김범수

DQN

DQN

By POSCAT

Contents

Q-learning + NN

- 지금까지 배웠던 Q-learning의 치명적인 단점 중 하나는 continuous state space에서 적용할 수 없다는 것입니다.
- 예를 들어 자동차 운전에서 자동차의 위치와 속력은 1, 2, 3과 같이 정수가 아니라 1.345, 2.124같은 실수로 표현이 됩니다. 따라서 우리가 지금까지 사용했던 리스트로 action value를 저장하는 방식은 사용하기 어렵습니다.
- continuous state space를 구간을 나누어 불연속적인 것 처럼 취급하는 방법도 존재하지만, 이러한 방법으로는 한계가 존재합니다.
- 따라서 action value를 리스트가 아니라 neural network에 학습시키는 기법을 도입하게 됩니다. Q-learning에 Deep learning을 적용했다고 하여 Deep Q-learning, DQN이라 부릅니다.
- 이번에는 Gym의 acrobot을 DQN으로 풀어보면서 DQN에 대해 배워봅시다.

Q-learning + NN

- acrobot은 기계팔의 관절을 움직여서 로봇팔이 선 위로 넘어가도록 하는 문제입니다.
- 먼저 acrobot의 state는 총 6개의 실수로 구성되며, 팔 각도의 sin값, cos값, 팔 관절의 각속도 등의 정보로 구성됩니다.
- action은 1, 0, -1의 세 종류가 가능하며, 각각 팔 관절에 가하는 힘에 해당됩니다.
- reward는 게임이 종료될 때까지 -1의 보상이 주어집니다.

```python
import gym

env = gym.make("Acrobot-v1")

print(env.observation_space)
print(env.action_space)
```

```
Box(-28.274333953857422, 28.274333953857422, (6,), float32)
Discrete(3)
```

Q-learning + NN

- 먼저 Q-learning의 목적은 현재 state s에서 action a를 취하고, state s'으로 전이했을 때,
  Q(s, a) = R + r * max Q(s', a')의 관계가 성립되도록 action value를 update하는 것입니다.
- 이 사실을 이용하여 neural network를 학습시켜 봅시다.
- 먼저 gym, random, torch, copy 모듈을 가져옵니다. torch.nn은 자주 사용되므로, 편의를 위해 nn으로 가져옵니다.

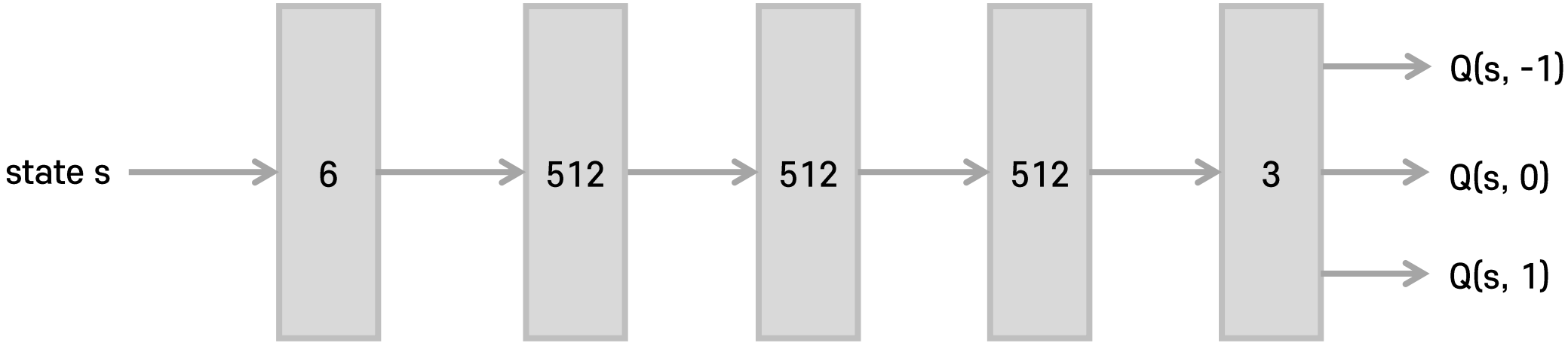
```
1   import gym
2   import random
3   import torch
4   import torch.nn as nn
5   from copy import deepcopy
```

- 그 후 acrobot 환경을 가져오고, 기본적인 상수들을 정의합니다.

```
 7   env = gym.make("Acrobot-v1")
 8
 9   alpha = 0.001 # learning rate
10   gamma = 1 # discount factor
11   max_episode = 10000 # the number of epsiodess
12   epsilon = 0.05 # epsilon for e-greedy
```

Q-learning + NN

- 6개의 실수로 이루어진 state를 입력으로 받아, 3개의 action에 대한 action value를 output으로 하는 NN을 설계합니다.
- 중간에 512개의 node를 가진 3개의 hidden layer를 추가합니다.



Q-learning + NN

- NN을 코드로 구현합니다.
- 게임의 진행 과정은 model의 history라는 리스트에 저장할 예정입니다.

```
19 class DQN(nn.Module):
20
21     def __init__(self):
22         super().__init__()
23         self.layer = nn.Sequential(
24                 nn.Linear(6, 512),
25                 nn.ReLU(),
26                 nn.Linear(512, 512),
27                 nn.ReLU(),
28                 nn.Linear(512, 512),
29                 nn.ReLU(),
30                 nn.Linear(512, 3)
31             )
32
33         self.history = []
34
35     def forward(self, x):
36
37         out = self.layer(x)
38         return out
```

Q-learning + NN

- 편의를 위해 state를 입력으로 받아 greedy action을 반환하는 함수를 제작합니다.
- torch.argmax 함수를 사용하면 최댓값을 가진 원소의 index를 tensor형으로 반환합니다. 예를 들어, t = [1, 0, 2]라는 tensor가 있을 때, torch.argmax(t)는 [2] 가 됩니다. item() 함수로 2라는 값만 가져옵니다.
- action은 [-1, 0, 1]이지만, index는 0~2 사이의 값이므로, 1을 뺀 값을 반환합니다.

```
141  def select_action(state):
142
143      action_value = model(state)
144      action = torch.argmax(action_value).item() - 1
145
146      return action
```

Q-learning + NN

- Q-learning하는 과정을 살펴보면, 먼저 하나의 에피소드(게임)을 플레이하고, 그 에피소드 과정을 바탕으로 action-value를 계산합니다.
- 먼저 에피소드를 플레이하는 것을 구현해봅시다.
- env.reset() 함수로 에피소드(게임)을 시작할 준비를 합니다.
- 그 후 게임이 종료될 때까지 e-greedy로 에피소드를 플레이합니다.

Q-learning + NN

- episode에서 얻은 총 보상을 episode_reward 리스트에 저장합니다. 이는 실제로 학습이 잘 돼서 total reward가 커지는지 확인하기 위해 저장합니다.

```
77   episode_reward = []
78
```

- 먼저 환경을 reset하고, total reward 저장할 변수를 하나 만듭니다.

```
79   for i_episode in range(1, max_episode + 1):
80
81       state = env.reset()
82
83       total_reward = 0
```

Q-learning + NN

- epsilon의 확률로 랜덤한 action을 선택하고, 1-epsilon의 확률로 가장 좋은 action을 선택합니다.

```
85      for i in range(499):
86
87          action = -1
88
89          if random.random() > epsilon:
90              action = select_action(torch.FloatTensor(state))
91          else:
92              action = random.choice([-1, 0, 1])
```

- action에 따라 게임을 진행하고, 그 결과를 model에 저장합니다.

```
94          next_state, reward, done, _ = env.step(action)
95
96          model.history.append((state, action, reward, done, next_state))
97
```

Q-learning + NN

- action을 취한 후에는 total reward를 계산해주고, state를 next state로 옮깁니다. 그 후 done 변수를 확인하여 에피소드를 종료합니다.

```
 98          total_reward += reward
 99
100          state = next_state
101
102          if done:
103              break
```

- 하나의 에피소드가 종료된 후에는 episode_reward에 에피소드의 총 보상을 저장합니다. episode_reward에는 최근 100 에피소드의 총 보상을 저장하며, 에피소드가 끝날 때마다 100 에피소드의 총 보상의 평균을 출력합니다. 만약 평균 총 보상이 -90 이상이 된다면 문제를 풀었다고 정의하고, 모델을 저장한 뒤 학습을 종료합니다.

```
105      episode_reward.append(total_reward)
106
107      if len(episode_reward) == 101:
108          del episode_reward[0]
109
110      print("[Episode {}] Reward : {:.3f}, Average reward : {:.3f}".format(i_episode, episode_reward[-1], sum(episode_reward) / len(episode_reward)))
111
112      if sum(episode_reward) / len(episode_reward) > -90:
113          torch.save(model.state_dict(), "acrobot.pt")
114          print("Clear at episode {}".format(i_episode))
115          break
116
117      finish_episode()
```

Q-learning + NN

- 위 내용을 코드로 구현하면 아래와 같습니다.

```
77  episode_reward = []
78
79  for i_episode in range(1, max_episode + 1):
80
81      state = env.reset()
82
83      total_reward = 0
84
85      for i in range(499):
86
87          action = -1
88
89          if random.random() > epsilon:
90              action = select_action(torch.FloatTensor(state))
91          else:
92              action = random.choice([-1, 0, 1])
93
94          next_state, reward, done, _ = env.step(action)
95
96          model.history.append((state, action, reward, done, next_state))
97
98          total_reward += reward
99
100         if len(model.history) == 100000:
101             del model.history[0]
102
103         state = next_state
104
105         if done:
106             break
```

Q-learning + NN

- finish_episode() 함수에서는 model의 history를 바탕으로 학습을 진행합니다.
- (S, A, R, S')이라는 history가 존재하면, Q(S, A) = R + r * max Q(S', a)가 성립하도록 학습시키는 것이 우리의 목표입니다. 따라서 loss function을 (Q(S, A) - R - r * max Q(S', a))^2 으로 정의합니다.
- 만약 S가 terminal state라면, Q(S, A) = R이 됩니다.
- Adam optimizer를 이용하여 loss function으로 action value를 학습을 진행합니다.

Q-learning + NN

- 우리는 model.history에 (state, action, reward, done, next_state)를 action을 취할 때마다 저장하였습니다. 따라서 model.history는 아래와 같은 모양을 띄고 있습니다.
  [(s1, a1, r2, d1, s2), (s2, a2, r3, d2, a3)..]
- 따라서 model.history 리스트에 있는 각각의 history에 대해 loss function을 계산한 뒤 학습을 진행해야 합니다. 우리의 목표는 Q(s, a) = R + r * max (s', a')가 성립하도록 action value를 정의하는 것이므로, loss function은 아래와 같이 정의됩니다.

$$loss = \sum \begin{cases} \left(Q(s_i, a_i) - r_i - \gamma * \max_a Q(s_{i+1}, a)\right)^2 & if\ d_i\ is\ False \\ (Q(s_i, a_i) - r_i)^2 & if\ d_i\ is\ True \end{cases}$$

Q-learning + NN

- 먼저 각 history에 대한 loss를 저장할 list를 만듭니다.

```
52  def finish_episode():
53
54      loss = []
```

- 모든 history에 대해 loss function을 계산합니다.

```
56      for (state, action, reward, done, next_state) in model.history:
57
```

- 먼저 model을 이용해서 Q(s, a)를 구합니다.

```
58          state_value = model(torch.FloatTensor([state]))
59
60          state_value = state_value[0][action + 1]
```

- 그 후 model을 이용해서 R + r * max Q(s', a')을 계산합니다.

```
62          if done:
63              total_reward = reward
64          else:
65              total_reward = reward + gamma * model(torch.FloatTensor([next_state])).max().item()
66
```

Q-learning + NN

- 따라서 finish_episode()는 아래와 같이 구현할 수 있습니다.

```
47  def finish_episode():
48
49      loss = []
50
51      for (state, action, reward, done, next_state) in model.history:
52
53          state_value = model(torch.FloatTensor([state]))
54
55          state_value = state_value[0][action + 1]
56
57          if done:
58              total_reward = reward
59          else:
60              total_reward = reward + gamma * model(torch.FloatTensor([next_state])).max().item()
61
62          loss.append((state_value - torch.tensor([total_reward])) ** 2)
63
64      loss = torch.stack(loss).sum() / len(loss)
65
66      optimizer.zero_grad()
67      loss.backward()
68      optimizer.step()
```

Q-learning + NN

- loss function을 리스트에 추가합니다.

```
68            loss.append((state_value - torch.tensor([total_reward])) ** 2)
```

- 총 loss function은 각 loss function의 합으로 정의하는 방법과, 각 loss function의 평균으로 정의하는 방법이 있습니다.
  여기서는 loss function의 평균으로 계산합니다. 그 이후, optimizer를 사용하여 학습을 진행합니다.
  해당 에피소드에 대한 학습이 종료된 후에는 history 리스트를 비워줍니다.

```
70        loss = torch.stack(loss).sum() / len(loss)
71
72        optimizer.zero_grad()
73        loss.backward()
74        optimizer.step()
75
76        del model.history[:]
```

Replay memory

- 이렇게 하나의 에피소드가 끝날 때마다 학습을 시켜주면 가장 기본적인 DQN의 형태가 됩니다. 하지만 이러한 학습에는 문제점이 존재합니다. 예를 들어, 마리오 게임을 학습시키는 상황을 생각해봅시다. 먼저 1스테이지에서 여러 번 죽어가면서 1스테이지를 깨는 방법을 익힐 것입니다. 그 후에는 2스테이지에서 여러 번 죽어가면서 2스테이지를 깨는 법을 익힙니다. 하지만, 2스테이지를 깨는 법을 익히면서 1스테이지를 깨는 법을 잃어버리게 됩니다.
- 과거에 학습했던 내용을 잃어버리는 이유는 history를 한 번 학습한 뒤 버리기 때문입니다. 따라서 history를 한 번만 학습하는 게 아니라 여러 번 학습하기 위해 replay memory(buffer)에 에피소드 기록을 저장하고, 여러 번 학습하는 방법을 고안합니다.
- replay memory에 있는 모든 내용을 학습하면 시간이 너무 오래걸리기 때문에, 이 중 랜덤하게 몇 개만 뽑아 학습시킵니다.
- replay memory의 용량을 무한하게 잡을 수는 없으므로, 적당히 크게 잡고, 용량을 초과하면 가장 오래된 데이터부터 replay memory에서 삭제합니다.

Replay memory

- Replay memory의 용량을 100000으로 정하고, 한 에피소드가 끝날 때마다 replay memory에서 1000개의 history를 뽑아서 학습시켜봅시다.
- 먼저 에피소드를 플레이하는 부분은 오른쪽과 같이 변경됩니다.
- 단순히 history의 용량이 1000000이 되면 가장 오래된 history를 제거해 줍니다.

```
100             if len(model.history) == 100000:
101                 del model.history[0]
```

```
85      for i in range(499):
86
87          action = -1
88
89          if random.random() > epsilon:
90              action = select_action(torch.FloatTensor(state))
91          else:
92              action = random.choice([-1, 0, 1])
93
94          next_state, reward, done, _ = env.step(action)
95
96          model.history.append((state, action, reward, done, next_state))
97
98          total_reward += reward
99
100         if len(model.history) == 100000:
101             del model.history[0]
102
103         state = next_state
104
105         if done:
106             break
```

Replay memory

- 학습을 진행하는 finish_episode() 함수는 아래와 같이 변경됩니다.

```
52  def finish_episode():
53
54      loss = []
55
56      for i in range(1000):
57
58          (state, action, reward, done, next_state) = random.choice(model.history)
59
60          state_value = model(torch.FloatTensor([state]))
61
62          state_value = state_value[0][action + 1]
63          if done:
64              total_reward = reward
65          else:
66              total_reward = reward + gamma * model(torch.FloatTensor([next_state])).max().item()
67
68          loss.append((state_value - torch.tensor([total_reward])) ** 2)
69
70      loss = torch.stack(loss).sum() / len(loss)
71
72      optimizer.zero_grad()
73      loss.backward()
74      optimizer.step()
```

Replay memory

- 리스트 내의 모든 history에 대해 학습하지 않고, 랜덤한 1000개의 history를 뽑아 학습시킵니다.

```
56      for i in range(1000):
57
58          (state, action, reward, done, next_state) = random.choice(model.history)
```

- 또한 학습이 끝나고 history를 비우는 코드를 제거합니다.
- 이러한 방식으로 변경하면, 과거의 내용도 여러 번 학습하기 때문에 좀 더 효율적인 학습을 진행할 수 있습니다.
- 하지만 이러한 방식을 도입해도 여전히 문제점은 남아있습니다.

2개의 NN을 이용하여 학습하기

- 우리는 Q(s, a) = R + r * max Q(s', a')가 되도록 학습을 진행합니다. 하지만, Q(s', a')값이 학습을 할 때마다 바뀌게 됩니다. 다시 말하면, Q(s, a)의 학습 기준이 한 번 학습할 때마다 바뀐다는 것입니다. 따라서 학습 자체가 굉장히 불안정할 수밖에 없습니다.
- 우리는 학습을 안정시키기 위해 NN을 2개를 사용할 것입니다. 하나의 NN는 우리가 학습시키는 NN이고, 다른 하나의 NN는 max Q(s', a')을 계산하기 위해 사용합니다. 전자의 NN을 train_model이라 하고, 후자의 NN을 base_model이라 칭하겠습니다.

2개의 NN을 이용하여 학습하기

- 에피소드를 플레이하는 부분은 이전과 똑같이 train_model을 이용하여 게임을 진행합니다.
- 하지만 학습을 할 때, base_model을 이용하여 max Q(s', a')을 계산해 train_model을 학습합니다.
- base_model값도 변경을 해야 하므로, 일정 episode가 끝날 때마다 base_model을 train_model로 변경해줍니다. 여기서는 50 episode마다 base_model을 train_model로 update합니다.

2개의 NN을 이용하여 학습하기

- 먼저 처음에 동일한 내용을 가지는 2개의 NN을 생성합니다.

```
40    base_model = DQN()
41    model = DQN()
42    model.load_state_dict(base_model.state_dict())
```

- 아래와 같이 50 episode마다 base_model을 train_model로 값을 덮어써서 update를 진행합니다.

```
110         if len(episode_reward) == 101:
111             del episode_reward[0]
112
113         if i_episode % 50 == 0:
114             base_model.load_state_dict(model.state_dict())
```

- 학습을 진행하는 코드에서는 max Q(s', a')를 base_model로 계산하도록 변경해 줍니다.

```
63              if done:
64                  total_reward = reward
65              else:
66                  total_reward = reward + gamma * base_model(torch.FloatTensor([next_state])).max().item()
```

DQN 학습 결과

- 전에 말했듯이 학습 종료의 기준은 최근 100 에피소드 total reward의 평균이 -90 이상이 되는 것으로 정의하였습니다. 따라서 학습이 얼마나 빠르게 종료 되느냐를 기준으로 모델의 성능을 파악할 수 있습니다.
- 학습 성능은 learning rate, gamma, epsilon 등에 영향을 받습니다. 그 외에도 episode 당 최대 step의 수 등 여러가지 요인에 의해 변경됩니다.
- Acrobot은 episode 당 최대 step의 수가 500으로 설정되어 있습니다. env의 _max_episode_step 변수를 변경하여 이 값을 바꿀 수 있습니다.

```
18    env._max_episode_steps = 1000
```

- 또한 reward의 정의를 바꾸어 학습을 가속시킬 수 있습니다. 예를 들어, Acrobot에서는 단순히 -1을 주는 게 아니라 로봇 팔이 밑에 있을수록 낮은 점수를 주면, 로봇 팔이 위로 가도록 유도되어 더 빨리 학습이 가능합니다.

연습문제

- 팔의 맨 끝 점의 높이를 reward로 하여 DQN을 학습시켜봅시다. 팔의 맨 끝점의 높이는 어떻게 알 수 있을까요? acrobot의 source code를 통해 알 수 있습니다. source code로부터 팔의 맨 끝점을 구하는 공식을 찾고 이를 계산해 봅시다.
- 우리가 넘어야 하는 선의 높이는 1로 정의되어 있습니다. 따라서 (팔의 맨 끝점) – 1을 reward로 하여 학습을 진행합니다. 이 때, 성공 조건은 최근 100 에피소드의 평균 reward가 -200 이상일 때로 정의합니다.
- 만약 학습에 성공한다면, 해당 모델을 저장해봅시다.
- 새로운 코드를 짜서 저장했던 모델을 불러오고, greedy로 에피소드를 플레이해서 실제로 학습이 되었는지를 확인해봅시다.