ChainMDP

1. 1차& 2차 제출 – PSRL

첫번째와 두번째 제출에서는 tabular 방법인 PSRL 방법을 이용하였다.[1] 알고리즘은 다음과 같다.

Algorithm: Posterior Sampling for Reinforcement Learning (PSRL)

Data: Prior distribution f, t=1
for *episodes* $k = 1, 2, \ldots$ **do**
  sample $M_k \sim f(\cdot|H_{t_k})$
  compute $\mu_k = \mu^{M_k}$
  for *timesteps* $j = 1, \ldots, \tau$ **do**
    sample and apply $a_t = \mu_k(s_t, j)$
    observe r_t and s_{t+1}
    $t = t + 1$
  end
end

처음에 state space S와 action space A에 대한 prior distribution으로부터 학습을 시작한다. k번째 episode마다 history H_{tk}에 conditioned 된 posterior distribution으로부터 MDP M_k를 sampling한다. 그로부터 policy μ_k를 계산하고, 그에 따라 action을 sampling한다. 이 때 Transition posterior와 Reward posterior에 대해서 각각 Dirichlet과 gamma distribution을 이용하였다.

코드에 대한 설명은 다음과 같다. 처음 agent를 생성하면, sample_posterior()를 통해서 transition posterior과 reward posterior가 각각의 distribution에 맞게 생성되고, 이 posterior를 이용해서 sample_posterior_and_update_continuing_policy()를 통해 Q value와 policy pi를 계산한다. 이후 update 과정인 update_after_step() 과정에서는 update_posterior()를 통해 posterior를 update하고 다시 sampling 후 Q value와 policy pi를 계산하는 과정을 거친다.[2]

Performance & Sample Efficiency

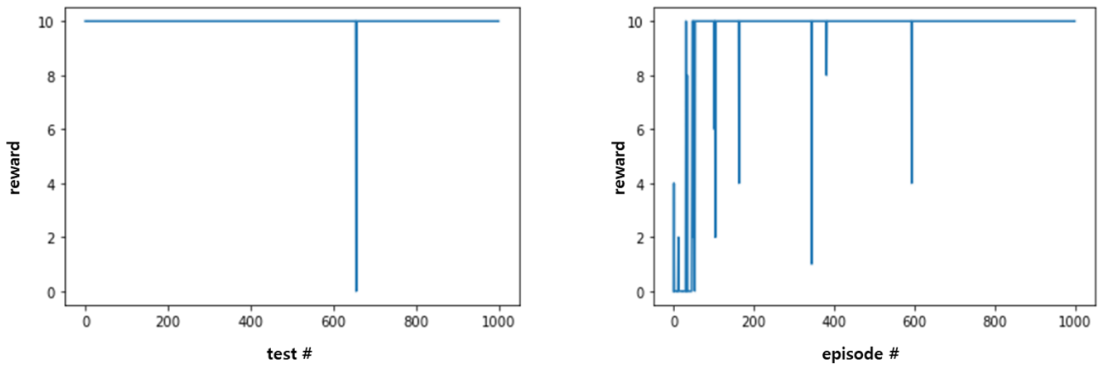


Figure 1. Performance(left) and Sample Efficiency (right)

Performance의 경우 1000번의 test동안 episode 1000에 대해서 학습시킨 후, 각각 학습시킨 agent에 대해서 10번의 episode에 대한 평균 reward를 측정하였다. 평균적으로 최대값인 10의 reward를 가짐을 확인할 수 있다. Sample Efficiency의 경우 9432정도로, 아래 그림과 같이 episode 100개 정도에 대해 확인해 보았을 때 episode 40개 정도 지나면 학습이 거의 완료되는

[1] Ian Osband, Daniel Russo, and Benjamin Van Roy. (more) efficient reinforcement learning via posterior sampling, 2013.

[2] E. Markou. https://github.com/stratisMarkou/sample-efficient-bayesian-rl,2019.

것을 볼 수 있었다.

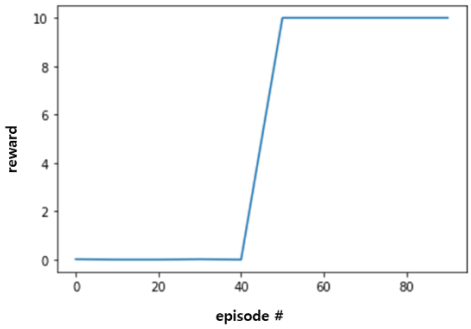


Figure 2. Sample Efficiency for 100 episodes

3차 제출.

간단한 문제였기에 tabular method 를 사용한 1, 2 차 제출을 넘어서 environment 의 변화에 대응가능한 알고리즘의 구축을 위해 function approximation 방법을 사용하였다. Deep exploration 을 위한 tabular method 를 제시한 Ian Osbond 의 이후 논문 “Deep Exploration via Bootstrapped DQN” 논문에서 function approximation method 를 활용한 deep exploration 기법을 제시한 것을 참고해서 코드를 작성하였다.

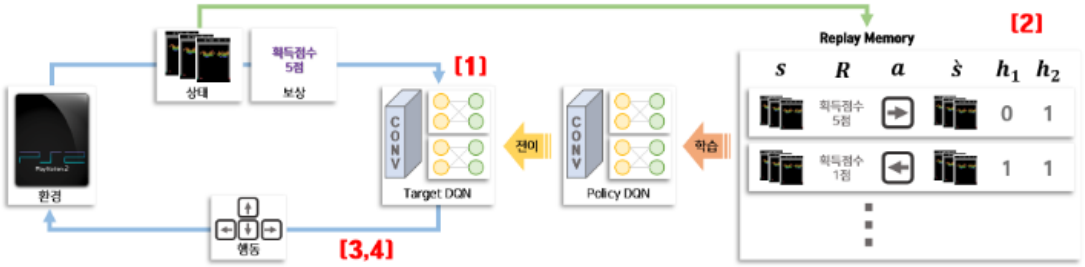
Osbond 는 본인이 제기한 bootstrapped DQN 방법이 다양한 환경에서 기존의 neural net 을 dithering 하는 다양한 기법들을 학습 속도 면에서 압도한다고 제시하였다.

Algorithm 1 Bootstrapped DQN

Input: Value function networks Q with K outputs $\{Q_k\}_{k=1}^{K}$. Masking distribution M.
Let B be a replay buffer storing experience for training.
for each episode **do**
  Obtain initial state from environment s_0
  Pick a value function to act using $k \sim \text{Uniform}\{1, \ldots, K\}$
  for step $t = 1, \ldots$ until end of episode **do**
    Pick an action according to $a_t \in \arg\max_a Q_k(s_t, a)$
    Receive state s_{t+1} and reward r_t from environment, having taking action a_t
    Sample bootstrap mask $m_t \sim M$
    Add $(s_t, a_t, r_{t+1}, s_{t+1}, m_t)$ to replay buffer B
  end for
end for

i

논문에서 제시한 Bootstrapped DQN의 알고리즘은 다음과 같다.
이를 아키텍쳐 형태로 표현하면 다음과 같다.



Bootstrapped DQN 강화학습 과정 Overview

ii

Bootstrapped DQN은 과제 3에서 다루었던 DQN 구조에서 여러 Q network를 추가하면서 Bootstrapping 방법을 적용하여 만든 앙상블 모델이라고 할 수 있다. 따라 구현 상에서 DQN 아키텍쳐와 크게 4가지 부분에서 차이가 있었다.

1. 논문에서는 위 그림 [1]에서 표현한 듯이 state 정보를 Convolutional network 를 통과시켜 feature 벡터로 표현하는 방법을 제시한다. 하지만 본 프로젝트 environment에서 표현하는 state 정보는 압축적으로 정보를 모두 표현하고 그 자체로 feature vector가 되므로 별도의 CNN layer는 생략하였다. 논문에서도 이 프로젝트와 같은 실험에서 똑같은 벡터를 사용하였다고 언급하였다.

2. 가장 특징적인 부분으로 Bootstrapped DQN은 K개의 Target Network-Principal

Network를 유지하면서 bootstrapping을 통해 statistically exploration을 가능하게 한다. 따라 K개의 네트워크 쌍을 조금씩 다르게 학습시켜 exploration을 도모해야한다. 이 학습을 위해 DQN에서 수행한 것 처럼 replay memory를 사용하지만 각 item에 mask 정보를 붙여 어느 네트워크 쌍에 학습에 참여할지 선택한다. 이는 item이 replay memory에 들어갈 때 각 네트워크 쌍에 대해 p확률로 결정된다.

3. 4, DQN에서는 state에서 action을 선택할 때 state의 각 action에 대한 q value를 계산하고 이를 그 최댓값을 선택하거나 ε확률 하에 무작위 액션을 통해 exploration을 수행한다. Bootstrapped DQN에서는 K개의 Q network 쌍에서의 action 선택을 모아 voting하면서 exploration을 수행한다.

따라 이를 구현하기 위해서는 DQN architecture를 사용하되 차이점이 있는 부분들에 대한 적절한 조정이 필요했다. 특징적인 코드는 다음과 같다.

```
class agent():

    def __init__(self, nState, nAction):
        self.nState = nState
        self.nAction = nAction
        self.hidden_dims = [10, 5]
        self.learning_rate = 0.3
        self.targetQ_list = [DQN(self.nState, self.nAction, self.hidden_dims, self.learning_rate) for _ in range(10)]
        self.principalQ_list = [DQN(self.nState, self.nAction, self.hidden_dims, self.learning_rate) for _ in range(10)]
        self.buffer = ReplayBuffer(10000, 10, 0.9)
        self.total_step = 0
```

Agent의 initialize 과정에서 agent가 한 개의 target-pricipal Q network를 가지는 것이 아닌 K(=10)개의 Q network 쌍을 가지게 한다. K가 커짐에 따라 더 Exploration을 잘 할 수 있다 생각할 수 있지만 논문에서는 작은 K로도 충분한 exploration이 가능하다고 제시하였다.
Buffer에 0.9의 확률로 K(=10)개의 네트워크 쌍에 대해 마스킹한다.

```
    def action(self, state):
        vote_list = []
        for Qprincipal in self.principalQ_list:
            Q = Qprincipal.compute_Qvalues(state)
            action = np.argmax(Q)
            vote_list.append(action)

        return 1 if np.array(vote_list).sum() >= 5 else 0
```

Voting을 사용하는 action과정은 다음과 같이 구현하였다. K개의 네트워크가 도출하는 action을 모으고, 과반수 이상인 액션을 선택한다.

```
def update_after_step(self):
    if self.buffer.number < 20:
        return
    samples = self.buffer.sample(5)

    experiences = [[[],[],[]] for _ in range(10)]
    for sample in samples:
        mask = sample[-1]

        for index, item in enumerate(mask):
            if item == 1:
                s = sample[0]
                a = sample[1]
                if sample[4]:
                    d = sample[3]
                else:
                    d = sample[3] + 0.99 * np.max(self.targetQ_list[index].compute_Qvalues(sample[2]))

                experiences[index][0].append(s)
                experiences[index][1].append(a)
                experiences[index][2].append(d)

    for index, Qprincipal in enumerate(self.principalQ_list):
      if experiences[index] != [[],[],[]]:
        Qprincipal.train(np.array(experiences[index][0]),np.array(experiences[index][1]), np.array(experiences[index][2]))

    self.total_step += 1

    if self.total_step % 100 == 0 :
        for i in range(10):
            self.targetQ_list[i].update_weights(self.principalQ_list[i])
```

Masking 과정을 적용한 Q network 학습은 다음과 같다.
Masking이 1로 되어 있다면 해당 Q network의 학습에 참여하는 구조이다.

논문의 구현을 마치고 성능을 체크하였다. 네트워크의 복잡성을 결정하는 hidden dims, learning rate, buffer의 max length, 네트워크 개수 K, 베르누이 확률 p, gamma 등 학습의 성공을 결정할만한 여러가지 파워풀한 하이퍼파라미터가 있었고, 이를 최적화 하기 위해 그리드 서치 등의 방법을 이용해서 실험했으면 좋았겠지만, Network가 꽤 무겁기에 컴퓨팅 리소스 등의 한계가 있었다. 가장 파워풀했던 파라미터는 learning rate로 bootstrapping 과정이 있기에 공격적으로 learning rate를 올려도 수렴성은 유지하면서 exploration을 하고, 학습을 굉장히 빠르게하는 효과가 있었다.

30 episode로 학습한 agent는 performance test에서 500점을 받으며 최적 policy를 학습하고 있음을 확인했다.

[i] Osband, I., Blundell, C., Pritzel, A., & Van Roy, B. (2016). Deep exploration via bootstrapped DQN. *Advances in neural information processing systems*, *29*.

[ii] https://joungheekim.github.io/2020/12/06/code-review/ 참조

Lava

1. **1차 제출(tabular method)**

- Model selection & python code generation

첫번째 제출에서는 일반적인 Q-learning을 사용하여 코드를 구현하였다. 주어진 문제가 discrete한 environment에서 정의되었고 유한한 MDP의 형태를 띄고 있었으므로, function base algorithm은 적절치 않다고 여겨 이와 같은 접근을 시도하였다.

기존 Q-learning 알고리즘의 action 선택 과정 중, epsilon greedy policy를 조정하여 성능 개선을 이루고자 하였다. 이는 코드 구현 과정에서 reference code를 일부 수정하여 구현하였으며, 자세한 설명은 아래와 같다.

```
def action(self, s, greedy = True):

    if type(s) == np.ndarray:
        state_index = np.where(s==1)[0][0]
        s=state_index

    if np.random.uniform(0,1) < self.epsilon/(math.log10(self.episode_num)+1) and not greedy:
        action = random.randint(0, len(self.action_space)-1)
    else:
        q_values_of_state = self.q_table[s]
        max_value = max(q_values_of_state.values())
        action = np.random.choice([k for k, v in q_values_of_state.items() if v == max_value])


    return action
```

Figure 1. Lava action policy code

위 [Figure 1]의 코드는 제출한 agent의 내부 메서드로 작성된 action() 함수이다. 해당 메서드는 인수로 action을 선택하고자 하는 state와 greedy search여부를 받아 선택된 action을 반환한다. 조정된 전략은 두가지로, 첫번째는 epsilon의 값을 episode 단계의 log scale로 decay 시킨다. 이를 통해 episode 진행에 따라 epsilon 값을 줄여가며 Exploration을 감쇄하고자 하였다. 두번째로, episode 300 이후로는 epsilon을 0으로 조정하여 greedy한 search만 진행하였다. 실험 결과 평균적으로 300회 이내 episode에서 모델이 최적경로를 빠르게 찾는 것을 확인하였기에 이처럼 디자인하였다. 하지만 이런 approach는 모델의 robustness에 악영향을 미칠 수 있으므로 이후 제출 과정에서는 삭제를 계획하였다.

- Parameter tuning & result (PF, SE)

Parameter tuning은 1) epsilon, 2) learning rate, 3) gamma에 대해 진행하였으며, 결과적으로 각각 0, 0.1, 1의 값을 주었다. Epsilon의 경우 계획한 것과 다르게 0의 값으로도 충분히 좋은 결과가 나오는 것을 확인하였으며, 이는 문제의 state space가 적고, lava와 마주치기 쉬운 환경에서 epsilon greedy와 greedy 전략의 차이가 크게 발생하지 않았기 때문이라고 여긴다.

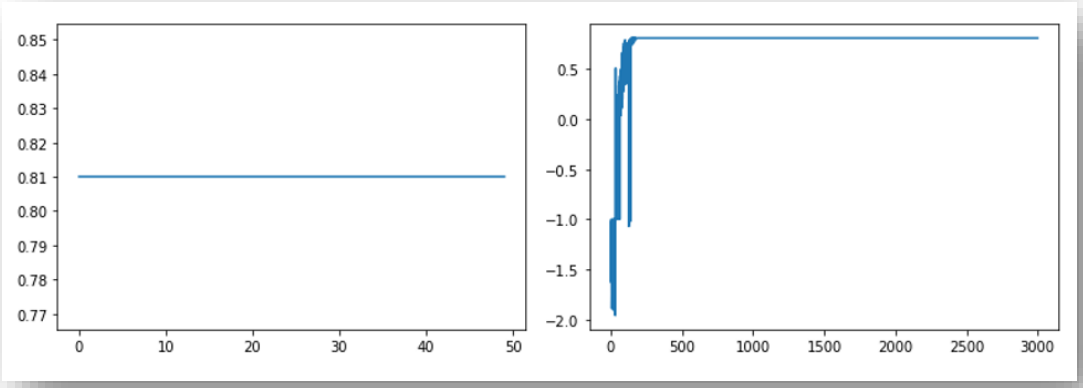


Figure 2. Performance (left), Sample Efficiency (right)

[Figure 2]의 결과는 각각 최종 제출한 Performance와 Sample Efficiency의 샘플링 된 값이다.

Performance의 경우, MDP session과 비슷하게 1000회 test를 진행한 결과 모든 test의 최종 결과는 0.81로 나타났다. 이는 최적 결과이며, epsilon 정책을 사용하지 않았기 때문에 이처럼 일관된 결과가 나온 것으로 해석한다.

Sample Efficiency의 경우 약 2298.02 수준이며, 평균적으로 300회 이전 최적 상태에 도달하는 것으로 확인되었다.

2. 2, 3차 제출 (Functional method)

- Model selection & python code generation

1차 제출 피드백 이후 변화하는 환경에 적응하기 위해서 Functional method의 필요성을 느꼈으며, DQN으로 모델 변경을 진행하였다. 2차 제출기간동안 model building을, 3차 제출기간동안 parameter tuning을 진행하였다.

```
self.model = torch.nn.Sequential(
    torch.nn.Linear(self.nS,self.nA,bias=False)
)
```

Figure 3. Q network

초기에 설정하였던, Q network는 hidden layer 2개를 보유한 NN 모델을 사용하였지만, 이는 지속적으로 goal 조차 찾지 못하는 모습을 보였다. 내부 parameter(loss function, reward, max steps, learning rate, gamma, epsilon, hidden layer)를 바꾸며 시도해보았지만, 유의미한 개선을 발견할 수 없었다. 여러 번의 실험 이후, 간단한 환경에서 빠른 학습을 위해 [Figure 3]와 같이 Q network를 최대한 단순화하여 디자인하였고, 이후부터 goal을 찾기 시작하였다. 이는 앞선 tabular method를 시도한 이유와 일맥상통한다.

Exploration의 경우 Epsilon greedy를 사용하는 대신 행동 선택 확률에 random noise를 주는 방식으로 진행하였다. 더불어, 실제 움직임(Figure 4)과 loss function을 구할 때 maxQ를 계산하는 과정(Figure 5) 각각에 다른 noise parameter를 사용함으로써 능률적인 탐색을 추구하였다. 이 noise vector는 step 값으로 나눠주어 decay 시킴으로써 exploitation의 영향을 강화하였다.

```python
def action(self, state):
    self.step += 1

    if type(state) == np.int64:
        state = np.eye(self.nS)[state]
    state = torch.from_numpy(state).float()

    self.qval_temp = self.model(state)
    qval_ = self.qval_temp.data.numpy()

    # add random noise to actions
    if self.noise1 != None:
        qval_ += np.random.rand(self.nA) * self.noise1/(self.step+1)

    a = np.argmax(qval_)

    return a
```

Figure 4. Agent.action(state) & noise1

```python
def update(self, new_state, action, reward):
    new_state = torch.from_numpy(new_state).float()

    with torch.no_grad():
        newQ = self.model(new_state)

    # add random noise to actions
    if self.noise2 != None:
        newQ += torch.randn(self.nA) * self.noise2/(self.step+1)

    maxQ = torch.max(newQ)
    if reward == -0.01:
        Y = reward + (self.gamma * maxQ)
    else:
        Y = reward

    X = self.qval_temp.squeeze()[action].reshape([1])
    Y = torch.Tensor([Y]).detach()
    loss = self.loss(X, Y)

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Figure 5. Agent.update(ns, action, reward) & noise2

- Parameter tuning & result (PF, SE)

Parameter tuning은 1) learning rate, 2) gamma, 3) noise 1, 4) noise 2에 대해 진행하였으며, 결과적으로 각각 0.005, 0.95, 50, 1의 값을 주었다.

learning rate	0.005	0.01
gamma	0.95	1
noise1	50	100
noise2	None	1

Table 1. Parameter Sample

대상 parameter는 위의 [Table 1]과 같으며, 16가지 모든 조합에 대해 각 10회씩 SE AUC를 계산하였다. 결과는 아래 [Table 2]와 같다.

	learning rate	gamma	noise1	noise2	SE AUC (10 seeds)
2	0.005	0.95	50	1	2171.48
13	0.01	1	50	None	2170.01
6	0.005	1	50	1	2167.638
14	0.01	1	50	1	2166.787
5	0.005	1	50	None	2165.642
15	0.01	1	100	None	2160.866
3	0.005	0.95	100	None	2145.868
16	0.01	1	100	1	2144.14
8	0.005	1	100	1	2144.138
4	0.005	0.95	100	1	2132.663
7	0.005	1	100	None	2125.14
1	0.005	0.95	50	None	1655.582
11	0.01	0.95	100	None	1116.697
10	0.01	0.95	50	1	637.07
12	0.01	0.95	100	1	614.459
9	0.01	0.95	50	None	133.556

Table 2. Sample Efficiency by parameter

위의 결과에 따르면, learning rate가 0.01일 경우 일부 case에서 goal에 도달하지 못해 SE AUC 값이 비교적 적게 나타난 것을 볼 수 있다. 또한, noise 1이 50일 경우 비교적 좋은 결과를 보였으며, noise 2의 경우 존재 여부가 큰 차이를 낳지는 않는 것으로 보였다. 가장 좋은 결과는 상술하였듯, learning rate=0.005/gamma=0.95/noise 1=50/noise 2=1에서 2171.48로 나타났다.

Parameter tuning 결과대로 3000 episode 동안 model을 훈련하고 PF와 SE 값을 비교하였다.

주어진 1, 10, 100, 1000, 10000 seed 값에서 결과를 비교하였을 때, PF는 36.686, SE는 2134.594의 값을 보였다.

Reference: Lava

[1] Watkins, Christopher J. C. H., and Peter Dayan. “Q-Learning - Machine Learning.” SpringerLink. Kluwer Academic Publishers. Accessed June 8, 2022. https://link.springer.com/article/10.1007/BF00992698.

[2] Michaeltinsley. “Michaeltinsley/Gridworld-with-Q-Learning-Reinforcement-Learning-.” GitHub. Accessed June 8, 2022. https://github.com/michaeltinsley/Gridworld-with-Q-Learning-Reinforcement-Learning-.

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).