Markov Decision Process

Brandon Yang
University of Virginia
Charlottesville, VA, USA
jqm9ba@virginia.edu

Dynamic Programming

Dynamic Programming solutions for MDP assume a known environment in order to take the expectation over all possible next states and rewards. The process is done iteratively to compute the optimal policy / value function given MDP.

Value Iteration

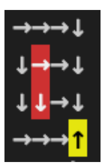
Value Iteration, for estimating $\pi \approx \pi_*$

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

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

Computed Policy (with tolerance = 0.01):



Value Function:

52.98272805	58.65479586	71.80603574	77.09290223	
46.03800916	-5.15258579	77.83147962	84.1414826	
56.78207149	1.29847647	84.86729996	91.7816501	
68.76914229	76.10763148	91.7816501	100	0

Number of Iterations:

16

Code Snippet:

```
def valueIteration(self, initialV, nIterations=np.inf,
tolerance=0.01):

        policy = np.zeros(self.nStates)
        V = initialV
        iterId = 0
        epsilon = tolerance

        while iterId < nIterations:
            iterId += 1
            delta = 0
            for current_state in range(self.nStates):
                v = V[current_state]
                max_value = -np.inf
                for action in range(self.nActions):
                    value = sum(self.T[action, current_state] *
(self.R[action][current_state] + (self.discount * V)))
                    if value > max_value:
                        max_value = value
                V[current_state] = max_value
                delta = max(delta, abs(v - V[current_state]))
            if delta < epsilon:
                break
        # output a deterministic policy
        for current_state in range(self.nStates):
            max_value = -np.inf
            selected_action = int(policy[current_state])
            for action in range(self.nActions):
                value = sum(self.T[action, current_state] *
(self.R[action][current_state] + (self.discount * V)))
                if value > max_value:
                    max_value = value
                    selected_action = action
            policy[current_state] = selected_action

        return [policy, V, iterId, epsilon]
```

Policy Iteration v1:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

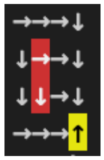
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

The first version of policy iteration involves solving a system of linear equations for policy evaluation. The policy is initialized so that all states would initially choose action 0.

Computed Policy:



Value Function:

52.98550684960492	58.65553357510296	71.80623279814883	77.09295575797236
46.03871770330745	-5.152410959209803	77.83151901332299	84.14149058571167
56.782261266602845	1.298514747683356	84.86730581429448	91.78165088658342
68.7691941384811	76.10763930920807	91.78165088658342	100.0
0.0			

Number of Iterations:

5

Code Snippet:

```
def policyIteration_v1(self, initialPolicy, nIterations=np.inf,
tolerance=0.01):

        policy = initialPolicy
        V = np.zeros(self.nStates)
        iterId = 0

        while iterId < nIterations:
            iterId += 1
            # Policy Evaluation Linear Systems of Equations
            V =
self.evaluatePolicy_SolvingSystemOfLinearEqs(policy)
            # Policy Improvement
            policy, policy_stable = self.extractPolicy(V)
            if policy_stable is True:
                break

        return [policy, V, iterId]


def evaluatePolicy_SolvingSystemOfLinearEqs(self, policy):

        # construct the Transition Matrix following current policy
        Transition_Matrix = []
        for i in range(self.nStates):

Transition_Matrix.append(self.T[int(policy[i])][i].tolist())
        Transition_Matrix = np.array(Transition_Matrix)
        # Calculate Value Function using system of linear
equations
        V = np.dot(np.linalg.inv((np.identity(self.nStates) -
(self.discount * Transition_Matrix))), self.R[0])

        return V

def extractPolicy(self, V):

        policy_stable = True
        for current_state in range(self.nStates):
            old_action = int(policy[current_state])
            # pick action that maximizes value
            max_value = -np.inf
            selected_action = old_action
            for action in range(self.nActions):
                value = sum(self.T[action, current_state] *
(self.R[action][current_state] + (self.discount * V)))
                if value > max_value:
                    max_value = value
                    selected_action = action
            policy[current_state] = selected_action
            if old_action != policy[current_state]:
                policy_stable = False

        return policy, policy_stable
```

Value Iteration v2

The second version of policy uses the same idea as the first version, in which both versions include alternating between policy evaluation and policy improvement. Version 2 involves evaluating the policy iteratively for n times. The relationship between evaluating the policy n times and the number of total policy iterations is shown in the table below. (tolerance = 0.01)

Number of iterations in policy evaluation ($\boldsymbol{n}$)	Number of iterations needed by policy iteration to converge
1	7
2	5
3	5
4	5
5	5
6	5
7	5
8	5
9	5
10	5

The number of iterations in policy evaluation affects the total number of iterations needed to converge to the optimal policy. Based on the table, when the number of iterations is 1 in policy evaluation, the total number of iterations for the policy to converge is 7. For all other numbers of iterations in policy evaluation, the results are all 5 iterations. Generally, more iterations in policy evaluation should lead to less iterations in the policy iteration process. However, due to the tolerance level, and the fact that initial policy chooses action 0 in all states, the number of overall iterations stayed same.

The iterative method of policy evaluation estimates the value function, while solving a system of linear equations calculates the exact value from the policy in each iteration. The complexity of using the matrix form of policy evaluation is $\boldsymbol{O}(|\boldsymbol{S}|^3)$, while the complexity of solving for the value function iteratively is $\boldsymbol{O}(|\boldsymbol{A}| \times |\boldsymbol{S}|^2)$. This means that iterative policy evaluation is faster than solving a system of linear equations, because the iterative method is estimating the value function and passing it to policy improvement to see if the estimated value function is good enough. If that is not the case, then the policy would be evaluated again. A system of linear equations ensure that the value functions are exact for the policy that is evaluated. In this environment, when the initial policy has action 0 for all states, it would take 5 iterations to converge to an optimal policy following the matrix form of policy evaluation. It would also take 5 iterations to converge to an optimal policy using iterative policy evaluation that has more than 2 iterations. This means that 2 iterations for policy evaluation is good enough to achieve the optimal policy for this environment. Both versions of policy iteration perform better than value iteration. This is because value iteration is evaluating the value function until a certain tolerance level, while policy iteration stops when policy remains consistent. This means that policy iteration does not need to calculate the exact value function for each state to obtain the optimal policy.

Code Snippet:

```
def policyIteration_v2(self, initialPolicy, initialV,
nPolicyEvalIterations=10, nIterations=np.inf, tolerance=0.01):

        V = initialV
        policy = initialPolicy
        iterId = 0
        epsilon = tolerance

        while iterId < nIterations:
            iterId += 1
            # Partial Policy Evaluation using iteratively method
            policy_iterId = 0
            while policy_iterId < nPolicyEvalIterations:
                policy_iterId += 1
                delta = 0
                for current_state in range(self.nStates):
                    v = V[current_state]
                    action = int(policy[current_state])
                    V[current_state] = sum(self.T[action,
current_state] * (self.R[action][current_state] + (self.discount *
V)))
                    delta = max(delta, abs(v - V[current_state]))
                if delta < tolerance:
                    break
            # Policy Improvement
            policy, policy_stable = self.extractPolicy(V)
            if policy_stable is True:
                break

        print(iterId)

        return [policy, V, iterId, epsilon]
```

Model-Free Control

Model-Free assumes the environment is unknown to the agent. The agent needs to sample sequences of states, actions, and rewards from interacting with the environment to learn an optimal policy. In this assignment, state transitions follow the state transition table, and the reward is sampled from a Gaussian distribution with the actual reward of each state as mean and standard deviation 1.0.

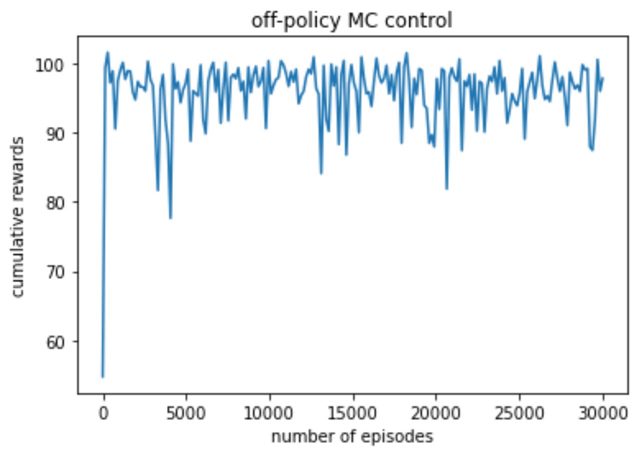
Off-Policy MC Control

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

This version of off-policy MC control uses a uniformly random soft policy to take an action at any state to generate its experience. The target policy is a greedy policy with respect to the currently estimated Q-function. Below is a graph produced by off-policy MC control after 30,000 episodes, averaged over 10 runs.



Code Snippet:

```
def OffPolicyMC(self, nEpisodes, epsilon=0.1):
        total_return = []
        Q = np.zeros([self.mdp.nActions,self.mdp.nStates])
        policy = np.zeros(self.mdp.nStates,int)
        C = np.zeros_like(Q)
        nSteps = 100_000_000
        for i_episode in range(1, nEpisodes+1):
            if i_episode % 1000 == 0:
                print("\rEpisode {}/{}.".format(i_episode,
nEpisodes), end="")
                sys.stdout.flush()
            # Generate an episode using b: S0, A0, R1,...,ST−1,
AT−1, RT
            episode = []
            # make starting state random
            state = np.random.randint(16)

            for t in range(nSteps):
                action = np.random.randint(4)
                reward, next_state =
self.sampleRewardAndNextState(state, action)
                episode.append((state, action, reward))
                if next_state == 16:  # if the terminal state is
reached, break
                    break
                state = next_state

            G = 0.0
            W = 1.0

            for t in range(len(episode))[::-1]:
                state, action, reward = episode[t]
                G = self.mdp.discount * G + reward
                C[action][state] += W
                Q[action][state] += (W / C[action][state]) * (G -
Q[action][state])
                policy[state] = np.argmax(Q[:, state])
                if action != policy[state]:
                    break
                W = W / (1/self.mdp.nActions)

            total_return.append(G)

        return [Q, policy, total_return]
```

Off-Policy TD Control

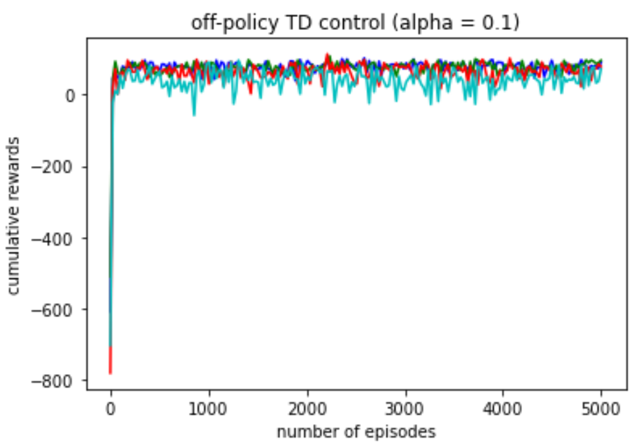
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

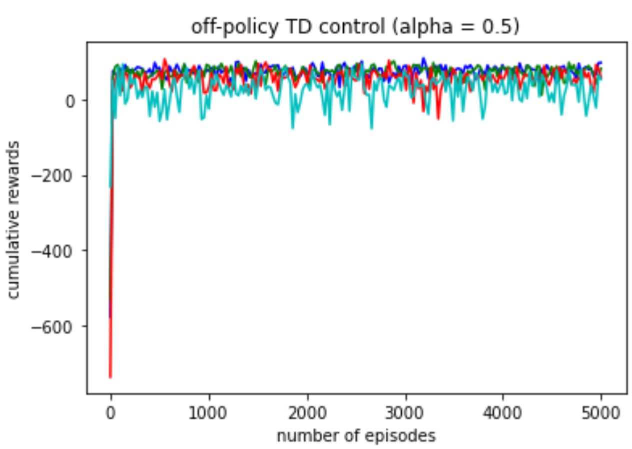
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

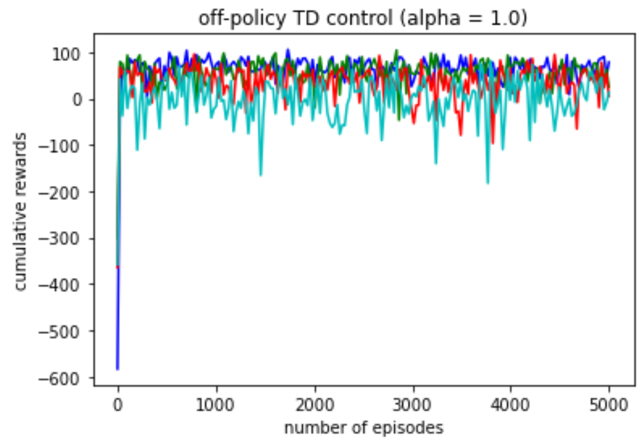
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

This is the Q-Learning algorithm. In this implementation, an extra hyperparameter α, or the learning rate is implemented according to the pseudocode. A figure is produced with exploration probability epsilon = 0.05, 0.1, 0.3 and 0.5. Three experiments are run with $\alpha = 0.1, 0.5, 1.0$. All experiments run for 5000 episodes.

Color:	Epsilon value:
blue	0.05
green	0.1
red	0.3
cyan	0.5







Changing the exploration probability epsilon influences the overall cumulative rewards per episode earned during training as well as the resulting Q-values and policy. Epsilon is a value that determines the probability of exploring a random action rather than taking the action that maximizes the Q-value for the current state. When epsilon is small, the algorithm would mostly exploit its current knowledge, and chooses an action based on the Q-value table; when epsilon is large, the algorithm would choose more random actions to explore the environment. This grid-world environment is a very simple environment, which has only 16 states. This means that the values in the Q-table are likely very good estimates of the environment. Therefore, for this specific environment, exploitation heavy settings would perform better than exploration heavy. Based on all three graphs, the algorithms with the smallest value of epsilon perform the best, because they accumulate the greatest number of cumulative rewards per episode. Algorithms that run with higher epsilon values can converge to an optimal policy, but the cumulative rewards per episode become nosier. This is more prevalent when the learning rate α is set to a higher value, such as 1. The Graph *off-Policy TD control (alpha = 1.0)* shows that although the total cumulative rewards are increasing over time for all runs with different epsilon values, the least value of epsilon accumulated the most amount rewards with the least variance, while the greatest value of epsilon accumulated the least amount of rewards with the highest variance.

Code Snippet:

```
def OffPolicyTD(self, nEpisodes, epsilon=0.0, alpha=0.1):
        Q = np.zeros([self.mdp.nActions,self.mdp.nStates])
        policy = np.zeros(self.mdp.nStates,int)
        cumulative_reward = []
        for i_episode in range(1, nEpisodes+1):
            if i_episode % 1000 == 0:
                print("\rEpisode {}/{}.".format(i_episode,
nEpisodes), end="")
                sys.stdout.flush()

            sum_rewards = 0
            state = np.random.randint(16)
            while True:
                policy[state] = np.argmax(Q[:, state])
                # epsilon greedy
                explore = np.random.binomial(1, epsilon)
                if explore == 1:
                    action = np.random.randint(4)
                else:
                    action = policy[state]
```

```
                reward, next_state =
self.sampleRewardAndNextState(state, action)
                sum_rewards += reward
                best_next_action = np.argmax(Q[:, next_state])
                Q[action][state] += alpha * (reward +
(self.mdp.discount * Q[best_next_action][next_state]) -
Q[action][state])
                state = next_state
                if next_state == 16:  # if the terminal state is
reached, break
                    break
            cumulative_reward.append(sum_rewards)
        return [Q,policy, cumulative_reward]
```