is defined as

$$V_\pi(s) = E[G_t|S_t = s]$$

.

Every finite MDP satisfies the bellmen equation, which is given below.

$$V_\pi(s) = E_\pi[R_t + \gamma V_\pi(S_{t+1})|S_t = s]$$

which is equivalent to.

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{r \in R, s' \in S} [r + \gamma V_\pi(s')]p(r, s'|s, a)$$

The value $p(r, s'|s, a)$ is the transition probability. It is the probability that after taking $A_t = a$ at $S_t = s$ the agent arrives at state, $S_{t+1} = s'$ and receives a reward, $R_{t+1} = r$. When the transition probabilities are not known then we have to rely on the monte carlo methods. We will see the monte carlo and other simulation based algorithms later.

Iterative Policy evaluation

The initial approximation, v_0, is chosen arbitrarily (except for the terminal state, all terminal state values are initialised to 0). The update rule for value of a state follows from the bellmen equation itself.

$$V_{k+1}(s) = \sum_{a \in A} \pi(a|s) \sum_{r \in R, s' \in S} [r + \gamma V_k(s')] p(r, s'|s, a)$$

$v_k = v_\pi$ is a fixed point for this update rule because of the Bellman equation for v_π. The sequence $\{v_k\}$ converges to v_π for all s as k approaches ∞.

We implement the sweeping version of the algorithm where we maintain a dictionary of all the state value functions initialized arbitrarily. We then at each iteration update the state value of the particular state by using the current values stored in the dictionary for all the states. We apply the update rule given above.

Plot for convergence

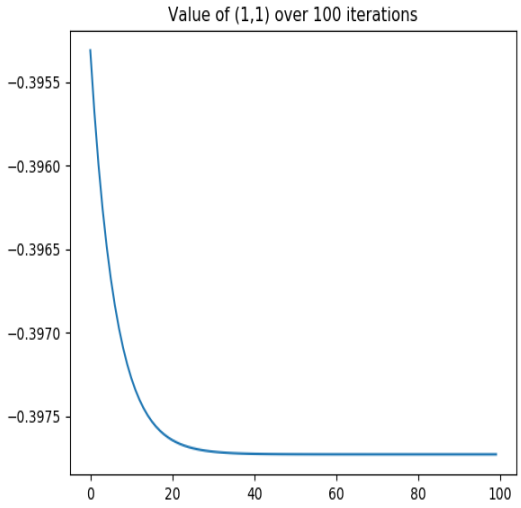

Figure 3: State value of the cell (1,1) over 100 iterations. The value converges to -0.4.

In order to also get the optimal policy we implement the policy improvement algorithm.

Policy improvement

In the similar fashion, state-action values, $Q_\pi(s, a)$ is defined as expected total discounted returns from selecting action a at state s.

A policy is optimum if and only if, for all the states, the actions which gives the maximum state-action value, are the only ones assigned a non-zero probability.

For an optimum policy:

$$\pi(a|s) > 0 \Rightarrow a \in \textit{argmax}\, Q_\pi(s, a)$$

Policy improvement theorem suggest that we can improve upon a given policy(if not optimal) by transferring the non-zero probability of a non optimal actions to any of the optimal actions. It provides us with an iterative algorithm to deterministic choose a better action for all the states at a given iteration. We then use policy evaluation to update

the value function and repeat the process, till we converge to an optimal policy.
This algorithm requires us to find a way to compute $Q_\pi(s, a)$ values for all state-action pairs. We know that

$$Q_\pi(s, a) = E[G_t|S_t = s, A_t = a]$$

Upon expansion, we get

$$Q_\pi(s, a) = \sum_{r\in R, s'\in S} [r + \gamma V_\pi(s')]p(r, s'|s, a)$$

So, we can extract the state-action function from the state functions, if we know the state functions and the transition probabilities. Note that, on the other hand, in order to get the state values it is sufficient to only know the state-action value of all the states as the below expressing would suggest.

$$V_\pi(s) = \sum_{a\in A} \pi(a|s)Q_\pi(s, a)$$

Now that we can compute all that is required for the algorithm, lets implement it, and get an optimum policy. I will be attaching the python code along with the document.

We print the optimal actions as a numpy array. We denote the actions choosen as "L", "R", "U" and "D", for actions left, right, up and down. Each of the action at the given state is performed with probability 1 (deterministic policy)

```
[['R' 'R' 'R' 'R' 'R' 'D']
 ['U' 'U' 'R' 'R' 'R' 'D']
 ['U' 'U' 'U' 'R' 'R' 'D']
 ['U' 'U' 0 'R' 'R' 'D']
 ['U' 'U' 0 'R' 'R' 'D']
 ['U' 'L' 'R' 'R' 'R' 0]]
```

Figure 4: The deterministic optimal policy as a matrix. Where optimal action at cell (i,j) is the ith row jth entry.

If we give the same problem to a human, chances are, they will come up with same set of actions. Notice how the agent learns to avoid the restricted (6,3) cell. IF we place the agent on cell (6,2) it wont take the action up, since then we still have 0.05 probability of entering the restricted (6,3) cell. By choosing left it ensures that it will not jump to the cell (6,3).

2.3 Monte Carlo simulations

2.4 Temporal difference algorithms

2.5 Gradient descent approach

2.6 Neural Networks