<IAB 실무응용 1 코딩기술서>

Applying AI in Game

-In the Super Hexagon-

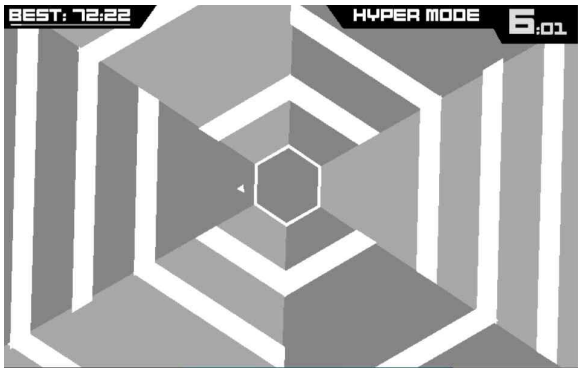
자연과학대학 생명과학부
2015-18432
유현진

※ 목차

I. Introduction

1.1. Game Rule and Previous Research

Super Hexagon은 Terry Cavanagh가 2012년 발매한 간단한 액션 게임으로, 육각형 모양의 맵 안에서 여러 방향으로 다가오는 벽을 좌우로 움직여 피하는 단순한 규칙의 게임이다. 각 단계는 60초 이상 버틸 경우, 클리어가 가능하지만, 매우 빠른 속도와 무작위적 회전, 그리고 다양한 패턴과 미세 조작의 어려움 등으로 난이도가 꽤 어려운 편이다. 총 6단계까지 있으며, 다음 Fig. 1은 이번 프로젝트에서 주로 다뤄볼 6단계 플레이의 일부 장면이다.



▲Fig. 1. Super Hexagon 게임 6단계 장면

이전의 다른 연구에서 CNN 기법을 활용한 강화 학습기법을 이용하여 게임을 학습 시도하였고, 1, 2단계를 클리어 가능한 상태로 학습시킨 전례가 있었다[1]. 인공 지능 기법을 적용한 사례는 아니지만, 정교한 이미지 전처리와 벽의 상태에 따른 최적의 이동 방향을 찾는 방법을 이용한 알고리즘을 통해 6단계를 마침내 클리어한 연구도 있었다[2]. 뒤에서 다뤄볼 연구의 전략에서 이 연구에서 사용한 기법을 일부 참조하였다.

1.2. Outline of Project

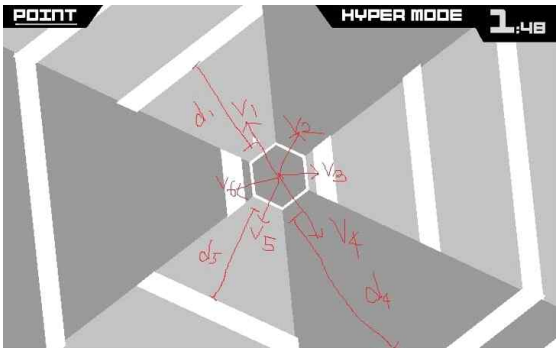
이번 연구에서는 위의 두 전례 연구에서 사용한 방법을 더하여 벽의 상태에 따른 정보를 가지고, 인공 지능 기법을 적용하여 게임 학습을 시켜보는 것을 목표로 하였다. 목표 단계는 6단계로 정하였는데, 색의 변화가 없는 유일한 단계이며, 가운데 도형이 오각형 및 사각형으로 변하는 패턴이 없어 가장 다루기 쉬울 것 같아 실제로는 가장 어려운 단계지만 도전해보기로 하였다. 참고로, 3단계에 적용하는 방법도 후술하여, 색이 변하는 패턴에 대해서도 시도를 해볼 수 있도록 하였다.

input data로 벽의 상태를 사용하여 expert data를 수집하고 이 데이터를 바탕으로 학습을 시켜보는 지도 학습 방법과 같은 input data이지만, DQN 기법을 사용하여 강화학습을 적용하는 방법을 모두 시도해보았다. 또한, CNN을 통해 이미지 전체를 input data로 사용한 방법의 결과와도 비교해보았으며, 지도 학습에서는 Tensorflow 프레임워크를, 강화학습에는 Pytorch 프레임워크를 사용했었다.

II. Supervised Learning Process and Result

2.1. Measuring Method

목표로 설정한 6단계에서는 벽이 다가오는 방향은 6개로 고정되어 있으므로, 방향의 개수가 바뀌는 상황은 우선 고려하지 않았었다. 이 방법에서 고려한 전략은 현재 커서의 위치를 정확하게 찾아내고, 그 커서를 기준으로 60도씩 회전한 총 6개의 방향에서 벽이 얼마나 가까이 위치했는지를 알아내어 어느 방향으로 갈지를 결정하는 것이었다. 전반적인 전략의 개요는 Fig. 2와 같았다. 이 방법은 이전 연구[2]에서 사용한 방법에 착안해 고안했었고, 사진 전체의 데이터를 사용하지 않고 일부 정보만 추출하더라도 인공지능 기법 적용이 가능할지를 연구 주제로 삼았다.



▲Fig. 2. Measuring Method의 개요

1) Capture Image and Get Array

학습을 진행하기 위한 첫 번째 과정은 게임 화면의 이미지를 캡처하고 그 이미지를 numpy array로 변환하여 각 픽셀의 정보를 캐내는 과정이다. 여기서는 게임을 우선 창모드 실행으로 바꾼 뒤, 게임이 실행되는 위치를 고정(보통 좌측 상단에 창을 밀착시켜두는 것이 편하다.)하고, 같은 위치에서 실시간으로 이미지를 캡처하는 코드를 작성했었다. 이미지 캡처에는 pyautogui 라이브러리를 사용했었다. 구현 코드에 창의 위치를 찾는 법도 같이 서술해 두었으며, 해당 게임 화면 이미지에 대한 numpy array를 반환하는 함수는 다음과 같았다.

```
def get_pic_array():
    pic = pyautogui.screenshot(region=(20, 40, 950, 600))
    pic_array = np.array(pic)
    return pic_array
```

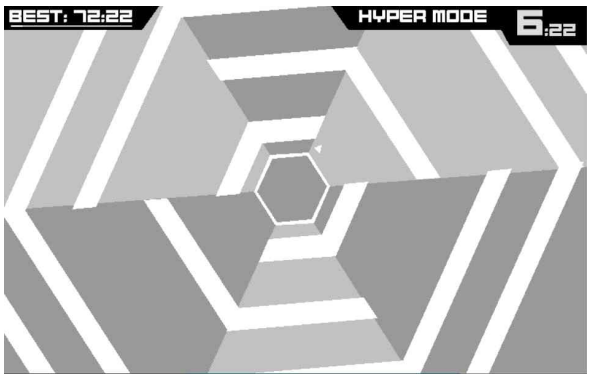
2) Find Location of Ship

현재 위치를 기준으로 어느 방향으로 이동하는 것이 좋을지를 정하기 위해서는 현재 위치를 정확히 알아내는 것이 매우 중요했었다. 6단계의 경우 가운데 육각형과 커서 그리고 벽이 흰색이고 나머지는 회색과 검정 계열의 색으로 이루어져 있었다. 흰색 픽셀의 값은 R, G, B 색상에 대한 값이 모두 240 이상 정도로 측정되었고, 나머지 색은 200 이하의 숫자들로

측정이 되기 때문에 이를 이용해 해당 픽셀이 흰색인지를 알아낼 수 있었다.

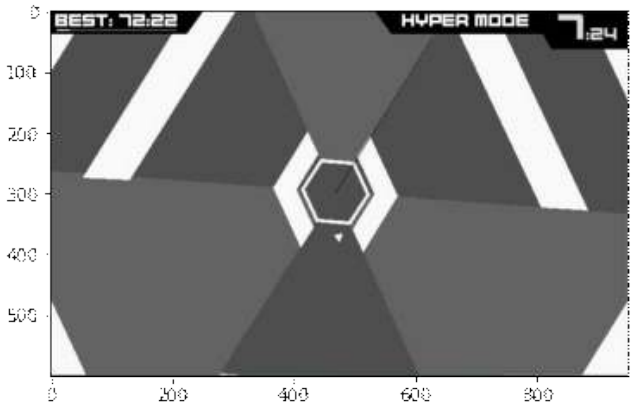
그러나, 가운데 육각형, 커서, 벽 중 커서만을 흰색으로 구분하는 작업은 생각보다 쉽지 않았다. 순간마다 가운데 육각형의 크기도 유동적이며, 회전 방향도 불규칙적이고 벽도 중앙 위치까지 밀착해서 접근하기 때문이다. 그래서 이를 효과적으로 구분하기 위하여 100장 이상의 사진 표본을 분석해본 결과, 다음과 같은 기준이 커서를 구분하는 기준으로 꽤 유효한 것으로 확인이 되었다.

1번. 중앙부터 탐색하여 첫 번째 흰색이 감지되는 부분은 가운데 육각형이고, 중앙에서부터 적당히 멀리 떨어지지 않은 곳에서 두 번째로 흰색이 감지되는 부분이 커서일 수 있다는 발상을 했었다. 그러나, Fig. 3.과 같은 경우에 벽이 커서로 감지될 수 있었다.



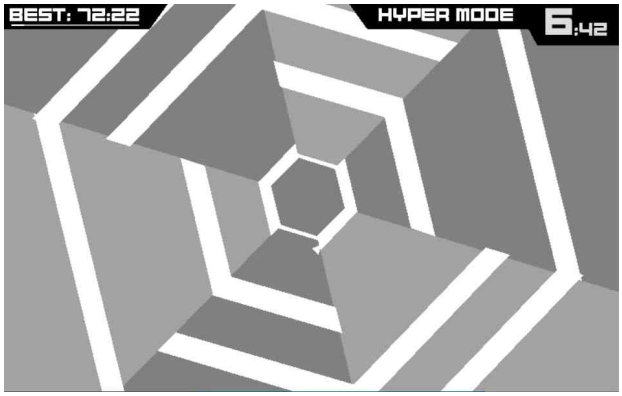
▲Fig. 3. 1번 기준으로 커서를 정확히 찾기 어려운 경우

2번. 중앙에서부터 탐색했을 때, 두 번째로 흰색이 감지되는 경우, 벽은 두께가 커서보다 훨씬 두꺼워서 커서는 특정 두께 이하로 흰색이 감지될 것으로 추측했었다. 그러나 Fig. 4.와 같은 경우, 미세한 측정 오차로 벽이 시작되는 부분에서 흰색이 감지되었다가 곧바로 흰색이 아닌 픽셀이 나와 커서가 있는 것처럼 인식되는 경우가 일부 있었다.



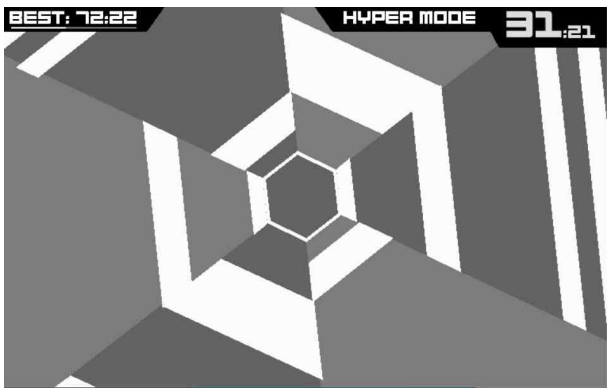
▲Fig. 4. 2번 기준으로 커서를 정확히 찾기 어려운 경우

3번. 커서가 감지된 위치를 기준으로 8도 정도 떨어진 위치에서 흰색이 다시 감지되는지를 확인했었다. 이 기준을 적용하면, 우연히 벽과 커서가 아주 딱 달라붙은 경우도 극히 일부 경우를 제외하고는 어느 정도 정확하게 인지되는 것을 확인했었다. Fig. 5.같은 경우에서도 정확하게 인지가 되었었다.



▲Fig. 5. 모호한 경우 중 1, 2, 3번 기준을 적용했을 때 커서를 정확히 찾는 경우

그러나, 게임의 특성상 Fig. 6. 같이 커서가 아예 숨어버리는 경우도 존재했었고, 이 경우는 당연히 커서를 찾기가 어려웠다. 그래도 표본 사진 중 98%에서 커서 위치를 오차 범위 내에서 정확하게 찾았었다.



▲Fig. 6. 커서가 완전히 숨어버려서 찾기 어려운 경우

중심의 위치를 찾는 방법과 중심에서 떨어진 각도와 거리에 해당하는 픽셀값을 반환하는 함수도 구현 코드 내에 작성해두었다. 12시 방향을 0°로 가정하고 2°씩 Brute Force 탐색을 하는 방법으로 1, 2, 3번 조건을 모두 만족 시 커서가 해당 각도에 있다고 생각하며 탐색을 종료하는 함수를 다음과 같이 작성했었다.

```
def measure_radian(pic_array):
    rad = 0
    for j in range(180): # 2 * 단위로 Brute Force 탐색
        cnt = 0 # 흰색 픽셀의 굵기를 감지하려는 변수
        seq = 0 # 두 번째로 흰색 감지되는 경우를 찾기 위한 변수(1번 조건)
        if rad != 0: # 커서 위치 찾으면 탐색 종료
            break
        for i in range(35,100,2): # 중심 기준 35픽셀~100픽셀 떨어진 범위 2픽셀씩 전진탐색
            if calculate_color(i, 2*j, pic_array) >= 235: # 흰색 감지
                if seq == 0:
                    seq = 1
                if seq == 2:
                    cnt += 1
            else: # 흰색 미감지
                if seq == 1: # 흰색이 앞에서 감지된 경우 업데이트
                    seq = 2
                if seq == 2 and cnt in [2, 3, 4]: # 굵기가 특정 범위 내에 있어야 커서로 인식(2번 조건)
                    ch = 0
                    for k in range(-4, 5): # 3번 조건 탐색
                        if calculate_color(i+k, 2*j-8, pic_array) >= 235 or ₩
                        calculate_color(i+k, 2*j+8, pic_array) >= 235:
                            ch = 1
                            break
                    if ch == 0: # 모든 조건 만족시 rad 업데이트
                        rad = 2 * j
                    break
    return rad
```

3) Measure Distance

위에서 찾은 커서의 위치를 바탕으로, 현재 위치를 기준으로 60°씩 회전한 총 6개의 방향에 대해 벽의 거리를 측정하고, 이 거리에 대한 리스트를 input data로 사용했었다. 첫 번째 벽의 정보만 이용하는 것보다는 두 번째 벽의 정보도 이용하는 것이 더 효과적인 것으로 나타나, 6개의 방향에서의 2개의 정보씩 총 12개의 값에 대한 리스트를 최종 input data 형태로 결정했었다. 단, 두꺼운 벽은 연속해서 벽이 2개 있는 형태로 가정했었으며, 빠른 게임의 특성상 탐색 시간을 줄이기 위하여 8픽셀씩 전진 탐색했었다. 사용한 함수의 코드는 다음과 같았었다.

```
def measure_distance(pic_array):
    dist_1 = [30] * 6 # 최대값 미리 부여하고 시작(오류 발생 가능성 제거)
    dist_2 = [30] * 6 # 두 번째 벽도 인식
    ch = 0
    rad = measure_radian(pic_array) # 커서 위치 측정
    for i in range(6): # 60°씩 회전하여 측정
        rads = (rad + 60 * i) % 360
        for dist in range(92, 392, 8): # 중심에서 92픽셀 떨어진 위치 부터 해당 방향으로 8 픽셀씩 전진
            if ch > 0: # 흰색 나온 직후 픽셀 감지 생략
                ch -= 1
                continue
            try:
                if calculate_color(dist, rads , pic_array) < 50: # 메뉴바 부분 픽셀 감지
                    break
                if calculate_color(dist, rads , pic_array) >= 235: # 흰색 픽셀 감지
                    if dist_1[i] == 30: # 아직 벽이 인식이 안되었던 경우
                        dist_1[i] = round((dist - 92) / 10)
                        ch = 5 # 40픽셀 점프하여 다시 계산 유도
                    else:
                        dist_2[i] = round((dist - 92) / 10)
                        break
            except:
                break
    return dist_1 + dist_2 # 0~5번 인덱스는 첫번째 벽(현재위치부터 60도씩 회전), 6~11번 인덱스는 두번째 벽과의 거리이다.
```

4) Collect Data

input data의 형태를 정의하였으니, 지도 학습을 위하여 데이터를 모으는 과정이 필요했었다. 게임오버인 상태에서 데이터가 수집되는 현상을 방지하기 위하여 게임오버일 때 검정색으로 변하는 부분의 픽셀의 상태를 이용해 게임오버 여부를 감지하는 함수도 작성하였으며, input data를 매 순간 생성해내는 함수, 그리고 keyboard 입력 상태를 입력받아 라벨을 생성해내는 함수도 같이 작성했었다. keyboard 입력 상태는 keyboard 라이브러리를 사용하여 감지할 수 있었다.

그 후, 게임을 플레이하여 이번에 실행했던 부분을 전체 데이터 set에 저장하고 싶으면 y를 입력하고 이번에 원하는 대로 플레이하지 못했다면 n을 입력하여 데이터를 버리는 코드를 작성했었다. 단, 각 게임마다 게임오버 직전의 데이터는 bad data로 생각하여 수집을 포기하는 방향으로 코드를 작성하였고, 양방향 모두 이동한 경우에는 오른쪽으로 일괄 이동하는 등의 방법으로 expert data의 quality를 높였었다. 이번에 수집한 데이터를 다음 번에도 사용이 가능하도록 pickle 라이브러리를 통해 저장 및 불러오기를 하는 방법도 같이 서술하였다. 구현 코드에 자세한 코드와 실행 방법을 수록하였다.

5) Training

Tensorflow 프레임워크(버전 2)를 사용하여 위에서 모은 데이터를 가지고 지도 학습을 진

행했었다. 이번 연구에서는 약 1만 개 정도의 데이터를 모았었으며, 라벨은 왼쪽은 0, 정지는 1, 오른쪽은 2로 정수형으로 변환해주었었다. train set 80%, test set 20%를 기준으로 학습을 진행했었고, test accuracy가 가장 높았던 하이퍼 파라미터의 조합을 코드 내에 작성했었다. 가장 높게 나타났던 test accuracy는 약 88.6%이었다(Fig. 7.).

```
Epoch 1/10
282/282 [==============================] - 1s 2ms/step - loss: 0.4857 - accuracy: 0.8318
Epoch 2/10
282/282 [==============================] - 1s 3ms/step - loss: 0.4202 - accuracy: 0.8564
Epoch 3/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3966 - accuracy: 0.8583
Epoch 4/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3838 - accuracy: 0.8634
Epoch 5/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3684 - accuracy: 0.8700
Epoch 6/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3600 - accuracy: 0.8752
Epoch 7/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3559 - accuracy: 0.8727
Epoch 8/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3466 - accuracy: 0.8748
Epoch 9/10
282/282 [==============================] - 1s 2ms/step - loss: 0.3424 - accuracy: 0.8762
Epoch 10/10
282/282 [==============================] - 1s 3ms/step - loss: 0.3389 - accuracy: 0.8780
56/56 - 0s - loss: 0.3196 - accuracy: 0.8863

[0.3195633888244629, 0.8862612843513489]
```

▲Fig. 7. Result of Supervised Learning by Measure Method

6) Play Game

위에서 학습했던 모델을 저장해두고, 이를 다시 불러와 게임을 플레이하는 코드를 작성해두었다. 게임 창의 위치를 학습했던 위치와 같은 위치에 고정시킨 뒤, 게임을 플레이하면 된다. 아쉽게도 위에서 얻었던 모델의 성능으로는 게임을 원활하게 플레이하는 것이 불가능했었다.

7) 번외 : 다른 단계에 적용하기

6단계가 아닌 다른 단계에도 이 방법을 물론 적용할 수 있다. 다만, 다른 단계는 색이 계속 변하고, 가운데 모양이 바뀌는 등 다른 패턴이 나와 이 부분에 대하여 코드를 재작성 해주어야 한다. 여기서는 색이 바뀌는 3단계에 대해서 학습이 가능한 코드를 작성해보았었다. 여기에서도 커서, 벽, 가운데 육각형은 모두 같은 색(환경보다 더 진한 색이다.)이며, 가운데 모양이 바뀌는 패턴은 나오지 않는다. 커서와 벽의 색은 가능한 모든 색에서 R, G, B 값 중 하나 이상이 200 이상으로 측정되는 특성이 있었고, 환경의 색은 R, G, B 값이 모두 200 미만으로 측정되어 이를 이용해 구분할 수 있도록 함수를 변형했었다. Fig. 8.은 3단계에서 나올 수 있는 색상의 예시이다.

▲Fig. 8. 3단계에서 등장하는 색의 예시들(노란색, 하늘색 등도 등장한다.)

2.2. CNN Method

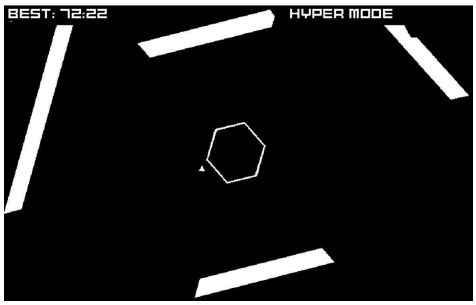
1) Plan to Using CNN

CNN(Convolutional Neural Network)은 이미지 전체의 공간적인 정보에서 오는 패턴을 효과적으로 학습하기 위하여, 인공 신경망에 여러 패턴을 학습시킬 수 있는 많은 필터를 접목하여 만든 기법으로, pooling layer를 통해 공간적인 상대 위치를 유지하며, 정보를 압축해 이미지 정보 학습을 효과적으로 할 수 있도록 Yann LeCun이 고안해낸 모델이다[3].

이전 연구 중, 같은 게임에 CNN을 어느 정도 성공적으로 적용했던 연구가 있었으며[1], CNN을 이용한 모델 학습이 이번 연구의 주목표는 아니었지만, 이미지 전체의 정보를 사용하려는 CNN과 위에서 고안했던 이미지 전체의 정보 중 위치에 관련된 일부 정보만을 사용했던 방법을 비교해보는 작업을 해보고 싶어 모델 학습에 일부 정보만을 가져와 학습해 보는 것이 효과적일 수 있는지를 알아보는 것이 목표였다.

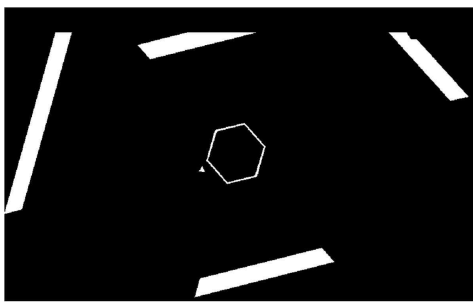
2) Collect Data

2.1. 절에서 사용했던 많은 함수와 방법들이 여기에서도 거의 동등하게 사용할 수 있다. 다만 input data의 형태가 위에서는 각 방향에서의 벽과의 거리정보만을 사용했지만, 여기서는 이미지 전체의 픽셀 정보를 사용한다는 점에서 차이가 있었다. R, G, B의 구분은 6단계에 적용하는 데에는 의미가 없다고 생각하였고, 벽, 커서, 가운데 육각형 부분만 드러내면 된다 생각하여 이전 다른 연구[2]에서 제안했던 방법과 같이 역치 이상인 부분만 드러내는 방법을 사용했었다. (Fig. 9.)



▲Fig. 9. 역치 이상인 부분만 드러낸 이미지 예시

다만, 이미지 상단에 최고 기록과 HYPER MODE라고 나오는 부분도 역치 이상으로 나오기 때문에 noise를 줄이기 위하여 전처리를 미리 해주었었다. (Fig. 10.)



▲Fig. 10. 최상단의 픽셀들을 전처리한 후의 이미지

다음 전처리를 자동으로 처리하여 input data를 형성하는 make_data 함수를 다음과 같이 작성했었다. 다른 절차들은 2.1. 4)에서 작성해두었던 코드를 그대로 사용해도 무방하다.

```
def make_data():
    pic_array = get_pic_array()
    cv2.rectangle(pic_array, (0,0),(949,50),(0,0,0), -1) # 시간이 나오는 윗 부분은 노이즈로 작용할 수 있어 제거
    pic_array = cv2.cvtColor(pic_array, cv2.COLOR_RGB2GRAY)
    pic_array = cv2.threshold(pic_array, 235, 255, cv2.THRESH_BINARY)
    return np.expand_dims(pic_array[1], axis = 2)
```

3) Training

2.1. 5)와 같이 몇 가지 전처리를 진행한 뒤, 간단한 CNN 모델을 작성하여 해당 image가 좌/우/정지 중 어느 방향으로 가는 것이 좋은지를 예측해보는 모델의 학습을 진행해봤었다. 다만, 이번 연구에서 사용했던 image size가 950 * 600으로 매우 컸기 때문에 CPU로는 학습이 어려웠고 GPU를 사용했었으며, data size도 10000 이하로 제한해서 실험을 진행했었다. 2.1.과 같이 Tensorflow 2 버전으로 작성되었으며, maxpooling layer를 총 5회 통과하는 모델이었으며, 과적합 방지를 적용하지 않았을 때의 실험 결과는 다음과 같았다. (Fig. 11.)

```
9000/9000 [==============================] - 880s 98ms/sample - loss: 0.7082 - acc: 0.6921
Epoch 2/10
9000/9000 [==============================] - 876s 97ms/sample - loss: 0.5991 - acc: 0.7491
Epoch 3/10
9000/9000 [==============================] - 876s 97ms/sample - loss: 0.4873 - acc: 0.7960
Epoch 4/10
9000/9000 [==============================] - 876s 97ms/sample - loss: 0.3848 - acc: 0.8382
Epoch 5/10
9000/9000 [==============================] - 878s 98ms/sample - loss: 0.2735 - acc: 0.8882
Epoch 6/10
9000/9000 [==============================] - 880s 98ms/sample - loss: 0.1837 - acc: 0.9252
Epoch 7/10
9000/9000 [==============================] - 881s 98ms/sample - loss: 0.1193 - acc: 0.9588
Epoch 8/10
9000/9000 [==============================] - 880s 98ms/sample - loss: 0.0744 - acc: 0.9732
Epoch 9/10
9000/9000 [==============================] - 883s 98ms/sample - loss: 0.0517 - acc: 0.9832
Epoch 10/10
9000/9000 [==============================] - 880s 98ms/sample - loss: 0.0456 - acc: 0.9847

1716/1716 - 27s - loss: 1.6272 - acc: 0.7617
[1.6272181172470945, 0.76165503]
```

▲Fig. 11. 과적합 방지를 사용하지 않았던 경우 CNN 모델의 학습 결과

train accuracy는 98% 이상이었지만, test accuracy는 76% 정도로 2.1.에서 시도했던 방법보다 더 낮은 예측 결과를 나타냈었다. 과적합 방지를 위하여 Tensorflow의 Earlystopping을 적용하면 epoch 3~4 정도에서 학습이 바로 종료되었었는데, 이 경우도 test accuracy가 위의 결과가 크게 다르지 않아 아쉬움이 많았었다.

4) Play Game

이 과정은 2.1. 6)과 거의 비슷하여 부연 설명할 부분이 많지는 않다. 다만, 리스트를 그대로 input으로 사용하는 것이 아니라, numpy array 형태로 자료형을 변환해주는 과정이 필요하다는 차이점이 있었다. 이 과정은 train, test dataset을 생성할 때도 같게 진행됐었다.

III. Reinforcement Learning Process and Result

3.1. Measuring Method

1) Plan to Using DQN

DQN(Deep Q Network) 기법은 구글 딥마인드에서 개발한 강화학습 기법으로, 게임의 리워드를 정하고 현재의 상태와 다음 상태, 그리고 다음 상태에서 얻을 수 있는 최대 리워드를 Q Network를 이용하여 계산해 점점 더 높은 리워드를 얻을 수 있도록 Q Network를 학습시키는 방법이다[4].

지도 학습에서 인위적으로 라벨링하는 과정으로 해결하지 못했던 문제들을 인공지능에게 모두 맡겨 학습을 시켜본다는 가정을 하여 위 참고 논문[4]에서 Atari 게임을 학습시켰던 과정과 비슷한 느낌으로 이번 게임에 적용을 해보고 싶었다.

일반적으로 DQN 기법에서는 CNN을 사용[4]하여 픽셀 전체 이미지를 가져와 이를 이용해 학습을 시키는데, 여기서는 다양한 시도를 위해 픽셀 정보를 축소했던 방법인 Measuring Method와 CNN을 모두 시도해보았다. 여기서는 Pytorch 프레임워크를 사용했었으며, 데이터를 뽑는 방법은 지도 학습때와 거의 같은 과정으로 진행했었다.

위 참조 논문[4]에서 사용했던 Replay buffer와 Target Q Network의 개념도 같이 도입하여 사용했었는데, 다음 데이터와의 상관관계를 막기 위하여 deque에 최근 10000프레임의 데이터를 저장하여 이 중 32개를 뽑아 학습시키는 과정을 진행했었으며, Target Q Network를 10게임 실행마다 새로 업데이트하는 방법을 선택했었다.

2) Make Data

지도 학습과는 다르게, 이번 과정에서는 데이터를 모을 필요가 없었다. 커서를 찾고 벽을 찾아 거리를 재는 방법과 사용할 input data의 형태는 모두 2.1.에서 소개했던 과정과 동등하게 진행했었다. 일부 함수는 2.1.에서 사용한 함수를 그대로 사용했고, 데이터를 모으는 과정 등은 여기서는 생략하였다.

3) Define DQN Model

우선, 이 코드의 뼈대는 해당 깃허브 주소에 있는 코드를 참조했었다.
출처 : https://github.com/seungeunrho/minimalRL/blob/master/dqn.py

유튜브 팡요랩 컨텐츠 분께서 이 코드와 코드 설명을 자세히 해주는 영상을 공유해주셨는데, 감사의 말씀을 올립니다. (영상 주소 : https://youtu.be/__NYgfkUr-M)

이 부분에 대한 설명이 잘 이해가 되지 않는다면 위의 영상을 참조하면 이해가 잘 될 것이다. 위의 코드를 이번 연구에 사용할 수 있도록 개조하였다.

첫 부분은 모델 학습에 필요한 하이퍼 파라미터를 정의하였다. 시간에 따른 미래 가치 감소를 정의하기 위한 감가율이 필요했었는데, 1초에 수십 프레임이 학습되므로, 0.99 정도로

높게 감가율을 설정했었다. buffer 사이즈는 더 크게 하는 경우가 많은 것으로 알려져 있으나, 여기서는 cpu 내에서도 메모리 용량이 버틸 수 있도록 10000프레임 정도의 데이터만 deque에 모으도록 했었으며, 한 학습에 32개의 임의 추출 데이터를 이용하여 학습시켰었다.

이 과정에서 사용했던 Replay buffer와 Q Network는 class로 정의해서 사용했었는데, 자세한 코드는 다음과 같았다.

```
class ReplayBuffer():
    def __init__(self): # deque 자료형으로 선언
        self.buffer = collections.deque(maxlen=buffer_limit)

    def put(self, transition): # 데이터 buffer 내에 삽입하는 함수
        self.buffer.append(transition)

    def sample(self, n): # state, action, reward, next state, game over 여부 를 각각 저장 후 샘플링
        mini_batch = random.sample(self.buffer, n)
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])

        return torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
               torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst)

    def size(self): # buffer 내에 현재 존재하는 데이터의 개수 반환
        return len(self.buffer)
```

```
class Qnet(nn.Module):
    def __init__(self):
        super(Qnet, self).__init__()
        self.fc1 = nn.Linear(12, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 3)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def sample_action(self, obs, epsilon): # epsilon의 확률로 초기 랜덤 action
        out = self.forward(obs)
        coin = random.random()
        if coin < epsilon:
            return random.randint(0,2)
        else :
            return out.argmax().item()
```

Target Q Network도 같은 class 객체를 적용하여 Q Network와 구조가 동등한 Network를 사용하되, 다만 가중치 업데이트를 10게임에 1번만 되도록 한정했었다.

4) Training

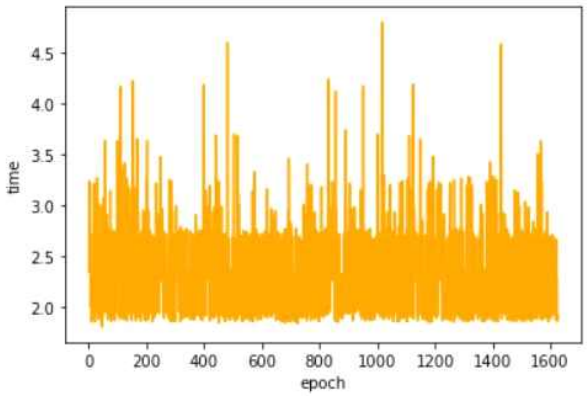
이제, 게임이 실행되면서 데이터를 모으고, 1게임이 끝났을 때, deque 내에 저장된 데이터 중 32개씩 임의 추출을 진행해 학습하는 과정을 10번씩 반복할 수 있는 모델 학습 함수를 작성했었다. 각 게임의 마지막 데이터에서는 다음 게임 첫 데이터가 반영되지 못하도록 done_mask 변수를 이용하여 조정했었다. 여기서는 loss function을 두 값의 절댓값 차이의 합인 L1 loss를 사용했었는데, L2 loss나 cross entropy 등 다른 loss function을 적용해도 무관하며, 오히려 다른 실험 결과가 나올 수도 있을 것으로 생각된다.

5) Play Game and Doing Reinforment Learning

이 코드를 실행시키면 최대 10000 게임 동안 계속 반복 실행되면서 모델이 자동 학습될

수 있게 설정해두었다. 리워드는 여기서는 마지막 게임 종료 직전의 bad action에는 –100, 나머지 action에는 0.1점을 부여했었는데, 다른 값을 부여해도 상관없고, 게임에서 버틴 시간을 이용하여 일괄 reward를 부여하는 방법 등도 존재할 것으로 보인다. 초기에는 높은 확률로 random action을 하다 점점 random action 확률을 줄일 수 있도록 epsilon 값도 설정했었다.

아쉽게도, 이번 시도에서는 epoch 반복에도 성능이 크게 개선되지는 못하였다. (Fig. 12.) 이에 대한 토의는 4장에서 서술할 예정이다.



▲Fig. 12. Measuring Method DQN 학습 결과

3.2. CNN Method

1) Make Data

opencv사용 역치 처리 및 numpy array로 변환해주는 과정은 2.2.와 동일하게 진행되었으나, Pytorch 프레임워크에 적용하기 위하여 reshape를 해주는 방법이 약간 달랐었다. 여기서 약간 바꾸어 새로 적용한 함수를 사용하면 된다.

2) Define DQN Model

이번에는 Pytorch 프레임워크로 2.2.에서 사용했던 모델과 거의 비슷한 모델을 다시 작성했었다. Tensorflow에 비해 코드가 조금 더 간결해진 모습을 관찰할 수 있었다. 만약 사용한 이미지의 픽셀 크기가 조금 다르다면, 직접 input data의 차원을 다시 계산해도 좋고, 출력된 오류 메시지를 보고 숫자를 재조정해도 괜찮다.

3) Training and Play Game

이 과정에서 사용했던 코드는 3.1.에서 사용한 코드와 거의 비슷하므로 설명을 생략하겠다. 다만, record로 게임 플레이 시간과 학습에 걸리는 시간이 더해서 산출되는데, 학습에 걸리는 시간에 차이가 있어서 이에 맞게 record에 일정한 값을 3.1.보다 더 빼주었다. 아쉽게도 이 방법으로는 학습에 지나치게 오랜 시간이 걸려 학습을 직접 해보는 작업은 실패했었다.

IV. Discussion

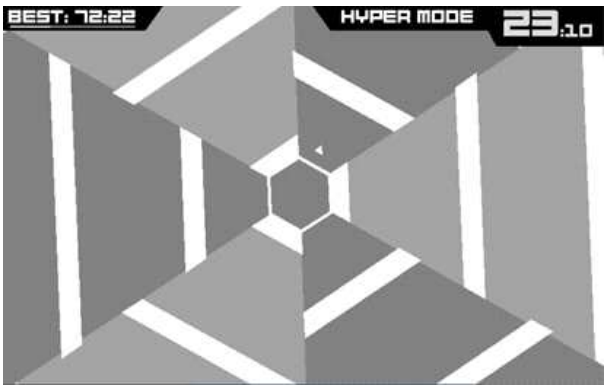
1) Difference between Measuring Method and CNN Method

이미지에서 정보를 학습할 때는 픽셀 전체의 이미지를 가져온 뒤, 픽셀의 정보를 최대한 많은 수의 필터를 이용하여 추측해낼 수 있도록 CNN을 사용하는 것이 유용한 방법일 수 있다[3]. 그러나, 만약 찾아야 하는 정보가 명확하게 명시되어있는 상황이고, 이를 찾는 것이 충분히 간단히 이해가 될 수 있는 상황이라면 이번 연구에서 수행했었던 Measuring Method 같은 방법도 도움이 될 수 있다는 것을 알 수 있었다.

물론, 이전 연구[1]와 같이 CNN을 더 섬세한 기법으로 적용해보았던 것은 아니었지만, 제한적인 상황에서는 꼭 이미지 전체의 정보를 사용하지 않더라도 모델 학습에서 좋은 결과를 낼 수 있다는 가능성을 보았었다. 실무에서 필요한 단계의 이미지 분류작업에서는 이렇게 간단하게 정보를 추출할 수 있는 상황이 거의 없겠지만, 그래도 필요한 부분이 어느 정도 암시된 경우라면 데이터 전처리를 통해 필요한 부분을 강조하여 사용할 수 있도록 해준다면 좋은 결과를 기대해볼 수도 있을 것 같다.

2) Challenge and Limitation

먼저 지도 학습에서 모델을 실제 학습시켜보기 전에 생각했던 예상 문제점은 다음과 같았었다. 우선, 게임의 특성이 매우 빠른 게임이기 때문에 이미지 캡처, 데이터 추출 및 학습까지 시간을 최대한 단축할 필요가 있었다. 이미지 캡처는 고정된 시간이 걸리는 작업이었고, Measuring Method 같은 경우는 학습 소요시간은 매우 적었기 때문에, 데이터 추출을 최대한 빠르게 할 수 있어야 했었다. 커서의 위치를 찾을 때 모든 각도를 찾는 방법 외에는 방법이 따로 없었고, 벽의 거리를 찾을 때도, 이진 탐색 등의 방법을 적용하기는 매우 어려웠으므로, brute force를 사용하되 탐색하되, 탐색하는 픽셀의 개수를 최소한으로 줄이는 방법을 사용했었다. 예를 들면, 1픽셀씩 탐색하는 것이 아닌, 벽을 탐색하는 범위를 8픽셀씩 전진으로 바꾸는 방법이나 커서 위치 탐색 범위를 최소한으로 줄이는 방법 등이 사용됐었다. 정확도가 아주 조금 떨어질 수는 있으나, 1프레임을 뽑고 학습시키는 데에 평균 0.1초 이하로 줄여 게임 학습의 가능성을 가져올 수 있었다.



▲Fig. 13. 라벨링이 모호한 경우 예시

그러나 지도 학습을 진행하는 과정에서 expert data가 불완전할 수밖에 없다는 점을 발견했었다. 예를 들어, 다음과 같은 경우 왼쪽, 오른쪽 모두 이동해도 무방하며 여기서 움직여도 좋고 한 프레임 전에 움직였어도 생존 가능한 상태였다(Fig. 13.). 양쪽 이동이 모두 가능한 경우는 오른쪽 이동으로 통일하여 라벨링 작업을 했었으나, 모호함을 모두 피할 수는 없었다. 위의 이미지보다 라벨링이 더 모호한 경우도 많았으며, 커서가 아예 가려진 경우(Fig. 6.) 등도 expert data가 완벽할 수 없었다.

이 모호함은 CNN 방법에서도 마찬가지로 존재했었는데, 순간순간의 생존할 수 있는 이동 방향의 정답이 1개만 정해진 것은 아니며, 우연히 생존했지만 사실 더 좋은 다른 액션이 있는 경우도 꽤 많이 관찰됐었다. 또한, CNN 방법에서는 image size가 일반적인 이미지 분류기에 사용되는 이미지보다 훨씬 큰 950 * 600 size였기 때문에, 학습 과정에서 적지 않은 시간이 소모되었었다. 이 과정은 필터의 개수와 모델의 복잡도를 조절하는 일이 중요한 과정이었다.

강화학습에서는 보상을 정하는 방법의 모호함이 가장 큰 한계였다고 생각이 들었다. 3.1. 의 5)에서 설명한 것처럼 여러 시도를 해볼 수 있는데, 학습이 가장 잘되는 보상의 조합을 찾지 못해 epoch 반복에도 성능이 크게 개선되지 못하는 현상을 관찰했었다. 다만, 이를 잘 설정해주어 만약 인공지능이 인위적으로 정하기 어려운 이동 방향의 질 등을 판별할 수 있게 된다면, 학습에 성공할 수 있게 될 수 있다는 가능성을 생각해보게 해주었었다.

또한, pyautogui 라이브러리를 통한 실제 모니터 내 화면의 픽셀 정보 이용과 keyboard 라이브러리를 통해 컴퓨터 내 키보드 입력을 직접 조작하는 방식으로는 여러 모듈을 동시에 구동하는 것이 어려운 일이었고, 따라서 학습량과 속도에 한계가 있었던 것 같았다. 그래서 보상을 정하는 방법이나 다양한 하이퍼 파라미터의 실험이 필요했었지만 많은 시도를 하지 못했었다. Measuring Method 같은 경우는 그나마 실행 속도가 어느 정도 나오지만, CNN을 사용할 때는 특히 image size도 매우 큰 편이라, 1 epoch에 수 분씩 걸리는 현상을 목격했었다. 이 문제는 직접 화면 내 띄우지 않아도 학습이 되는 원격 학습 모듈 구현 및 GPU 사용을 통해서 해결해야 하는 문제라고 생각했었다.

References

[1] Jason Lewis. Playing Super Hexagon with Convolutional Neural Networks (Milestone), Stanford CS231N (2015).

[2] Trimaille, Valentin. "Super Hexagon Bot.", CrackedOpen Mind (2015).

[3] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner. GradientBased Learning Applied to Document Recognition, Proceedings of the IEEE (Volume: 86 , Issue: 11) (1998).

[4] Volodymyr Mnih et al. (Google Deepmind) Playing Atari with Deep Reinforcement Learning, NIPS Deep Learning Workshop 2013 (2013).

some of source code : https://github.com/seungeunrho/minimalRL/blob/master/dqn.py