Reinforcement Learning: Assignment 3

1 Aug 2024

Github: https://github.com/nhienchic/Assignment3

Part 1

0. Environment Setup

Consider a simple 5×5 gridworld problem:

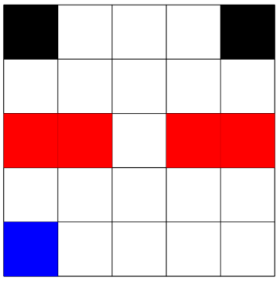

Fig. 1: Gridworld.

Each of the 25 cells of the gridworld represents a possible state of the world. The gridworld is implemented in the code as

```
self.map = [['T','W','W','W','T'],
            ['W','W','W','W','W'],
            ['R','R','W','R','R'],
            ['W','W','W','W','W'],
            ['B','W','W','W','W']]
```

Fig. 2: Gridworld map in code.

where W, B, R and T represent the white, blue, red, and black grid (terminal), respectively. For example, $self.map[0][4] = 'T'$, $self.map[2][3] = 'R'$.

An agent in the gridworld environment can take a step up, down, left, or right, which is expressed as

```
# Available actions
#                      left      down      right     up
self.action =       [ [0, -1],  [1, 0],   [0, 1],   [-1, 0]]
self.action_text =  ['\u2190', '\u2193', '\u2192', '\u2191']
```

Fig. 3: Actions in code.

where the $self.action$ represents the actual action that is taken. For example, $self.action[0] = [0, -1]$ means keeping the index of the current row intact while reducing the index of the current column by 1. The second variable $self.action\_text$ is the Unicode text used for visualization, e.g. $\backslash u2190$ represents ←.

After the map and action settings are introduced, the model of this gridworld problem is described as follows. Based on the current state and action (s, a), the model returns a sequence (p, s', r, t) where p denotes the probability of transiting from state s to state s' under action a, and t is a bool value which is $'True'$ or $'False'$ representing whether the next state s' is a terminal state.

The blue grid, i.e. $self.map[row][col] = 'B'$, is considered as the starting grid. For each episode, the agent starts from the blue grid and tries to find the best route. The agent moves among the blue grid and the white grids. If the agent attempts to step off the grid, a reward of -1 will be received. If the agent moves to the red grid, it receives a reward of -20 and moves back to the blue grid. If the agent moves to the black grid, the episode is terminated. The model is implemented in code as

```
# Blue
if self.map[row][col] == 'B':
    if outsidecheck:
        self.model[s][a].append([1.0, s, -1.0, False])  # if want to move out, stay and -1.0
    else:
        self.model[s][a].append([1.0, state_, -1.0, False])

# White
elif self.map[row][col] == 'W':
    if outsidecheck:
        self.model[s][a].append([1.0, s, -1.0, False])  # if want to move out, stay and -1.0
    elif self.map[row_][col_] == 'R':  # if move to R, jump to start point
        self.model[s][a].append([1.0, self.state_0, -20.0, False])
    elif self.map[row_][col_] == 'T':  # if move to terminal state, end
        self.model[s][a].append([1.0, state_, -1.0, True])
    else:
        self.model[s][a].append([1.0, state_, -1.0, False])

# Red
elif self.map[row][col] == 'R':  # if current is red, just for indexing
    self.model[s][a].append([1.0, s, 0.0, False])

# Black (Terminal)
elif self.map[row][col] == 'T':  # if current is terminal, just for indexing
    self.model[s][a].append([1.0, s, 0.0, True])

else:
    raise ValueError("Unknown value !!!!!!!!!!")
```

Fig. 4: Model in code.

Intuitively, an agent with a good policy should try to avoid the red grid and find the fastest way from the blue grid to the black grid.

1. Algorithms

1.1 Sarsa

This algorithm can be implemented according to the pseudocode given in the book.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

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

Fig. 5: Sarsa method.

1.2 Q-learning

Similarly, this algorithm can be implemented according to the pseudocode given in the book.

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

Fig. 6: Q-learning method.

2. Results

With a discount factor $\gamma = 0.99$, epsilon greedy action selection with epsilon = 0.2, learning rate alpha = 0.5, and a policy with equiprobable moves, the results of the two methods are shown in the following figures.

Fig. 7: (Left) Optimal policy; (Right) Trajectory under the optimal policy.

According to the derived optimal policy, we can see that Q-learning algorithm performs better than the Sarsa algorithm. Q-learning algorithm can almost give all the correct choices at each state (except for the bottom right corner), while Sarsa can only give some of the correct choices. For example, at the starting grid, Q-learning algorithm shows 2 choices: go left and go up, while the Sarsa algorithm only shows 1 choice: go up.

The trajectory given by the two algorithms are generally the same, both avoid the red grids and move to the black grid in the smallest steps. We think the similar results are caused by the nature of this 5 by 5 grid world problem. This problem is symmetric, as long as the agent does not try to move out, move back towards the starting point, or move to the red grid, any trajectories can be one of the best choices.

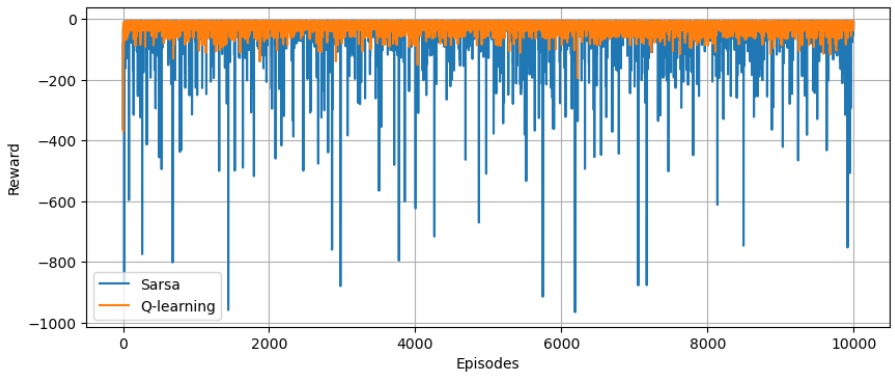

Fig. 8: Sum of rewards.

Fig. 8 shows the sum of rewards over each episode for these two algorithms. It is obvious that the sum of rewards for Q-learning is much higher than that of Sarsa's, which means that Q-learning is less likely to move to the red grid and move to the black grid faster than Sarsa algorithm.

Part 2

0. Environment Setup

Considering the random walk in a 7 by 7 gridworld problem:

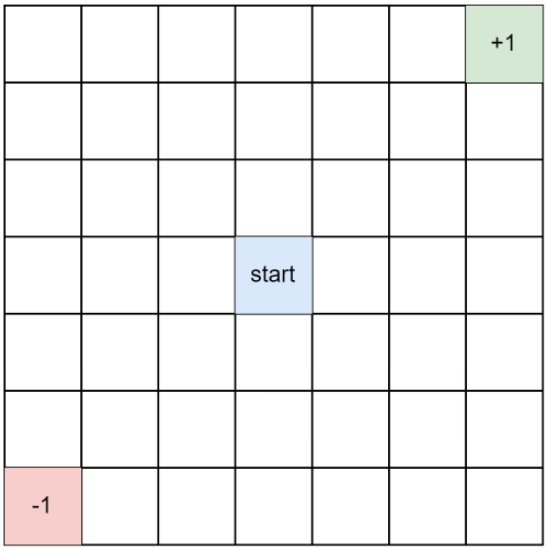

Fig. 9: 7 by 7 Gridworld problem.

The new gridworld is implemented in the code as

```
self.map = [['W','W','W','W','W','W','G'],
            ['W','W','W','W','W','W','W'],
            ['W','W','W','W','W','W','W'],
            ['W','W','W','S','W','W','W'],
            ['W','W','W','W','W','W','W'],
            ['W','W','W','W','W','W','W'],
            ['R','W','W','W','W','W','W']]
```

Fig. 10: New Gridworld map in code.

where 'S' is the start state, 'G' and 'R' are terminal states with rewards of $+1$ and -1, respectively.

For each episode, the agent starts from the starting grid and try to find the best route. The agent moves among the starting grid and the white grids. If the agent attempts to step off the grid, a reward of 0 will be received. If the agent moves to the red grid, it receives a reward of -1 and ends the episode. If the agent moves to the green grid, it receives a reward of $+1$ and ends the episode. The model is implemented in code as

```
# Start grid
if self.map[row][col] == 'S':
    self.model[s][a].append([1.0, state_, 0.0, False])  # normal movement

# White
elif self.map[row][col] == 'W':
    if outsidecheck:
        self.model[s][a].append([1.0, s, 0.0, False])  # if want to move out, stay
    elif self.map[row_][col_] == 'R':
        self.model[s][a].append([1.0, state_, -1.0, True])  # red terminal
    elif self.map[row_][col_] == 'G':
        self.model[s][a].append([1.0, state_, 1.0, True])  # green terminal
    else:
        self.model[s][a].append([1.0, state_, 0.0, False])  # normal movement

# Red or Green
elif self.map[row][col] == 'R':  # if current is red, just for indexing
    self.model[s][a].append([1.0, s, 0.0, True])
elif self.map[row][col] == 'G':  # if current is green, just for indexing
    self.model[s][a].append([1.0, s, 0.0, True])

else:
    raise ValueError("Unknown value !!!!!!!!!!")
```

Fig. 11: Model in code.

Intuitively, an agent with a good policy should try to avoid the red grid and move to the green grid.

1. Algorithms

1.0 Affine function approximation

In the code, the Affine function approximation is realized as

```
def Affine(state, env):
    """
    Affine function approximation
    Input:
        state (int) - state number
        env (class) - gridworld environment with the following parameters:
            env.n_state (int) - number of states
    Output:
        fv (1d array) - encoded feature vector
    """

    fv = np.zeros(env.n_state)  # initialize the feature vector
    fv[state] = 1  # use one-hot encoding for the input state

    return fv
```

Fig. 12: Affine function approximation.

Different from the 1-D random walk example shown in the book, this is a 2-D random walk, which makes it not possible to group a set of states. Therefore, here we only focus on the exact state, we set the coefficient in feature vector for the state as 1, and other coefficients as 0.

```
x = Affine(trace[0], env)  # feature vector based on current state
w += alpha * (G - np.dot(w, x)) * x  # update weight
```

Fig. 13: Weight update.

Then, as shown in Fig. 13, $\hat{v}$ is simply the dot product of w and x, and the gradient of $\hat{v}$ is simply x.

1.1 Gradient Monte Carlo Algorithm

This algorithm can be implemented according to the pseudocode given in the book.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Fig. 14: Gradient Monte Carlo Algorithm

1.2 Semi-gradient TD(0) Algorithm

Similarly, this algorithm can be implemented according to the pseudocode given in the book.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Fig. 15: Semi-gradient TD(0) Algorithm

1.3 Compute the exact value function

Based on the Bellman equations shown on the right panel of Fig. 16, we set up a linear equation as shown in the left panel of Fig. 16.

```
for s in range(env.n_state):  # sweep all the s
    A[s,s] = 1  # VF[s] = sum(policy[s,a] * p *
    for a in range(env.n_action):  # sweep all
        for p, s_, r in env.model[s][a]:  # swe
            A[s,s_] -= policy[s,a] * p * gamma
            b[s] += policy[s,a] * p * r  # keep
```

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

Fig. 16: Building the equation set according to the Bellman equations.

where A is initialized as a zero matrix. For each state s, $A[s, s]$ is set as 1, which represents the coefficient of $v(s)$, and $A[s, s']$ shown in the fifth line on the left panel in Fig.16 represents the coefficient of $v(s')$ after moving to the left side of the equation so a the minus sign is used. $b[s]$ shown in the last line on the left panel in Fig.16 represents the constant in the Bellman equation for state s. After this step, the linear equation is transferred to the matrix multiplication $Ax = b$, where A is a 49×49 matrix representing the coefficients for all the states in the Bellman equation, x is a 49×1 matrix representing the 49 unknown $v(s)$, b is also a 49×1 matrix representing the constant in the Bellman equation. x can be simply solved by multiplying the inverse of matrix A with b.

2. Results

With a discount factor $\gamma = 0.99$, learning rate alpha = 0.01, and a policy with equiprobable moves, the results are shown in the following figures.

```
Gradient Monte Carlo method:
[[ 0.109  0.112  0.234  0.3    0.45   0.618  0.   ]      [['→' '→' '→' '→' '→' '→' 'o']
 [-0.008  0.066  0.118  0.202  0.297  0.474  0.568]       ['↑' '→' '↑' '↑' '→' '↑' '↑']
 [-0.058 -0.012  0.022  0.127  0.234  0.44   0.432]       ['↑' '↑' '→' '→' '→' '↑' '↑']
 [-0.258 -0.181 -0.068  0.068  0.164  0.317  0.306]       ['↑' '↑' '→' '→' '→' '↑' '↑']
 [-0.4   -0.385 -0.178 -0.062  0.047  0.177  0.177]       ['↑' '→' '→' '↑' '→' '↑' '↑']
 [-0.64  -0.501 -0.264 -0.156 -0.086  0.019  0.033]       ['↑' '→' '→' '↑' '↑' '↑' '↑']
 [ 0.    -0.623 -0.327 -0.221 -0.196 -0.166 -0.091]]      ['o' '→' '→' '↑' '↑' '↑' '↑']]

Semi-gradient TD(0) method:
[[ 0.001  0.024  0.093  0.21   0.371  0.6    0.   ]      [['→' '→' '→' '→' '→' '→' 'o']
 [-0.03  -0.003  0.056  0.143  0.282  0.428  0.561]       ['↑' '→' '→' '→' '→' '↑' '↑']
 [-0.088 -0.062 -0.     0.081  0.196  0.266  0.364]       ['↑' '→' '→' '→' '↑' '↑' '↑']
 [-0.2   -0.145 -0.077  0.017  0.093  0.165  0.198]       ['↑' '↑' '→' '→' '↑' '↑' '↑']
 [-0.34  -0.263 -0.157 -0.065  0.011  0.064  0.09 ]       ['↑' '↑' '→' '↑' '↑' '↑' '↑']
 [-0.581 -0.437 -0.271 -0.134 -0.049  0.001  0.036]       ['↑' '↑' '→' '→' '↑' '↑' '↑']
 [ 0.    -0.603 -0.362 -0.195 -0.084 -0.023  0.008]]      ['o' '→' '→' '→' '→' '→' '↑']]

Exact value function:
[[ 0.     0.031  0.093  0.193  0.344  0.585  0.   ]      [['→' '→' '→' '→' '→' '→' 'o']
 [-0.031  0.     0.06   0.149  0.27   0.423  0.585]       ['→↑' '→' '→' '→' '→' '→↑' '↑']
 [-0.093 -0.06   0.     0.08   0.173  0.27   0.344]       ['↑' '→↑' '→' '→' '→↑' '↑' '↑']
 [-0.193 -0.149 -0.08   0.     0.08   0.149  0.193]       ['↑' '↑' '→↑' '→↑' '↑' '↑' '↑']
 [-0.344 -0.27  -0.173 -0.08   0.     0.06   0.093]       ['↑' '↑' '→↑' '→↑' '↑' '↑' '↑']
 [-0.585 -0.423 -0.27  -0.149 -0.06   0.     0.031]       ['↑' '→↑' '→' '→' '→↑' '↑' '↑']
 [ 0.    -0.585 -0.344 -0.193 -0.093 -0.031  0.   ]]      ['o' '→' '→' '→' '→' '→↑' '↑']]
```

Fig. 17: (Left) Optimal policy; (Right) Trajectory under the optimal policy.

As shown in the left-hand side of the Fig. 17, the value functions estimated by the two algorithms is compared with the value function derived from the Bellman equations. Semi-gradient TD(0) method gives a closer approximation compared to the Gradient Monte Carlo method, the mean square error is shown in the following figure.

```
MSE between Gradient Monte Carlo method and the exact value function:
0.005529755102040816

MSE between Semi-gradient TD(0) method and the exact value function:
0.00012297959183673465
```

Fig. 18: MSE of the two approaches.

On the right-hand side of the Fig. 17, the optimal policy is shown based on the three value functions. Both the Semi-gradient TD(0) method and the Gradient Monte Carlo method give correct (although not all) choices of the random walk in the 7 by 7 grid world.