Assignment 2: Value Iteration

1 Introduction

Value iteration is an algorithm falling under the umbrella of dynamic programming. Dynamic programming can be used for optimal sequential decision making. That is, what action should I take given I am in a particular state. The perceived worth of an immediate action is based on an evaluation of how good that action is in the long run. In practice, dynamic programming is often difficult to implement because it is computationally expensive and requires that the dynamics of the environment are completely known. However, it serves as the fundamental structure for understanding many of the later algorithms developed for reinforcement learning.

The interaction between an agent and its environment is formulated as a Markov Decision Process (MDP) where we assume a finite set of states $\mathcal{S}$, actions $\mathcal{A}$, and numerical rewards $\mathcal{R}$. The dynamics can be described by a function $p(r, s'|s, a)$, $\forall s \in \mathcal{S}, a \in \mathcal{A}, r \in \mathcal{R}, s' \in \mathcal{S}$. If the task is episodic (i.e. can be broken into natural subsequences with a clear ending point) than $s' \in \mathcal{S}^+$ where $\mathcal{S}^+$ contains $\mathcal{S}$ as well as terminal or end states. Here we will treat the task as a continuing one.

Figure 1 describes the sequential interaction between agent and environment. At each time point t, an agent receives a representation of the state it is in (s_t). It then takes an action (a_t) which has some effect on the environment. This effect is fed back to the agent in the form of a next state (s_{t+1}). Additionally, the agent receives feedback for that transition in the form of a numerical reward (r_{t+1}).

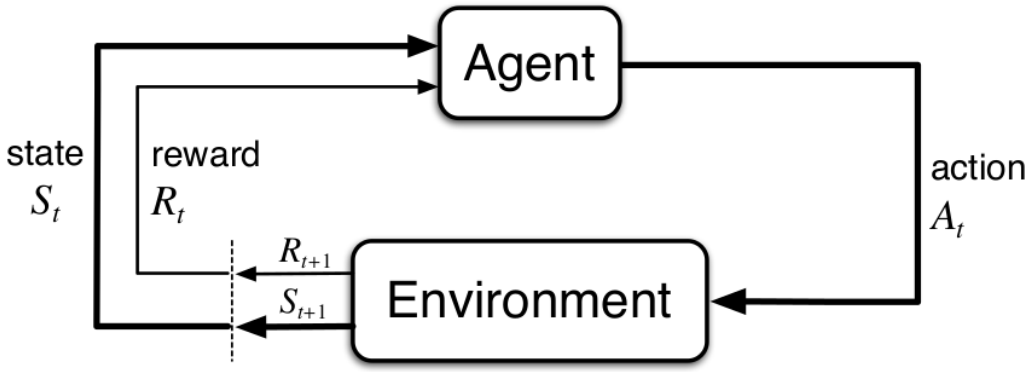

Figure 1: From Sutton and Barto 2018

In this assignment, you will implement value iteration to find the optimal policy: the best action to take for each state. The policy is written $\pi(a|s)$, or sometimes $\pi(s)$, and is the probability of taking action a given you are in state s. Intuitively, for the optimal policy ($\pi^*(a|s)$), we want to choose actions that maximize the amount of total rewards we expect to accumulate in the future, starting from that state. This quantity is represented by a state-value function $V(s)$, measuring how good it is to be in each state while adopting a policy. We can think of $\gamma \in [0, 1]$ as a discounting factor, a measure of how quickly the worth of future rewards diminish.

Finding the optimal policy requires two pieces: being able to evaluate a policy and improving on it. To evaluate a policy, we estimate the value of the states under that policy. Once we evaluate a policy, we want to know whether to stick with the policy or if doing something different would lead to improvements.

If we can find an action to take for even one of the states that leads to higher accumulated expected rewards than our current policy, that new policy is better than the existing one. Thus, in value iteration we go through each state and perform evaluation and improvement simultaneously until convergence. At each step of the algorithm, the value function is equivalent to the expected accumulation of rewards from following a particular policy.

We use $\sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$ to quantify how good each possible action is for the current state (evaluation) and choose the action that maximizes that value using $\max_a$ (improvement).

Value Iteration, for estimating $\pi \approx \pi_*$

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 $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$

Figure 2: From Sutton and Barto 2018

For further background, refer to Chapter 3 and 4 of Sutton and Barto 2018.

2 Problem Setup

Please implement value iteration using the provided code structure and algorithm above[1]. You will be given an initialized value table, a threshold for convergence, and all of the information about the environment dynamics in the form of transition table and reward table.

State space and action space: The environment is a 2D game board where each state is a tuple containing the (x,y) coordinates of that state on a grid. From each state, an agent can choose to take one of four actions: move up, move right, move down, and move left which are represented by the tuples (0,1), (1,0), (0,-1), (-1,0) respectively.

Transition: If an agent tries to move to a spot not on the board, it will stay where it is. Furthermore, the board is slippery and the transitions are not deterministic. For example, if the agent tries to move up to a valid coordinate, it will most likely move there with some small probability of slipping in a different direction.

Reward: There will be at least one goal state on the board which has a positive reward and trap states with a large negative reward (i.e. a high cost). Additionally, each move incurs a small cost. As the agent is learning a policy to maximize long-term rewards, it should take actions most likely to lead it to the goal state, while avoiding the trap, as quickly as it can.

Again, please leave names of files/functions unchanged, though you may write any other helper functions you need.

[1] The one difference between this algorithm and your implementation is your policy is not deterministic. If there is more than one best action, your policy should choose each with equal probability

3 Functions

The ValueIteration class is initialized with:

- **Attributes:**
  - **transitionTable:** $P(s'|s,a)$ The state transition, a nested dictionary of form state : action : next state : probability. The transition only includes state-action-nextstate combinations that have non-zero probability.
  - **rewardTable:** $R(s',s,a)$ The deterministic reward function, a nested dictionary of form state : action : next state : reward
  - **valueTable:** $V(s)$ the initialized value of each state, a dictionary of form state : value. All states' values are initialized to 0.
  - **convergenceTolerance:** A small scalar threshold determining how much error is tolerated in the estimation
- **__call__(self):** Returns the new value table, optimal policy in a list.
  - **valueTable:** the dictionary of form state : value where the values are the state-values of the optimal policy (i.e. what are the state values after convergence)
  - **policyTable:** $\pi(a|s)$, a dictionary of form state : action : probability giving the approximate optimal policy of an agent. You only need to include actions with non-zero probability. Note: this differs from the algorithm in Figure 2.