Multi-Agent Maze Solving Bot

Using Q-Learning

Abstract

Solving maze from inside perspective has been difficult for human. This means find a route without an overview of the maze or the surroundings. It takes a lot of time and mistakes, and possibly a great memory which human may not have. Artificial intelligence however are made to do jobs like this. By using Q-learning agents can now solve mazes.

To take a step further, I developed a bit of coordination between multiple agents so they can work together to achieve a higher efficiency. This is done by letting one of the agents be the solver while the other one only needs to be an explorer. This will reduce the mistakes made by the solver thus making it quicker to reach the final destination.

Contents

1 Introduction ……1

1.1 Inspiration ……1

1.2 Solving a maze with AI ……2

2 Algorithm

2.1 Reinforcement learning ……3

2.2 Q-Learning ……4

3 Implementation ……6

3.1 Generate environment ……6

3.2 Define actions and rules ……6

3.3 The reward system ……7

3.4 Reinforcement learning iteration ……7

3.5 Multi-agent ……9

4 Evaluation……10

4.1 Single agent……10

4.2 Multi-agent……12

5 Conclusion……13

Chapter 1

Introduction

1.1 Inspiration

OpenAI is a very strong artificial intelligence research laboratory. They are considered as a competitor to another AI research lab DeepMind, which created many powerful AIs including the famous AlphaGo, which smashed the best human Go player in the world in this complicating game. OpenAI's most recent work is called Generative Pre-trained Transformer 3 (GPT-3), which understands any text you input, then predicts and completes any following context for you.

This project was initially inspired by a research done by OpenAI, called *Emergent Tool Use From Multi-Agent Interaction*, where they put multiple agents into an environment containing walls and movable bricks and slopes and letting them compete in a game of *hide-and-seek.* The result came out to be very interesting as agents successfully developed strategies requiring sophisticated tool use and coordination. Figure 1 shows at one stage, the hiders (blue) learned to surround themselves with movable walls to escape from the seekers (red).

At first, my goal was to try to understand and replicate something similar, but after several iterations of the idea, it came out to be solving cheese mazes with unsupervised agents and find how coordination could be built up in this process.

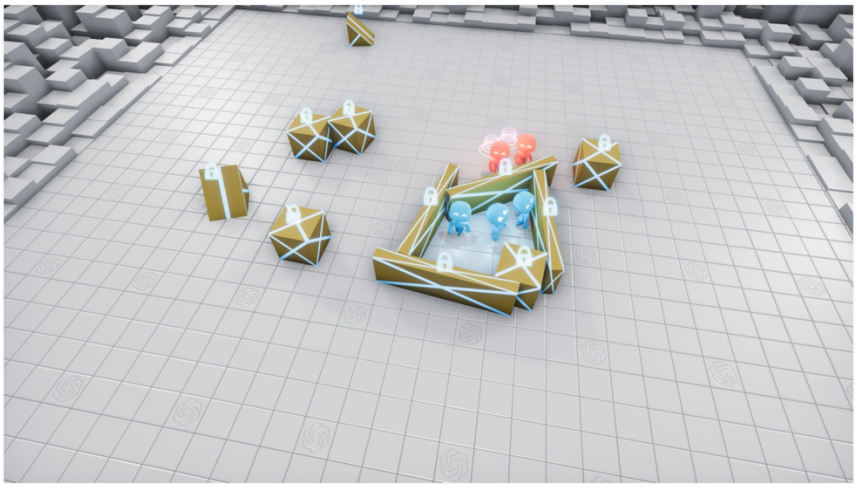


Figure 1

1.2 Solving a maze with AI

Solving a maze is like finding a path on Google map, except when you are actually in a maze, you will not see beyond the walls. This makes it much harder to plan the route to take, because very often, you will only know it's a dead end by walking into it. I created a similar environment for the AI agents, where they can only know the existence of walls by bumping into them. They don't know where the boundaries are, nor do they know the location of the destination.

The advantage about solving maze with AI is that they will never get "tired", so they can try and fail and try another way endlessly. This is called trial and error.

Chapter 2

Algorithm

2.1 Reinforcement learning

Reinforcement learning is a type of strategy learning in artificial intelligence. The word ‘reinforcement learning’ comes from behavioural psychology, it treats the behaviour of learning as experimenting repeatedly, thus maps the dynamic environment state to corresponding actions. This method is different from the supervised learning technique, which uses positive and negative examples to tell which behaviour to take, but uses trial and error method to find the optimal behaviour strategy. Its properties of self-improving and online learning make reinforcement learning become one of most important machine learning methods.

Through continuous trial and error and the method of getting rewards and punishments from the environment, agent can learn independently which actions have the greatest value in different states, so as to find or approach the strategy that can get the greatest reward. It is similar to the traditional experience of “A fall into the pit, a gain in your wit”.

If an agent's behaviour strategy leads to a positive reward (enhancement signal), then the agent's tendency to produce this behaviour strategy in the future will be strengthened. The objective of an agent can be defined as a reward or payoff function, which assigns a numerical value to the different actions taken by the agent from different states, namely the immediate payoff. The task of the agent is to perform a series of actions, observe the results, and then learn the control strategy. In the upper strategy of cheese maze solving, we hope that the control strategy is to select actions in

any initial discrete state, so that the agent can find the optimal strategy over time to achieve the expected discount reward (reward) and maximum.

As shown in Figure 2, the agent selects an action for the environment. The environment receives changes in state after the action and generates a reward feedback to the agent. The agent selects the next action according to the strengthening signal and the current state of the environment, in the principle of increasing the probability of receiving positive reinforcement.

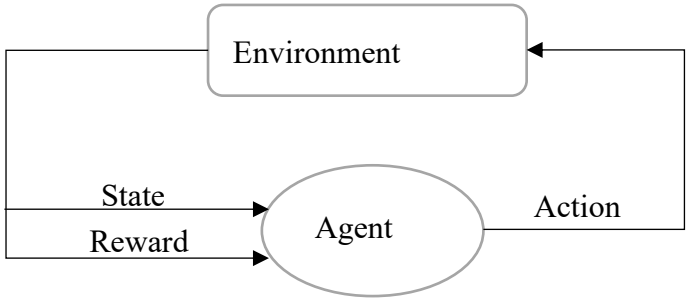


Figure 2

2.2 Q-learning

Q learning is a reinforcement learning algorithm that finds an optimal policy telling an agent what action to take under what circumstances for any finite Markov decision process (FMDP). The FMDP provides a mathematical framework for decisions to be made under a partly under the control of a decision maker and partly random scenario. For every time step, we denote the current state as *s*, the decision maker can choose any action a available in this state. *s* therefore moves to a new state *s'* and returns a corresponding reward $r_a(s, s')$.

However, it is very difficult for agents to directly learn the optimal strategy in any environment, because training samples in the form of <s, a> are not provided in the training. The process will be made easier by learning a numerical evaluation function defined on states and actions, and then implementing the optimal strategy in the form

of that evaluation function.

In Q learning, Q is the expected outcome of action a performed in state s. s is the state vector, a is the action vector, r is the immediate reward obtained, and γ is the discount factor. Then the algorithm has a function that calculates the quality of a state-action combination:

$$Q: s \times a \rightarrow r$$

Its value is the maximum discounted cumulative expected reward that can be obtained when starting from state s and using a as the first action. It can be formalised by the following rule:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t) \times (r_{t+1} + \gamma \max Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

where t is discrete time step, $\alpha_t(s_t, a_t)$ is the learning rate (or step size parameter) and the $\max Q_t(s_{t+1}, a)$ is the estimate of the optimal future value.

Chapter 3

Implementation

3.1 Generate environment

To start everything up, we first need to have a maze for the game to take place. I created six maps by setting up walls in different locations. There are two reasons I choose to do this manually instead of generating the map randomly. First, randomly generating walls may lead to maps with inaccessible exit. Secondly, I want to make sure my agent is tested on different types of "terrains", i.e., single path maze, multi-path maze and random obstacle maze. The maze is a 7 by 7 square filled will black blocks indicating walls. The agent starts from the bottom-left corner and needs to go to the top right corner. Figure 3 shows the coordinates of the walls in the maps. The actual look of the maps is in appendix A.

```
23  dynamic_walls = {0:[(1, 1), (2, 2), (3,0), (4,2), (6,3),(3,4), (4,5)],
24                   4:[(0, 4), (0, 5), (2,6), (4,5), (3,4),(1,3), (2,3), (6,2), (5,1), (4,1), (3,1)],
25                   2:[(4, 0), (6,2), (3,3),(3,1), (3,2), (2,6), (1,5), (1,4), (1,3)],
26                   5:[(4, 0), (5,5), (1,2), (6,2), (5,4), (4,3),(3,1), (3,2), (2,6), (1,5), (1,4), (1,3)],
27                   3:[(0, 2), (0, 3), (1, 3), (2,3), (3,1), (4,1), (1,5), (2,5), (5,1), (6,2), (3,3)],
28                   1:[(1, 1), (2, 2), (5,0), (4,2), (5,3),(3,3), (6,3)]}
29
30  walls = dynamic_walls[walli]
31  specials = [(6, 0, "red", 30)]
```

Figure 3: lines 23-31, *Gui.py*

3.2 Define actions and rules

For all states, the agent have four possible actions: Up, Down, Left and Right. Bumping into a wall counts as a move but its state will remain unchanged.

3.3 The reward system

After the agent has taken an action, the environment needs to make a feedback, that is, the reward. The environment needs to feedback the next state s_{t+1} and the reward for the action taken $r_t(a)$.

The rewards for different states after taking an actions are:

- -0.1 for taking each step
- -1 for hitting a boundary
- 30 for reaching the destination

The Q values are stored in a 2-D array, where the dimensions are states and actions responsively, corresponding to $Q_t(s_t, a_t)$. All values in this array are initialised to 0.1 at the beginning. For s_t =a wall, Q is set to 15.

The *try_move* function in *Gui.py* will both calculate both the new Q score and agent's new position after taking an action.

3.4 Reinforcement learning iteration

The main part of reinforcement learning is actually all about how to update Q and what to do before and after that. Figure 4 illustrates the logic of the iteration in a very clear way. Figure 5 shows the key part of the code related to this iteration. *max_Q* is a function that returns the maximum Q value and the corresponding action the agent can get at given state. α takes part in the *inc_Q* function and I made it dynamic so that the agent can be more of an explorer at the beginning but becomes more and more greedy as the number of iterations increases. This means it will tend to follow a certain explored route instead of finding new ways as it learns more and more about the map. This will increase its efficiency for finding the best route.

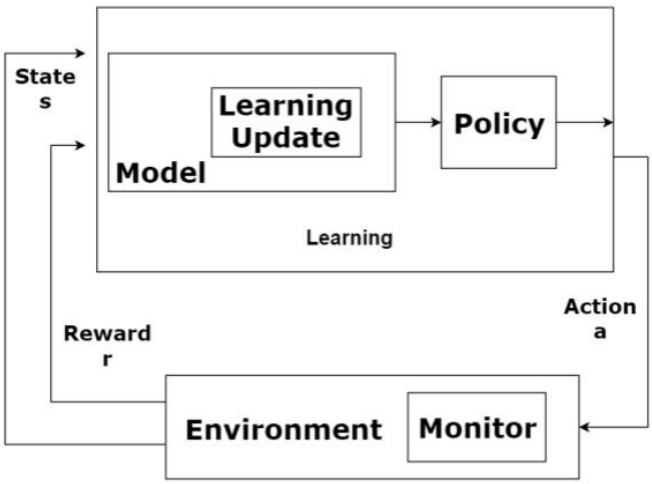


Figure 4

```
while True:
    i = 0
    ff += 1
    # Pick  action
    s = Gui.player
    #print Q[(0,4)]
    max_act, max_val = max_Q(s)
    (s, a, r, s2) = do_action(max_act)


    # Update Q
    max_act, max_val = max_Q(s2)
    inc_Q(s, a, alpha, r + discount * max_val)
```

```
def inc_Q(s, a, alpha, inc):
    Q[s][a] *= 1 - alpha
    Q[s][a] += alpha * inc
```

Figure 5

Each time the agent reaches the destination, we record the final Q score, number of moves and total time used. If the agent reaches the destination with a same Q score for 8 successive times, we consider it has found the best route, or it won't improve anymore,

so we move onto next map.

3.5 Multi-agent

To add another agent to aid the maze solving process, I first copied the existing one and let it run on another thread. However this way these two agents won't coordinate and does not help improving the result. My solution is to divide the work between them, i.e., one agent be explorer (blue block in demonstration), another be the solver, and allowing them to work on a same Q value table. The solve works at half the rate of the explorer. The explorer does not follow the Q value to decide its next move (does not pick a in $\max Qt(s_{t+1}, a)$). It helps explore the map and update the Q table thus the solver can use the result directly.

Chapter 4

Evaluation

4.1 Single agent

The Q-score, number of moves and time cost are recorded for each time the agent reaches destination. Detailed results for this part can be seen in Appendix B.

Figure 6 shows the results for map 1 solved by a single agent. It takes 7 iterations for the agent to find the best route. We can see that in general, the Q-score is increasing, time cost and number of moves are reducing for each literation, which indicates the agent is learning the map. Similar trends can be noticed on all of the 7 maps.

On the other hand, if we look at the results of map 6 (figure 7), we can notice the second iteration achieved quite good result, but the third iteration became bad again. This is because the agent does the exploration at complete random in early stages, or in another word, “luck”.

Figure 6

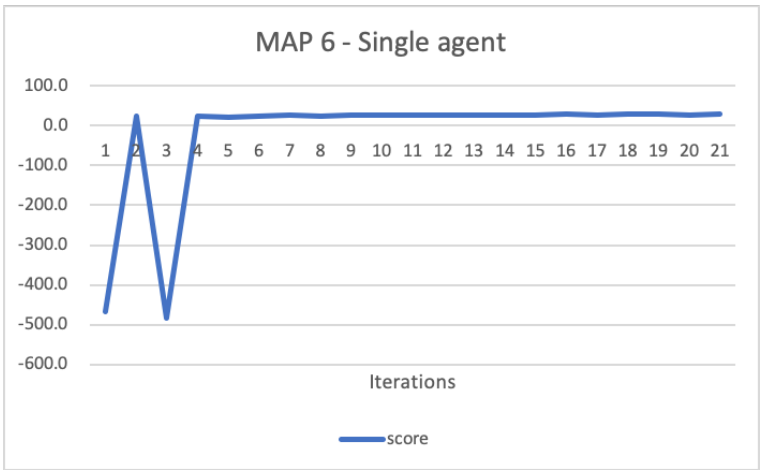


Figure 7

4.2 Multi-agent

Detailed results for this part can be seen in Appendix C.

We can notice is that to achieve the same result, it takes significantly less iterations for multi-agent to work together (see table 1).

map	iterations used	
	single agent	multi-agent
1	7	2
2	12	1
3	9	9
4	24	11
5	10	6
6	21	16

Table 1

Chapter 5

Conclusion

By using Q-learning algorithm, artificial intelligent agents can solve a maze through iterations. Multiple agents working together will highly improve the efficiency of finding the solution. The key to making them coordinate is through split working and put all the results in a shared Q-value table.

In this project I only experimented with one explorer, but with more explorers the solver will be able so solve larger, more complicated mazes in less iterations.

In practical application, many searching problems are very similar to solving a maze. This research may give them a train of thought on how to improve the searching.

Bibliography

MA Yong, LI Long-shu and LI Xue-jun (2008). *Research and application about defensive strategy based on Q Learning*. In *Computer Technology and Development* Vol. 18, No. 12.

GAO Yang, CHEN Shi-fu and LU Xin (2004). *Research on reinforcement learning technology: A review*. In *ACTA Automatica Sinica* Vol. 30 No. 1.

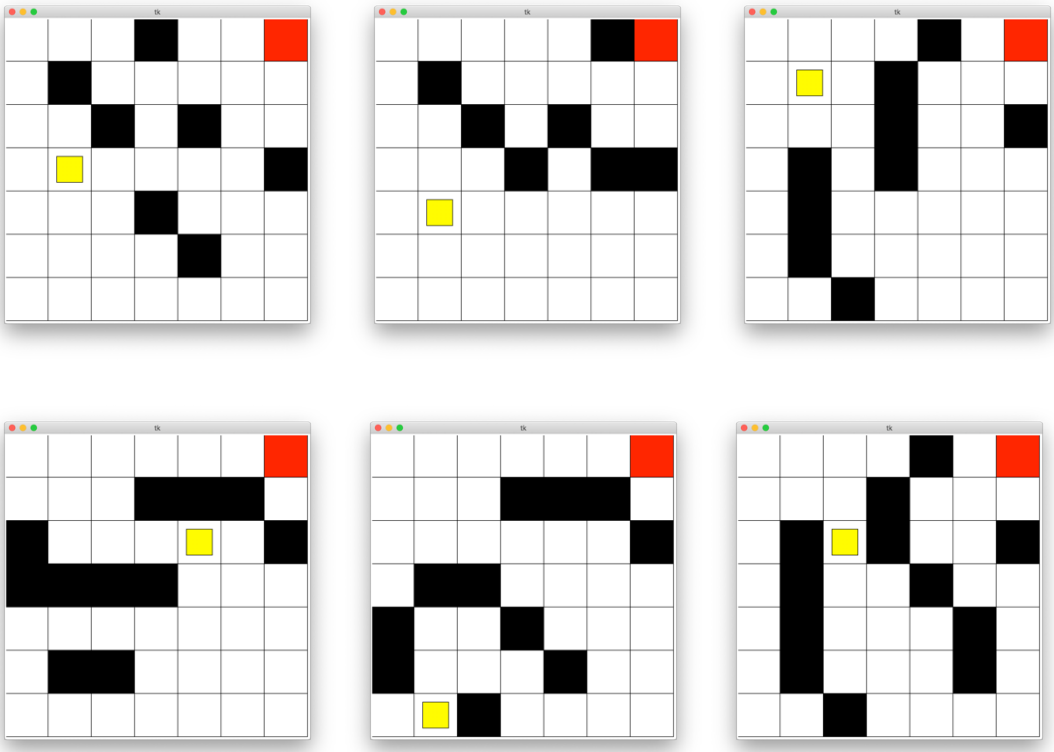
Bowen Baker, Ingmar Kanitscheider, Todor Markovm, Yi Wu, Glenn Powell, Bob McGrew and Igor Mordatch (2020). *Emergent tool use from multi-agent autocurricula*. Published as a conference paper at ICLR 2020.

Tamás Tompa, Szilveszter Kovács (2015). *Q-learning vs. FRIQ-learning in the Maze problem*. In Cognitive Infocommunications (CogInfoCom), 2015 6th IEEE International Conference on (pp. 545-550).

Gilgame (2019). *[Reinforcement learning] Q-learning maze algorithm example*. https://blog.csdn.net/Gilgame/article/details/90669032.

Appendix A

The following figure shows how the six mazes tested in this project looks like. Black blocks are walls, yellow is the agent and red is its destination. Note that the location of the yellow block does not indicate its start position. The agent always start from bottom-left and need to go the top right.



Appendix B

MAP 1

ITERATION	score	move	time	total time
1	21.4	16	3.246	3.246
2	9.1	218	5.295	8.541
3	20.0	108	2.640	11.181
4	29.6	14	0.385	11.566
5	22.7	84	2.041	13.607
6	29.0	18	0.517	14.124
7	29.7	14	0.365	14.489

MAP 2

ITERATION	score	move	time	total time
1	2.1	180	7.720	7.720
2	4.3	166	6.369	14.089
3	-1203.9	10626	298.874	312.963
4	-11.0	338	9.901	322.864
5	28.6	20	0.590	323.454
6	20.9	82	2.453	325.907
7	26.9	38	1.022	326.929
8	24.6	52	1.561	328.490
9	26.6	40	1.129	329.619

10	24.0	60	1.760	331.379
11	29.5	16	0.400	331.779
12	29.7	14	0.341	332.120

MAP 3				
ITERATION	score	move	time	total time
1	-18.7	192	11.697	11.697
2	-288.5	2604	75.100	86.797
3	14.4	140	3.899	90.696
4	28.1	22	0.727	91.423
5	26.8	30	1.007	92.430
6	27.9	22	0.767	93.197
7	29.3	16	0.429	93.626
8	28.7	18	0.578	94.204
9	29.5	16	0.394	94.598

MAP 4				
ITERATION	score	move	time	total time
1	-1086.8	9158	280.984	280.984
2	10.0	154	4.759	285.743
3	17.6	106	3.200	288.943
4	16.8	106	3.475	292.418
5	20.4	78	2.525	294.943

6	25.5	38	1.375	296.318
7	24.8	48	1.509	297.827
8	25.4	46	1.357	299.184
9	26.2	34	1.101	300.285
10	24.1	58	1.648	301.933
11	27.5	26	0.831	302.764
12	28.2	20	0.691	303.455
13	26.5	34	1.113	304.568
14	27.6	26	0.876	305.444
15	29.1	18	0.505	305.949
16	28.8	18	0.576	306.525
17	28.8	20	0.576	307.101
18	27.9	26	0.772	307.873
19	29.3	16	0.454	308.327
20	29.5	16	0.424	308.751
21	29.3	16	0.467	309.218
22	29.4	16	0.424	309.642
23	29.3	16	0.452	310.094
24	29.5	16	0.408	310.502

MAP 5				
ITERATION	score	move	time	total time

1	-221.5	1786	60.145	60.145
2	26.6	34	1.064	61.209
3	28.1	24	0.719	61.928
4	26.5	34	1.064	62.992
5	28.9	20	0.517	63.509
6	28.1	24	0.688	64.197
7	28.3	24	0.639	64.836
8	29.0	20	0.502	65.338
9	28.8	20	0.575	65.913
10	29.1	20	0.477	66.390

MAP 6				
ITERATION	score	move	time	total time
1	-468.0	3140	118.663	118.663
2	22.5	54	2.132	120.795
3	-481.8	4244	120.842	241.637
4	23.1	544	1.917	243.554
5	20.1	78	2.655	246.209
6	22.7	60	1.984	248.193
7	25.5	40	1.343	249.536
8	23.4	52	1.871	251.407
9	26.7	32	1.050	252.457

10	25.5	44	1.344	253.801
11	25.3	42	1.406	255.207
12	25.6	42	1.331	256.538
13	27.4	30	0.916	257.454
14	26.7	34	1.100	258.554
15	26.7	32	1.127	259.681
16	27.9	28	0.799	260.480
17	27.1	32	1.002	261.482
18	28.0	28	0.782	262.264
19	28.3	26	0.710	262.974
20	27.7	28	0.837	263.811
21	28.5	26	0.666	264.477

Appendix C

MAP 1

ITERATION	score	move	time	total time
1	29.4	14	4.257	4.257
2	29.7	14	3.057	7.314

MAP 2

ITERATION	score	move	time	total time
1	29.7	14	3.067	3.067

MAP 3

ITERATION	score	move	time	total time
1	8.0	168	47.538	47.538
2	25.9	40	10.823	58.361
3	26.4	32	9.841	68.202
4	28.7	20	5.129	73.331
5	28.9	18	4.701	78.032
6	29.3	16	3.884	81.916
7	29.2	18	4.295	86.211
8	29.2	16	4.114	90.325
9	29.5	16	3.479	93.804

MAP 4

ITERATION	score	move	time	total time
1	-40.9	532	148.021	148.021
2	25.7	40	11.253	159.274
3	26.5	34	9.625	168.899
4	27.6	28	7.419	176.318
5	27.6	28	7.523	183.841
6	28.3	20	5.959	189.800
7	28.5	20	5.536	195.336
8	28.5	22	5.532	200.868
9	29.5	16	3.468	204.336
10	29.4	16	3.685	208.021
11	29.5	16	3.510	211.531

MAP 5

ITERATION	score	move	time	total time
1	12.3	34	50.284	50.284
2	25.7	22	6.156	56.440
3	28.2	24	6.966	63.406
4	27.8	22	5.142	68.548
5	28.7	20	4.316	72.864
6	29.1	20	4.313	77.177

MAP 6

ITERATION	score	move	time	total time
1	-445.9	450	157.907	157.907
2	20.6	744	21.697	179.604
3	22.0	456	18.999	198.603
4	27.2	612	8.150	206.753
5	23.8	46	15.038	221.791
6	24.7	48	13.341	235.132
7	26.8	36	8.967	244.099
8	27.4	28	7.773	251.872
9	27.3	32	7.992	259.864
10	26.5	28	9.585	269.449
11	27.6	28	7.331	276.780
12	28.6	24	5.325	282.105
13	28.1	26	6.353	288.458
14	28.7	24	5.115	293.573
15	28.5	26	5.516	299.089
16	28.8	24	5.111	304.200