# COSE361 Final Project 2/2

2018320141 이승준

## 1. Introduction

이번 final project 2/2에서는 제공된 팩맨 게임 룰을 활용하여 게임을 전략적으로 플레이 할 수 있는 인공지능 agent를 설계하는 것이 목표였다. 각각 2개의 agent로 이루어져 있는 red team과 blue team이 경쟁하여, 어느 팀이 상대방 영역에 있는 음식을 더 많이 먹는 지에 관해 승부를 가리는, 일종의 땅따먹기 게임이었으며, 해당 게임 내에서 눈여겨볼 특징은 다음과 같다.

- 음식을 먹고 반드시 자신의 구역으로 들어와야 점수가 누적된다. 따라서, 상대편 팩맨이 우리 쪽 음식을 먹어도, 자신의 구역으로 들어가기 전에 저지하면, 상대편 점수는 높아지지 않는다. 반대로, 우리 팀 입장에서는 상대편 음식을 먹는 것도 중요하지만, 먹은 음식을 우리 쪽 구역으로 가져와 점수화 하는 것 또한 매우 중요하다.
- 2개의 agent는 둘 다 팩맨이 될 수도, 혹은 둘 다 ghost가 될 수도 있다.
- 캡슐을 적절히 활용하면 높은 점수를 확보할 수 있다.
- 음식의 위치에 따라, 안전하게 먹을 수 있는 음식이 있고, 위험을 감수해야만 먹을 수 있는 음식이 있다.

이번 프로젝트에서는 위에서 언급한 특징들을 기반으로 agent를 설계하였다.

## 2. method

Method 단원에서는 이번 final project에서 구현된 여러 팩맨 알고리즘에 관하여 간략히 설명하도록 하겠다. 이번 프로젝트에서는 총 3개의 알고리즘을 설계하였으며, 아래와 같은 이름의 파이썬 파일로 구현되어있다.

- **your_baseline1.py**
- **your_baseline2.py**
- **your_baseline3.py**

각각의 파이썬 파일에는 총 2개의 agent가 구현되어 있다. 하나는 주로 상대방 구역의 음식을 먹는 역할을 하며, 나머지 하나는 자신의 구역에 들어온 상대방 팩맨을 저지하는 역할을 한다. 이 둘을 각각 attacker와 defender로 부르도록 하겠다.

### 2.1. your_baseline1.py

your_baseline1.py는 a-star 알고리즘을 기반으로 작성되었다. 해당 알고리즘에서 가장 중요한 부분은 a-star을 적용할 때, goal state을 상황에 따라 다르게 설정하여, 효율적으로 문제를 해결하도록 구현하였다는 점이다. goal state에 따라 팩맨을 총 7개의 작은 문제로 분할하였으며, 다음과 같다.

**Food Problem:** agent를 중심으로 가장 가까운 위치에 있는 상대방 구역의 음식을 goal state으로 설정한다. attacker를 위한 problem이며, 코드에는 *FoodProblem*이라는 함수로 구현되어 있다.

**Easy Food Problem:** your_baseline1.py에서는 상대편 음식을 음식의 위치와 주변 지리에 따라 쉬운 음식(easy food)과 어려운 음식(hard food), 총 2부류로 나누어 해결하고 있다. 쉬운 음식은 상대편 음식의 위치로 부터, 자신의 구역으로 돌아갈 수 있는 경로가 2개 이상인 음식을 말하며, 어려운 음식은 자신의 구역으로 돌아갈 수 있는 경로가 한 개 이하인 음식을 말한다[1]. Easy Food Problem에서는 agent의 위치를 기준으로 가장 가까운 쉬운 음식을 goal state으로 설정한다. attacker를 위한 problem으로, 코드에는 *EasyFoodProblem*이라는 함수로 구현되어 있다.

**Hard Food Problem:** agent를 중심으로 가장 가까운 위치에 있는 어려운 음식을 goal state으로 설정한다. attacker를 위한 problem이며, 코드에는 *HardFoodProblem*이라는

[1]해당 게임에서는 agent가 음식을 먹은 뒤, 반드시 자신의 구역으로 안전하게 돌아와야만 점수가 누적되는 규칙을 적용하고 있다. 따라서, 음식을 먹은 뒤, 자신의 구역으로 안전하게 돌아올 수 있는 경우의 수에 따라 음식을 나누어, 문제를 효율적으로 해결하였다.

함수로 구현되어 있다.

**Capsule Problem:** agent를 중심으로 가장 가까운 위치에 있는 상대편 캡슐을 goal state으로 설정한다. attacker를 위한 problem이며, 코드에는 *CapsuleProblem*이라는 함수로 구현되어 있다.

**Back Home Problem:** 자신의 구역을 goal state으로 설정한다. 해당 문제에서 agent는 자신의 구역으로 가는 최단 경로를 계산하여 돌아간다. attacker를 위한 problem이며, 코드에는 *BackHomeProblem*으로 구현되어 있다.

**Emergency Problem:** attacker가 위급한 상황일 때 사용하는 문제이다. 해당 문제에서 agent는 상대편 캡슐과 자신의 구역 중 가장 가까운 곳을 goal state으로 설정한다. 코드에는 *EmergencyProblem*으로 구현되어 있다.

**Invader Problem:** defender를 위한 problem으로, 자신의 구역으로 들어온 상대편 팩맨을 goal state으로 설정한다. 코드에는 *InvaderProblem*이라는 함수로 구현되어 있다.

your_baseline1.py에는 *Attacker*와 *Defender*라는 클래스로 2개의 agent를 구현하고 있다. 각각의 agent는 매번 다음 action을 고를 때 마다, 위 7개의 문제 중 자신의 상황에 가장 필요한 문제를 골라 a-star 알고리즘을 수행한다. 먼저 *Attacker*는 다음과 같은 방식으로 문제를 선정한다.

- 자신을 쫓는 ghost와의 거리가 5보다 작으면 Emergency Problem을 선택한다.
- agent의 numCarrying[2] 수가 15보다 많거나, 남은 시간이 얼마 남지 않았을 때는 Back Home Problem을 선택한다.
- 상대편 agent가 scared 상태가 되었고, 원래대로 복귀될 때 까지 15초 이상의 시간이 남았으면 Hard Food Problem을 선택한다. 어려운 음식은 해당 음식을 먹었을 때, 자신의 구역으로 돌아가는 경우의 수가 적어, 점수화 하기 어려운 음식들이기 때문에, 상대편 agent가 scared 상태일 때 먹는 것이 더 안전하다.
- 어려운 음식의 개수가 남았고, 상대편 agent의 scared 시간이 10초 이하이면 Capsule Problem을 선택한다.
- agent의 numCarrying 수가 3이하이면 Easy Food Problem을 선택하여, 음식을 더 먹도록 유도한다.
- 위의 상황에 모두 해당되지 않으면 Food Problem을 선택한다.

Defender 또한 Attacker 처럼 상황에 맞는 문제를 선택하여 다음 action을 결정한다. 다만 흥미로운 점은, Defender 같은 경우, 특수한 상황에서 마치 Attacker 처럼 행동하도록 설계되었다는 점이다. Defender의 action 결정 방식은 다음과 같다.

- 상대편 agent가 scared 상태이고, scared time이 20초보다 많으면 Food Problem을 선택해서 Attacker인 것처럼 상대편 음식을 먹는다.
- 상대편 agent가 scared 상태일 때, 남은 scared time에 비해, 자신의 구역으로 돌아갈 시간이 특정 임계치보다 적다면, Back Home Problem을 선택하여, agent가 먹은 음식을 점수화한다.
- 자신의 구역으로 들어온 팩맨의 수가 0이어서 방어할 팩맨이 없다면, Food Problem을 선택하고, 상대편 구역으로 들어가 음식을 먹는다. 이때, 자신의 구역으로 부터 7보다 멀리 있는 지역은 가지 않으며, numCarrying 수가 4보다 많거나, 자신을 잡으러 온 ghost와의 거리가 5보다 작으면 Emergency Problem을 선택하여, Defender가 ghost에 먹히지 않도록 한다[3].
- 위의 상황에 모두 해당되지 않으면, Invader Problem을 선택하여 자신의 구역으로 들어온 상대편 팩맨을 저지한다.

이처럼 your_baseline1.py는 a-star 알고리즘을 기반으로, 각각의 상황에 알맞는 문제를 채택하여 다음 action을 선택해 나가도록 구현되어 있다. Figure1에서 확인할 수 있는 것처럼, 해당 알고리즘은 기존의 baseline.py과 비교하여 승률 100%와 평균 점수 12.2점을 기록하며 압도적으로 높은 성능을 보였다.

### 2.2. your_baseline2.py

your_baseline2.py도 2개의 agent를 구현하였으며, 하나는 attacker 역할을, 나머지 하나는 defender 역할을 한다. attcker 역할의 agent는 your_baseline1.py의 *Attacker* 클래스를 그대로 사용하였으며, defender 역할의 agent는 approximate-Qlearning을 기반으로 구현되었다. 따라서 defender 같은 경우 action을 생성할 때마다 사전에 설

[2]numCarrying은 agent가 상대편 음식을 먹고, 아직 자신의 구역으로 가져오지 않은 음식의 개수를 말한다. 쉽게 말해, 팩맨이 먹었지만, 아직 점수화 되지 않은 음식의 개수를 말한다.

[3]Defender는 먹히지 않는 것이 정말 중요하다.

정해 놓은 features들의 가중치를 업데이트 해나간다. features는 baseline.py 코드를 참고하여 설정되었으며 다음과 같다.

- **invaderDistance:** 자신의 구역에 들어온 팩맨까지의 거리
- **onDefense:** 자신이 pacman이 아닌, Defender 역할을 하고 있으면 1.0을 할당하고, 아니면 0.0을 할당
- **numInvaders:** 자신의 구역에 들어온 팩맨의 수
- **stop, reverse:** agent가 다음으로 선택한 action에 관한 features
- **DistToCapsules:** 자신의 구역에 있는 캡슐까지의 거리[4]

defender 역할의 agent는 *DefensiveQAgent* 클래스로 구현되어 있으며, your_baseline1.py의 *Defender* 클래스와 유사하게 동작한다. 다만, *Defender* 클래스가 Invader Problem을 선택하는 상황에서 *DefensiveQAgent* 클래스는 a-star 알고리즘을 사용하지 않고, approximate-Qlearning을 사용하여 다음 action을 결정한다.

Approximate-Qlearning을 적용하여 가중치를 업데이트 할 때 많은 어려움이 있었는데, 정리해보면 다음과 같다.

- features와 reward 설정이 굉장히 까다롭다.
- 훈련을 하는데 굉장히 많은 시간이 소요된다.
- 훈련 진행 상황을 확인할 수 있는 지표 설정이 굉장히 어렵다.
- 특정 맵에서 가중치를 잘 훈련시켜도, 다른 랜덤맵에서는 저조한 성능을 보이는 경우가 있다.[5]

특히 2번째와 3번째 문제가 가장 까다로웠다. 딥러닝에서는 훈련을 진행할 때, 보통 loss와 average를 활용하여 진행 상황을 확인하는데, 팩맨같은 경우 훈련 상황과 loss의 움직임이 크게 관련이 없어보였다.[6] 그 이유로 크게 2가지를 추측해보았는데, 그 중 첫 번째 문제로 손실함수를 되짚어보았다. 해당 학습에서는 손실함수로 mse를 사용하였는데, 팩맨같은 복잡한 문제에서 다소 단순한 손실함수를 사용한 것이 느린 학습속도를 유발한 것으로 추측하였다. 두 번째로, loss는 모든 agent의 움직임과 맵 내부의 상황에 따라 결정되는데, 만약 agent의 움직임에 랜덤성이 부여된다면, 학습이 잘되는 것과는 상관없이 어느 정도 loss가 진동할 수 있다는 생각을 해보았다. 실제로 baseline.py 같은 경우 agent의 행동에 어느 정도의 랜덤성이 부여되어 있다.[7].

위와 같은 문제를 해결해보고자, 학습과정에서 다양한 전략을 시도해보았다. 먼저, 가중치를 업데이트할 때, 학습률을 조정해주는 여러 알고리즘들을 사용해 보았으며, 다음과 같다.

- Gradient descent
- AdaGrad
- Adam
- Newton

첫 번째 gradient descent는 고정되어 있는 학습률을 적용하여 가중치를 업데이트 해나가나는 가장 보편적인 경사하강법이며, 아래 3개의 방식은 딥러닝에서 자주 활용되는 학습 기법들이다. 3개 중 가장 안정적으로 학습이 되었던 기법은 AdaGrad였으며, 실제로 점진적으로 학습률을 줄여나가며, 안정적으로 학습을 진행해 나가는 모습을 보였다. Adam 같은 경우 AdaGrad 보다 더 부드럽게 loss 함수가 감소하였으나, 일정 epoch 이상으로 학습을 할 경우 agent가 이상행동을 보이는 문제가 발생하였다. Newton은 gradient exploding이 발생하여 학습이 전혀 진행되지 않는 모습을 보였다.

위와 같은 학습전략을 시도해도 agent의 가중치를 적절히 학습시키기 위해서는 적절한 feature 설정과, reward 설정이 필요하며, 학습을 진행시킬 맵 등이 필요하다. 과제에 주어진 시간과, 여러 상황을 고려해봤을 때, 경사하강법으로 agent를 학습키셔 높은 성능을 기대하는 것은 현실적인 어려움이 있었다. 따라서 your_best.py로는 채택하지 않았다.

### 2.3. your_baseline3.py

your_baseline3.py는 앞에서 설명한 두 방식의 성능을 평가하기 위해 만들어진 agent들이며, attacker로는 your_baseline1.py의. *Attacker* 클래스를 사용하였고, de-

[4] defender가 캡슐을 사수하게 하는 것이 중요하므로 해당 feature를 추가하였다.
[5] overfitting이 일어난 경우이다.
[6] 사실 거의 대부분의 경우에서 크게 진동을 하였다.
[7] 가장 적절한 움직임으로 판단되는 action이 2개 이상이면 이중 무작위로 하나를 골라 선택한다.

fender로는 baseline.py의 *DefensiveReflexAgent*를 사용하였다.

## 3. Result

output

| | your_best(red) |
|---|---|
| | <Average Winning Rate> |
| your_base1 | 0.0 |
| your_base2 | 0.6 |
| your_base3 | 1.0 |
| baseline | 1.0 |
| Num_Win | 3.0 |
| Avg_Winning_Rate | 0.65 |
| | <Average Scores> |
| your_base1 | 0.0 |
| your_base2 | 0.6 |
| your_base3 | 1.0 |
| baesline | 12.2 |
| Avg_Score | 3.4500000000000000 |

*Figure 1.* your_best.by와 나머지 알고리즘을 비교하여 산출한 결과표이다.

여러 실험을 해본 결과, your_baseline1.py의 성능이 가장 안정적으로 높게 나와 your_best.py로 설정하고, 최종 결과물을 산출해보았다. Figure.1에서 그 결과를 확인할 수 있다.

your_baseline1.py와 your_best.py는 동일한 알고리즘이기 때문에 10번의 시뮬레이션에서 모두 무승부를 기록했고, 따라서 승률 0%를 기록하였다. your_baseline2.py와 your_baseline3.py와의 비교에서는 높은 승률을 기록하기는 했으나, 모든 승부가 매우 근소한 점수 차로 결정되었다[8]. 가장 중요한 것은 baseline.py와의 비교인데, 평균 12.2점의 차이로 100%의 승률을 기록하며 압도적인 성능 차이를 보였다. 그러나, 간혹 -18점으로 baseline.py에게 패배하는 경우가 낮은 확률로 발생하였는데, 해당 케이스를 분석해본 결과, 공통적으로 상대편 agent가 캡슐을 먹은 뒤의 움직임에 의해 발생한 사례들이었다. 따라서 defender 부분에 capsule에 관한 여러 알고리즘들을 추가해주면, 더 높은 성능도 기대해 볼 수 있을 것이다.

[8] your_baseline1.py, your_baseline2.py, your_baseline3.py 모두 비슷한 알고리즘을 사용했기 때문에 이와 같은 결과가 나타났다.

## 4. Discussion

이번 프로젝트를 진행하면서 가장 고민을 많이 했던 부분은 your_baseline2.py의 approximate-QLearning을 구현하는 부분이었다. '어떻게 하면 좀 더 빠르고, 안정적인 학습을 유도할 수 있는 가'에 대한 고민을 하였고, 딥러닝에서 사용되고 있는 여러 학습 기법들을 시도하여 해당 고민을 해소해보고자 했으나, 만족할만한 성과를 보이는데 실패했다. 그러나 아직 시도해 보지 못한 기법들이 많으며, 아래와 같은 전략들을 도입하면, 더 안정적인 학습 효과를 기대해 볼 수 있을 것이다.

- gradient clipping
- early stopping
- l1, l2 regularizer
- feature normalization

이 외에도, approximate-QLearning 뒷 부분에 2-layer MLP를 추가하여 더 높은 일반화 성능을 기대해 보는 것도 생각해 보았으나, 해당 기법을 사용할 경우 훨씬 더 많은 시간을 학습해야 하므로 사실상 실현하기 어려운 기법이었다.