Reinforcement Learning Model of Automatic Car Parking

자율주차 문제에 대한 강화학습 모델

https://github.com/KimDomiz/22-2-Reinforcement-Learning/tree/main/Final%20Project%20Team8

2018312856 고성원

2018314103 김민성

2018313350 손다인

2018311438 유현지

목차

1. 문제 정의

1 가구당 자가용의 수가 늘어나고 있고, 인공지능이 등장함에 따라 자율주행 기술이 나날이 발전하고 있지만 여전히 안정성에 대한 의문이 있습니다. 자율 주행 기술은 기술 수준에 따라 1단계부터 5단계까지 나뉘는데, 완전한 자율주행 단계인 5단계 자율주행 까지는 현재 상용화가 잘 되지 못하고 있습니다. 이에 대한 이유로는 기술적인 어려움도 있지만, 운전 중에 겪을 수 있는 여러가지 돌발 상황에 대한 대처가 미흡할 것이라는 이용자들의 불신도 크게 작용하고 있습니다. 저희는 이용자들이 운전하면서 어려워하는 일 중 하나인 주차 문제를 여러가지 강화학습 알고리즘으로 해결해보기로 했습니다. 현재까지의 자율 주차 연구는 인간이 만든 공식을 적용하는 것도 많습니다. 따라서 주행의 원리에 대한 수학 공식을 만들어서 예측 모델에 적용하는 것이 나은지, 강화학습 알고리즘으로 최적화를 시키는 것이 나은지 비교할 것입니다. 또한 실생활과 유사한 주차 환경과 주행 상황에서 어떤 알고리즘이 잘 작동하는지, 장애물이 존재할 때 어떤 알고리즘이 잘 작동하는지를 비교해보고 최적의 모델을 구현해보기로 하였습니다.

2. 관련 연구

먼저 강화학습 기반 자율주차 연구를 위한 시뮬레이터를 개발한 연구 (Eom et al.) 가 있습니다. 본 연구는 애커만 조향 기하학 모델이 제안하는 수식을 기반으로 실제 직각주차 상황과 차량의 물리적 특성을 고려하여 설계하였습니다. DDPG알고리즘을 구현된 시뮬레이터에 적용하여 효과적인 학습이 가능함을 입증하였습니다.

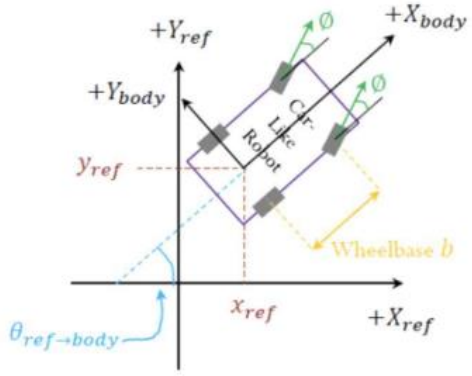


그림 2. 유클리디안 공간상의 애커만 조향 기하학 모델
Fig. 2. Ackerman Model in Euclidean Space

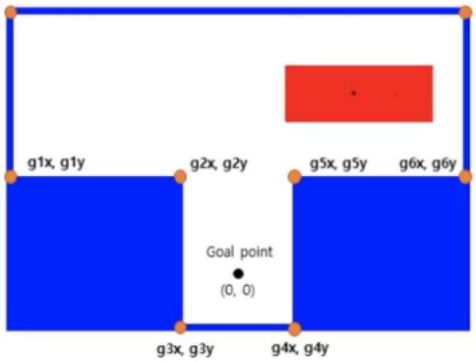
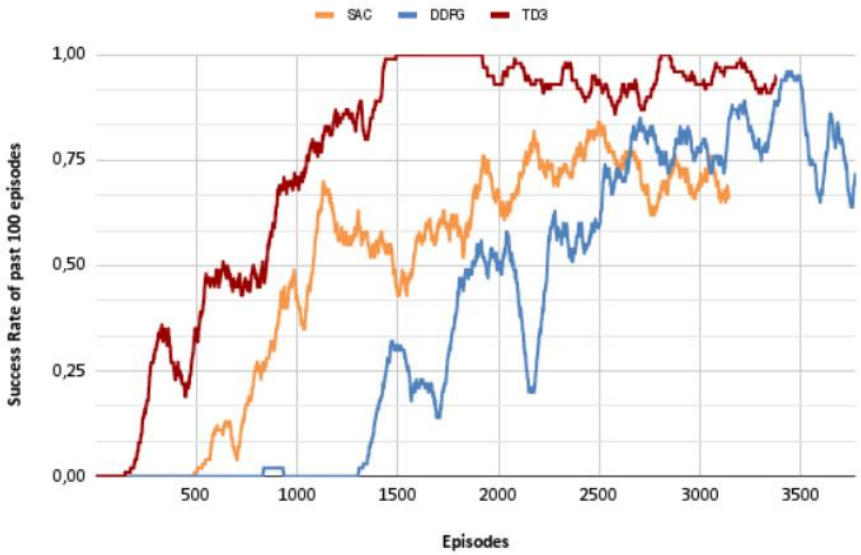
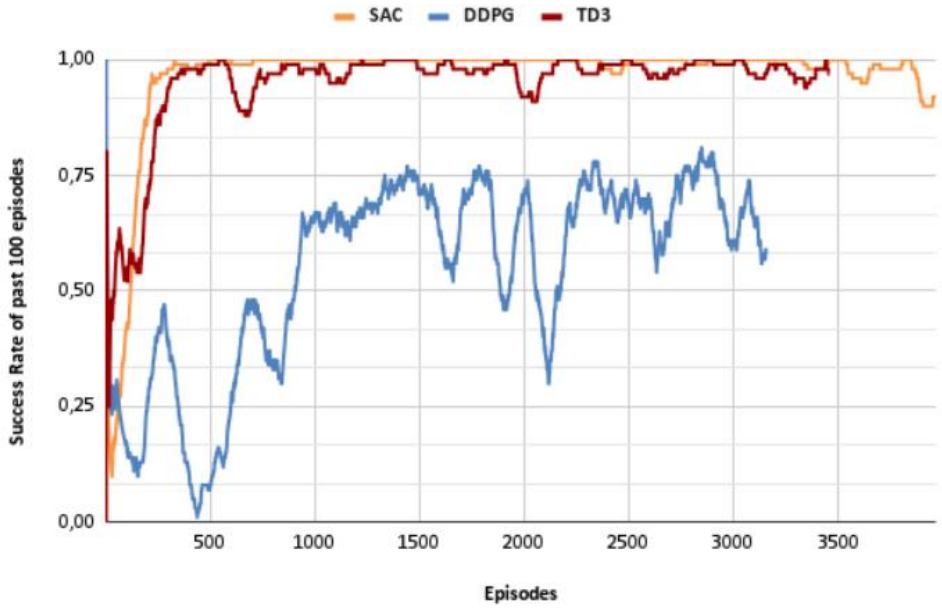


그림 3. 자율주차 시뮬레이터 화면 (직각주차)
Fig. 3. Autonomous Parking Simulator (Perpendicular Parking)

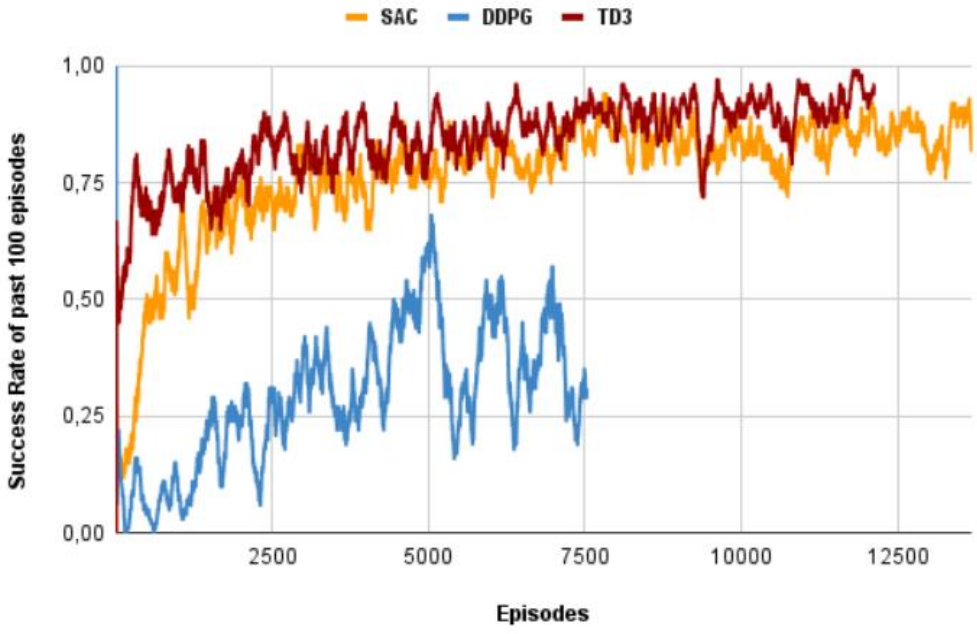
차량이 사전 정의된 경로를 따를 수 있는 방법론을 개발하는 것은 어려운 작업입니다. 또한 주변 장애물과의 충돌을 방지하기 위해 경로와 일관된 거리를 유지할 수 있어야 합니다. 이러한 특성을 가진 사전 정의된 경로를 통해 제한된 공간 내에서 차량을 안내하는 방법을 심층 강화학습으로 에이전트를 가르치는 방법이 제안되었습니다. (Moreira) Soft Actor Critic, Deep Deterministic Policy Gradient, Twin Delayed Deep Deterministic Policy Gradient인데 각 알고리즘의 성능을 비교한 결과는 다음과 같습니다.



[SAC, DDPG 및 TD3 알고리즘을 사용하여 트레이닝 1단계에서 100회를 고려한 성공률]



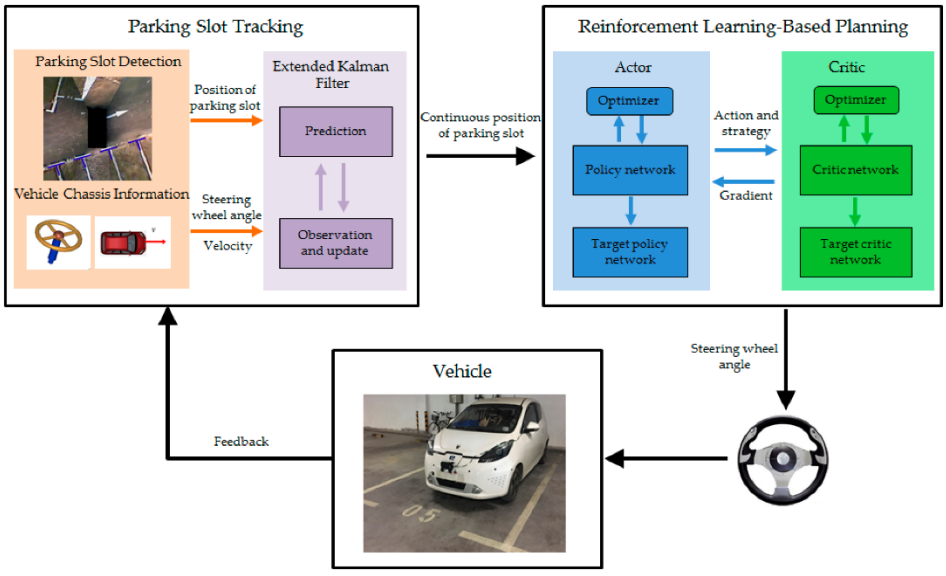
[SAC, DDPG 및 TD3 알고리즘을 사용하여 트레이닝 2단계에서 100회를 고려한 성공률]



[SAC, DDPG 및 TD3 알고리즘을 사용하여 트레이닝 3단계에서 100회를 고려한 성공률]

또한 자율주행 주차를 하기 위해 강화학습 기반의 종단간 주차 알고리즘을 제안한 연구도 있습니다. (Zhang et al.) 기존의 주류 자동 주차 시스템에 따르면, 먼저 센서가 감지한 주차 공간을 기반으로 주차 경로를 계획하고 따

라갑니다. 그러나 차량이 비선형 동적이기 때문에 경로 추적 오류가 발생할 수밖에 없고 주차 기울기 및 편차가 발생합니다. 이에 대해 종단간 주차 알고리즘을 제안하는데, 차량은 수많은 주차 시도를 통해 지속적으로 학습하고 경험을 축적한 다음 다양한 주차 공간에서 최적의 각도 명령을 학습할 수 있습니다. 이러한 종단 간 주차를 기반으로 경로 추적으로 인한 오류를 방지할 수 있습니다. 또한 학습과정에서 주차 공간을 지속적으로 획득할 수 있도록 vision과 차량 샤시 정보의 조합을 기반으로 주차 공간 추적 알고리즘을 제안했습니다. 또한 학습 네트워크 출력이 수렴하기 어렵고 주차 과정에서 local minimum에 빠지기 쉽다는 점을 고려하여 주차 조건 측면에서 몇 가지 강화 학습 훈련 방법이 개발되었습니다.



[Overview of the reinforcement learning-based end-to-end parking method]

3. 제안 방법

저희는 highway-env의 parking-v0 환경을 사용하여 현재 공개되어 있는 여러 알고리즘들을 적용해보았습니다. 실험에 사용된 알고리즘들은 Deep Q-Network(DQN), Deep Deterministic Policy Gradient(DDPG), Dynamics + Cross Entropy Method(CEM) 입니다. 세 알고리즘을 선택한 이유는, DQN은 강의에서 가장 친숙하게 배운 알고리즘이었기에 parking에 적용할 때 어느 정도의 성능을 낼 수 있는지 확인하고 싶었습니다. DDPG와 CEM같은 경우는 실험 과정에서 발생한 DQN 알고리즘의 한계점들을 다른 알고리즘을 통해서 해결해 보기 위해 선정하였습니다. DDPG는 DQN에서 continuous action space를 실행할 수 있도록 발전된 코드이고, CEM은 완전 다른 방향에서의 접근이었습니다. 이 알고리즘들은 추후 코드와 함께 서술하겠습니다.

4. 코드 수정

알고리즘 코드들을 parking-v0에 적용하는 과정에서 코드가 제대로 실행되지 않는 문제가 많이 발생하였습니다. Highway-env의 코드와 stable-baseline3 모듈의 상세 코드들을 확인하여 문제의 원인을 파악했습니다. 주된 이유는 highway-env와 다른 알고리즘 코드들 간의 버전 차이에서 오는 충돌이었습니다. 여러가지 버전의 문제점 때문에 계속해서 오류가 발생해서 저희는 colab이 아닌 vscode를 통해 로컬로 코드를 실행해야 했습니다. 아래는 colab에서 실행할 시 나오는 오류 메시지입니다.

```
stable-baselines3 1.7.0a5 requires gym==0.21, but you have gym 0.26.2
which is incompatible.
```

오류 내용을 보면 stable-baseline3는 gym 0.21버전을 지원하지만 highway-env는 gym 0.26.2에서 제대로 동작하였습니다. 따라서 저희가 원하는 알고리즘들을 실행하기 위해선 모듈의 상세 내용을 직접 수정해야 했습니다. 첫 번째로 수정한 파일은 다음과 같습니다.

stable_baselines3\common\vec_env\dummy_vec_env.py

dummy_vec_env 파일은 SB3 모듈에서 observation을 처리하는 기능을 가지고 있습니다. 원래 parking-v0의 env는 observation 값으로 딕셔너리 형태의 값을 반환합니다. 이전 버전의 stable-baseline이라면 문제없이 딕셔너리 값을 처리하지만, 해당 버전은 더 이상 사용할 수 없기 때문에, 딕셔너리 형태의 입력 값을 처리하도록 수정하였습니다.

```
def _save_obs(self, env_idx: int, obs: VecEnvObs) -> None:
    for key in self.keys:
        if key is None:
            self.buf_obs[key][env_idx] = obs
        else:
            if len(obs) == 2:
                self.buf_obs[key][env_idx] = obs[0][key]
            elif len(obs) == 3:
                self.buf_obs[key][env_idx] = obs[key]
```

else: 이후 부분이 수정한 부분입니다. parking-v0의 obs가 상황에 따라 다른 딕셔너리 값들을 반환하기 때문에 해당 부분을 수정하였습니다. 다음은 두 번째 수정 파일입니다.

stable_baselines3\common\monitor.py

해당 부분은 env.step을 할 때 parking-v0이 다른 환경들과는 달리 인자를 하나 더 반환하기 때문에 아래와 같이 코드를 수정했습니다.

```
if self.needs_reset:
    raise RuntimeError("Tried to step environment that needs reset")
observation, reward, done, _, info = self.env.step(action)
self.rewards.append(reward)
```

_로 인자를 하나 추가해줘서 오류가 발생하지 않도록 하였습니다. 그 외에도 코드 실행 부분에서 이전 모듈을 사용했던 것을 교체하거나, input값들을 수정하여 로컬에서는 코드 실행이 원활하도록 만들었습니다. 다음은 저희 실험에 사용하기

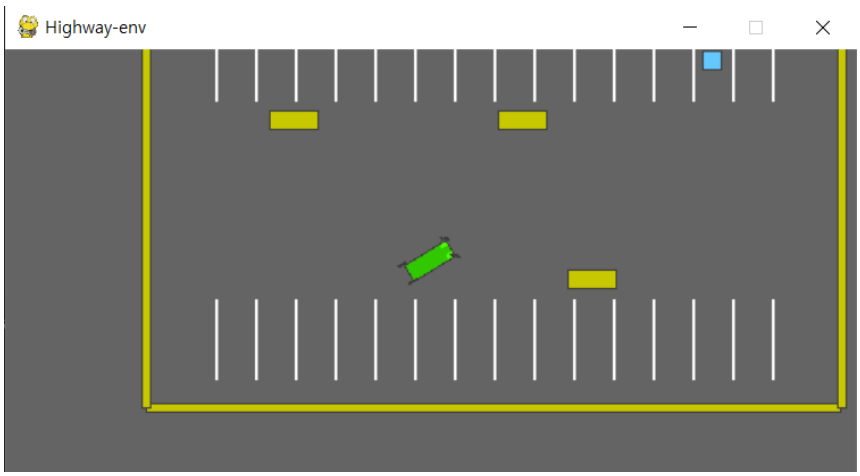
위해 수정한 사항입니다.

highway_env\envs\parking_env.p

이번에 수정한 파일은 parking_env 파일입니다. 저희가 원하는 환경은 아무것도 없는 상황, 사람들이 있는 상황, 자동차들이 주차된 상황이었습니다. 이를 위해서 parking_env 파일에 obstacle들을 추가해서 환경을 만들었습니다.

```
obstacle1 = Obstacle(self.road, [10, 8], np.pi)
obstacle1.LENGTH, obstacle1.WIDTH = (5, 2)
obstacle1.diagonal = np.sqrt(obstacle1.LENGTH**2 + obstacle1.WIDTH**2)
self.road.objects.append(obstacle1)

obstacle2 = Obstacle(self.road, [3, -8], np.pi)
obstacle2.LENGTH, obstacle2.WIDTH = (5, 2)
obstacle2.diagonal = np.sqrt(obstacle2.LENGTH**2 + obstacle2.WIDTH**2)
self.road.objects.append(obstacle2)

obstacle3 = Obstacle(self.road, [-20, -8], np.pi)
obstacle3.LENGTH, obstacle3.WIDTH = (5, 2)
obstacle3.diagonal = np.sqrt(obstacle3.LENGTH**2 + obstacle3.WIDTH**2)
self.road.objects.append(obstacle3)
```

Obstacle()이라는 함수를 활용하여 각각의 장애물들을 만들어주었습니다. self.road는 parking 환경이 정의된 road인자를 말하는 것이고 [10,8]은 해당 장애물의 위치, np.pi는 해당 장애물이 보는 방향입니다. 해당 코드를 통해 아래 사진처럼 여러가지 위치에 다양한 크기의 장애물들을 배치할 수 있습니다.



지금까지 언급한 코드 수정 외에도, 다양한 부분에서 오류가 발생하여 그때마다 오류를 수정하는 방법으로 문제를 해결했습니다. 따라서 미처 언급하지 못한 부분이 있을 수 있습니다. 결론적으로, 위의 오류사항들을 모두 수정하면 colab 환경에서도 잘 작동합니다. 하지만 colab은 gym.wrappers의 RecordVideo를 활용하여 시각화를 할 수 없습니다. colab에는 display 장치가 없기 때문에 실험 결과를 시각적으로 볼 수 있는 방법이 없었습니다. 따라서 저희는 위의 모든 코드들을 수정한 뒤, 실험 결과의 시각화까지 정상적으로 작동하는 로컬에서 모든 실험을 진행하였습니다.

5. Highway-env

저희가 실험에 사용한 환경은 open AI gym의 third-party인 highway-env입니다. 자동차가 처할 수 있는 여러가지 상황을 구현하여 기계학습을 편하게 할 수 있도록 만들어진 환경입니다. 사용할 수 있는 환경의 종류는 차선을 변경할 때 사용할 수 있는 highway, 차선이 합쳐지는 상황인 merge, 교차로 환경인 intersection 등 여러가지 환경이 존재하며, 저희는 parking-v0이라는 자율주차 환경을 사용했습니다.

Parking env의 action space는 가속 범위가 -5~ +5, 회전 범위가 -45도에서 +45도까지로 설정되어 있습니다. 이는 highway 환경에서 기본적으로 정의된 Action space입니다. 하지만 알고리즘별로 이산형 값이 필요한 경우도 있기 때문에, 이는 저희가 강화학습 알고리즘에 맞춰 이산형 값으로 정의하거나, 연속형 값으로 정의하여 사용했습니다.

Highway-env의 state space는 다음의 features들을 포함합니다. x, y로 표현된 자동차의 위치, vx와 vy로 나타내는 자동차의 x, y축 상에서의 속도, 마지막으로 cosin h, sin h로 표현되는 자동차의 방향입니다.

Highway-env는 매 step 별로 observation, reward, done, info 값들을 반환합니다. Observation에는 현재까지 도달한 지점인 observation과 achieved_goal, 최종 목표 지점인 desired_goal의 위치를 알려줍니다. Reward는 현재 도달한 위치와 목표 위치 간의 거리, 즉 자동차가 목표에 얼마나 근접했는지를 계산하여 반환합니다.

Done은 자동차가 목표위치에 도달하거나 장애물을 만나 crashed됐는지를 알려줍니다. 마지막 info에는 기본적으로 목표 도달 여부가 포함되어 있으며, 실험에 필요한 다른 추가적인 정보들을 포함시킬 수 있습니다.

6. Deep Q Learning Network

가장 먼저 사용한 알고리즘은 DQN입니다. Action Q-value 값을 통해 학습하는 Q-learning에 심층신경망을 적용한 알고리즘입니다. 이미 수업에서 많이 다룬 내용이기 때문에 알고리즘에 대한 설명보다는 코드 설명 위주로 작성하였습니다.

```
for steering in np.linspace(-0.5, 0.5, 11):
        for acceleration in np.linspace(0.8, 0.4, 4):
            candidate_actions.append(torch.Tensor([acceleration, steering]))
print(candidate_actions)
```

우선 DQN은 continuous한 action space를 사용하지 않습니다. 하지만 앞서 highway-env의 설명을 보시면 기본적으로 parking-v0 환경은 continuous 하게 정의되어 있습니다. 따라서 DQN을 하기 전 action space를 discrete하게 설정했습니다. 해당 수치는 언제든지 조정 가능합니다.

```
class DQN(nn.Module):

    def __init__(self, state_size, hidden_size, action_size):
        super(DQN, self).__init__()
        self.conv1 = nn.Linear(state_size, hidden_size//2)
        self.conv2 = nn.Linear(hidden_size//2, hidden_size)
        self.conv3 = nn.Linear(hidden_size, hidden_size)
        self.conv4 = nn.Linear(hidden_size, hidden_size)

        linear_input_size = hidden_size
        self.head = nn.Linear(linear_input_size, action_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.relu(self.conv4(x))
        return self.head(x.view(x.size(0), -1))
        #x = self.head(x)
```

```
        return x
```

다음은 신경망 부분입니다. Torch.nn을 사용하여 구성되었습니다. 4개의 neural layer가 사용되었고 활성화 함수는 relu입니다.

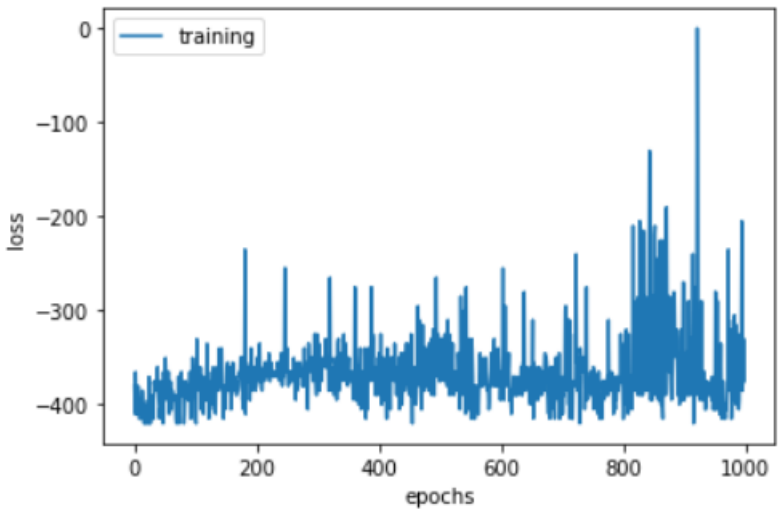
```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. *
                    steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        ret_val = torch.tensor([[random.randrange(n_actions)]]).cuda()
        return ret_val
```

행동을 결정하는 부분입니다. 특정 eps 값보다 크면 max Q value 인 action을 반환하고, 그렇지 않으면 임의의 action을 반환합니다.

```
for i_episode in range(num_episodes):
    obs, done = env.reset(), False
    loss_value = 0
    for t in range(100):
        if len(obs) == 2 :
          action = select_action(torch.from_numpy(np.array([obs[0]["observation"]])).type(dtype))
        elif len(obs) == 3:
          action = select_action(torch.from_numpy(np.array([obs["observation"]])).type(dtype))

        obs_next, reward, done, info, _ = env.step(candidate_actions[action])

        reward = torch.tensor([reward]).cuda()
        if(t%99==0 and t!=0):
          print("Episode: ", i_episode, ", iteration: ", t, ", reward: ", reward)

        if len(obs) == 2:
          memory.push(obs[0]["observation"], action, obs_next["observation"], reward)
        elif len(obs) == 3:
          memory.push(obs["observation"], action, obs_next["observation"], reward)

        obs = obs_next
```

마지막으로 학습하는 부분입니다. 설정한 에피소드 수만큼 반복을 실행합니다. If와 elif로 나눠진 부분은 이전에 언급한 오류 수정을 위해서 수정한 부분입니다. 우선 action을 선택하고, 선택된 action으로 env.step을 실행합니다. 그 후 replay

memory에 현재 상태들을 저장하고, obs를 다음 state로 변경합니다. 이 과정을 반복하며 학습합니다. 저희는 1000 epochs로 설정하여 학습을 진행하였습니다. 학습 결과는 아래와 같습니다.



결과를 보면 계속 학습을 해도 일정하게 높은 결과를 보여주지는 않습니다. 실제로 해당 학습 결과를 시뮬레이션으로 돌려봐도 자동차가 목표 방향으로 가는 것 같지만 제대로 학습되었다고 판단되지는 않았습니다. 저희는 이러한 문제가 discrete한 action space에서 학습하기에 발생하는 문제점이라 보고, 이를 개선한 DDPG 알고리즘을 사용해보았습니다.

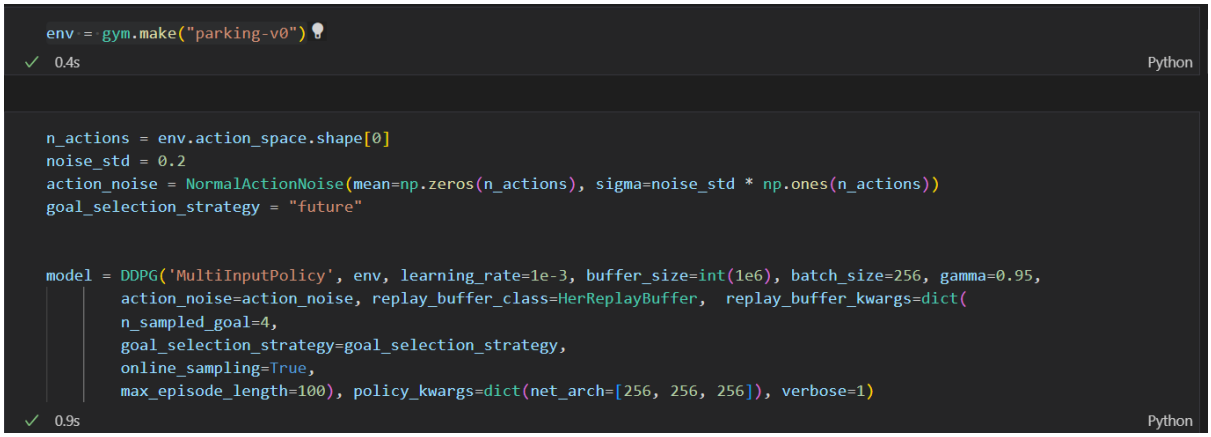
7. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient는 DQN에 continuous action을 사용할 수 있도록 개선한 알고리즘입니다. 앞서 DQN는 discrete action space를 사용했기 때문에 highway-env처럼 continuous한 action space에서 작동하려면 action을 따로 정의하는 과정이 필요했습니다. 또한, discrete하지만 action의 개수가 많아지면 차원이 매우 커지는 차원의 저주에 걸릴 위험이 있습니다. 하지만 DDPG는 continuous한 학습이 가능한 Deterministic Policy Gradient(DPG)에 DQN을 접목한 형태로 둘의 장점을 가져와 학습합니다.

DDPG에는 critic network와 actor network가 존재합니다. Critic network는 현재 상

태와 action의 가치를 평가합니다. Actor network는 policy gradient 알고리즘을 사용하여 actor의 policy를 업데이트 합니다. 해당 과정을 반복하며 파라미터 값을 최적화하는 알고리즘이라고 할 수 있습니다.

DDPG 알고리즘은 stable-baseline3 모듈을 통해 사용할 수 있습니다. 하지만 highway-env와의 버전 충돌 문제로 인해, 모듈 내부의 코드들을 수정하여 실험에 사용했습니다.



DDPG는 위와 같은 코드를 통해 실행할 수 있습니다. MultiInputPolicy는 parking-v0의 input 값이 딕셔너리 형태이기 때문에 해당 옵션으로 학습을 진행해야 합니다. Action_noise는 continuous한 action space 상황에서 exploration을 할 때 매우 비슷한 행동에 빠져 같은 행동을 반복하는 것을 막아줍니다. Replay buffer로는 Hindsight Experience Replay(HER) 알고리즘을 적용했습니다. HER 알고리즘은 학습 과정에서 최종 목표에 도달하지 못하더라도 임의의 보상을 만들어 해당 과정에서도 학습을 진행하는 알고리즘입니다. Stable baseline3 이전 버전에서는 HER이 독립적인 학습 알고리즘으로 사용할 수 있었으나 SB3에서는 Parking-v0에서도 중간에 장애물이 있거나 하는 상황에서 학습이 용이할 것이라 판단하여 DDPG에 HER을 추가로 사용했습니다.

```
model.learn(int(1500))

model.save('her_ddpg_highway')
```

✓ 25m 59.6s

```
--------------------------------
| rollout/           |         |
|    ep_len_mean     | 89.8    |
|    ep_rew_mean     | -540    |
|    success_rate    | 0       |
| time/              |         |
|    episodes        | 4       |
|    fps             | 17      |
|    time_elapsed    | 20      |
|    total_timesteps | 359     |
| train/             |         |
|    actor_loss      | 0.534   |
|    critic_loss     | 1.13e+03 |
|    learning_rate   | 0.001   |
|    n_updates       | 340     |
--------------------------------
```

앞서 model을 설정해주고 해당 모델을 사용하여 학습을 진행하였습니다. 우선 1500회를 학습시켰습니다. 1500회로 설정한 이유는 다음에 사용할 CEM과 학습 성능을 비교하기 위해서 동일하게 설정했습니다.

```
model = DDPG.load('ddpg_parking', env=env)


env = gym.make("parking-v0")
env = RecordVideo(env, video_folder='./videos', episode_trigger=lambda e:
True)
env.unwrapped.set_record_video_wrapper(env)
for episode in trange(4, desc="Test episodes"):
    obs, done = env.reset(), False
    while not done:
        if len(obs) ==2:
            action, _ = model.predict(obs[0])
        elif len(obs) ==3:
            action, _ = model.predict(obs)
        obs, reward, done, _, info = env.step(action)
env.close()
show_video('./video')
```

이후 위의 코드로 저장된 모델을 불러온 뒤 시뮬레이션을 해보면 자동차는 제자

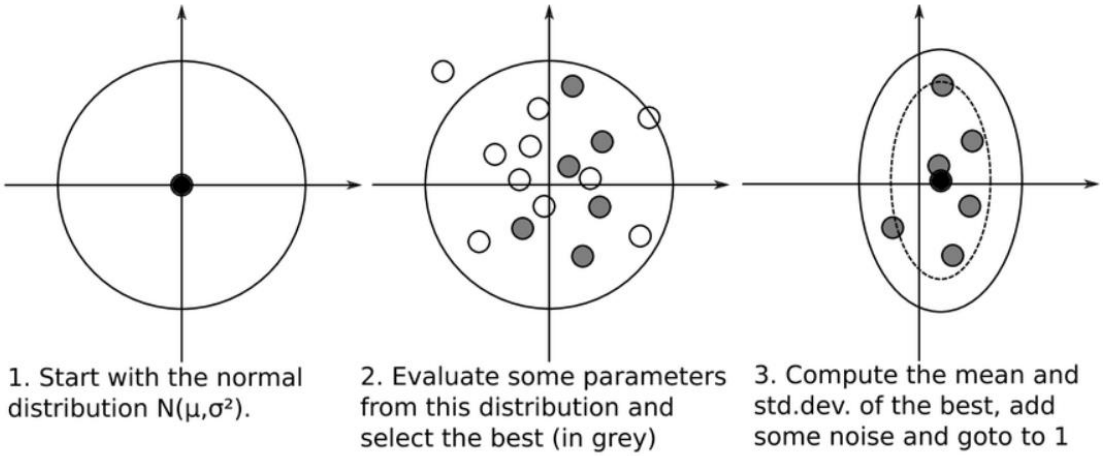
리에서 계속 돌기만 하고 앞으로 가거나 목표를 향해 다가가지 않습니다. 또한 학습 시간도 매우 오래 걸리며, 다른 알고리즘에 비해 학습 시간 대비 성능이 매우 좋지 못했습니다.

이런 결과가 나온 이유에 대해 조원들과 토의한 결과, DDPG 알고리즘이 가진 문제점이 원인이라고 생각했습니다. DDPG는 학습 과정에서 학습 초반에 제대로 된 policy를 학습하지 못하면 한방향으로만 움직이는 행동을 반복할 수 있습니다. 즉, local optima에 빠져서 제자리에서 계속해서 돌기만 하는 것입니다. 학습 횟수가 부족하여 생긴 문제라고 판단하여 colab에서 코드 수정을 통해 더 많은 횟수를 학습시켜도 여전히 local optima에 빠져 한자리에서 계속 도는 문제가 발생하였습니다. 만약 DDPG의 학습 횟수를 계속해서 늘리고 학습을 시도하면 제대로 학습된 policy가 나올 수도 있겠지만, DDPG는 저희가 가진 computing power로는 학습 시간이 매우 오래 걸렸습니다. CPU를 사용하는 로컬 환경 뿐만 아니라 gpu를 사용하는 colab에서도 학습 시간이 수 시간 걸렸습니다. 코드를 수정한 뒤 시뮬레이션을 돌리기 위해선 비효율적으로 많은 시간이 필요했기에 비교적 적은 시간 내에 학습이 되고 성능이 더 좋은 알고리즘을 찾아보았습니다.

8. Dynamic Model을 적용한 Cross Entropy Method

연속적인 action 을 가지는 문제를 풀 수 있는 알고리즘을 찾아보던 중, CEM 과 Dynamic Model 을 적용한 논문에서 아이디어를 얻었습니다. Cross Entropy Method (이하 CEM)는 간단하게 설명해서 샘플링을 이용한 강화학습 알고리즘이라고 할 수 있습니다. 큰 연못에서 낚시를 하는 상황을 예시로 들어보겠습니다. 연못 내부, 그리고 낚싯대, 미끼와 낚시 결과의 상관관계에 대한 정보가 없는 상황에서 가장 큰 물고기를 잡을 수 있는 낚시 포인트를 알려면 어떻게 해야 할까요? CEM 의 방법을 대입해 보면 다음과 같습니다. 랜덤하게(혹은 임의의 분포에 따라) 추출된 100 개의 지점에서 낚시를 해봅니다. 100 번의 낚시에 대한 데이터를 기록하고, 이 100 개의 지점들 중 가장 무거운 물고기를 낚은 상위 10 개의 지점을 선택합니다. 이렇게 선택된 지점들을 '엘리트(elite)'라고 하고, 이 엘리트들의 평균과 분산을 구합니다. 그 후, 이 평균과 분산을 따르는 정규분포에

의해 100 개의 새로운 지점들을 샘플링 합니다. 이 지점들에서 또 낚시를 한 후 또 상위 10 개의 지점들을 추출, 평균과 분산을 계산합니다. 이와 같은 과정을 만족할 만한 결과가 나올 때까지 반복합니다.



이 과정을 반복하다 보면 결국 보상의 기대 값이 가장 높은 지점을 추정할 수 있다는 것이 바로 CEM 입니다. 분포에 따라 샘플링 된 데이터를 action 으로 선택한다는 점에 있어서 action 이 연속형 변수인 문제에 대응할 수 있다는 것이 장점입니다. CEM 은 가장 강한 상위 option 들만 계속해서 살아남는다는 점에서 진화론의 자연선택설과 비슷한 맥락으로도 이해할 수 있습니다.

저희가 참고한 논문에서는 CEM과 함께 Dynamic Model을 사용했습니다. 차량을 운행하는 문제에 있어서, 단순히 현재 state 에 따른 다음 action 을 결정하는 과정을 넘어서 안정적인 경로로 주행하기 위한 path-planning 능력이 꼭 필요하기 때문에 Dynamic Model 을 사용한다고 설명하였습니다.

환경과 에이전트를 처음 생성했을 때, 에이전트는 자신의 행동이 환경 안에서 어떤 변화를 가져오는지 알지 못한다.

따라서 1000번 동안
랜덤하게 action 한 쌍을 샘플링하고,
처음 랜덤하게 초기화된 위치에서 시작해서 4번
'**같은 action**'으로 움직였을 때 어떤 변화가 일어나는지
기록해서 Transition의 형태의 데이터로 저장.

즉, 각 observation에서 어떤 action을 취했을 때
다음 observation에 대한 정보를 수집하는 단계.

Action 1개에 대해 4번씩 탐색 / 저장하고
Action을 1000개 추출하므로 총 4000개의
[state, action, next_state] 정보를 data에 저장하게 됨.

다른 방식으로 이 과정의 필요성을 설명하자면, 가장 처음에는 agent 는 자신이 어떤 action 을 취했을 때 환경과 다음 state 에 어떤 영향을 미치는지 알지 못합니다. DQN 에서 차량이 목표 지점에 가까이 가더라도, 멈추지 않고 속도를 줄이지 못하는 이유가 바로 이 지점에서 발생한다고 생각했습니다. Dynamic model 을 도입하게 되면 현재 state 와 action 을 결정했을 때, 실제로 움직이지 않고도 미리 다음 state 를 예측할 수 있는 능력을 기를 수 있게 됩니다. 예를 들어, 현재 방향대로 가면 5 step 이후 목표지점에 도달할 수 있다고 판단이 된다면, action 을 그에 맞게 action 정해서 적절히 감속을 하게끔 agent 를 유도할 수 있다는 것이 핵심입니다.

코드를 보면서 설명을 하자면, 가장 초기에 1 가지 action 을 랜덤하게 샘플링한 후, 해당 action 을 고정한 채로 4 번의 time step 을 거치게 됩니다. 각각의 time step 을 거칠 때 마다 환경에서 관찰되는 다음 state 를 포함하여 [state, action, next_state] 쌍을 data 에 저장합니다. size 만큼 action 을 샘플링하고, action_repeat 변수만큼 반복되므로 현재 상태에서는 총 1000 x 4 = 4000 개의 [state, action, next_state] 데이터를 저장하게 됩니다. 이 데이터는 이후 Dynamic Model 을 훈련하는데 input data 와 정답 데이터로 사용하게 됩니다.

```python
class DynamicsModel(nn.Module):
    STATE_X = 0
    STATE_Y = 1

    def __init__(self, state_size, action_size, hidden_size, dt):
        super().__init__()
        self.state_size, self.action_size, self.dt = state_size, action_size, dt
        A_size, B_size = state_size * state_size, state_size * action_size
        self.A1 = nn.Linear(state_size + action_size, hidden_size)
        self.A2 = nn.Linear(hidden_size, A_size)
        self.B1 = nn.Linear(state_size + action_size, hidden_size)
        self.B2 = nn.Linear(hidden_size, B_size)

    def forward(self, x, u):
        """
            Predict x_{t+1} = f(x_t, u_t)
        :param x: a batch of states
        :param u: a batch of actions
        """
        xu = torch.cat((x, u), -1)
        xu[:, self.STATE_X:self.STATE_Y+1] = 0  # Remove dependency in (x,y)
        A = self.A2(F.relu(self.A1(xu)))
        A = torch.reshape(A, (x.shape[0], self.state_size, self.state_size))
        B = self.B2(F.relu(self.B1(xu)))
        B = torch.reshape(B, (x.shape[0], self.state_size, self.action_size))
        dx = A @ x.unsqueeze(-1) + B @ u.unsqueeze(-1)
        return x + dx.squeeze()*self.dt

dynamics = DynamicsModel(state_size=env.observation_space.spaces["observation"].shape[0],
                         action_size=env.action_space.shape[0],
                         hidden_size=64,
                         dt=1/env.unwrapped.config["policy_frequency"])
```

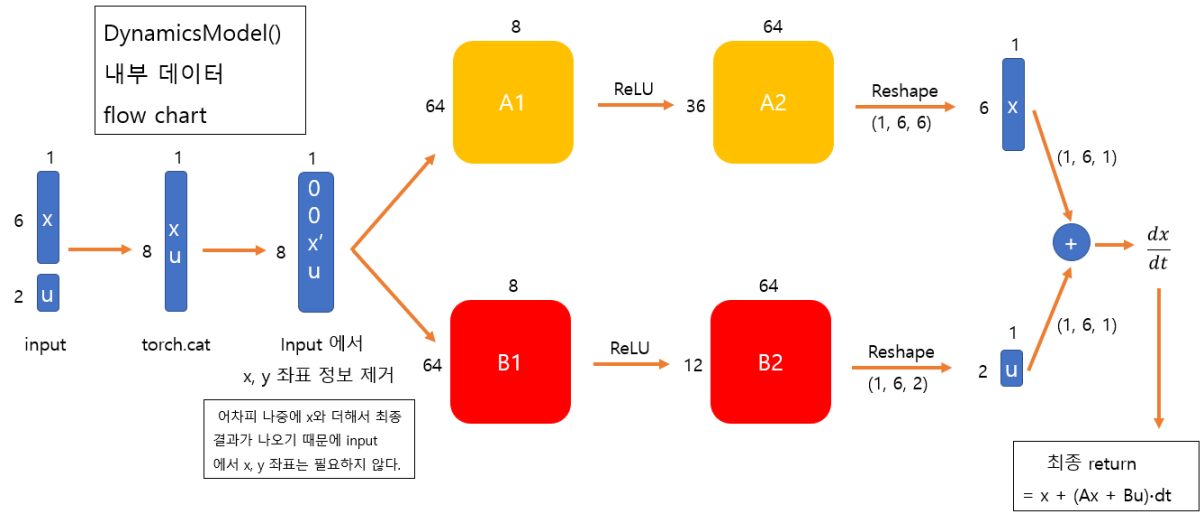
앞에서 환경과 상호작용 하면서 얻은 데이터를 기반으로 경로를 따라 주행하는 것을 가르치기 위해 모델 생성

$$\frac{dx}{dt} = Ax + Bu$$

Control theory 에서 나온 LTI system 모델을 전제로 함.
(LTI system: Linear, Time invariant system)

쉽게 생각해서 input 으로 state 와 action을 넣으면
Output 으로 next_state가 나오는 신경망 모델.
처음에는 랜덤한 값들로 초기화되어 있지만 앞에서 수집한 데이터들을 이용해 지도학습 (Adam) 방식으로 최적화시킨다.

$\frac{dx}{dt} = Ax + Bu$ 이므로, $dx = (Ax + Bu)\cdot dt$. 따라서
Next state = x + dx
= x + (Ax + Bu)·dt



DynamicsModel() 클래스 내부는 2 개의 신경망으로 나누어졌다가 최종적으로 다시 합쳐져서 계산되는 형태를 띄고 있는데, 이는 제어 분야의 LTI system 에서 기인한 것입니다. dx/dt = A(x, u)*x + B(x, u)*u 라는 기본 꼴에서 출발한 것으로, 쉽게 설명하면 현재 system 의 시간에 따른 변화량은 x 의 영향과 u 의 영향의 합으로 나타낼 수 있다는 것입니다. 이때 x 는 현재 system 의 상태, u 는 system 내부의 모터나 엔진과 같은 actuator 의 동작을 의미합니다. 이 문제에 대입해서 생각해 보면 x 는 현재 state, u 는 현재 선택한 action(가속력, 바퀴 방향)으로 볼 수 있습니다. A(x, u), B(x, u)는 일종의 행렬로 생각할 수 있는데, 내부 변수로 x 와 u 가 들어가 있습니다. A(x, u)*x + B(x, u)*u 를 계산해 dx/dt 를 계산한다면,

$X_{t+1} = X_t + dx$

$= X_t + (dx/dt)*dt$

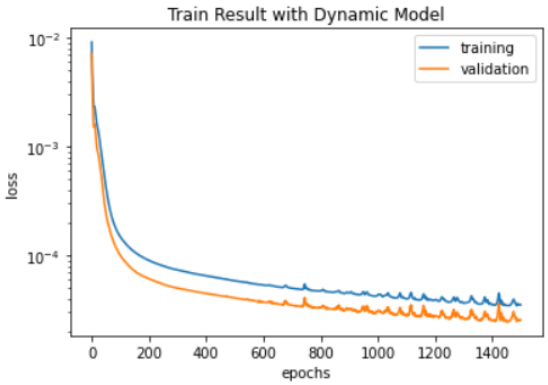
$= X_t + (A(x, u)*x + B(x, u)*u)*dt$

의 수식을 통해 다음 state, 즉 X_{t+1} 을 예측할 수 있게 됩니다. 쉽게 말해서, input 으로 현재 state 와 현재 action 을 넣어주면, output 으로 next_state 를 뱉는 모델을 얻을 수 있다는 것입니다. 이 문제에서 dt = 0.2 로, 1 초에 5 번씩 action 을 선택하고 다음 state 로 이동합니다.

전통적인 제어공학의 분야에서는 문제가 간단한 경우, 주어진 문제 상황의 특성을 집어넣어서 수학적으로 A(x, u)와 B(x, u)를 실제로 계산하지만, 이 과제에서는 앞에서 얻었던 [state, action, next_state]를 이용해 신경망을 최적화 시키는 방법을 선택했습니다. 즉, [state, action]을 input data 로 사용하고, next_state 를 모델의 ground truth, 즉 정답 데이터로 사용해 모델의 loss function 값을 최소화 시키면서 A(x, u)와 B(x, u)를 찾아가는 것입니다.

```
optimizer = torch.optim.Adam(dynamics.parameters(), lr=0.01)

# Split dataset into training and validation
train_ratio = 0.7
train_data, validation_data = data[:int(train_ratio * len(data))], \
                              data[int(train_ratio * len(data)):]

def compute_loss(model, data_t, loss_func = torch.nn.MSELoss()):
    states, actions, next_states = data_t
    predictions = model(states, actions)
    return loss_func(predictions, next_states)

def transpose_batch(batch):
    return Transition(*map(torch.stack, zip(*batch)))

def train(model, train_data, validation_data, epochs=1500):
    train_data_t = transpose_batch(train_data)
    validation_data_t = transpose_batch(validation_data)
    losses = np.full((epochs, 2), np.nan)
    for epoch in trange(epochs, desc="Train dynamics"):
        # Compute loss gradient and step optimizer
        loss = compute_loss(model, train_data_t)
        validation_loss = compute_loss(model, validation_data_t)
        losses[epoch] = [loss.detach().numpy(), validation_loss.detach().numpy()]
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # Plot losses
    plt.plot(losses)
    plt.title('Train Result with Dynamic Model')
    plt.yscale("log")
    plt.xlabel("epochs")
    plt.ylabel("loss")
    plt.legend(["training", "validation"])
    plt.show()

train(dynamics, data, validation_data)
```

이전에 수집했던 4000 개의 [state, action, next_state] 데이터 중 70%만 훈련에 사용하고, 나머지는 훈련 중 평가인 validation 을 위해 사용했습니다. Dynamics Model 의 loss function 은 일반적인 MSE loss 를 사용했고, 파라미터 최적화에는 우수한 성능을 보이는 것으로 알려진 Adam optimizer 를 사용해 1500 epoch 동안 훈련을 진행하였습니다. 훈련 결과는 아래 그래프와 같습니다.

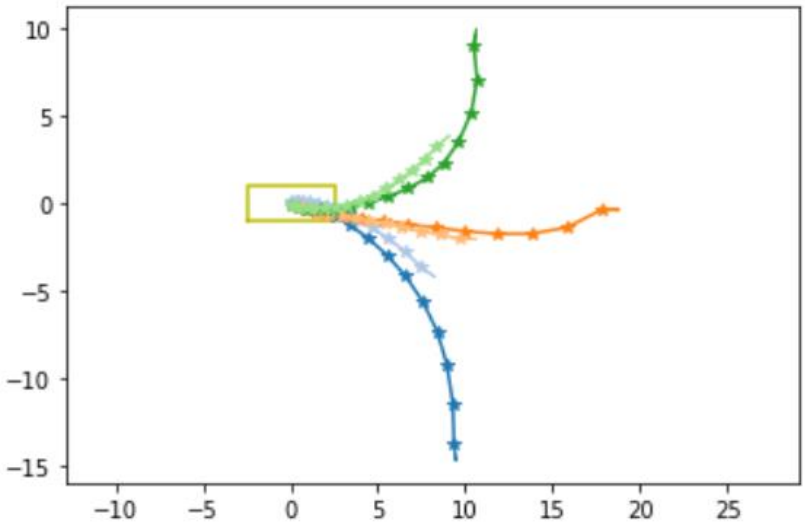


해당 그래프를 보면, 갈수록 training loss 가 감소하고, validation loss 를 보면 아주 근소한 차이 이지만 훈련이 진행될수록 training loss 보다도 낮은 통계를 보여줍니다. 즉, 제어 이론에 기반해서 만든 Dynamic Model 이 훈련이 잘 이루어 졌고, 처음 마주하는 state, action 에 대해서도 next_state 를 곧 잘 예측한다고 할 수 있습니다.

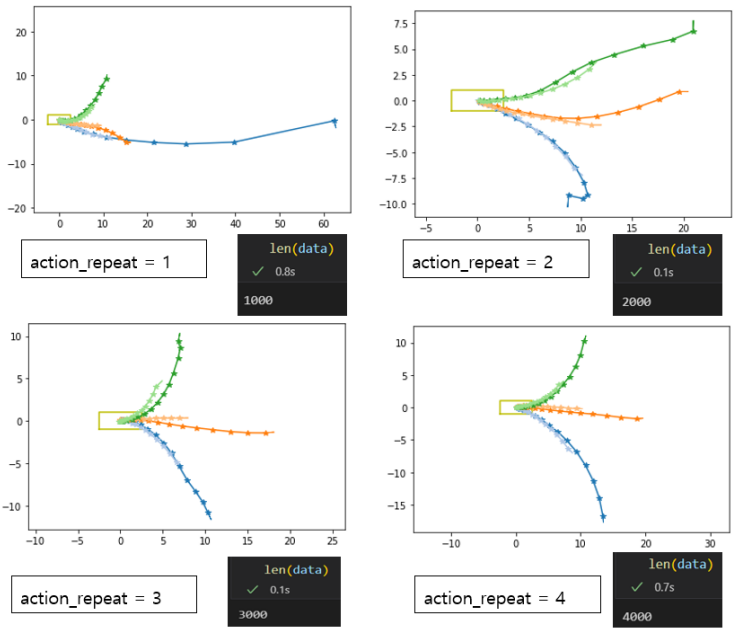
다음은 Dynamic Model 훈련의 결과를 시각화 하기 위한 코드입니다.

```
def predict_trajectory(state, actions, model, action_repeat=1):
    states = []
    for action in actions:
        for _ in range(action_repeat): #1개의 초기 acrion 당 15개의 경로 지점 예상
            state = model(state, action)
            states.append(state)
    return torch.stack(states, dim=0)

def plot_trajectory(states, color):
    scales = np.array(env.unwrapped.config["observation"]["scales"])
    states = np.clip(states.squeeze(1).detach().numpy() * scales, -100, 100)
    plt.plot(states[:, 0], states[:, 1], color=color, marker='*')
    plt.arrow(states[-1,0], states[-1,1], states[-1,4]*1, states[-1,5]*1, color=color)

def visualize_trajectories(model, state, horizon=15):
    plt.cla()
    # Draw a car
    plt.plot(state.numpy()[0]+2.5*np.array([-1, -1, 1, 1, -1]),
             state.numpy()[1]+1.0*np.array([-1, 1, 1, -1, -1]), 'y') # 자동차 그리는 부분
    # Draw trajectories
    state = state.unsqueeze(0)
    colors = iter(plt.get_cmap("tab20").colors)

    # Generate commands
    for steering in np.linspace(-0.5, 0.5, 3):
        for acceleration in np.linspace(0.8, 0.4, 2): # 초기 action 6가지 경우에 대해

            actions = torch.Tensor([acceleration, steering]).view(1,1,-1)
            print('one actions:')
            print(actions)
            # Predict trajectories
            states = predict_trajectory(state, actions, model, action_repeat=horizon)
            plot_trajectory(states, color=next(colors))
    plt.axis("equal")
    plt.show()


visualize_trajectories(dynamics, state=torch.Tensor([0, 0, 0, 0, 1, 0]))
```

쉽게 설명해서, 한 지점에서 서로 다른 6 가지의 sample action 에 대해, 각 action 으로 horizon (=15) 만큼 움직인다고 했을 때 예상되는 경로에 대한 그림을 그려줍니다. 결과는 아래와 같습니다.



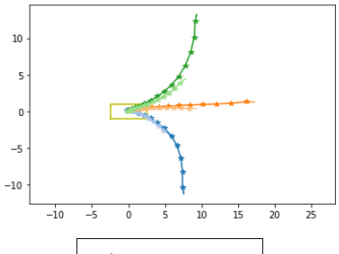
결과를 보면 각 action 에 대해 꽤 매끄러운 경로를 예상하고 있음을 알 수 있습니다. 이 지점에서 만약 초기에 수집했던 데이터가 앞으로 4step 이 아니라 이것 보다 더 적은 step 을 보게 된다면 Dynamic Model 의 학습에 어떤 영향을 미칠지가 궁금해졌고, 위의 action_repeat 변수를 바꿔가며 데이터를 새로 수집해 실험을 진행해 보았습니다.




```
for steering in np.linspace(-0.5, 0.5, 3):
    for acceleration in np.linspace(0.8, 0.4, 2):
```

6가지의 초기조건에 대한 모델의 경로 설계 결과
가장 초기에 학습단계에서 몇 step을 보는지에 따라 path planning 능력이 크게 달라진다.

action repeat = 1, 2일 때는 안정적으로 경로를 보지 못하다가 3, 4 이상으로 가면 안정적으로 설계해 나가는 경향을 보인다.



action_repeat 을 계속 늘린다고 해서 설계 능력이 계속 나아지지는 않는다. n=4에서 타협

실험 결과를 보면, action_repeat = 1, 2 일때는 경로 예상 능력과 설계 능력이 불안정함을 알 수 있고, action_repeat = 3 이상일 때부터 점점 안정적으로 경로를 예상 및 설계해 나갈 수 있음을 볼 수 있었습니다. action_repeat = 1 일 때는 매 time step 마다 새로운 action 을 추출해서 agent 를 움직여 가며 데이터를 수집하는데, 이 때문에 한가지 action 이 next_state 에 미치는 영향에 대한 경향성을 잘 파악하지 못했기 때문일 것으로 예상합니다. 반대로, action_repeat = 10 일때의 결과를 보면 action_repeat = 4 일때의 결과와 크게 다르지 않음을 알 수 있는데, 이를 통해 action_repeat 를 늘린다고 해서 계속해서 설계 능력이 향상되는 것은 아님을 알 수 있습니다. action_repeat 가 너무 적으면 설계 능력이 떨어지고, action_repeat 가 너무 많으면 데이터 수집 및 훈련에 시간이 오래 걸리므로, action_repeat=4 가 적당하다고 판단하였고, 이후 이 값으로 프로젝트를 진행했습니다.

다음으로, 해당 과제에서 사용한 reward 함수에 대해 보겠습니다.

```python
def reward_model(states, goal, gamma=None):
    '''
    reward = 루트((state-goal) * [1, 0.3, 0, 0, 0.02, 0.02])
    '''
    goal = goal.expand(states.shape)
    reward_weigths = torch.Tensor(env.unwrapped.config["reward_weights"])
    rewards = -torch.pow(torch.norm((states-goal)*reward_weigths, p=1, dim=-1), 0.5) #
    if gamma:
        time = torch.arange(rewards.shape[0], dtype=torch.float).unsqueeze(-1).expand(rewards.shape)
        rewards *= torch.pow(gamma, time)
    return rewards

obs = env.reset()[0]
```

State = [x, y, Vx, Vy, cos(h), sin(h)]

Goal = [x', y', Vx', Vy', cos(k), sin(k)]

reward_weight = [1, 0.3, 0, 0, 0.02, 0.02]

γ = discount factor (1)

Reward = $-\gamma^t\sqrt{(\text{state}-\text{goal})*\text{reward\_weight}}$

-> 현재 지점과 목표가 멀수록 작아지고,
현재 지점과 목표가 클수록 커진다.

이 과제에서 reward 는 현재 지점과 목표 주차지점까지의 거리와 지정된 각도를 지표로 가져가고 있습니다. State 안의 정보인 V_x 와 V_y 는 고려되지 않습니다. 이는 이전 DQN 에서 목표 지점에 가까워져도 속도가 줄어들지 않았던 이유 중 하나로 추측합니다. discount_factor 은 1 로 설정하고 있는데, 매 time step 마다 좌표 정보를 기반으로 reward 를 계산하는 구조이기 때문에 미래의 reward 도 확실하게 받을 수 있어서 감쇄시키지 않는 것입니다. 거리는 가까울수록 절댓값이 작고, 멀수록 크기 때문에, 앞에 (-)를 붙여서 가까울수록 리워드가 크고, 멀수록 리워드가 작은 형태를 띄고 있습니다. 또한, 단순히 거리가 아닌 reward_weight 를 곱해서 사용하는데, 이는 각 요소의 영향을 제어하기 위해서입니다. CEM 을 활용해 이 프로젝트를 실행해보면 agent 가 주차 지점 근처에서 x 좌표는 엄격해도 y 좌표에는 약간 느슨한 모습을 보이고, 주차가 되는 각도에 크게 민감하지 않은 양상을 보이게 되는데, 이것이 reward 를 산출할 때 곱해지는 reward_weight 때문인 것으로 예상하고 있습니다.

이어서 앞서 설명했던 CEM 부분의 코드를 보겠습니다.

```python
# Cross entropy method
def cem_planner(state, goal, action_size, horizon=5, population=100, selection=10, iterations=5):
    state = state.expand(population, -1)
    action_mean = torch.zeros(horizon, 1, action_size)
    action_std = torch.ones(horizon, 1, action_size)
    for _ in range(iterations):
        # 1. Draw sample sequences of actions from a normal distribution
        actions = action_mean + action_std * torch.normal(0, 1, size = (horizon, population, action_size))
        actions = torch.clamp(actions, min=env.action_space.low.min(), max=env.action_space.high.max())
        states = predict_trajectory(state, actions, dynamics, action_repeat=5)
        # 2. Fit the distribution to the top-k performing sequences
        returns = reward_model(states, goal).sum(dim=0)
        _, best = returns.topk(selection, largest=True, sorted=False)
        best_actions = actions[:, best, :]
        action_mean, action_std = best_actions.mean(dim=1, keepdim=True), best_actions.std(dim=1, unbiased=False, keepdim=True)
    return action_mean[0].squeeze(dim=0)


# Run the planner on a sample transition
action = cem_planner(torch.Tensor(obs["observation"]),
                     torch.Tensor(obs["desired_goal"]),
                     env.action_space.shape[0])
print("Planned action:", action)
```

1개의 state에서 100가지 action 샘플링해서 테스트
이때 분포: N ~ (0,1) 정규분포에서 추출

각 action에 대해 5개의 time step 으로 움직였을 때 Reward 각각 계산

각 action에 대해 5개의 Reward 합으로 최종 리워드 계산, 이 중 가장 값이 큰 10개 선택

10개의 Elite 들에 대한 평균과 표준편차 계산, 다시 루프로 돌아가서 샘플링/선택 반복 (5회)

최종적으로 선택되는 action : 학습된 분포에서의 평균값

```
Planned action: tensor([0.9044, 0.6767])
```

CEM 은 주어진 상황에 대해 여러가지 action 을 샘플링하고, 그에 대한 평가를 진행한 후 이에 기반하여 다음 샘플링을 이어 나가는 알고리즘입니다. 여기서 샘플링 된 action 을 평가하는 부분에서 Dynamic Model 을 사용합니다. 우선 100 가지의 action 을 정규분포에 따라 추출하고, 이 100 가지 action 의 즉각적인

reward 가 아닌, 앞으로 5 step 이후까지의 reward 의 총합을 각 action 의 가치로 판단합니다. 이때, 앞서 훈련했던 Dynamic model 을 이용합니다. State 정보와 목표 지점에 대한 정보만 있으면 reward 를 계산할 수 있으므로, Dynamic model 을 이용해 해당 action 으로 5 time step 만큼 진행하면 각각 어떤 state 에 도달하는지 예측한 후, 이를 종합하는 것입니다. 이 과정을 통해 가장 reward 가 큰 10 개의 지점을 선택하고, 이들의 평균, 분산을 구해 이 분포에 따라 다음 100 개의 action 을 샘플링 합니다. 이 과정을 iteration 이라는 변수만큼 반복하는데, 위 코드에서는 총 5 회 반복하게 됩니다. 이 과정을 통해 최종적으로 얻은 평균이 곧 CEM planner 가 최종적으로 주는 output, 즉 주어진 state 에 대한 최적의 action 입니다.

마지막으로, 실제로 CEM 을 활용해 주차를 하는 부분에 대한 코드입니다.

```
from tqdm import trange

env = gym.make("parking-v0")
# env.configure({"vehicles_count": 10})
env = RecordVideo(env, video_folder='./videos', episode_trigger=lambda e: True)
env.unwrapped.set_record_video_wrapper(env)
for episode in trange(20, desc="Test episodes"):
    obs, done = env.reset(), False
    t = 0
    while ((not done) and (t <70)) :
        if len(obs) == 2:
            action = cem_planner(torch.Tensor(obs[0]["observation"]),
                                 torch.Tensor(obs[0]["desired_goal"]),
                                 env.action_space.shape[0])
        elif len(obs) ==3:
            action = cem_planner(torch.Tensor(obs["observation"]),
                                 torch.Tensor(obs["desired_goal"]),
                                 env.action_space.shape[0])
        obs, reward, done, info, _ = env.step(action.numpy())
        t = t+1

env.close()
show_video('./video')
```

총 20 번의 episode 에 대해서 실행하고, 매 state 마다 cem_planner 를 통해서 action 을 결정합니다. 프로그램을 돌리다 보면 종료조건이 잘 맞지 않아서 충분히

근처에 갔음에도 episode 가 종료되지 않는 경우들이 간혹 있어서 70 번의 time step 을 넘기면 자동으로 episode 가 종료되도록 설정하였습니다. 결과를 보면, 매 episode 마다 주차 구역이 랜덤한 장소로 지정되는데, 각 지점에 알맞게 매끄럽게 주행해서 들어가는 모습을 볼 수 있습니다. 또한, DQN 때와는 달리 목표 지점에 가면 속도가 크게 줄어들어 정지한 상태가 됨을 알 수 있습니다. 이는 Dynamic Model 을 통해 미래의 state 를 예측한 후 실제 행동으로 옮길 수 있는 능력이 생겼기 때문에 가능한 것으로 예상합니다. Dynamic Model 을 적용한 CEM planner 는 이 주차 문제에서 매우 잘 작동했지만, 몇 가지 의문점이 들어 후속 연구를 진행했습니다.

· 의문점 1. 왜 꼭 dynamic model 을 사용해야 하는가?

가장 첫번째로 CEM 을 dynamic model 의 능력이 낮을 때, 즉 1step 앞의 모습만을 가지고 훈련한 Dynamic model 을 이용해 탐색하면 어떻게 동작할지가 궁금했습니다. 그래서 앞에서 했던 예시 중 action_repeat=1 인 상태로 dynamic model 을 새로 훈련해 CEM planner 로 테스트를 해보았습니다. 그 결과 주행 경로가 많이 불안해지고, 우여곡절 끝에 목표지점에 도달하는 episode 도 관찰되었지만 설정된 외벽에 부딪혀서 끝나는 episode 의 빈도가 월등히 높았습니다. 또한 이 경우에도 주차 공간 근처에 가도 속도를 줄이지 못했습니다. 이를 통해 주차 공간에 도달했을 때 정지할 수 있는 연속적인 action 을 설계하는 것에는 먼 미래의 상태를 연속적으로 예측하는 능력이 필수적이라는 결론을 얻을 수 있었습니다.

· 의문점 2. 왜 꼭 dynamic model 을 제어 공학 이론에 기반해서 설계해야 하는가?

Dynamic model 의 필요성을 깨달은 이후, 두번째로 들었던 의문은 왜 하필 LTI system 의 유형에 맞게 dynamic model 내부를 설계해야 하는 것이었습니다. 왜냐하면 쉽게 생각했을 때, 입력 데이터와 정답 데이터가 주어진 상황에서 모델 내부의 설계와 무관하게 loss function 은 최적화를 시킬 수 있다고 생각했기

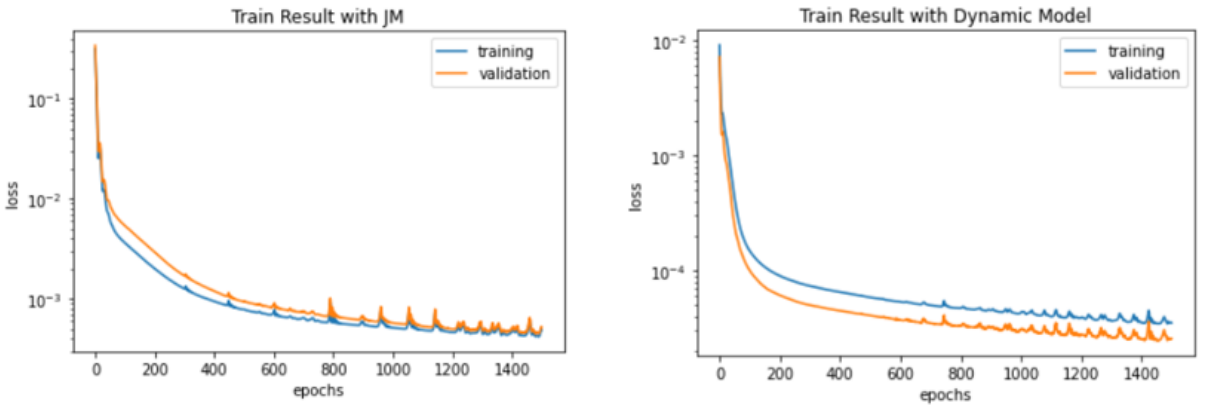
때문입니다. 단적인 예시를 들어서 기계학습의 아주 오랜 예시인 MNIST 손글씨 예측 문제를 해결한다고 했을 때, 단순하게 만들어진 완전연결층도 높은 정확도를 보여줍니다. 그러나 이 완전연결 층을 설계할 때에는 input 으로 들어오는 픽셀이나 숫자 이미지에 대한 사전 정보나 이론에 근거해서 설계한 것이 전혀 아닙니다. 모델의 입장에서는 input 은 그저 벡터, output 역시 그저 벡터일 뿐입니다. 이 dynamic 을 훈련시키는 문제에서도, [state, action]은 모델의 입장에서는 그저 숫자들의 나열일 뿐, 아무런 의미도 없지 않을까? 하는 의문이 들었습니다. 그래서 똑같이 input 으로 [state, action]을 주면 바로 [next_state]를 주는 아주 단순한 완전 연결 신경망(JustModel, 이하 JM)을 만들었고, 이를 통해 실험을 진행했습니다.

```
class JustModel(nn.Module):

    def __init__(self, state_size, action_size, hidden_size):
        super().__init__()
        self.state_size, self.action_size = state_size, action_size
        self.C1 = nn.Linear(state_size+action_size, hidden_size)
        self.C2 = nn.Linear(hidden_size, state_size)

    def forward(self, x, u):
        xu = torch.cat((x, u), -1)
        C = self.C2(F.relu(self.C1(xu)))
        next_state = torch.reshape(C, (x.shape[0], self.state_size))

        return next_state

JM = JustModel(state_size=env.observation_space.spaces["observation"].shape[0],
               action_size=env.action_space.shape[0],
               hidden_size=64)

next_state = JM.forward(data[10].state.unsqueeze(0), data[10].action.unsqueeze(0).detach())

print('current state: ',data[10].state.unsqueeze(0))
print('predicted next_state: ', next_state)
print('input state shape: ', data[10].state.unsqueeze(0).shape)
print('output state shape: ', next_state.shape)
```

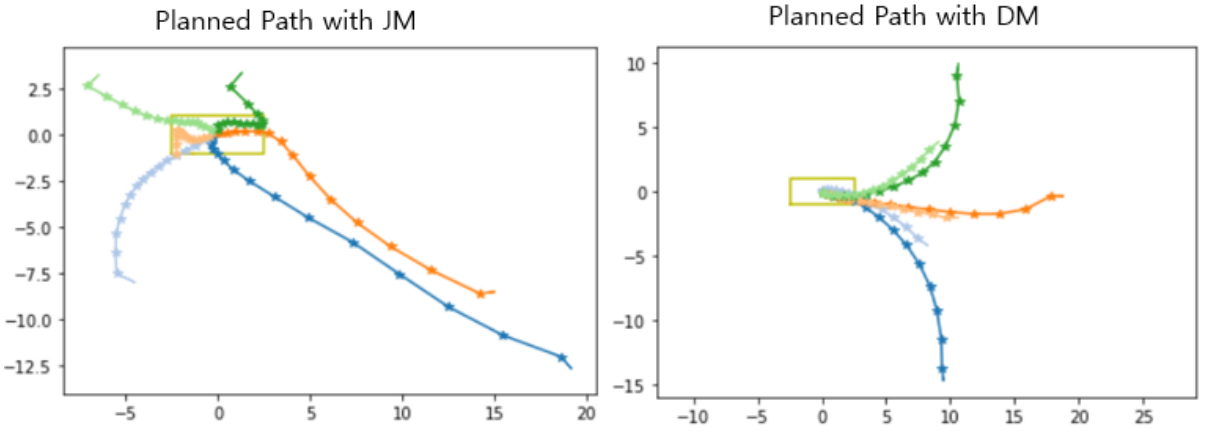
✓ 0.7s

```
current state:  tensor([[-0.0344, -0.0034, -0.1084, -0.0133, -0.9926, -0.1215]])
predicted next_state:  tensor([[-0.0290,  0.1259,  0.0754,  0.0286,  0.0645, -0.1300]],
       grad_fn=<ViewBackward>)
input state shape:  torch.Size([1, 6])
output state shape:  torch.Size([1, 6])
```

모델의 구조만 바꾸고 실험 데이터, 학습 방법, 횟수는 모두 이전의 DynamicsModel()을 학습시킬 때와 동일하게 진행하였습니다. 학습하면서 기록한 loss 값 결과를 보면 다음과 같습니다.



단순하게 만들어진 JustModel 의 훈련 결과를 보면, 예상했던 대로 loss 값이 점점 작아지면서 최적화되어 가고 있음을 볼 수 있습니다. Validation loss 역시 training loss 와 같이 값이 점점 작아지면서 처음 보는 state 에 대해서도 어느정도 예측 능력을 기르고 있다고 볼 수 있습니다. 그러나 앞서 훈련했던 DynamicModel (이하 DM)과 비교해 보면, 그 값에서 크게 차이가 나고 있음을 알 수 있습니다. 실제로 각 모델의 1200 ~ 1400 사이의 validation loss 값들의 평균을 보면 JM 의 경우 $0.347*10^{-3}$, DM 의 경우 $1.086*10^{-5}$ 로 단순히 비교하면 약 31 배 차이가 남을 알 수 있습니다.



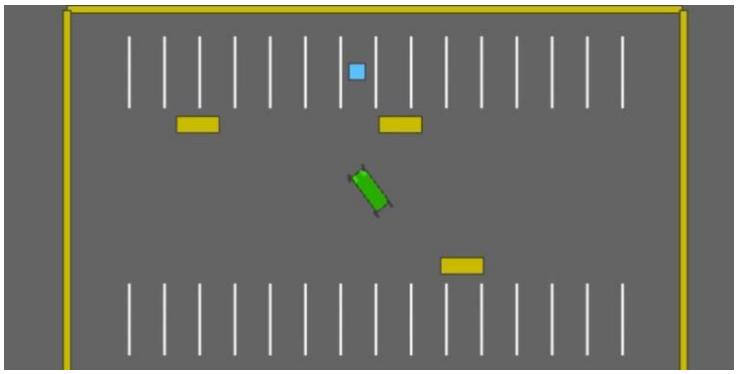
단순히 loss 값에서 보이는 차이 뿐만 아니라, 훈련된 결과로 같은 초기조건에 대해 다음 15 step 의 경로 예측에 대한 결과를 보면, 주어진 방향에 따라

안정적으로 경로를 예측하고 있는 DM 과 달리 JM 의 예측 결과들은 매 time step 마다 움직이는 간격과 방향이 일정하지 않고 불안정한 경향을 띄는 것을 확인할 수 있습니다.

또한 훈련된 JM 을 통해 CEM planner 로 실제 주차 문제를 해결하려고 해보면, 같은 데이터로 학습했음에도 불구하고 DM 을 적용했을 때와 비교했을 때 agent 가 굉장히 불안한 경로로 주행하고 문제 해결 능력이 현저히 떨어짐을 확인할 수 있습니다.

· 의문점 3. 주차장에 장애물이 있는 경우에도 대응할 수 있는가?

마지막으로, 이 알고리즘이 장애물이 있는 경우에도 대응할 수 있을지 궁금했습니다. 그래서 환경변수를 바꿔 주차장에 장애물이 있도록 바꿨고, 이 장애물들은 새로운 episode 가 실행되어도 위치가 바뀌지 않습니다. 이런 환경에서 가장 잘 작동했던 방식인 DM 을 이용한 CEM 방식으로 연구를 진행했습니다.



위 모습이 장애물이 추가된 환경의 모습이고, 연구 결과 아슬아슬하게 피해서 운전해 목적지에 도달하는 모습을 볼 수 있었는데, 주차 위치가 장애물들과 가깝게 형성된 경우 이를 잘 피하지 못하고 자주 충돌하는 모습을 보였습니다.

9. 결론 및 의의

앞의 의문점들을 해결하면서, Path planning 능력 없이 CEM 방법 만으로는 주차 문제를 해결할 수 없고, Dynamic model 이 아닌 일반 신경망으로 구성하면 path

planning 능력이 현저히 떨어지고, loss function 최적화가 잘 되지 않는다는 결론을 얻을 수 있었습니다. 즉, 수학적인 방식으로 Dynamic Model 의 A 행렬과 B 행렬의 값을 계산하는 것이 아니라 지도학습 방법으로 각 행렬을 최적화시키는 방법을 쓴다고 하더라도, 현 문제상황에 알맞은 system model 에 기반해서 내부를 설계해야 의도한 대로 모델을 학습, 사용할 수 있다는 것입니다.

최종적으로 DQN, DDPG, CEM (with Dynamic Model) 3 가지 방식으로 자율주행 주차 문제를 해결하려고 했을 때, 가장 좋은 퍼포먼스를 보여준 것은 바로 Dynamic Model 을 적용한 CEM 이었습니다. 이를 통해 agent 의 action 이 environment 와 next state 에 미치는 영향과 결과가 수학적으로 표현될 수 있는 경우에 이에 대한 역학적 이해를 바탕으로 agent 에게 미래를 예측할 수 있는 능력을 부여하여 각 state 의 action 선택 과정에 이를 포함시키면, agent 는 action 으로 인한 먼 미래의 state 까지 올바르게 예측할 수 있게 되고, 이는 학습 시간 단축과 agent 의 문제 해결 능력 향상에 기여할 수 있다는 결론을 내렸습니다. 또한, CEM 의 경우 장애물이 있는 상황에서는 어느 정도 대응하는 모습을 보여주긴 하지만 완벽하지는 않은 모습을 보여줌을 확인하였습니다.

이 프로젝트의 의의로는 우선 이 프로젝트의 결과를 통해 어떤 문제를 마주쳤을 때, 강화학습 알고리즘을 단독으로 적용하는 것이 아닌 다른 학문을 이용해 이를 바탕으로 문제를 더 효율적으로 접근해서 풀 수 있다는 가능성을 제시했다는 점에서 이 프로젝트의 의의가 있다고 생각합니다. 두번째로 이러한 접근을 할 때에, 관련 이론에 기반해서 모델을 설계해야 원하는 결과를 얻을 수 있고 단순하게 loss value 최적화만 시키는 모델은 실제 사용에 있어서 효과가 떨어진다는 점을 확인하였습니다. 세번째로, 문제 상황마다 잘 작동하는 강화학습 알고리즘이 다른 경우가 많은데, 이 주차학습의 경우 DQN, DDPG, CEM 중 샘플링을 기반으로 하는 CEM 이 가장 좋은 성능을 보였다는 점도 큰 의미가 있다고 생각합니다. 마지막으로, 기존에는 장애물이 있는 경우에 대한 연구가 없었는데, 이를 시도해 보고 결과를 확인하였다는 점도 의의가 있습니다.

10.한계점 및 후속 연구과제

이 프로젝트의 한계점으로는 우선 빈 주차장에서 주차하는 것을 일차적인 목표로 삼았기 때문에 문제 상황이 매우 이상적이었다는 것입니다. 물론 장애물을 추가한 환경에서의 실험도 진행했지만, 조금 더 다양한 상황에서 훈련을 진행하면 더 풍부한 결과를 얻을 수 있었을 것입니다. (보행자와 같이 움직이는 장애물, 기울어진 경사면 상에서 주차를 하는 상황, 등등) 또한, 실제 자율주행 차량은 다양한 센서를 통해 주변에 대한 정보를 얻어 종합하고 이를 바탕으로 판단하는데, 이 프로젝트에서는 센서는 없고 위치와 각도 기반으로만 리워드를 계산하고 훈련했기 때문에, 갑자기 다른 차가 돌진해오는 상황이나, 골목길에서 아이가 튀어나오는 상황에는 대처할 수가 없다는 것이 한계점입니다.

이를 보완하기 위해서, 후속연구에서는 차량에 여러 방향으로 물체와의 거리를 감지하는 센서와 그 값을 state 변수에 추가해서 이에 대한 리워드도 정의하고 학습한다면, 즉각적인 변화에 대해서도 대응할 수 있는 능력을 갖추게 될 것으로 기대하고, 이를 통해 더 현실적인 문제해결에 가까워질 수 있을 것입니다.

DQN 과 DDPG 의 모델기반 훈련은 훈련시간이 매우 오래 걸리고, 연산량이 매우 크며, 실제 주행 능력과 문제 해결 성능이 뛰어나지 못하다는 단점이 있었던 반면, 모델에 현재 state 를 넣으면 모델을 통과해서 바로 action 이 계산되므로 실시간 적용이 매우 빠르다는 장점이 있습니다.

CEM (with DM)은 결과적으로 문제해결 능력, 주행능력이 모두 뛰어났지만 샘플링 기반 이므로 매 time step 마다 다음 action 을 결정하는데 시간이 매우 오래 걸립니다.

두번째 후속 연구로 이 둘의 장점만을 합쳐서 DQN 이나 DDPG 에서 의사결정을 할 때에 미리 학습된 Dynamic Model 의 예측 결과를 반영해서 훈련을

진행한다면 주행 능력도 향상되고, 동시에 빠른 실시간 연산을 할 수 있으므로 더 나은 agent 를 만들 수 있을 것으로 기대합니다.

참고 문헌

Marcin Andrychowicz, et al. “Hindsight Experience Replay.” *Neural Information Processing Systems*, vol. 30, July 2017, pp. 5048–58.

Timothy P. Lillicrap, et al. “Continuous Control With Deep Reinforcement Learning.” *International Conference on Learning Representations*, July 2016.

Volodymyr Mnih, et al. “Playing Atari With Deep Reinforcement Learning.” *arXiv: Learning*, Dec. 2013.

Google Colaboratory. (n.d.). https://colab.research.google.com/github/eleurent/highway-env/blob/master/scripts/parking_model_based.ipynb

Eom, Hayoung, et al. “Autonomous Parking Simulator for Reinforcement Learning.” *Journal of Digital Contents Society*, vol. 21, no. 2, Digital Contents Society, Feb. 2020, pp. 381–86. https://doi.org/10.9728/dcs.2020.21.2.381.

Moreira, Dinis. “Deep Reinforcement Learning for Automated Parking.” *FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO*, Aug. 2021.

Zhang, Peizhi, et al. “Reinforcement Learning-Based End-to-End Parking for Automatic Parking System.” *Sensors*, vol. 19, no. 18, MDPI AG, Sept. 2019, p. 3996. https://doi.org/10.3390/s19183996.