보고서

박상욱 2020710425

개발환경

- OS: Ubuntu 18.04
- Python: 3.6.10
- IDE : pycharm

성능

대체적으로 공을 잘 받아치나 초기에 공이 랜덤하게 출발할 때 player위치에서 너무 멀어지면 공까지는 완전히 가지 못하는 단점을 지니고 있습니다. Match.py의 episode가 진행될 수 록 random한 player와는 격차가 점점 더 많이 벌어집니다.

모델 설명 Train_Model.py

메인 모델은 다음의 구조로 이루어져 있습니다.

class Model(tf.keras.Model)

tensorflow keras Model을 상속받으며 200의 unit을 가진 2개의 Dense layer(activation relu)와 3개의 유닛을 가진 마지막 logit layer가 존재합니다.

def __init__(self, model_num):

model_num을 입력으로 받아 model의 이름을 설정해 주며 모델을 구축합니다.

def call(self, inputs):

입력을 받아 모델을 포워딩을 하는 method입니다.

def action_value(self, obs):

obs: 입력되는 state입니다. state를 받아와 action을 prediction합니다. remember_buffer에 값을 집어 넣기 위한 prediction의 경우, state가 하나가 들어오며 이때는 action값과 q value를 출력합니다. 그렇지 않고 batch를 하는 경우에는 각각의 state에 알맞는 action 값만 출력합니다.

class DDQNAgent

Double DQN Agent입니다.

def __init__(self, ...):

```
def __init__(self, modelp, target_modelp, modelq, target_modelq,
             env, buffer_size=10000, learning_rate=.0015, epsilon=.1,
             epsilon_dacay=0.995,min_epsilon=.01, gamma=.9, batch_size=64,
             target_update_iter=100, train_nums=10000, start_learning=50,):
```

DDQNAgent의 hyperparameter 등을 설정합니다. 총 episode는 10000번을 돕니다. epsilon-greedy를 위해 epsilon 값과 epsilon_decay를 주고 minimum epsilon 값을 주어 epsilon 값이 특정 값 이하로 떨어지지 않도록 합니다.

생성자에서 optimizer를 주고 keras model을 compile합니다.

```
self.obs = np.empty((self.buffer_size,) + (4 * 12,))
self.actions = np.empty(( self.buffer_size,2), dtype=np.int8)
self.rewards = np.empty((self.buffer_size, 2), dtype=np.float32)
self.dones = np.empty((self.buffer_size, 2), dtype=np.bool)
self.next_states = np.empty((self.buffer_size, ) +(4 * 12,))
self.next_idx = 0
self.player, self.opponent = 0, 1
```

replay buffer를 사용하는 모습입니다. 모델의 입력으로는 4개의 frame을 사용하기 때문에 obs와 next_state는 (buffer_size, 48)의 shape 형태를 갖습니다. 입력 state는 10개의 left player의 state에 right player의 위치값 2개를 추가하여 생성합니다.

def train(self):

학습을 위해 replay_buffer에 값을 집어넣고 reward를 계산하는 부분입니다. 각각의 episode는 done값이 True가 되거나 step_count가 100000번을 넘으면 종료합니다. episode는 총 10000번으로 학습을 진행합니다.

`obs_window, n_obs_window`를 통하여 4개의 frame을 (1, 48) shape으로 만들어 줍니다. 그리고 해당 값으로 action을 뽑은 다음 e-greedy하게 explore를 진행합니다.

```
if reward[0] == 1:
    # 왼쪽이 이기고 오른쪽이 짐
    reward[0] = 1
    reward[1] = -10 * abs(ball_position[0] - obs[-2])
    win_step_count = 0

elif reward[1] == 1:
```

```
    reward[1] = 1
    reward[0] = -10 * abs(ball_position[0] - obs[0])
    win_step_count = 0

else:
    reward[0] = -1 * np.sqrt((obs[0] - ball_position[0])*
                             (obs[0] - ball_position[0]))
    reward[1] = -1 * np.sqrt((obs[10] - ball_position[0])*
                             (obs[10] - ball_position[0]))
```

reward는 위의 코드처럼 계산합니다. 한쪽이 승리하게 되면 패배를 한 다른 한쪽의 player는 떨어진 공의 y와의 L1 거리 만큼의 negative reward를 받게 됩니다. 승패가 결정나지 않은 경우 양쪽 모두 공과의 L2 distance를 계산하여 negative reward를 줍니다.

reward가 계산이 된 다음에는 다음의 코드를 통해 replay buffer에 저장하게 됩니다.

```
self.store_transition(obs_window, action, reward, n_obs_window, done)
self.num_in_buffer = min(self.num_in_buffer + 1, self.buffer_size)
```

학습은 buffer에 값이 충분히 찬 후에 진행합니다. 100 episode 마다 모델의 checkpoint를 저장합니다.

def train_step(self):

해당 method를 통해 실제 학습이 batch별로 이루어집니다. Double DQN을 위해서 action을 선택하거나 action을 평가하는 것을 분리해서 진행합니다. 그리고 나서 학습을 진행합니다.

def store_transition(self, obs, action, reward, next_state, done):

replay buffer에 값을 update 합니다. buffer의 크기는 index를 통해 통제됩니다.

```
n_idx = self.next_idx % self.buffer_size
self.obs[n_idx] = obs
self.actions[n_idx] = action
self.rewards[n_idx] = reward
# next_state = np.asarray(next_state[0] + next_state[1][:2])
self.next_states[n_idx] = np.asarray(next_state)
self.dones[n_idx] = done
self.next_idx = (self.next_idx + 1) % self.buffer_size
```