Deep Q-Networks

한양대학교 최솔비

Real-life value iteration

- Problems
    - the count of environment states and our ability to iterate over them

  ex) Atari 2600 : screen is 210 * 160 = 336000 pixels, and every pixels has one of 128 colors. 128^{33600} states...
  -> cannot calculate all those states and actions.

    - It limits us to discrete action spaces. Not suitable for continuous control problems

  ex) angle of a steering wheel, the force on an actuator, temperature of a heater.

Tabular Q-learning

- 5장의 예제처럼 state space 의 모든 state 들에 대해서 iterate 해야할 필요 없음.
- Just use states obtained from the environment to update values of states.
- 이러한 Value iteration method 를 Q-learning 이라고 한다. (Tabular 는 "표의")

Tabular Q-learning

1. 빈 표에서 시작해서 states 와 values of actions 를 맵핑한다.
2. environment 와 interact 하면서 ,
   $(state, action, reward, newstate)$ tuple 을 얻는다. (이때 어떤 action 을 취할지 결정하는 문제를 exploration versus exploitation problem이라고 함.)
3. $Q(s, a)$ 값을 Bellman approximation 을 이용하여 update 한다.

$$Q_{s,a} \leftarrow r + \gamma max_{a' \in A} Q_{s',a'}$$

4. 2번부터 반복한다.

Tabular Q-learning

- update 가 threshold 보다 적게 되거나, episode reward 가 일정 이상이 되면 완료한다.
- Q-value 를 update 할 때 "blending technique"을 사용하도록 한다. 그냥 표에 덮어쓰게 되면 학습이 unstable 해진다.
- "blending technique"이란, 이전 Q value 와 새로운 Q value 사이에 learning rate (alpha) (0 to 1)를 사용하여 averaging 하는 것이다.
- $$Q_{s,a} \leftarrow (1-\alpha)Q_{s,a} + \alpha(r + \gamma max_{a' \in A} Q_{s',a'})$$
- This allows values of Q to converge smoothly, even if our environment is noisy.

바뀐 알고리즘

1. Start with an empty table for $Q(s, a)$
2. Obtain (s, a, r, s') from the environment
3. Make a Bellman update

$$Q_{s,a} \leftarrow (1-\alpha)Q_{s,a} + \alpha(r + \gamma max_{a' \in A} Q_{s',a'})$$

4. Check convergence conditions. If not met, repeat from step2.

FrozenLake Tabular Q-Learning

```
class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state = self.env.reset()
        self.values = collections.defaultdict(float)
        # value table 만 만들면 됨.
```

```
def sample_env(self):
    # action은 random 하게 고름.
    action = self.env.action_space.sample()
    old_state = self.state
    new_state, reward, is_done, _ = self.env.step(action)
    self.state = self.env.reset() if is_done else new_sta
    return (old_state, action, reward, new_state)
```

```
def best_value_and_action(self, state):
    best_value, best_action = None, None
    for action in range(self.env.action_space.n):
        # 현재 state 에서 취할수 있는 action 들의 Q-value를 얻음.
        action_value = self.values.get[(state, action)]
        if best_value is None or best_value < action_valu
            best_value = action_value
            best_action = action
    # Q-value가 가장 높은 action과 value를 리턴.
    # 만일 table 에 저장된 Q-value가 없다면 0을 리턴.
    return best_value, best_action
```

```
# update values table using one step from the environment
def value_update(self, s, a, r, next_s):
    best_v, _ = self.best_value_and_action(next_s)
    new_val = r + GAMMA * best_v
    old_val = self.values[(s, a)]
    self.values[(s, a)] = (1 - ALPHA) * old_val
                                + ALPHA * new_val
```

- next state 에서 취할 수 있는 best value를 구해서 discount하여 현재 state 의 Q-value 구하는데 사용.

```
def play_episode(self, env):
    total_reward = 0.0
    state = env.reset()
    while True:
        # 현재 value table 에서 best action 을 찾아서 취함.
        _, action = self.best_value_and_action(state)
        new_state, reward, is_done, _ = env.step(action)
        total_reward += reward
        if is_done:
            break
        state = new_state
    return total_reward
```

- This method is used to evaluate our current policy to check the progress of learning.
- This method doesn't alter our value table.

main 함수

- do one step in the environment and perform a value update using the obtained data.

```
while True:
    ...
    # random 하게 action을 실행.
    s, a, r, next_s = agent.sample_env()
    # 결과를 가지고 Q-value table update.
    agent.value_update (s, a, r, next_s)
    reward = 0.0
    for _ in range(TEST_EPISODES):
        reward += agent.play_episode(test_env)
    reward /= TEST_EPISODES
    ...
```

- test 도중엔 Q-values table 을 수정하지 않는다. test 에서 얻어진 samples 을 이용하지 않는다.

Deep Q-Learning

- 앞에서 다룬 일반적인 tabular Q-learning 은 observable set of states 가 너무 클때는 문제 해결이 어렵다.
- state와 action을 value에 맵핑하는 nonlinear representation 구하는 문제를 "regression problem"이라고 한다.
- 이러한 nonlinear representation을 구하기 위하여 아까의 Q-learning 알고리즘에 deep neural network 를 추가하도록 한다.

1. $Q(s, a)$ 초기화
2. environment 와 interact 하여 (s, a, r, s') 튜플 얻기.
3. loss value 계산 : (첫번째 식은 episode 가 끝났을 때)

$$L = (Q_{s,a} - r)^2$$

$$L = (Q_{s,a} - (r + \gamma max_{a' \in A} Q_{s',a'}))^2$$

4. stochastic gradient descent(SGD) 알고리즘을 이용하여 loss value 를 최소화 시키는 방향으로 $Q(s, a)$ 를 업데이트 한다.
5. 수렴할때까지 step2부터 반복.

- 그렇지만 잘 되지 않는다고 한다.

Interaction with the environment

- random 하게 행동해서 이기려면 너무 많은 시간이 걸림. -> Q function approximation 을 사용하도록 하자.
- Q value 가 좋은 data 를 학습시켜야 하는데 Q approximation이 잘못 되면 문제가 생길수 있다. (예를 들면 학습 초반. Q value 를 따르는 것이 의미가 없음)
- exploration versus exploitation dilemma
    - agent 가 environment 를 explore(random) 해야하는 한편, environment와 효율적으로 interact 해야함(Q approximation)
    - random 행동은 학습 초반부에 하여, environment states 에 대한 uniformly distributed information 을 얻는다.
    - Q approximation은 학습 후반부로 갈수록 점점 비율을 높인다.

epsilon-greedy method

- ϵ 을 사용하여 random actions 의 비율을 조정한다.
- The usual practice is to start with $\epsilon = 1.0$ (100% random actions) and slowly decrease it to some small value such as 5% or 2% of random actions.

SGD optimization

- SGD optimization 을 위한 중요 요구조건은 학습 데이터가 independent and identically distributed (i.i.d) 해야한다는 것이다.
- 지금 논의하고 있는 Q-learning 적용방식이 이를 충족시키지 못한다.
    - samples are not independent. 어떤 batch를 학습시킬 때, 그 안의 sample 들이 주로 같은 episode 에 속해있기 때문에 서로 close 하다.
    - 학습 데이터의 분포가 optimal policy의 데이터 분포와 동일하지 않다.
        - supervised learning 하려면 labeled 된 best solution이 있어야하는데 이가 주어지지 않았다는 뜻 같음.

- 이 문제를 해결하기 위해서 replay buffer 라는 past experience 의 sample training data 를 저장하기 위한 큰 버퍼를 사용한다. (latest experience 만 사용하는 것이 아님.)
- replay buffer 를 사용하면, 거의 independent 한 데이터를 훈련할 수 있다.

Correlation between steps

- Bellman equation에서, $Q(s, a)$ 와 $Q(s', a')$ 는 한 step 이기 때문에 차이가 너무 적어져서 neural network이 둘을 구분하기 어렵다.
- network 파라미터 갱신할 때($Q(s, a)$ 를 desired result 와 비슷하게 만드려고), $Q(s', a')$ 및 근처 다른 states 와 연관된 값을 건드리게 된다.
- training 이 unstable 하게 된다. update 한 것이 서로 꼬리를 물게 됨.
- 이를 개선하기 위해 `target network` 를 이용하도록 한다 : keep a copy of our network and use it for the $Q(s', a')$ value in the Bellman equation.
- target network 는 N steps(1k 10k 처럼 큰수) 마다 main network 와 동기화 하도록 한다.

The Markov property

- 논의한 RL method 는 문제가 Markov property를 만족시킨다고 가정하고 있지만 많은 practice 들은 그렇지 않다.
- Pong게임에서도 한번의 observation으로 가장 적절한 action 을 찾는 것이 무리가 있다. 공이 어느 방향으로 가는건지도 알 수 없다.
- 이는 Markov property를 위배하며, 이러한 문제를 partially observable MDPs(POMDP)라고 한다.
- 푸는 문제를 MDP 범주안에 넣어주기 위해서
    - maintain several observations from the past and using them as a state.
    - Atari 게임에서는 여러개의 화면 프레임(보통 4개)을 같이 저장하여 이를 매 state 의 observation으로 사용한다.

The final form of DQN training

알고리즘은 다음과 같다.

1. $Q(s, a)$ 와 $\hat{Q}(s, a)$ 를 위한 파라미터를 랜덤하게 초기화 한다. 입실론은 1.0으로 설정한다. replay buffer 를 비운다.
2. ϵ 의 확률에 따라서 random action a 또는 $a = argmax_a Q_{s,a}$
3. action a 를 실행한 후 , reward 와 next state를 관측한다.
4. (s, a, r, s') transition 을 replay buffer 에 저장한다.
5. replay buffer 에서 랜덤하게 minibatch 를 고른다.

6. buffer 의 매 transition마다

   target $y = r + \gamma max_{a' \in A} \hat{Q}_{s',a'}$ 를 계산한다. (episode 가 끝났다면, $y = r$)

7. loss value 를 계산한다: $L = (Q_{s,a} - y)^2$
8. SGD 알고리즘을 이용하여, loss를 최소화 하도록 $Q(s, a)$ 를 갱신한다.
9. N step마다 Q 에서 $\hat{Q}_t$ 로 weight 을 카피한다.
10. 수렴할 때까지 2부터 반복한다.