A.I assignment2 report

2015147533 유현석

이번 과제는 새로운 파일인 NEW game.py를 사용하였습니다.

1. How did you define the state and reward for this game?

Reachability 함수는 동전과 시계가 도달할 수 있는 새로운 list를 만들어주는 것입니다.

하지만 바구니가 담을 수 있는 아이템의 범위가 정해져 있고 그것을 구해보았습니다. 그 결과가 바로 itembox입니다. itembox의 각각의 11개의 배열은 2개의 값을 가지고 있는데 그 값들 중 왼쪽 값은 바구니가 받을 수 있는 최소의 좌표이고 오른쪽 값은 바구니가 받을 수 있는 최대의 좌표입니다. 예를 들어 5번째에 떨어지는 item은 [2,3]을 가지고 있는데 이 의미는 바구니가 2에 있을 때와 3에 있을 때 받을 수 있다는 의미입니다. 이를 토대로 아이템에 도달하기 위해 바구니의 가로의 이동수가 동전이 바닥에 도달할 때까지의 시간보다 짧다면 도달 할 수 있고 그 반대라면 도달 할 수 없습니다. 이를 토대로 도달할 수 있는 아이템의 좌표들만 filtered로 저장해준 후 return 해줍니다.

```
def reachability(basket_loc, target_locs):
    itembox=[[0,0],[0,1],[0,2],[1,2],[2,3],[3,4],[4,5],[4,5],[5,6],[6,7],[7,8],[7,8]]
    filtered = []
    for tl in target_locs:
        if(itembox[tl[1]][0]>basket_loc):
            if np.abs(itembox[tl[1]][0] - basket_loc) <= 9 - tl[2]:
                filtered.append(tl)
        elif(item abs [1]][1]<basket_loc):
            if np.abs(itembox[tl[1]][1] - basket_loc) <= 9 - tl[2]:
                filtered.append(tl)
        elif(itembox[tl[1]][0]<=basket_loc and itembox[tl[1]][1]>=basket_loc):
            filtered.append(tl)
        #ipdb.set_trace()
    return filtered
```

위에 함수를 토대로 게임 화면상에 존재하는 아이템들 중 도달 할 수 있는 아이템의 좌표만을 get_state 안에서 저장해줍니다. 맨 앞이 1이면 동전 2이면 시계이기 때문에 구분하여 저장해줍니다.

```
## this function return state from given game information.
def get_state(counter, score, game_info):
    basket_location, item_location = game_info
    """
    FILL HERE!
    you can (and must) change the return value.
    """
    coin_locs = [item for item in item_location if item[0] == 1]
    coin_locs = reachability(basket_location, coin_locs)
    #coin_dists = [np.linalg.norm(np.array([item[1],item[2]]) - np.array([basket_location,9])) for item in coin_locs]
    clock_locs = [item for item in item_location if item[0] == 2]
    clock_locs = reachability(basket_location, clock_locs)
    #clock_dists = [np.linalg.norm(np.array([item[1],item[2]]) - np.array([basket_location,9])) for item in clock_locs]
```

위에 도달할 수 있는 좌표들와 지금 현 시점의 바구니의 위치를 비교하여 3가지의 액션을 정할 수 있습니다. 오른쪽으로 이동, 왼쪽으로 이동, 가만히 있기 이렇게 3개입니다. 그렇다면 이러한 경우를 가지고 state를 구분지어 질 수 있습니다. 저는 총 16개의 state를 지정해주었습니다. 먼저 크게 도달할 수 있는 동전의 존재 여부, 시계의 존재 여부로 4개로 구분 질 수 있습니다. 그 후, 각각의 4개의 영역에서 존재한다면 3가지의 액션을 해야하는 상황이 있을 것이고 만약 없다면 취할 수 있는 액션은 존재하지 않을 것입니다. 결국 3*3 + 3*1 + 1*3 + 1*1 = 16개의 stata가 존재하게 됩니다.

도달할 수 있는 여부	동전 존재 O (3가지 액션)	동전 존재 X (액션 없음)
시계 존재 O (3가지 액션)	9가지 경우의 수 (시계와 동전 모두 right, left, stay)	3가지 경우의 수 (시계만 right, left, stay)
시계 존재 X (액션 없음)	3가지 경우의 수 (동전만 right, left, stay)	1가지 경우의 수 (존재 X)

이를 다음과 같은 코드로 표현하였습니다.

1. 시계와 동전 모두 존재하는 9개의 경우의 수

```
coin_state = (coin_loc_array  - np.array([0, basket_location, 9])).tolist()
clock_state = (clock_loc_array  - np.array([0, basket_location, 9])).tolist()

if (len(coin_state) > 0 and len(clock_state)>0):
    coinx=coin_loc_array[0]
    clockx=clock_loc_array[0]
    if(itembox[coinx[1]][0]<=basket_location and basket_location<=itembox[coinx[1]][1]):
      coinmovedir=0
    elif(itembox[coinx[1]][1]<basket_location):
       coinmovedir=1
    elif(itembox[coinx[1]][0]>basket_location):
        coinmovedir=-1

    if(itembox[clockx[1]][0]<=basket_location and basket_location<=itembox[clockx[1]][1]):
        clockmovedir=0
    elif(itembox[clockx[1]][1]<basket_location):
        clockmovedir=1
    elif(itembox[clockx[1]][0]>basket_location):
        clockmovedir=-1
```

```
if coinmovedir > 0 and clockmovedir >0:
    final_state = '0'
elif coinmovedir > 0 and clockmovedir==0:
    final_state = '1'
elif coinmovedir > 0 and clockmovedir <0:
    final_state = '2'
elif coinmovedir == 0 and clockmovedir >0:
    final_state = '3'
elif coinmovedir == 0 and clockmovedir == 0:
    final_state = '4'
elif coinmovedir == 0 and clockmovedir <0:
    final_state = '5'
elif coinmovedir < 0 and clockmovedir >0:
    final_state = '6'
elif coinmovedir < 0 and clockmovedir ==0:
    final_state = '7'
elif coinmovedir < 0 and clockmovedir <0:
    final_state = '8'
```

2. 동전만이 존재하는 3개의 경우의 수

```
elif(len(coin_state)> 0 and len(clock_state)<=0):
    coinx=coin_loc_array[0]
    if(itembox[coinx[1]][0]<=basket_location and basket_location<=itembox[coinx[1]][1]):
      coinmovedir=0
    elif(itembox[coinx[1]][1]<basket_location):
       coinmovedir=1
    elif(itembox[coinx[1]][0]>basket_location):
        coinmovedir=-1

    if coinmovedir > 0:
        final_state = '9'
    elif coinmovedir == 0:
        final_state = '10'
    elif coinmovedir < 0:
        final_state = '11'
```

3. 시계만이 존재하는 3개의 경우의 수 + 아무것도 존재하지 않는 1개의 경우의 수

```
elif(len(coin_state)<=0 and len(clock_state)>0):
    clockx=clock_loc_array[0]
    if(itembox[clockx[1]][0]<=basket_location and basket_location<=itembox[clockx[1]][1]):
        clockmovedir=0
    elif(itembox[clockx[1]][1]<basket_location):
        clockmovedir=1
    elif(itembox[clockx[1]][0]>basket_location):
        clockmovedir=-1
    if clockmovedir > 0:
        final_state = '12'
    elif clockmovedir == 0:
        final_state = '13'
    elif clockmovedir < 0:
        final_state = '14'
else:
    final_state ='-1'


#print(state)
return final_state
```

위에 16개의 state로 나눈 후 행동에 대한 가중치를 주기 위해서 get_reward 함수를 이용하여 결과의 방향성을 지정해주었습니다. 제 코드의 방향성은 시계를 먹는 것이 동전을 먹는 것보다 무조건 우선시되는 reward입니다. 아래의 그림에서도 보다시피 counter: 시간(시계를 먹은 보상)가 증가할 때 기본적으로 5000점이 주어집니다. 만약 동전을 놓치지 않았다면 10000점까지의 보상 역시 받을 수 있습니다. 이와 반대로 동전을 먹었을 때는 100점, 그 동전마저 놓친다면 -100점을 하였습니다. 이를 토대로 동전보다 시계가 중요한 아이템임을 지정해주었습니다.

```
## this function return reward from given previous and current score and counter.
def get_reward(prev_score, current_score, prev_counter, current_counter):
    """
    FILL HERE!
    you can (and must) change the return value.
    (current_score /current_counter) - (prev_score/ prev_counter)
    """
    score =0
    if((current_counter - prev_counter) > 0):

        if((current_score-prev_score)==0):
            score =10000
        else:
            score = 5000
    else:
            if((current_score-prev_score)>0):
                score = 100
            else:
                score =-100

    return score
```

2. How did you modify the policy function / Q-score

The policy function : 이번 과제에서는 Epsilon-greedy policy를 사용한 Q learning을 진행하였습니다. 여기서 Greedy Algorithm이란 미래를 생각하지 않고 각 단계에서 가장 최선의 선택을 하는 기법입니다. 즉, 각 단계에서 최선의 선택을 한 것이 전체적으로도 최선이길 바라는 알고리즘입니다. 미래의 가치를 고려하지 않기 때문에 항상 최선의 결과를 반환하지는 않습니다. 그렇기 때문에 입실론의 중요성이 나옵니다. 일정한 확률로 랜덤으로 state를 선택하여 현재의 가치 뿐만 아니라 미래의 가치를 고려할 수 있습니다.

Q-score: Q-learning을 이용한 강화학습에서 Q-learning에서는 어떤 state에서 어떤 action을 했을 때 그 행동이 가지는 가치를 계산하는데 사용되는 것이 Q-value입니다. 각각의 에피소드에 대해서 종료되기전까지 reward의 총합의 예측 값을 계산하여 시간이 흐른 후에 얻는 보상을 discount factor와 함께 계산합니다. 그 discount factor는 0~1사이의 값으로 현재의 보상과 미래의 얻게 되는 보상을 비교하여 중요도를 따지는 값입니다. 그렇다면 아래의 식과 같이 매 시간마다 agent는 행동을 하고 예측되는 최대의 미래의 값과 discount factor와 보상을 learning rate와의 곱으로 적용시켜 줍니다. 그리고 새로운 행동마다 보상을 받고 새로운 상태로 이동 Q값이 갱신되는데 이때의 Q값이 바로 Q-score가 됩니다. 이전의 값과 새로운 정보의 가중합을 이용하는 기법입니다.

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right)$$

3. Learning parameters (learning rate, epsilon, etc)

Q= train(num_episodes, [epsilon, learning_rate, 0.5]

사용한 parameter 값들은 default 값으로 다음과 같습니다.

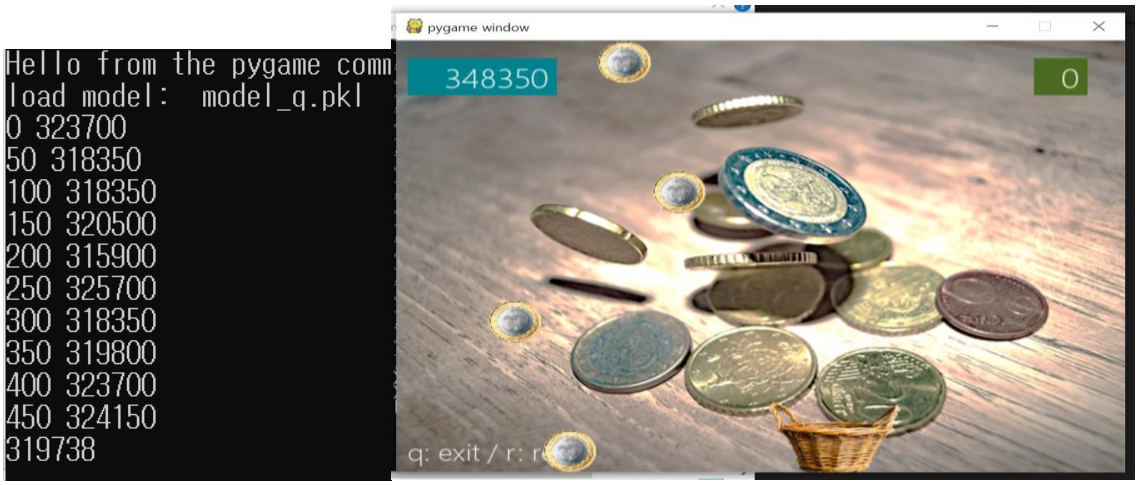
Learning rate: default 0.1 epsilon: default 0.1

주어진 default 값을 사용했음에도 원하던 결과가 잘 나왔습니다.

4. Result and analysis for your agent

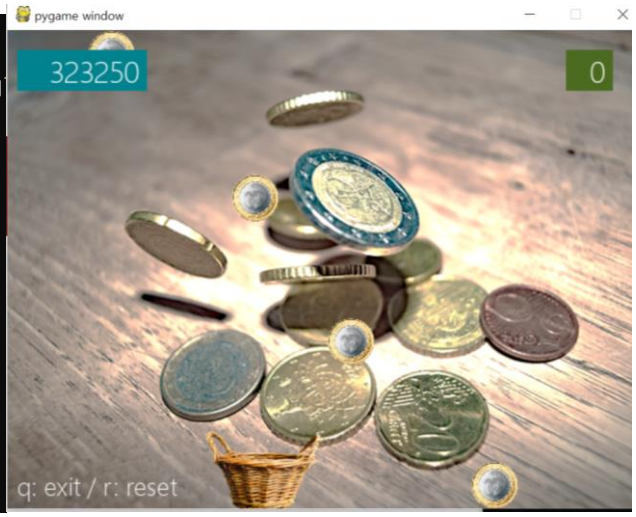
4-1Quantitative results (ex: the avg scores for many games)

N=100 Episode 100/100 (Score: 344250)



N=1000

```
pygame 1.9.6
Hello from the pygame community. h
Episode 100/1000 (Score: 350750)
Episode 200/1000 (Score: 352950)
Episode 300/1000 (Score: 321400)
Episode 400/1000 (Score: 344550)
Episode 500/1000 (Score: 342150)
Episode 600/1000 (Score: 343350)
Episode 700/1000 (Score: 346050)
Episode 800/1000 (Score: 341950)
Episode 900/1000 (Score: 352600)
Episode 1000/1000 (Score: 355700)
```

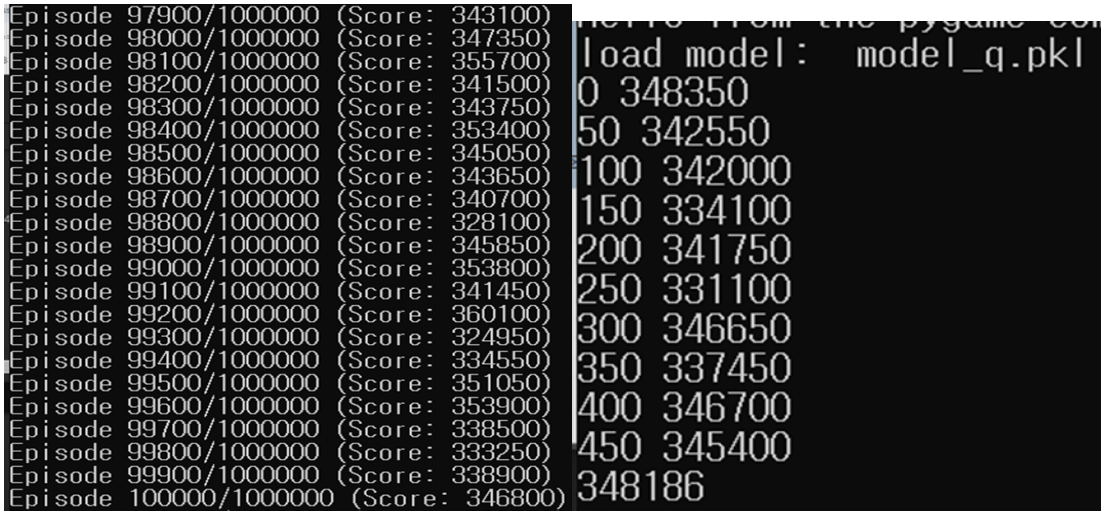



```
load model:  model_q.pkl
0 347850
50 346550
100 352250
150 344250
200 326150
250 348750
300 345850
350 354350
400 352050
450 350300
348090
```

N=10000


```
pygame 1.9.6
Hello from the pygame community. https:
load model:  model_q.pkl
0 341450
50 353050
100 339200
150 352100
200 348900
250 353750
300 349450
350 348450
400 355000
450 352850
348351
```

N=100000



100번의 에피소드와 1000번의 에피소드, 10000번의 에피소드에 대한 결과 값은 위에 그림들과 같습니다. 이를 보시면 100번일때 보다 1000번 100000번일때 평균 점수가 증가함을 알 수 있습니다. 그리고 100번일때에서 1000번까지의 증가폭은 1000번에서 10000번까지의 증가폭에 비해서 낮음을 알 수 있습니다. 이는 1000번에서 충분히 학습이 되었음을 확인할 수 있습니다. 즉 1000번이상에서 state에 대한 reward가 수렴함을 알 수 있습니다. 또한 학습된 에피소드이 숫자에 따라 실제 게임에서 바구니의 움직임에서 차이가 살짝 보였음을 확인할 수 있었습니다. 보다 부드럽고 자연스럽고 놓치는 시계 아이템의 숫자가 적었음을 확인할 수 있었습니다. 후에 100000번을 학습시킨 후 비교를 하였을 때 역시 수렴을 하였는지 결과값에는 큰 차이가 없었습니다.

학습 횟수	100	1000	10000	100000
평균값	319738	348090	348351	348186

또한 결과가 좋을수록 게임 진행되는 시간이 길어 에피소드마다 출력되는 결과값에 도달하는 시간이 길어져 걸리는 시간이 오래되는 것을 확인할 수 있었습니다.

4-2 Analysis for successful case / failure case

중간에 점수가 낮게 나오는 경우가 있습니다. 그러한 경우를 failure case로 볼 수 있는데 이러한 경우에는 시계 아이템이 별로 안 나오는 경우가 있어서 점수를 높이기전에 끝나는 경우도 존재할 거 같습니다. 혹은 시계 아이템에 우선순위를 뒀기 때문에 시계 아이템이 양 끝에서 나오고 많은 숫자의 동전들이 반대편에서 나온다면 점수가 낮게 나올 수 있을 거 같습니다. 하지만 시계 아이템만 충분히 나온다면 점수가 낮게 나올 가능성은 거의 없다고 생각합니다. 시간이 충분하다면 동전을 먹었을 때 500점이 늘어나고 못 먹는다면 -150점이 낮아져 둘 사이에 차이가 크기 때문에 점수를 충분히 나중에 높일 수 있다고 생각합니다.