Convergence Capstone Design

Final Project Report

PPO 기반의 강화학습을 이용한 자율주차

2조

김현우 : PPT제작 및 발표

김창기 : 보고서제작

김범진 : 보고서제작

안종원 : 보고서제작

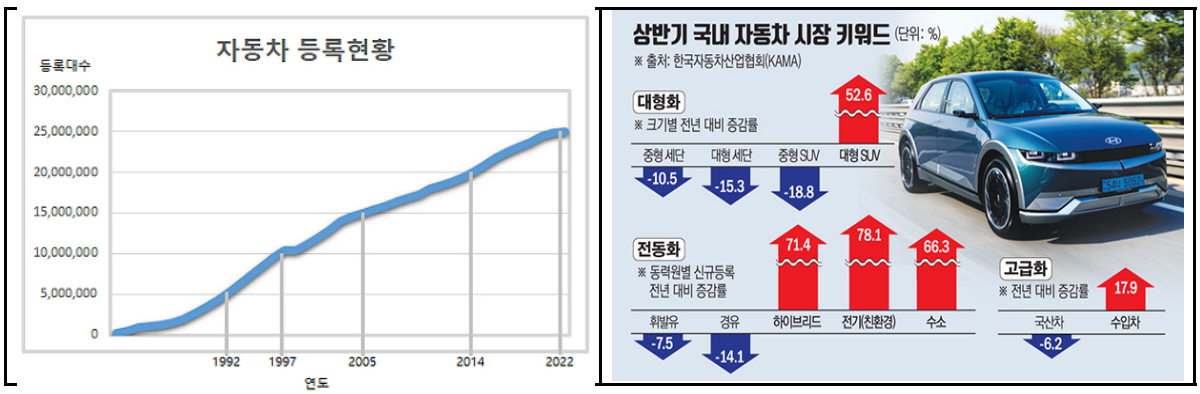
목 차

1. 서론

1. 프로젝트 개요

1.1 프로젝트 선정 배경

21세기 과학기술 분야에서 자주 거론되는 이슈 중 하나는 자율주행이다. 최근의 무인자동차 기술은 차량에 탑재된 센서를 통해 주변 상황을 인식하고 컴퓨터가 차량을 조정하는 인공지능에



[그림1]

기반한 자율주행이 가능한 수준으로 발전하였고 자동차 등록대수는 [그림1]과 같이 매 해 늘어나고 있고 중형 자동차는 줄어들고 있는 반면, 대형 SUV는 늘어나고 있는 추세이다. 주차 공간은 정해져 있는데 자동차 크기는 대형화되는 추세이므로 주차 하는데 어려움이 발생하게 된다. 이에 우리조는 자율주행 주차를 통해 주차하는데 어려움을 줄이고자 한다.

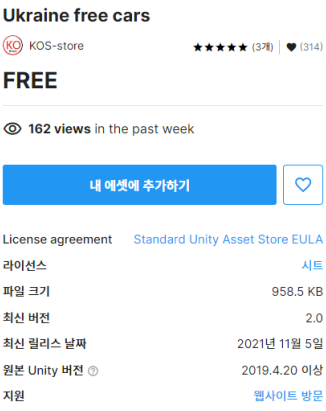
1.2 프로젝트 목표

Unity ML Agent 기반의 PPO 강화학습을 이용해 한정된 주차공간 내에서 양 옆에 주차된 차에 부딪히

지 않고 정해진 라인안에 주차 하는 것을 최종 목표로 한다.

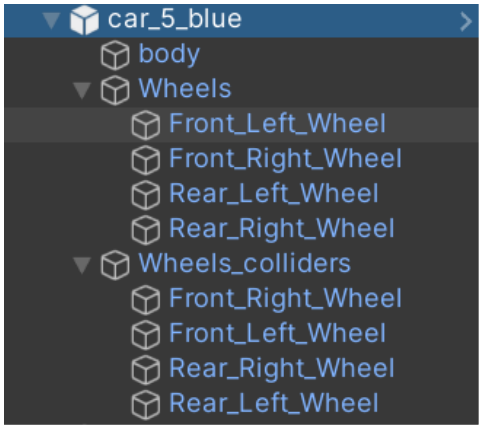
2. 본론

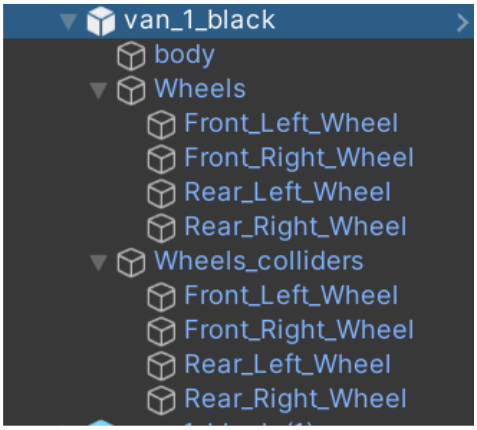
2.1 프로젝트 환경



유니티 에셋 스토어에서 무료 에셋인 Ukraine Free Cars의 car_5_blue를 에이전트로 사용하고 두 대의 van_1_black을 환경에 사용한다.

각 Area는 에이전트인 car_5_blue와 2대의 van, 주차공간인 parking place 그리고 벽과 바닥으로 구성된다.





에이전트 car_5_blue와 두 대의 van은 body, Wheels, Wheels_colliders로 구성되며 Wheels와 Wheels_colliders는 자동차의 바퀴위치에 따라 4개의 요소로 나누어진다.

2.2 PPO 알고리즘

학습데이터를 재사용하는 모델로, Episode 단위로 반영하는 것이 아닌 Step 단위로 학습데이터를 만들어 내어 학습하는 방식이다. PPO는 MM(Minorize-Maximization) 알고리즘을 기반으로 모델링 수식이 구성된다. MM 알고리즘에서는 반복할 때 마다 대리 함수 M을 찾는 것을 목표로 하며, 그 때마다 Optimal point M을 찾는다. 그리고 도출한 point M을 현재 훈련에 사용할 기반정책으로 사용한다. 여기서 정책 M을 기반으로 lower bound를 재계산하며 이 과정을 반복해 나가는 것이 MM 알고리즘이다. 그러나 정책은 학습의 결과에 영향을 미치는 중요한 요소이므로 이 알고리즘을 반복적으로 수행할 때마다 정책을 지속적으로 향상시킨다. PPO의 objective function은 이전 학습시 저장해 둔 기존 보상값과 clip된 보상값들 중 작은 값을 취하는 형태로 되어 있다. 이를 통해서 예외적 결과인 변화가 큰 값의 정책 업데이트는 줄여 안정화된 학습이 되도록 유도한다.

```yaml
behaviors:
  CarBehaviour:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 5120
      learning_rate: 0.00035
      beta: 0.0025
      epsilon: 0.3
      lambd: 0.95
      num_epoch: 5
      learning_rate_schedule: linear

    network_settings:
      normalize: true
      hidden_units: 264
      num_layers: 3

    reward_signals:
      extrinsic:
        gamma: 0.95
        strength: 0.99

    keep_checkpoints: 15
    checkpoint_interval: 100000
    time_horizon: 264
    max_steps: 100000000
    summary_freq: 100000
    threaded: true
```

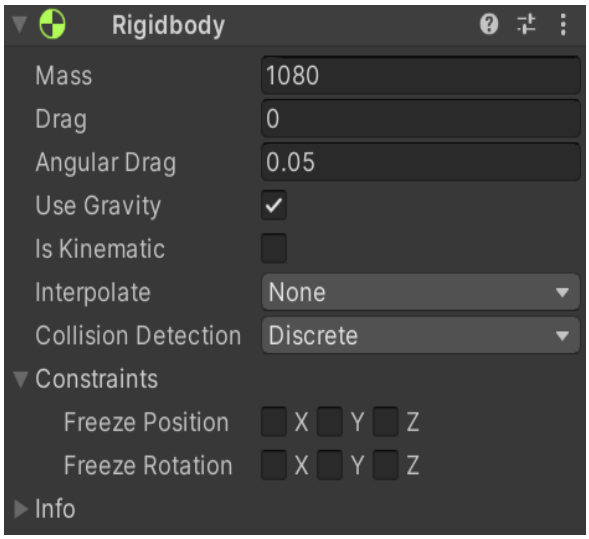
Trainer_type ppo로 설정하고, 연속적인 행동 알고리즘이므로 batch size를 1024로 크게 설정했다. 이외의 learning rate, beta, epsilon, lambd 파라미터 값의 경우 학습을 여러번 수행해 보며 안정적으로 학습을 수행하는 동시에 적절한 학습속도를 갖도록 설정하였다.

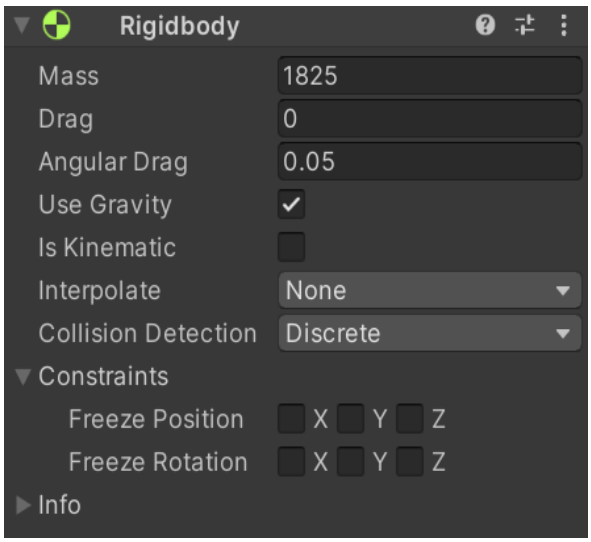
2-3 컴포넌트 설명

1) Rigidbody

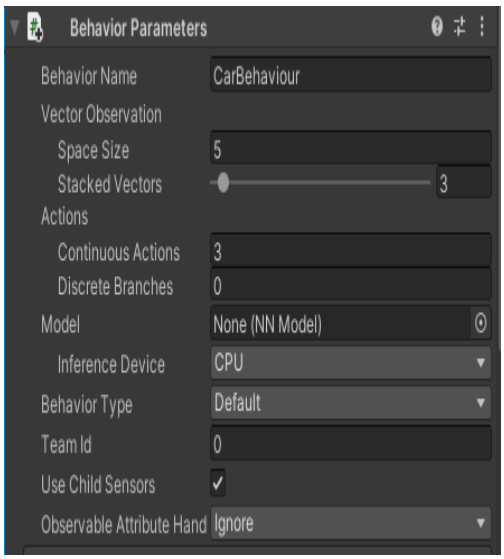
에이전트 차량 car_5_blue와 두 대의 van에 Rigidbody 컴포넌트 사용.

Rigidbody로 에이전트와 van 오브젝트에 물리엔진을 적용. Mass(질량)는 van에 더 큰 값을 부여해 van 환경이 크게 변화하지 않도록 함, 중력 적용, 이산적 충돌 감지





2) Behavior Parameters



Behavior Parameters 컴포넌트를 통해 에이전트의 ML-Agents를 통한 학습에 관련된 다양한 Parameter를 설정

Space Size(벡터 관측의 크기) = 5

Stacked Vector(관측의 누적 횟수) = 3

Continuous Actions = 3

Discrete Branches = 0

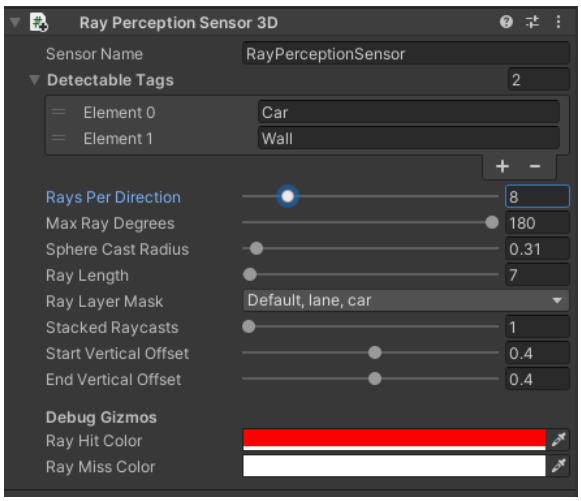
연속적 행동은 3개의 축으로 행해지며 이산적 행동은 없다.

3) Decision Requester

Decision Requester 컴포넌트를 통해 에이전트가 어떤 Action을 취할지 정책에 결정을 요청한다.

Decision Period: 에이전트가 새로운 행동을 결정하는 스텝의 간격, 5스텝마다 한 번씩 행동을 결정하도록 한다.

4) Ray Perception Sensor 3D



Ray Perception Sensor 3D 컴포넌트를 통해 주차선의 간격, 주차된 차들의 간격을 인식해 라인에 알맞게 주차도록 한다.

2-4 ML-Agents를 이용한 강화학습 구현

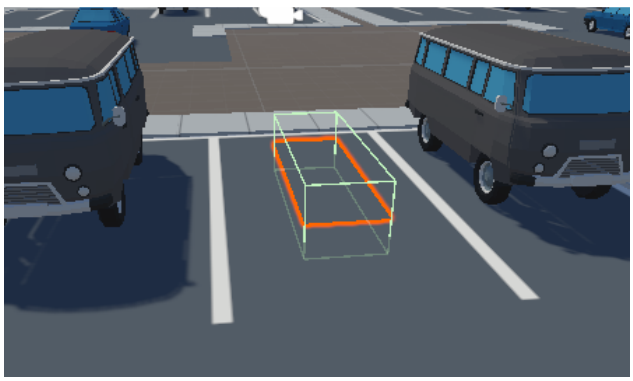
1. Agent, State, Action, Reward 정의

1) Agent: car_5_blue 차량

2) State

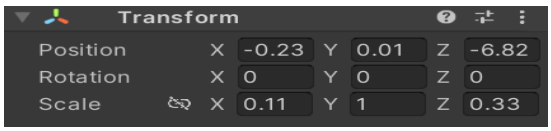
State는 Parking place 사이의 거리, Agent와 Parking place 사이의 각도, Agent의 현재 속도를 설정하였다. Parking place 사이의 거리는 Agent의 현재 좌표에서 Parking place의 좌표를 뺀값을 통해 두 차량 사이에 주차가 잘 되었는지 확인한다. Agent와 Parking place 사이의 각도를 State로 설정한 이유는 두 차량 사이에 주차가 이루어 진 경우라도 일직선으로 되었는지 사선으로 주차가 이루어졌는지에 따라 Reward를 다르게 주기 위함이다. 거리, 각도만으로도 주차가 잘 되었는지 확인 할 수 있으나, Agent의 현재 속도를 추가 설정한 이유로는 실제 차량이 빠른속도로 부딪힐 때 파손이 심한 것을 나타내기 위함이다. 이는 Agent가 부딪혔을 때 현재속도를 계산해 속도가 클수록 더 많은 -reward를 받도록 설정한다. 만약 차선 사이로 주차가 되고 속도가 2미만으로 벽과 부딪히지 않고 주차에 성공한다면 +reward를 받도록 설정한다.

① Agent와 Parking place 사이의 거리

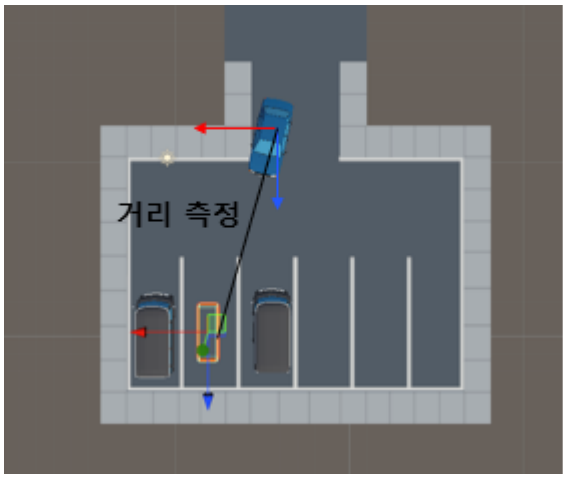


Parking Place는 주차 공간에서 양쪽 van 사이의 위치로 고정하며 이 지점의 x,y,z 좌표는 아래와

같이 일정한 값으로 설정된다.



한편 Agent 차량은 출발 지점에서부터 주차 공간에 도달할 때까지 지속적으로 값이 변화한다. 이로부터, transform.position 함수를 사용해 Agent의 현재 좌표에서 Parking Place의 좌표를 뺀 값을 계산하여 거리를 측정한다.



이 거리를 계산하는 함수를 C# 스크립트 상에서 작성하면 아래와 같다.

```
float distanceToTargetX = Mathf.Abs(transform.position.x - target.transform.position.x);
float distanceToTargetZ = Mathf.Abs(transform.position.z - target.transform.position.z);
```

transform.position을 사용해 Agent의 현재 좌표에서 Parking Place의 좌표를 뺀값을 사용한다.

② Agent와 Parking place 사이의 각도

Agent가 주차공간에 들어오면 Angle 함수를 이용해 Agent와 Parking Place 간의 각도를 0도에서 180도 사이의 값으로 받아와 Reward에 반영한다.

```
if(parkingplace){
    float angleToTarget = Vector3.Angle(transform.forward, target.transform.forward);
```

③ 에이전트 차량의 현재 속도

```
private float _currentSpeed;
```

```
public float CurrentSpeed{ get { return _currentSpeed; }}
```

Agent의 현재 속도를 받아 충돌시 빠른속도로 충돌할 경우 Reward를 더 크게 감점하여 성공적인 주차가 되도록 한다.

```
void OnCollisionStay(Collision collision)
{
    if(collision.gameObject.tag == "Car")
    {
        float reward = -Mathf.Abs(carController.CurrentSpeed) * 5f;
        AddReward(reward);
        EndEpisode();
    }
}
```

3) Action: steering, accel, reverse

Action의 경우 차량의 동작을 담당하는 CarControll script와 연동되어 차량의 조향과 전후의 움직임을 나타낸다. 우선 차량의 동작 같은 경우는 전체적으로 Unity의 기능인 WheelCollider를 사용하여 구현을 진행하였다. WheelCollider는 굉장히 현실적인 자동차의 동작을 구현할수 있도록 만들어놓은 장치로 차량의 단순한 이동뿐만 아니라 깊이있게 들어가면 서스펜션이나 댐퍼, 타이어의 마찰, 각종 토크등 차량에 대하여 굉장히 자세한 부분까지 컨트롤이 가능하도록 구성되어 있다. 여기선 차량의 자세한 특성보단 차량의 움직임이 주된 요소임으로 그부분에 집중하였고, 바퀴에 Quaternion 좌표계를 구현하여 회전에대한 축도 설정하여 더 자세하게 바퀴의 동작을 구현할 수 있게 만들었다.

해당 스크립트 내에서 차량의 구동을 위해서 Move함수를 만들어 조향과 전진 후진에대한 데이터를 줄수 있게 하였고

```
for (int i = 0; i < 4; i++)
{
    Quaternion quat;
    Vector3 position;
    m_WheelColliders[i].GetWorldPose(out position, out quat);
    m_WheelMeshes[i].transform.position = position;
    m_WheelMeshes[i].transform.rotation = quat;
}
```

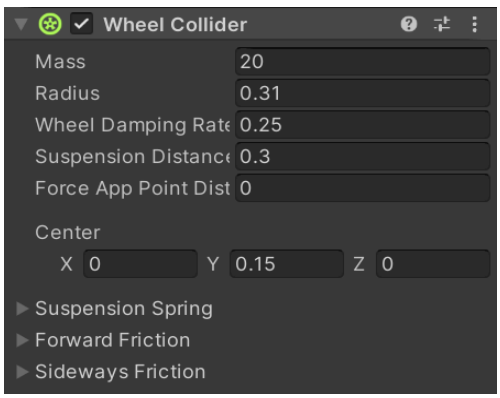
위와 같은 for문을 이용하여 각각의 바퀴들에 대한 정보를 줄 수 있게 하였다. Wheelmeshes[n]은 각각의 바퀴를 뜻하기 때문에 앞바퀴와 뒷바퀴에 대한 각각의 설정으로 조향을 할때는 앞바퀴만 움직이도록, 브레이크를 밟을때는 뒷바퀴만 멈출수 있게 구성되어 있다.

```
//clamp input values
steering = Mathf.Clamp(steering, -1, 1);
AccelInput = accel = Mathf.Clamp(accel, -1, 1);
BrakeInput = footbrake = -1*Mathf.Clamp(footbrake, -1, 0);
handbrake = Mathf.Clamp(handbrake, 0, 1);
```

Move 함수의 구동부분은 위와같이 입력된 값에 대하여 Clamp되어 -1~1 사이의 값이 입력될수 있도록 설정되어 있다.

```
public override void OnActionReceived(ActionBuffers actions)
    float steering = actions.ContinuousActions[0];
    float accel = actions.ContinuousActions[1];
    float reverse = actions.ContinuousActions[2];
```

Agent에 구현한 Action은 3가지로 자동차의 조향을 위한 wheel의 steering, 자동차가 앞으로 움직이도록 wheel이 회전하는 accel, 자동차가 뒤로 움직이도록 wheel이 회전하는 reverse이다. 사실 앞에서 말한 Move 함수에는 Steering과 accel 이 두개로 움직임이 결정되는데 accel의 +에 대한 움직임은 accel이, -에 대한 움직임은 reverse를 이용하여 각각의 움직임을 얻을수 있ㄱ 하였고, Continuous 한 값을 입력받아서 Move함수를 동작시킬 수 있도록 하였다.



각각의 바퀴에 Wheel Collider 컴포넌트를 적용해 Action이 구현되도록 하였다.

4) Reward

Reward를 설명하기 앞서 본 프로젝트 팀에서는 성공적인 주차를 고정된 Parking Place를 기준으로 Agent가 wall이나 van 등과 충돌하지 않고 lane을 밟지 않고 Parking Place의 지정된 좌표에 수직으로 들어온 상태라고 정의하였다. 이런 정의를 기준으로 아래 상황들에 대한 reward의 값을 다음과 같이 정리하였다.

① 주차공간에 들어왔을 때의 reward

Target과 Agent 간의 각도를 Θ라 정의하고 Θ의 값에 따라 reward를 아래의 함수로 정리할 수 있다.

$$R_1(\theta) = \begin{cases} 2\ (\theta = 0) \\ 0\ (\theta = \frac{\pi}{2}) \end{cases}$$

한편, target과 agent 간의 거리를 r, agent의 속도를 v라고 하고 앞서 정의한 agent의 성공적인 주차가 이루어졌다고 가정했을 때, reward 값은 아래와 같이 정의하였다.

$$R_2\left(\theta, r, v \mid \theta < \frac{\pi}{60}, r < 1, v < 2.5\right) = 10$$

코드 상에서 살펴보면,

```
float angleToTarget = Vector3.Angle(transform.forward, target.transform.forward);
```

먼저 target과 Agent 간의 각도를 변수 angleToTarget으로 설정

```
angleToTarget = Mathf.Clamp(angleToTarget, 0f, 90f);
float angleReward = (-(1f/45f) * angleToTarget) + 1f;
totAngleChangeReward = angleReward + 1f;
```

각도 차이가 적을 수록 많은 reward가 부여되도록 해 각도가 0도 일 때 +2의 reward, 90도 일 때 0의 reward가 부여되도록 한다.

```
if(angleToTarget < 3f && distanceToTarget < 1f && Mathf.Abs(carController.CurrentSpeed) < 2.5f){
    Debug.Log("parking success!");
    reward += 10f;
    EndEpisode();
}
```

주차공간과 차와의 각도가 3도 미만, 거리가 1 미만, 현재 속도가 2.5 미만일 때를 성공적인 주차로 판단하고 +10의 reward를 부여 후 에피소드를 종료한다.

② 주차 공간에 들어오지 않았을 때의 reward

```
float distanceToTargetX = Mathf.Abs(transform.position.x - target.transform.position.x);
float distanceToTargetZ = Mathf.Abs(transform.position.z - target.transform.position.z);

float lastDistanceToTargetX = Mathf.Abs(lastPosition.x - target.transform.position.x);
float lastDistanceToTargetZ = Mathf.Abs(lastPosition.z - target.transform.position.z);

float directionChangeX = lastDistanceToTargetX - distanceToTargetX;
float directionChangeZ = lastDistanceToTargetZ - distanceToTargetZ;


totDirectionChangeReward = (directionChangeX + directionChangeZ) * 1f;
totDirectionChangeReward = Mathf.Clamp(totDirectionChangeReward, -0.5f, 0.5f);

float distanceRewardX = (1f - distanceToTargetX/20f);
float distanceRewardZ = (1f - distanceToTargetZ/20f);
```

x좌표와 z좌표에 대한 과거의 target 까지의 거리와 현재의 target 까지의 거리의 차를 계산하여 합산

```
float distanceRewardX = (1f - distanceToTargetX/20f);
float distanceRewardZ = (1f - distanceToTargetZ/20f);

totDistanceReward = (distanceRewardX + distanceRewardZ) / 20f;
```

과거의 거리는 고려하지 않고 현재 target 까지의 거리가 짧을수록 높은 reward

```
reward += totDirectionChangeReward + totDistanceReward;
```

위 두개의 reward 값을 더해서 최종 reward 값을 부여한다.

위에 작성된 코드로부터 agent와 target 간의 거리의 초기 좌표갑과 현재 좌표값 사이의 차이를 모두 합산한다. 이로부터 얻은 값을 Δd라고 정의한다. 그리고 현재 agent의 위치와 지정된 target 간의 거리를 d라고 정의한다. Δd와 d의 값으로부터 위에 작성된 코드로부터 각각의 reward 값을 계산하고 그 두 개의 reward 값들을 합산하여 거리에 대한 최종 reward 값을 계산한다. 이는 아

래의 식으로 정리할 수 있다.

$$R_3 = R_{\Delta d} + R_{d'}$$

③ 주차선을 밟았을 때

```
void OnTriggerStay(Collider other)
{
    if(other.gameObject.tag == "lane")
    {
        AddReward(-2f);
    }
}
```

주차선을 밟으면 -2의 reward를 부여하고 초기화는 진행하지 않는다.

$$R_4 = -2$$

④ 벽과 충돌했을 때

```
void OnCollisionEnter(Collision collision)
{
    print(collision.gameObject.tag);
    if (collision.gameObject.tag == "Wall")
    {
        AddReward(-1f);
        EndEpisode();
    }
}
```

벽과 충돌 시 -1의 reward를 부여하고 에피소드를 종료한다.

$$R_5 = -1$$

⑤ 차와 충돌했을 때

```
void OnCollisionStay(Collision collision)
{
    if(collision.gameObject.tag == "Car")
    {
        float reward = -Mathf.Abs(carController.CurrentSpeed) * 5f;
        AddReward(reward);
        EndEpisode();
    }
}
```

지정된 Parking place 양 옆에 위치한 van과 충돌했을 경우이며 이 때에는 Agent의 속도 값을 reward 값에 반영되도록 설정하였으며 속도의 값이 더 커질수록 reward의 – 값이 커지도록 계산식을 구성하였다. Agent의 속도를 v로 정의했을 때, reward는 아래의 식으로 정리해 볼 수 있다.

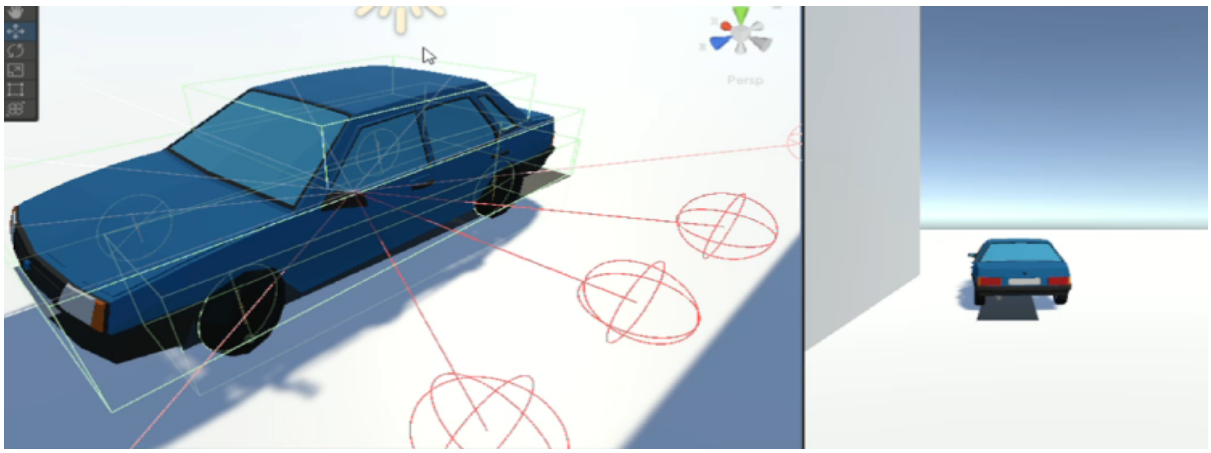
$$R_6(v) = -5|v|$$

위에서 제시된 5가지 경우에 따라 계산되는 최종 Reward 값 R은 $R = R_1+R_2+R_3+R_4+R_5+R_6$ 의 식으로 종합하여 정리해 볼 수 있다.

2-5 강화학습 수행 결과

최종 학습 환경을 구성하기 위해 먼저 매우 간단한 환경을 구성하여 PPO 기반의 강화학습이 제대로 수행되는 지 확인하고 학습 환경을 추가해 나갔다.

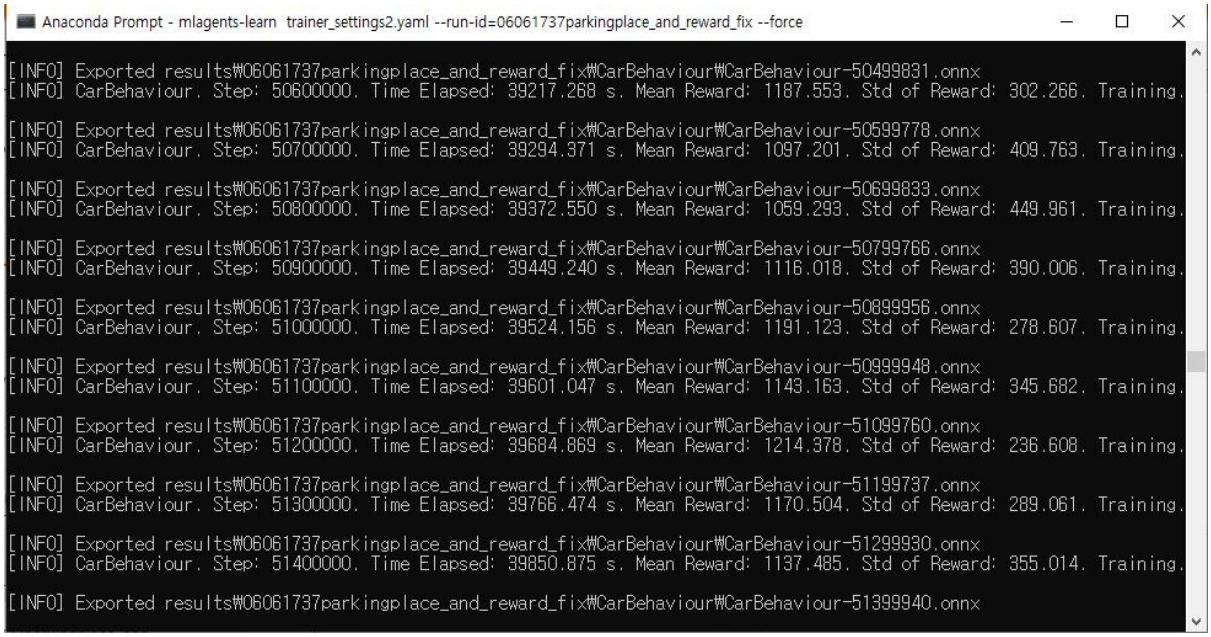
처음에는 주차공간과 Agent 차량만 있는 환경을 구성해 학습을 수행했다.



벽과 주차선 그리고 다른 차량이 없기 때문에 Reward 부여가 단순하게 이루어지며 매우 짧은 시간내에 충분한 학습이 완료되어 성공적으로 주차하는 것을 확인할 수 있었다.

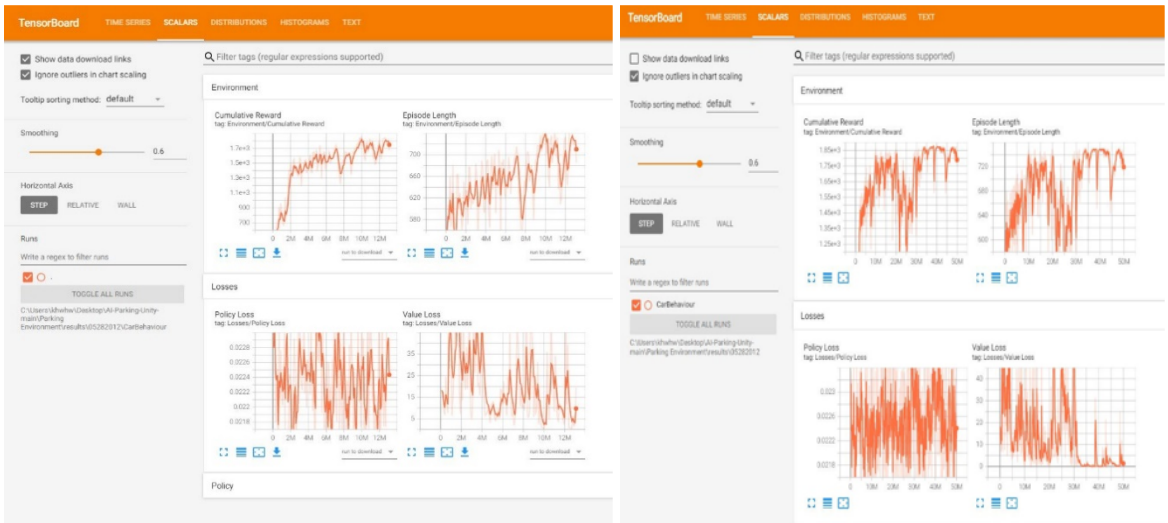
다음은 오브젝트를 직접 만들어 주차장을 구현했다. 주차선을 추가하고 벽을 설정해 선이나 벽과 충돌했을 때 -Reward를 받도록 구현했다. 주차공간만 있을 때 보다 학습 시간이 오래 걸렸다.

최종적으로 주차공간 양 옆에 두 대의 van을 추가하여 환경 구성을 완료했다.



Mean Reward 값은 계속 증가하여 1200에 가까운 값으로 수렴하였고 안정적으로 주차하는 것을 확인할 수 있었다.

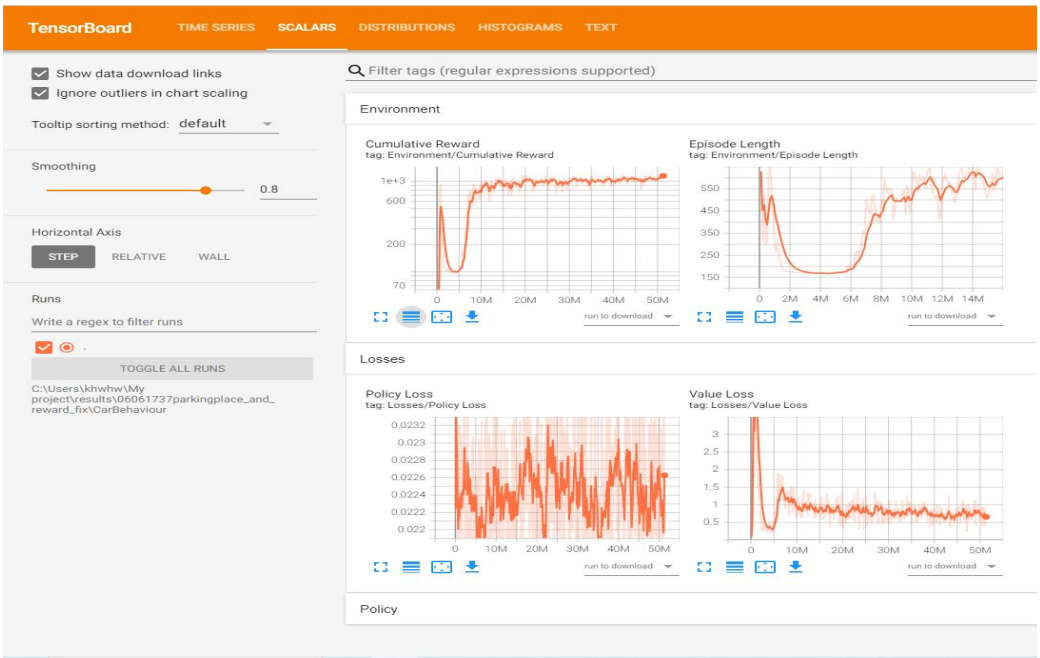
2-6 텐서보드를 이용해 확인한 결과

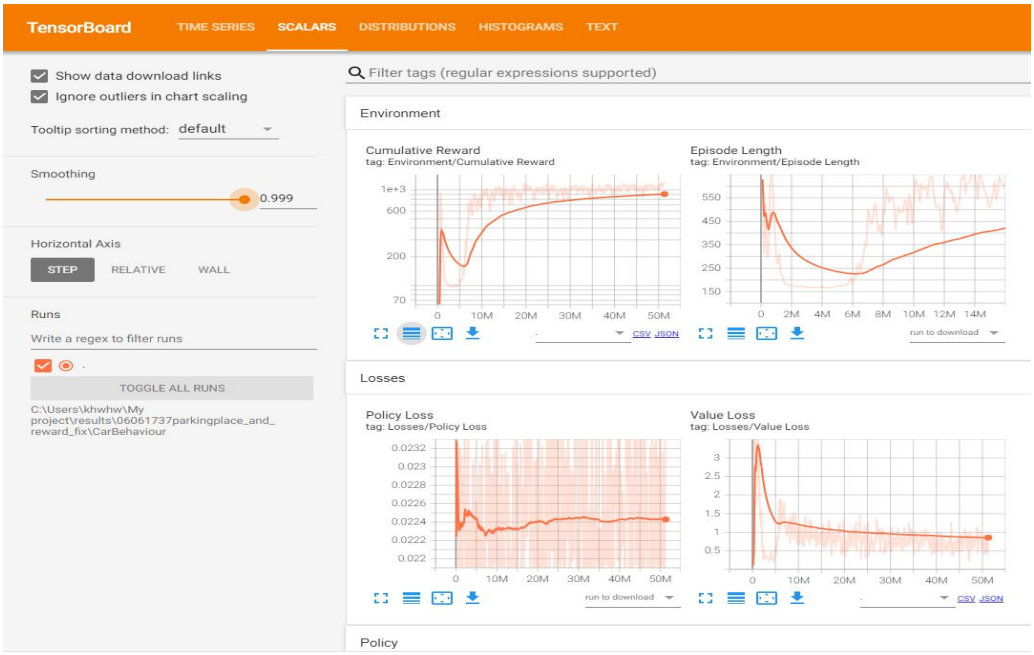


강화학습 명령창에서는 Mean Reward 값이 꾸준히 상승하여 수렴하였으나 텐서보드를 이용해 확

인해본 결과 cumulative reward 그래프가 계속해서 상승하지 않고 변동이 심했고 Loss 값 또한 변동 폭이 매우 커 이상적인 그래프와는 거리가 멀었다.

이러한 현상은 reward 체계의 복잡성과 랜덤한 행동을 했을때에 너무 극단적인 +reward와 -reward로 인하여 발생한다고 판단이 되었다. 한가지 예시로 거리에 대한 reward는 fixed update를 통해 연속적이게 들어가므로 그 과정에서 너무 많은 reward들의 경우가 중첩되다보니 설계할때의 의도와 맞지 않게 여러 reward가 맞물려 굉장히 불안정한 reward를 얻는 경우를 확인이 가능하였고, 더 문제가 되었던 점은 reward의 기본적인 크기가 전체적으로 크다보니, agent가 exploitation을 진행하며 random한 행동을 계속해서 학습이 진행되는데 이로인하여 random한 행행동 할 때 빠른속도로 벽이나 차에 부딪히는 등의 큰 -reward가 발생하기도하고 어쩌다가 한번 주차에 성공하여 큰 +reward가 들어오는 등 random한 행동에 대해서 reward가 그대로 반영되어 불안정한 모습을 보였기 때문에 우선 reward 체계를 간략화하고, 전체적인 reward의 크기를 약 1/10, 1/100 꼴로 감소시켰으며 PPO의 하이퍼파라미터 중 랜덤한 행동에대한 빈도수를 담당하는 beta값을 약간 감소시키자





위와같이 reward의 경우 약 1000에서 안정적으로 수렴하는 모습의 그래프를 얻을 수 있었고, Loss의 경우는 시작했을때만 급격하게 증가하였다가 이후 학습이 진행되며 약 1보다 작은값으로 수렴해가고 우하향하는 모습을 확인이 가능했다.

위의 tensorboard 화면에서 확인할수 있는 한가지 문제점은 Episode length에 대한 것인데, 학습이 진행되고 reward가 수렴해가는 단계에서 일반적으로 생각해보면 학습이 정상적으로 진행되는 것이므로 Episode length가 점차 줄어가야한다. 하지만 위의 tensorboard 결과를 보면 학습이 진행될 수록 점차 증가하는 Episode length를 확인 할 수 있는데 이는 학습성공의 기준. 즉 주차의 기준을 너무 엄격하게 잡아서 학습완료가 이루어지지 않는것이다. 여러 번 학습기준을 풀어주며 시험해 보았는데 이로인해 다시 reward 부분이 수렴하지 않는 등의 문제가 발생하여 이런식으로 결과를 내게 되었다. 다음 프로젝트에선 이런 부분에대한 보완이 필요할것이다.

3. 결론

본 프로젝트 팀은 자동주차 시스템을 Unity 환경 상에 구현하는 것을 목표로 프로젝트를 진행하였고 이를 위해 Asset store와 직접 구현 등의 수단으로부터 해당 주차 환경을 설정하였고 PPO 기법을 이용해 환경 상에 고정된 주차 지점으로 Agent 차량이 성공적으로 주차되는지에 대한 여부를 강화 학습을 통해 확인하였다.

실습 결과, 초기에는 정상적으로 학습이 진행되기는 하나, 불완전하게 주차가 이루어지는 모습을 확인하였고 이러한 점으로 인해 reward 값이 불안정하게 튀는 모습을 tensor board 상에서 확인할 수 있었으며 loss 값 역시 굉장히 불안정한 결과 값으로 나타나는 것을 board 상으로 확인할 수 있었다. 이를 보완하기 위해 angle 함수를 추가로 구현해주고 reward 함수에 부여된 값의 수정을 거친 뒤 Ray Perception Sensor 3D에 대한 설정을 추가로 해 준 후, 다시 강화학습을 진행할 결과 이전보다 훨씬 안정적인 모습으로 주차가 이루어짐을 Unity 환경 상으로 확인할 수 있었으며 가상환경 화면에서 나타나는 mean reward 값 또한 stable하게 나타났다. 또한, tensor board 상의 reward 값과 loss 값도 초기에 진행했던 강화학습 때와는 달리 일정한 값으로 수렴하는 형태의 그래프를 나타내는 것을 확인할 수 있었다.

결론적으로, 본 프로젝트 팀이 구현한 자동주차 프로그램이 Unity 환경 상에서 성공적으로 구현되었다고 볼 수 있으며, 향후 다른 형태로 구성된 주차환경에서도 자동주차 프로그램이 정상적으로 구동될 가능성을 이번 프로젝트로부터 보였음을 주장할 수 있다. 하지만, 다른 주차환경에서도 정상 구동되기 위해서는 지금 경우보다 훨씬 더 다양한 경우와 결과값을 모두 reward에 대한 함수에 반영하고 그에 맞는 강화학습의 형식을 모색하는 등의 보완점을 마련해야 할 것이고 그렇게 한다면, 실제 차량에 도입되는 날도 머지않아 실현될 것이다. 이러한 보완을 준비하는 것은 자동주차 프로그램을 구현하려는 연구자들에게 앞으로 주어진 과제이며 이러한 과제에 대한 간략한 설명을 끝으로 본 주제에 대한 설명을 여기서 마무리하는 바이다.