### **4.3 강화학습(reinforcement learning) 기반 공정제어**

|  |  |
| --- | --- |
| **학습 내용** | |
| [문제] | Van de Vusse’s reaction 이해 및 강화 학습 기반 공정제어 |
| [방법] | Van de Vusse’s reaction 모델 구축 및 강화 학습을 위한 MDP 정의 |
| [응용] | 강화 학습(DQN 알고리즘)을 이용한 공정제어 |
| [요약] | * Van de Vusse’s reaction 이해 및 모델링 * MDP 설정 및 DQN 이해 * 강화 학습을 사용한 공정제어 |

### **[이론] - Van de Vusse’s reaction control 문제에 DQN 알고리즘 바탕의 강화 학습을 사용한 공정제어.**

Van de Vusse’s reaction control 문제는 1964년에 제안된 isothermal CSTR반응기에서 진행되는 공정으로써 다음과 같은 두 개의 반응이 존재한다. (출처: Van de Vusse, J. G., “Plug-Flow Type Reactor versus Tank Reactor”, *Chemical Engineering Science*, **19**, 964 (1964))

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |

이때, 반응 상수와 농도를 사용하여 mass balance를 통해 해당 공정의 다이나믹스(dynamics)를 표현하면 다음과 같이 표현될 수 있다.

|  |  |
| --- | --- |
|  | (3) |
|  | (4) |

여기서 은 반응물 A의 농도, 는 중간 생성물인 B의 농도, 는 feed에 있는 반응물의 농도, 그리고 는 희석률(dilution rate)이다. 각 상수의 값은 , , , , 그리고 이다.

이번 실습문제에서는 불확실성을 포함하는 시스템을 다루고자 하는데 이를 위해 반응 상수 의 값이 연속균등분포(continuous uniform distribution)를 갖는다고 가정하고 CSTR반응기를 제어하고자 한다. 이러한 시스템 하에서 제어의 목적은 0.1 시간 동안 희석률()을 바꾸어 가며 반응 중간생성물의 농도()를 에 맞추는 것이며, 샘플링 시간은 0.002시간이다. 이때, 현실적이고 부드러운 제어를 위하여 현재의 희석률과 과거의 희석률의 차이(delta)를 줄이는 것도 목적중의 하나이다.

이러한 제어 문제를 풀기 위하여 강화 학습을 도입할 수 있다. 챕터 8에서 강화 학습의 개념이 설명이 되는데, 이는 에이전트가 MDP로 표현된 시스템 하에서 상태(state)를 관찰하여 여러 행동(action)을 취해본 뒤, 받는 보상(reward)을 토대로 스스로 최적의 행동 정책을 찾아가는 기계학습 방법론이다. 이번 예제에서 사용될 강화 학습 알고리즘은 DQN이다. 이는 deep q network의 줄임 말로써, Google의 DeepMind (알파고를 만든 회사)에서 개발한 알고리즘이다. 챕터 8에서 설명한 큐러닝 알고리즘과 같은 off-policy의 TD-learning 알고리즘이지만, 핵심적인 차이는 큐함수를 구함에 있어서 인공신경망을 사용하여 근사치를 구한다는 것이고, 또한 특정 상황에서 행동을 했을 때 받은 결과를 replay buffer에 저장을 한 후, buffer에서 여러 개의 batch를 랜덤하게 뽑아서 학습을 진행한다. 이는 강화학습을 진행함에 있어 데이터의 시계열 적인 상관관계를 깨서 더욱 안정적인 정책 학습을 가능하게끔 한다. 딥 마인드에서 아타리 게임에 DQN을 적용하여 게임 해 나가는 동영상으로 유명한 알고리즘이다 (<https://www.youtube.com/watch?v=V1eYniJ0Rnk>).

문제와 시스템 상황을 명확하게 표현할 수 있는 MDP를 정의하는 것은 강화 학습에서 가장 중요한 일이다. MDP는 챕터 8에서와 같이 상태, 행동, 보상, 상태 변환 함수, 그리고 할인률 5가지로 이루어진다. 본 챕터의 공정 문제를 MDP로 표현한다면 다음과 같이 표현이 가능하다:

* 상태:
* 행동:
* 보상:
* 상태 변환 함수: 본 챕터의 식 (3), (4)
* 할인율: 0.95

본 실습에서는 위와 같이 정의된 MDP에서 DQN 알고리즘이 300번 에피소드 동안 스스로 학습해 나가며 최적의 제어 정책을 찾아가는 과정을 실습할 것이다. 강화학습에서의 에피소드는 의사결정의 시작으로부터 특정한 정책을 가지고 의사결정을 해 나간 뒤, 종결 상태 (terminal state)에 도달하여 의사결정이 끝나기 까지의 과정을 의미한다. 이때, 큐함수는 특정 상태에서 어떠한 정책을 가지고 행동을 하였을 때, 에피소드가 끝나기 전까지 어떠한 보상을 받게 될 지에 대한 함수이다. 따라서, 강화학습에서 에이전트는 현재 상태에서 각 행동의 큐함수를 계산하여 큐함수의 값이 가장 높은 행동을 취하게 된다. 현 제어 문제에서 에피소드는 전체 제어의 지평선(horizon)인 0.1 시간이 끝나는 시점에 종결이 되게 된다. 큐함수는 수학적으로 식 (5)와 같이 표현될 수 있다.

|  |  |
| --- | --- |
|  | (5) |

이번 실습에서 에이전트가 관찰하게 되는 상태는 특정 시간의 농도이고, 할 수 있는 행동은 0~70까지의 5단위의 희석률이다. 일반적인 큐러닝 알고리즘을 통한 큐함수의 업데이트는 수식 (6) 과 같이 표현되지만, DQN의 경우, 큐함수를 근사하기 위하여 인공신경망을 사용하기에, 가 로 표현되는 특징이 있고, 이때 인공신경망의 파라미터인 를 업데이트 하기 위해서 사용하는 오류함수는 식 (7)과 같이 표현된다. 오류함수는 의 형태로 정의되는데 이때, 정답()을 내는 함수가 매 업데이트 마다 값이 계속 변하게 되면 러닝이 안정적으로 되지 않을 수 있기에, 오류 함수에서 정답 부분에 사용되는 인공신경망은 일정 시간 유지하다가 특정한 간격마다 업데이트를 하는 특징이 있다.

|  |  |
| --- | --- |
|  | (6) |
|  | (7) |

### **[문제] MDP로 표현된 Van de Vusse’s reaction control 문제를 코드화 하여 DQN 알고리즘이 학습될 수 있는 기반을 마련하고 실제로 에이전트를 트레이닝 하며 점점 나아지는 제어 성능을 관찰한다.**

- “main.py, Environment.py, Agent.py” 코드를 활용하여라.

- matplotlib, scipy, keras 라이브러리를 설치 하여라

- 챕터 8을 읽고 강화 학습의 이론을 이해하여라.

### **[방법] Van de Vusse’s reaction control 문제를 MDP로 표현하여 강화 학습을 위한 environment 구축 및 DQN 에이전트 셋업**

#### 관련 라이브러리를 설치하고 정의한 MDP를 environment로 코드화 해보자.

1. 필요한 라이브러리 등을 불러온 이후, Env()라는 class를 정의하게 되는데, 이때 class로 객체를 만들었을 때, \_init\_에서 MDP에서 표현한 것과 같이 state은 를 받아드릴 수 있게끔 2개의 차원을 할당, action은 15개의 가능성이 있기에 15개의 차원을 할당, 초기 의값을 설정 sampling time 설정, 그리고 시간을 초기화 한다. 이후 reset이라는 함수를 정의하여 에피소드가 끝난 후, 다시 새로운 환경이 주어질 수 있게끔 한다. 그리고 step, system\_functions, reward\_function이라는 함수를 Env() class 내에 정의하여 MDP를 코드로 표현한다.

|  |
| --- |
| class Env():  def \_\_init\_\_(self):  self.s\_dim = 2  self.a\_dim = 15 *#action (0, 5, 10, ..., 65, 70)* self.x0 = np.array([[0., 0.]])  self.u0 = np.array([[7.]])  self.dt = 0.002  self.t0 = 0.  self.system\_eval = partial(self.system\_functions)   def reset(self):  self.state = self.x0  self.action = self.u0  self.prev\_action = self.u0  self.done = False  self.set\_point = 1.2  self.time = self.t0  self.uncert\_param = np.random.uniform(45., 55.)  return self.state, self.action, self.uncert\_param, self.time   def step(self, u):  x = self.state  t = self.time  u = np.array([float(u)])  dx = partial(self.system\_eval, u=u)  x = np.reshape(x, [-1, ])  sol = solve\_ivp(dx, [0, self.dt], x)  next\_x = np.reshape(sol.y[:, -1], [1, -1])  t += self.dt  self.time = t  reward = self.reward\_function(next\_x, u)  done = self.done\_function(next\_x)  self.state = next\_x  self.prev\_action = u  return next\_x, reward, done, t   def system\_functions(self, t, x, u):  x = x  x1, x2 = np.reshape(x, [-1, ]) *#unpack states* u = np.reshape(u, [-1, ])  u = u \* 5 *# (0, 1, ... 14) -> (1, 5, 10, 15, ..., 7)* k1 = self.uncert\_param *#(h-1)* k2 = 100 *#(h-1)* k3 = 10 *#(l/mol/h)* x1f = 10 *#concentration of the reactant (mol/l)* dx = [- k1\*x1 - k3\*(x1\*\*2) + (x1f - x1)\*u,  k1\*x1 - k2\*x2 - x2\*u]  dx = np.reshape(dx, [1, -1])  return dx   def reward\_function(self, x, u):  x = x  x1, x2 = np.reshape(x, [-1, ])  u = np.reshape(u, [-1, ])[0]  prev\_u = np.reshape(self.prev\_action, [-1, ])[0]  reward = -(x2 - self.set\_point)\*\*2 - 0.01\*(u-prev\_u)\*\*2  return reward |

#### DQN agent를 코드로 표현해보자.

1. DQN class를 정의하기 위하여, 처음에 구동되는 \_init\_함수에는 기본적인 세팅이 들어가야 한다. 상태의 크기, 행동의 크기, 할인율, 그리고 러닝과 관련된 하이퍼 파라미터들(buffer의 크기, 입실론 관련 파라미터들)을 지정해준다. 하이퍼 파라미터인 입실론은 exploration에 초반에 중점을 두는 러닝에서 시간이 감에 따라 exploitation에 중점을 두는 러닝으로 서서히 바뀌게끔 유도를 한다. 이러한 이해를 기반으로 파라미터를 직접 바꾸어 가며 어떻게 러닝이 되는 양상이 바뀌어 가는지에 대한 이해를 늘려가는 것을 추천한다. 이후, DQN의 핵심인 인공신경망을 build\_model 함수에 정의한다. 이때 신경망의 깊이와 node의 개수, activation 함수의 형태 모두 바꿀 수 있는 하이퍼 파라미터이다. 또한 학습을 위한 remember 함수와, 상태를 받아서 어떠한 행동을 하는지에 대한 act함수를 정의함으로써 DQN 에이전트를 정의할 수 있다.

|  |
| --- |
| class DQNAgent:  def \_\_init\_\_(self, state\_size, action\_size):  self.state\_size = state\_size  self.action\_size = action\_size  self.memory = deque(maxlen=2000)  self.gamma = 0.95 *# discount rate* self.epsilon = 1.0 *# exploration rate* self.epsilon\_min = 0.001  self.epsilon\_decay = 0.9995  self.learning\_rate = 0.001  self.model = self.\_build\_model()   def \_build\_model(self):  *# Neural Net for Deep-Q learning model* model = Sequential()  model.add(Dense(24, input\_dim=self.state\_size, activation=**'relu'**))  model.add(Dense(24, activation=**'relu'**))  model.add(Dense(self.action\_size, activation=**'linear'**))  model.compile(loss=**'mse'**,  optimizer=Adam(lr=self.learning\_rate))  return model   def remember(self, state, action, reward, next\_state, done):  self.memory.append((state, action, reward, next\_state, done))   def act(self, state):  if np.random.rand() <= self.epsilon:  return random.randrange(self.action\_size)  act\_values = self.model.predict(state)  return np.argmax(act\_values[0]) *# returns action* |

### **[응용] 정의된 MDP내부에서 에피소드를 진행하며 강화 학습 기반 공정제어를 진행해보자.**

#### 한 에피소드 동안 특정 정책을 가지고 환경(environment)와 상호작용을 통해 정책 업데이트를 위한 reply buffer를 구축해보자.

1. 한 에피소드 내에서 상태, 행동 등을 reset한 뒤, 500번 동안 현재 가지고 있는 정책을 가지고 행동을 하여 그때 받은 보상들을 data\_history (replay buffer)에 차곡차곡 저장을 한다. 이때, 모은 데이터가 batch의 사이즈를 넘어가면 모인 데이터를 바탕으로 정책 업데이트를 진행한다. 또한 한 에피소드가 진행됨에 따라 얻어진 reward의 총 합은 score라는 변수 아래에 저장이 된다. (score에 저장되는 값은 강화학습에서 return이라고 일반적으로 지칭한다)

|  |
| --- |
| state, action, uncert\_param, t = env.reset() state = np.reshape(state, [1, state\_size]) score = 0 ep\_data\_history = np.zeros([1, 2 \* state\_size + 3]) *# time, state, action, next\_state, reward* for n in range(500):  action = agent.act(state)  next\_state, reward, done, t = env.step(action)  next\_state = np.reshape(next\_state, [1, state\_size])  agent.remember(state, action, reward, next\_state, done)  temp\_data\_history = np.append(t, state)  temp\_data\_history = np.append(temp\_data\_history, action\*5)  temp\_data\_history = np.append(temp\_data\_history, next\_state)  temp\_data\_history = np.append(temp\_data\_history, reward)  temp\_data\_history = np.reshape(temp\_data\_history, [1, -1])  ep\_data\_history = np.concatenate([ep\_data\_history, temp\_data\_history])  state = next\_state  score += reward   if len(agent.memory) > batch\_size:  agent.replay(batch\_size) |

#### 정책을 업데이트 해가며 300회 에피소드를 진행하고, 각 에피소드의 공정 제어 결과를 저장하여라.

1. env 라는 환경을 모사한 class와 agent라는 DQN 에이전트를 생성을 한다. 이후, reply buffer에서 몇 개를 뽑아서 정책 업데이트를 진행할지에 대한 하이퍼 파라미터인 batch\_size를 정의하고, 총 에피소드의 수를 300으로 결정한다. 이후 에피소드의 횟수만큼 A3. 의 코드를 진행하여 공정제어와 정책 학습을 반복해 나간다.

|  |
| --- |
| env = Env() state\_size = env.s\_dim action\_size = env.a\_dim agent = DQNAgent(state\_size, action\_size) *# agent.load("./save/project.h5")* done = False batch\_size = 32 EPISODES = 300 RANDOM\_SEED = 1234 np.random.seed(RANDOM\_SEED) ep\_score\_history = np.array([])  for e in range(EPISODES):  A3.의 코드 |

#### 1번째 에피소드, 101번째, 201번째, 300 번째 에피소드의 공정 제어 결과 (시간에 따른 score, , 의 그래프)를 나타내어라.

1. 아래와 같은 코드를 활용하여 1번째, 101번째, 201번째, 그리고 300번째 에피소드의 score값, ,의 시간에 따른 변화 값을 프린트 할 수 있다.

|  |
| --- |
| if done:  print(**"episode: {}/{}, score: {: .3}, e: {: .3}"**.format(e, EPISODES, score, agent.epsilon))  ep\_score\_history = np.append(ep\_score\_history, score)   plt.subplot(311)  plt.title(**"score"**)  plt.scatter(e, score, c=**'blue'**)   if (e % 100 == 0) or (e == EPISODES-1):  t\_history = ep\_data\_history[:, 0]  x2\_history = ep\_data\_history[:, 2]  u\_history = ep\_data\_history[:, 3]  plt.subplot(312)  plt.title(**"x2"**)  plt.plot(t\_history, x2\_history, label=(e+1))  plt.legend()  plt.ylim(0, 1.5)  plt.axhline(y=1.2, color=**'r'**, linestyle=**'--'**, linewidth=1)   plt.subplot(313)  plt.title(**"u"**)  plt.step(t\_history, u\_history, label=(e+1))  plt.legend() |

첫번째로 볼 수 있는 에피소드에 따른 score 값의 변화는 에피소드의 번호가 뒤로 감에 따라 (학습이 더 진행된 정책기반 의사결정) score 값이 증가함을 알 수 있다. 즉, 점점 reward를 많이 받는 행동을 매 의사결정마다 자주 했었다는 의미이고, reward를 많이 받았다는 것은 우리가 처음에 설정한 목적을 더욱 잘 달성했다는 의미이다. (그림1)

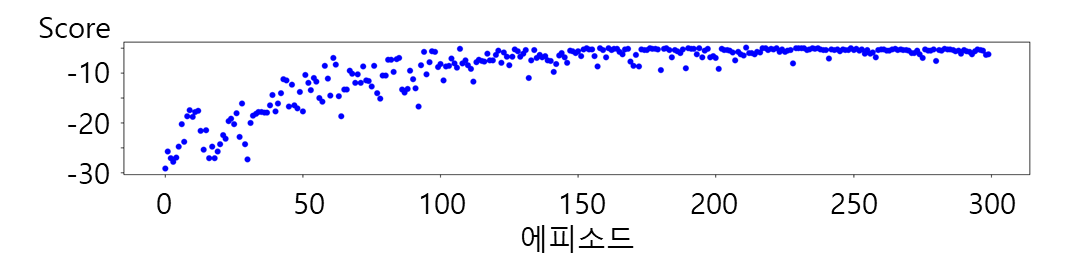


그림1. 에피소드 별 score 값

두번째로 불 수 있는 의 경우, 첫번째, 101번째, 201번째, 300번째의 에피소드로 학습이 진행됨에 따라 점점 기존의 set point였던 1.2 라는 값에 빠르고 안정적으로 안착을 하는 모습을 볼 수 있다. (그림2)

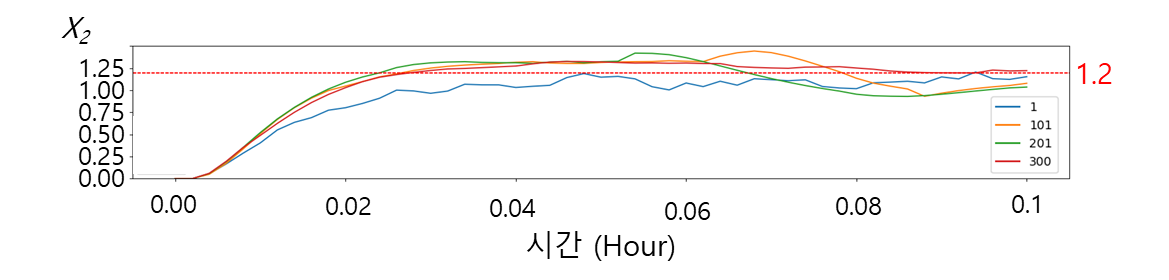


그림2. 에피소드 마다 시간에 따른 의 값

마지막으로 볼 수 있는 의 경우 1번째 에피소드의 경우 의 값이 각 스텝마다 굉장히 급격하게 바뀌는 모습을 솔 수 있지만, 101번, 201번, 300번 에피소드로 정책이 진화함에 따라 굉장히 부드럽고 천천히 바뀌기에, 안정적인 제어를 할 수 있는 노하우를 DQN 에이전트가 터득했다고 볼 수 있다. (그림3)

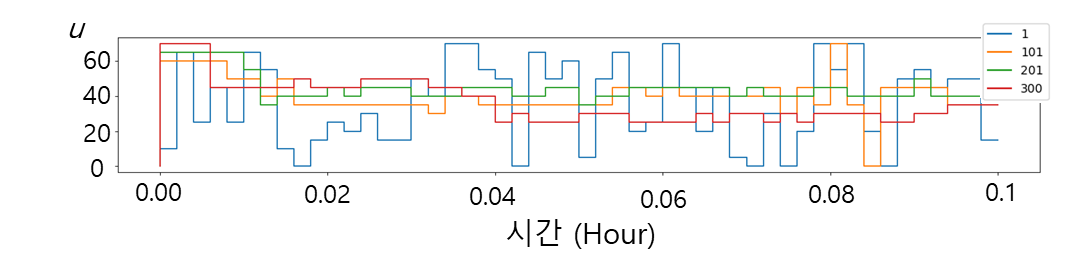


그림3. 에피소드마다 시간에 따른 의 값

### **[결론]**

본 장에서는 강화학습 기반 공정 제어를 Van de Vusse’s reaction control 문제에 적용하여 공정제어와 에이전트 학습을 실제로 진행해보았고, 이러한 공정 제어 결과를 에피소드 별로 plot 하여 에피소드가 진행함에 따라 처음 설정한 목적에 맞는 제어 방식을 강화학습 에이전트가 실시간으로 배워 나가는 과정을 몸소 확인해보는 공부를 하였다.

### **[학습 결과]**

* 학습 내용

Van de Vusse’s reaction control 문제를 이해하고 MDP로 코드화 시켜 보기

* 학습 결과 확인하기

DQN 알고리즘 기반 강화학습 에이전트를 구성해보기.

* 학습 결과 응용하기

생성된 강화학습 에이전트를 Van de Vusse’s reaction control 문제에 적용하여 공정 제어 퍼포먼스를 실시간으로 확인해보기