인공지능

Lab. Project #2

-Dynamic Programming-

컴퓨터정보공학부

2020202013

고가연

I. Introduction

-> 해당 프로젝트에서는 동적계획법(Dynamic Programming)을 이용한 environment의 model을 푸는 과제이다. 제안서에서 제시한 7x7 grid-world에서 Policy iteration과 Value iteration을 구현하여 policy를 최적화하는 것을 목표로 한다. 7x7 grid-world는 시작점과 끝점을 가졌고 초기값은 모두 -1이고 state 중에서 reward값이 -100을 가지는 함정 state가 존재한다. 그리고 action은 상하좌우 4가지이며 제일 바깥쪽 state에서 grid-world 밖으로 나가는 동작(action)을 취하게 되면 다시 제자리로 돌아오는 것을 원칙으로 한다. 또한, 처음 agent가 택하는 policy는 상하좌우 모두 같은 확률이 0.25로, uniform random policy이다.

II. Algorithm

1. Policy evaluation – random policy

-> 현재 주어진 Policy인 uniform random policy에 대한 value function을 구하는 것이고, one step back으로 구현하였다. Iteration number는 0부터 99까지 총 100번을 반복하며 one step씩 각 step의 value function을 update하는 과정을 출력하도록 코딩하였다. 한 step씩 아래의 식을 통해서 value function을 update하도록 개발했다. 이런 식을 무한대까지 계산하여 현재 random policy에 대한 true value function을 구할 수 있다.

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

-> Policy evaluation 알고리즘을 구현하기 위해 사용한 Method의 동작에 대해 설명하겠다. 이전 value 7x7 table을 저장하기 위해 post_value_table 2차원 행렬을 -1로 초기화하였다. for문을 사용해서 iteration을 0부터 99으로 총 100번을 반복할 수 있도록 했다. 그 반복문 안에 next_value_table 7x7 행렬을 생성하여 -1로 초기화하였다. 이때 종점은 0으로 초기화했다. 그리고 iteration 반복문 안에 이중 for문을 작성해 7x7 grid-world의 출발점부터 행순으로 각 state의 action(up, down, left, right)에 따라서 위의 식대로 value function을 계산한 다음 모두 합하여 next_value_table의 해당 state 칸에 update할 수 있도록 코딩했다. Action을 취할 때 for문을 사용해 4번 반복하는데 grid-world를 벗어나는 행동을 취할 경우 제자리로 돌아올 수 있도록 만들었다. 결과적으로 Iteration

0부터 Iteration 99까지의 value table을 순차적으로 출력하였다.

2. Policy Improvement

-> 위에서 설명한 알고리즘인 Policy evalution에서 Iteration을 반복하며 찾은 true value function을 사용해 이 policy를 따르는 것이 좋을지 안 좋을 지를 판단하고 Policy를 update해야 한다. 이를 통해서 현재 policy보다 더 나은 policy를 찾아가면 optimal policy에 가까워지게 되는데 해당 과정을 Policy Improvement라고 한다.

-> 해당 알고리즘을 구현하기 위해서 사용한 Method 동작에 대해 설명하겠다. 먼저 action_match 1차원 배열을 선언해 'Up', 'Down', 'Left', 'Right'을 저장해 놓았다. 그리고 policy evaluation에서 찾은 true value function을 이용하기 위해 이중 for문을 작성했다. 종점은 update하지 않도록 예외처리를 해주었고 각 state value를 토대로 현재의 policy를 action-value function을 이용해 더 좋은 action을 선택하는 policy로 결정하는 알고리즘을 구현하였다. 이때 각 state의 상하좌우 action 4개의 값 중 최대값을 선택하기 위해서 np.argmax() 메소드를 사용했다. 해당 메소드는 최대값에 해당하는 색인(index)를 찾는 동작을 수행한다. 인자로 policy 행렬을 넣어서 정상적으로 최대값인 action을 취할 수 있도록 작성했다. 또한, policy가 업데이트되는 과정을 policy 행렬을 사용해 출력하였다.

3. Value Iteration

-> Value Iteration은 Bellman Optimality Eqn.을 이용해서 계산할 수 있다. Evaluation 과정에서 max값을 취해서 greedy하게 value function을 구한 다음 improve 과정까지 진행하는 것이 Value Iteration 알고리즘이다. 이는 Policy evaluation 과정과 달리, value function의 action을 취할 확률을 곱해서 합치는 것이 아니라 max값을 취하는 것이 주의해야 할 점이다.

-> 해당 알고리즘을 구현하기 위해 이용한 Method의 동작을 설명하겠다. 먼저, 이전 state의 value값을 저장할 7x7 행렬 post_value_table을 선언했다. 마찬가지로 -1로 초기화했다. 그리고 Iteration 0~99까지 총 100번을 반복하도록 for문을 작성했다. 해당 반복문 안에서 update한 value를 저장하기 위한 7x7 next_value_table을 -1로 초기화하였다. 이중 for문을 사용해 7x7 행렬의 모든 state를 돌면서 종점을 제외하고 value값을 상하좌우 action 값 4개 중 가장 큰 값으로 update할 수 있도록 했다. 최대값을

고르기 위해 사용한 method는 max()이다. 이때 reward가 -100으로 지정된 함정 state에 대한 예외처리도 작성했다. 그리고 action을 취할 때 grid-world를 벗어나는 경우에 대해 다시 제자리로 돌아오도록 작성했다.

III. Result

1. Policy evaluation – random policy

-> Iteration 0, 1, 2, 3인 결과를 캡처한 화면이다.

```
Iteration: 0
[[-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.  0.]]

Iteration: 1
[[  -2.    -2.   -101.    -2.    -2.    -2.    -2.  ]
 [  -2.    -2.   -101.    -2.    -2.    -2.    -2.  ]
 [  -2.    -2.    -2.    -2.    -2.    -2.    -2.  ]
 [  -2.    -2.    -2.    -2.   -101.   -101.    -2.  ]
 [  -2.    -2.    -2.    -2.    -2.    -2.    -2.  ]
 [  -2.    -2.    -2.    -2.    -2.    -2.   -1.75]
 [  -2.    -2.   -101.   -101.    -2.   -1.75   0.  ]]
```

```
Iteration: 2
[[  -3.    -27.75  -151.5   -27.75   -3.     -3.     -3.   ]
 [  -3.    -27.75  -126.75  -27.75   -3.     -3.     -3.   ]
 [  -3.     -3.    -27.75   -3.    -27.75  -27.75   -3.   ]
 [  -3.     -3.     -3.    -27.75  -126.75 -126.75  -27.75 ]
 [  -3.     -3.     -3.     -3.    -27.75  -27.75   -2.938]
 [  -3.     -3.    -27.75  -27.75   -3.     -2.875  -2.438]
 [  -3.    -27.75  -151.5  -151.5   -27.688  -2.438   0.   ]]

Iteration: 3
[[ -10.188  -53.5   -183.438  -53.5   -10.188   -4.     -4.   ]
 [ -10.188  -41.125 -158.688  -41.125  -16.375  -10.188  -4.   ]
 [  -4.     -16.375  -34.938  -28.75   -41.125  -41.125  -16.375]
 [  -4.      -4.     -16.375  -34.938 -152.5   -152.5    -41.11 ]
 [  -4.      -4.     -10.188  -22.562  -41.125  -41.078  -16.219]
 [  -4.     -16.375  -47.312  -47.312  -22.516   -9.907   -3.063]
 [ -10.188  -47.312 -189.625 -189.61   -47.156   -9.25     0.   ]]
```

-> Iteration 99인 결과를 캡처한 화면이다.

```
Iteration: 99
[[-1166.788 -1230.24  -1368.117 -1201.439 -1097.058 -1031.089  -996.223]
 [-1138.97  -1191.084 -1311.278 -1172.878 -1091.474 -1032.051  -992.985]
 [-1094.312 -1118.718 -1151.129 -1120.415 -1095.923 -1043.745  -981.222]
 [-1059.87  -1072.454 -1087.255 -1093.574 -1158.456 -1094.871  -935.217]
 [-1046.607 -1057.089 -1063.72  -1038.263  -981.464  -872.054  -754.18 ]
 [-1055.733 -1077.626 -1102.695 -1042.314  -882.07   -679.376  -474.348]
 [-1075.084 -1126.121 -1256.244 -1172.24   -846.813  -505.224     0.   ]]
```

-> 중간에 함정인 부분(reward = -100)이 존재하기 때문에 Iteration 1부터 value function을 계산하여 update하는 부분부터 -101처럼 매운 큰 값이 보이는 것을 확인할 수 있다. Iteration 99인 부분은 각 state가 특정한 값에 수렴하는 것을 보여주고 있다.

2. Policy Improvement

-> 왼쪽의 2개의 이미지는 새로 갱신된 policy가 업데이트되는 과정을 보여주고 맨 오른쪽의 이미지는 각 state에서의 action('Up', 'Down', 'Left', 'Right')을 매트릭스로 만들어 결과를 출력한 화면이다.

```
Updated policy is :
[[[0. 1. 0. 0.]
  [0. 0. 1. 0.]
  [0. 0. 0. 1.]
  [0. 0. 0. 1.]
  [0. 0. 0. 1.]
  [0. 1. 0. 0.]]

 [[0. 1. 0. 0.]
  [0. 1. 0. 0.]
  [0. 1. 0. 0.]
  [0. 0. 0. 1.]
  [0. 0. 0. 1.]
  [0. 0. 0. 1.]
  [0. 1. 0. 0.]]

 [[0. 1. 0. 0.]
  [0. 1. 0. 0.]
  [0. 1. 0. 0.]
  [0. 0. 0. 1.]
  [0. 0. 0. 1.]
  [0. 0. 0. 1.]
  [0. 1. 0. 0.]]]
```

```
[[0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]]

[[0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]]

[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]]

[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 0.]]]
```

```
at each state, chosen action is :
[['Down' 'Left' 'Right' 'Right' 'Right' 'Right' 'Down']
 ['Down' 'Down' 'Down' 'Right' 'Right' 'Right' 'Down']
 ['Down' 'Down' 'Down' 'Down' 'Right' 'Right' 'Down']
 ['Down' 'Down' 'Down' 'Down' 'Down' 'Down' 'Down']
 ['Left' 'Left' 'Right' 'Right' 'Right' 'Down' 'Down']
 ['Up' 'Left' 'Right' 'Right' 'Right' 'Right' 'Down']
 ['Up' 'Left' 'Up' 'Right' 'Right' 'Right' 'T']]
```

-> policy evaluation에서 최종 수렴한 true value function을 가져와서 각 state에서 최대값을 취할 수 있는 action을 선택한 모습이다. 따라서, Optimal policy를 위와 같이 찾았다.

3. Value Iteration

-> 아래의 이미지는 Iteration 0, 1, 2, 3인 결과를 캡처한 화면이다.

```
Iteration: 0
[[-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.  0.]]

Iteration: 1
[[  -2.   -2. -101.   -2.   -2.   -2.   -2.]
 [  -2.   -2. -101.   -2.   -2.   -2.   -2.]
 [  -2.   -2.   -2.   -2.   -2.   -2.   -2.]
 [  -2.   -2.   -2.   -2. -101. -101.   -2.]
 [  -2.   -2.   -2.   -2.   -2.   -2.   -2.]
 [  -2.   -2.   -2.   -2.   -2.   -2.   -1.]
 [  -2.   -2. -101. -101.   -2.   -1.    0.]]
```

```
Iteration: 2
[[  -3.   -3. -102.   -3.   -3.   -3.   -3.]
 [  -3.   -3. -102.   -3.   -3.   -3.   -3.]
 [  -3.   -3.   -3.   -3.   -3.   -3.   -3.]
 [  -3.   -3.   -3.   -3. -102. -102.   -3.]
 [  -3.   -3.   -3.   -3.   -3.   -3.   -2.]
 [  -3.   -3.   -3.   -3.   -3.   -2.   -1.]
 [  -3.   -3. -102. -102.   -2.   -1.    0.]]

Iteration: 3
[[  -4.   -4. -103.   -4.   -4.   -4.   -4.]
 [  -4.   -4. -103.   -4.   -4.   -4.   -4.]
 [  -4.   -4.   -4.   -4.   -4.   -4.   -4.]
 [  -4.   -4.   -4.   -4. -103. -103.   -3.]
 [  -4.   -4.   -4.   -4.   -4.   -3.   -2.]
 [  -4.   -4.   -4.   -4.   -3.   -2.   -1.]
 [  -4.   -4. -103. -102.   -2.   -1.    0.]]
```

-> 아래의 이미지는 충분히 Iteration 값을 늘려 수렴하게 된 부분을 캡처한 화면이다.

```
Iteration: 99
[[ -12.  -11. -109.   -9.   -8.   -7.   -6.]
 [ -11.  -10. -108.   -8.   -7.   -6.   -5.]
 [ -10.   -9.   -8.   -7.   -6.   -5.   -4.]
 [  -9.   -8.   -7.   -6. -104. -103.   -3.]
 [  -8.   -7.   -6.   -5.   -4.   -3.   -2.]
 [  -7.   -6.   -5.   -4.   -3.   -2.   -1.]
 [  -8.   -7. -105. -102.   -2.   -1.    0.]]
```

IV. Consideration

-> 해당 프로젝트를 진행하면서 Dynamic Programming(DP)을 이용해서 제시된 7x7 grid-world에서 Policy iteration과 Value iteration을 구현해 policy를 최적화하는 코드를 python으로 구현해보았다. 사실 인공지능 이론수업 때 배웠을 때는 별로 어렵지 않게 생각했는데 막상 코딩으로 프로그램을 구현하려니까 굉장히 막막했다. 하지만 인공지능 강의자료와 과제 제안서에 있는 4x4 grid-world 예제를 완벽하게 이해하고 많이 참고하면서 진행하니까 정상적으로 결과가 잘 출력되는 것 같다. 이번 과제를 통해서 python도 계속 공부하면서 여러 메소드를 사용해볼 수 있었고 동적 계획법에 대해서 더 정확히 이해할 수 있었던 것 같다.