EECS 4401 Project Report.
Shiyi Du
214438469

Topic: AI Learning Game with Q-Learning.

Project Motivation

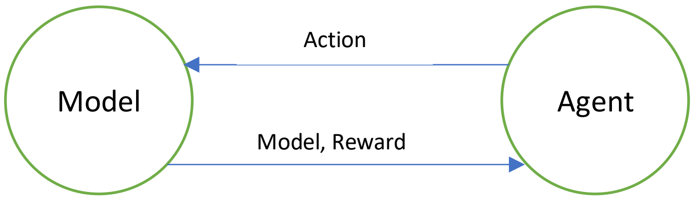
The motivation of this project is to explore and to learn more about the idea of reinforcement learning through the implementation of Q-Learning algorithm and the evaluation of the algorithm on some real-world problems in particular games.

The objective of the project is simple: implement the algorithm, evaluate it on some simple problems, evaluate it on some large problems, explore how to do state abstractions more efficiently when dealing with problems with large state space and see how the learning rate and the performance of the agent are effected.

Project Implementation

The nice thing about reinforcement learning is that it is model-free, meaning that I can implement the algorithm without worrying about how the problem would look like. Therefore, I can implement the algorithm once and evaluate it on many problems with a bit of state abstraction.

The relation between the problem and the algorithm is a classical client-supplier relation with the algorithm being the supplier. The algorithm accepts 2 things: the current **state** of the model (encoded as numbers), the **reward** from executing the last action, the algorithm then update the q-table based on the information given and **return a new action** it deemed necessary (exploration or exploitation).



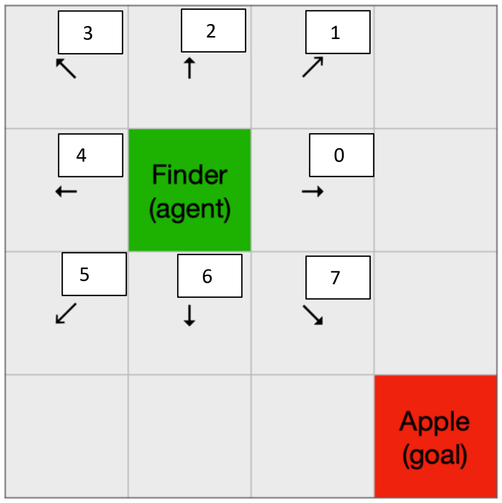
Whenever an action is executed and the state is changed, the Q value will be updated based on the following Temporal-difference value update rule(I modified a little for readability):

currentQ = qTable (oldState, action);
*newQ = (1 - learningRate) * currentQ + learningRate * (reward + discountFactor * futureReward));*
qTable (oldState, action) = newQ;

In which the *oldState* is the state where the *action* is being executed, the *reward* is the reward we received from the model by executing the *action* in *oldState*

The exploitation rule for Q-learning is to simply pick up the action that has the highest Q-value from the table. To introduce some exploration to the algorithm, I declared another variable *greedyFactor,* which is the probability that a random action would be taken; And every time we want to pick an action from the table, we would first generate a random number, if that number is smaller than *greedyFactor,* we would randomly pick an action from the table where each action is equally likely to be picked. This turns out to be a very important number for the problems I tried to solve, which are introduced below.

Problem I: Apple Finder game
To determine the usability of the algorithm, I first created a simple game. In this game, you control a single square and try to eat as much apple as you can. The only actions you can perform is to go up, go left, go right or go down.



To abstract the states, I calculated the relative angle of the apple based on finder's position and apple's position, and I cut the angle into 8 different directions as shown in the picture. For each direction, a number will be assigned to it representing the current state. In the above example, the apple is on the bottom right corner which is in the direction '↘', as can be seen, state 7 is assigned to this direction, so we will tell the algorithm that we are in state 7 right now.

Reward
In every state, the agent will pick up an action from the table, and the model will give reward to the algorithm. The rules I used is as follows:

If the finder gets further away from the apple, punish it by 1
If the finder gets closer, reward it by 1
If the finder eats the apple, reward it by 10

Parameters
greedyFactor(initial) = 0.6
discountFactor = 0.5
learningRate = 0.8

Note that the greedyFactor is changed dynamically, the more actions the agent does, the lower the greedyFactor (in this case it is decreased by 0.002 every time an action is taken)

Result
Since the state space is really simple (8 state and 4 actions), the agent learns within a hundred steps (within a second when running in full speed) that by getting closer to the apple, it will receive max reward.

I noticed that by giving the agent a higher greedyFactor, the agent tends to learn faster and gain experience more quickly at the start. And by slowly decreasing it, the agent will perform with less errors and performs better.

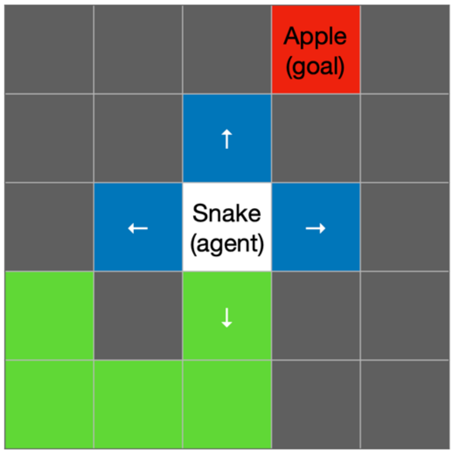
One interesting finding about the reward function is that the reward of eating the apple doesn't really affect the learning speed of the agent; My assumption is that since I didn't encode the distance to the apple into the states, the agent can never know if the next step will get us closer to the apple or just eat the apple; Therefore the agent can never associate the reward of eating the apple with any of the 8 giving states.

Problem II: Snake Game
To evaluate the algorithm on more complex problem, I created a snake game. In which the goal of the player is to eat as many apples as possible, if you eat an apple, you body will get longer. If the snake hits a wall or hits its own body, it dies. This requires the learning agent to maximize its reward while avoiding the risk of dying.

I defined 3 actions in this game, turn-left, turn-right or do nothing; and if the snake do nothing, it will simply move forward to its current direction.

State Abstractions

In this case, not only does the agent needs to know where the goal is, it also needs to know where the risks are. Therefore, I must also encode the obstacles around its head into the states as well as the direction of the apple.



As can be seen from the graph, the white square is the head of the snake, and the blue squares are the visions of the snake (front, left and right). The vision of the snake will be rotated based on the direction of the snake, so in any states, it will be able see its front, left and right.

To encode this data into a state number, we need a 3-bit number, where each bit represents a square. If the square is an obstacle, that bit is 1, if the square is an empty slot, then that bit is 0.

For example, it left, front and right are all empty, then the bit representation would be 000. If left and right are blocked and the front is empty, it would be 101. And if all three blocks are blocked, it would be 111.

Therefore, to represent the obstacle information as a state number, we would need 8 different states; but we also need to encode the goal into the states, which is the direction of the apple. To simplify the state space, instead of encoding 8 direction like the first example, I only encoded 4 directions (up, left, right, down) of the apple.

Therefore, for each direction, there are 8 different states, that gives us a total of $4 \times 8 = 32$ number of states.

<u>Reward</u>

If the snake eats an apple, reward it by 100
If the snake dies, punish it by 100
If the snake gets further away from the apple, reward it by 1 (for staying alive)
If the snake gets closer to the apple, reward it by 2

<u>Parameters</u>

greedyFactor(initial) = 0.2

discountFactor = 0.5
learningRate = 0.8

Note in the previous case, we can assume the agent is getting better and hence decreasing the greedyFactor by a constant value. However, this problem is a little more complex: if the greedyFactor is too high, the snake will die too soon before hitting high score; if the greedyFactor is too low, the snake might stuck in a loop and never try to eat apple (stuck in a local minimum) Therefore, I updates the greedyFactor based on the current high score; So for every step taken, the following rule will be used to update the greedyFactor

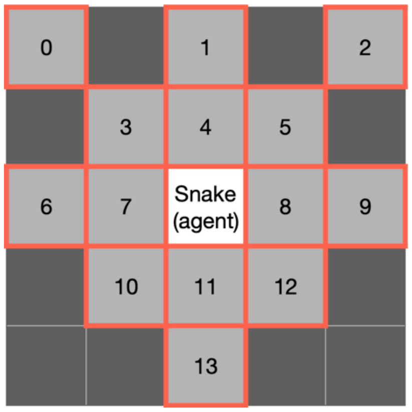
$$\text{greedyFactor} = (1 / \text{highestScore} \times 200)$$

<u>Result</u>
Since the state space is 8 times larger than previous example, the learning speed does gets slower, however, the training can still be done within a second with a steady average score of 15 on a 20×20 map. The limitation is obvious, the snake only knows to avoid immediate obstacles, it doesn't know if it will get to a dead end by doing so. Therefore, this score still have a lot more room to increase, which lead us to our next problem.

Problem III: Snake Game with More Visions.
In the previous example, the obvious problem is that the vision of the snake is limited, therefore we want to give the agent more knowledge by encoding more information into states.

However, the actions I defined is slightly different, I defined 4 actions: go left, go right, go up or go down. And the reason why I do this is simply because I don't have to rotate the vision of the snake as the direction of the snake changes by doing so.

<u>State Abstraction</u>
In this case, the first obvious thing is to encode more obstacle information into states. In previous example, I encoded information of 3 cells. But in this example, I want to encode more cells into states. Therefore I picked some key cells to encode into states as shown (the numbered cells are key cells):



As can be seen, there are in total 14 key cells that I need to encode. We can simply use the encoding rule from the previous example, except instead of a 3-bit representation, we now need to use a 14-bit representation.

The direction of the apple must also encode into the states, since we have 4 possible directions where the apple is, we need another 2-bit to represent the direction of the apple.

To make it more interesting, I decided to let the agent know which direction have less than 20% of remaining cells accessible(meaning those cells are not blocked), if it has less than 20% of remaining cells, then mark it with 1, otherwise 0. Because if a direction has less than 20% of remaining cells, it means going into that direction is a more dangerous action as it is more likely to hit a wall. I want to let the agent know this information on all 4 directions so that require another 4-bit

Therefore, in total, we have encoded all the information in 20-bit, which gives us a total state space of 1,048,576.

Reward

If the snake eats an apple, reward it by 3
If the snake dies, punish it by 100
If the snake gets further away from the apple, reward it by 1 (for staying alive)
If the snake gets closer to the apple, reward it by 2

As can be seen, the reward for eating the apple is decreased, because comparing to eat more apples, I think it is more important to stay alive. Therefore, I think the punishment for death should be more severe here.

Parameters

greedyFactor(initial) = 0.2
discountFactor = 0.8
learningRate = 0.5

the discount factor is slightly higher because I want the agent to value future rewards over the immediate reward.

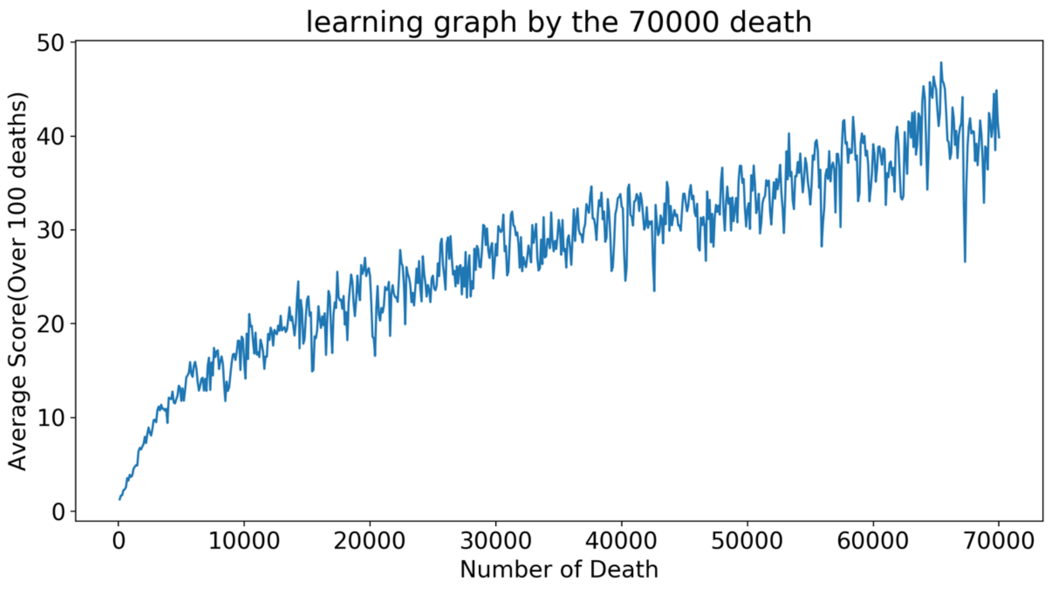
The greedyFactor update rule is the same as previous example. I think it gives a good balance between exploration and exploitation.

Result

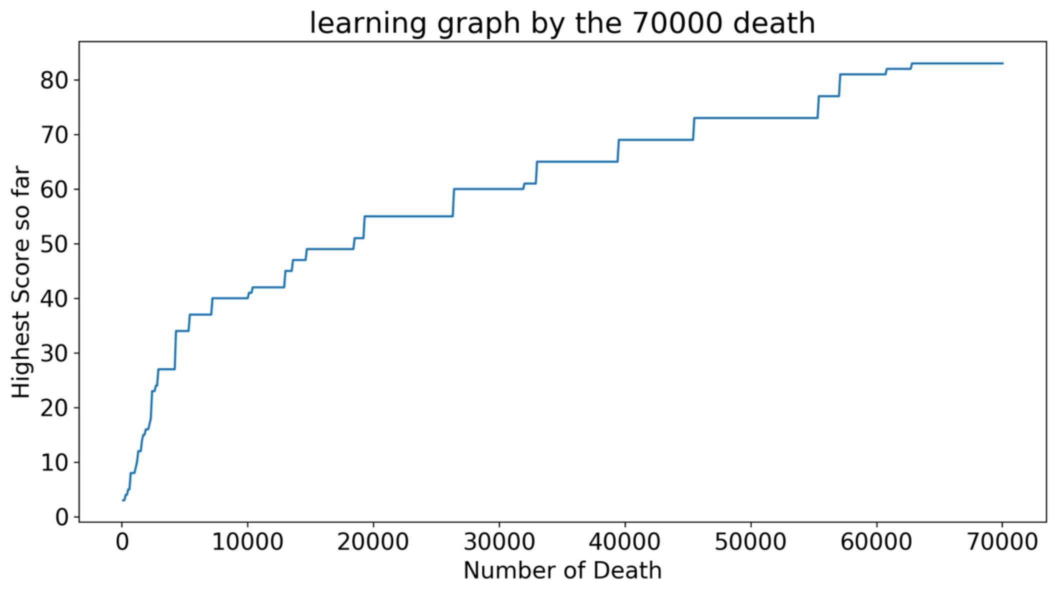
Since the state space is dramatically increased, the training speed is vastly slowed. In the previous example, the agent hit an average score of 15 within a second (hundreds of deaths), but in this example, by the time agent hit an average score if 10, it has already been trained for several minutes (3100 deaths)! And when it finally hit 15, its already been trained for about 20 minutes (6000 ~ 8000 deaths)!

The best result I have got by training this agent is an average score of 55+, however, in order to get this result, I have to pause the training several times, play around with the parameter a couple of times and train the agent for hours and hours with a total death of 230,000.

In order to collect data and draw the learning curve of the agent without me tweaking parameters in-between, I started training the agent from ground up. As I am writing this report, this agent has been training for almost 5 hours, and here is the learning curve graph of the agent by the 70,000th death:



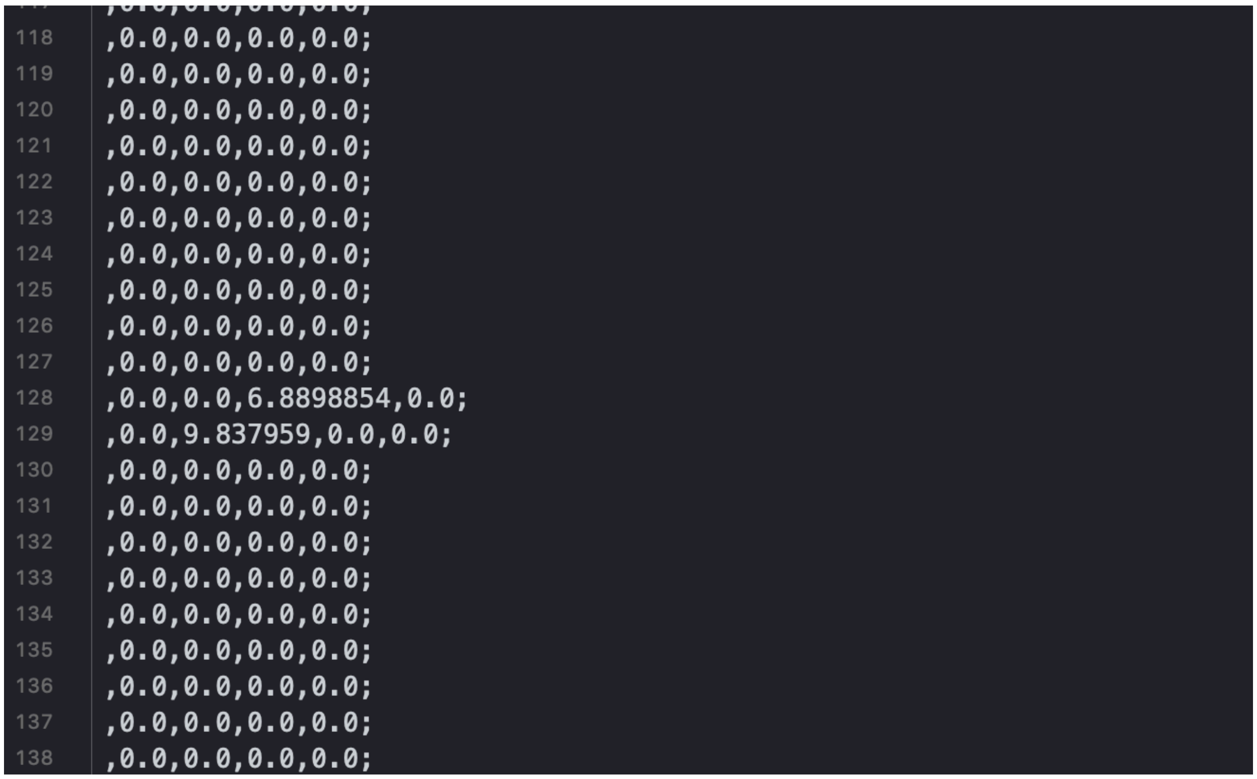
The following graph shows the highest score the agent achieved so far, which is basically the same shape:



Limitations

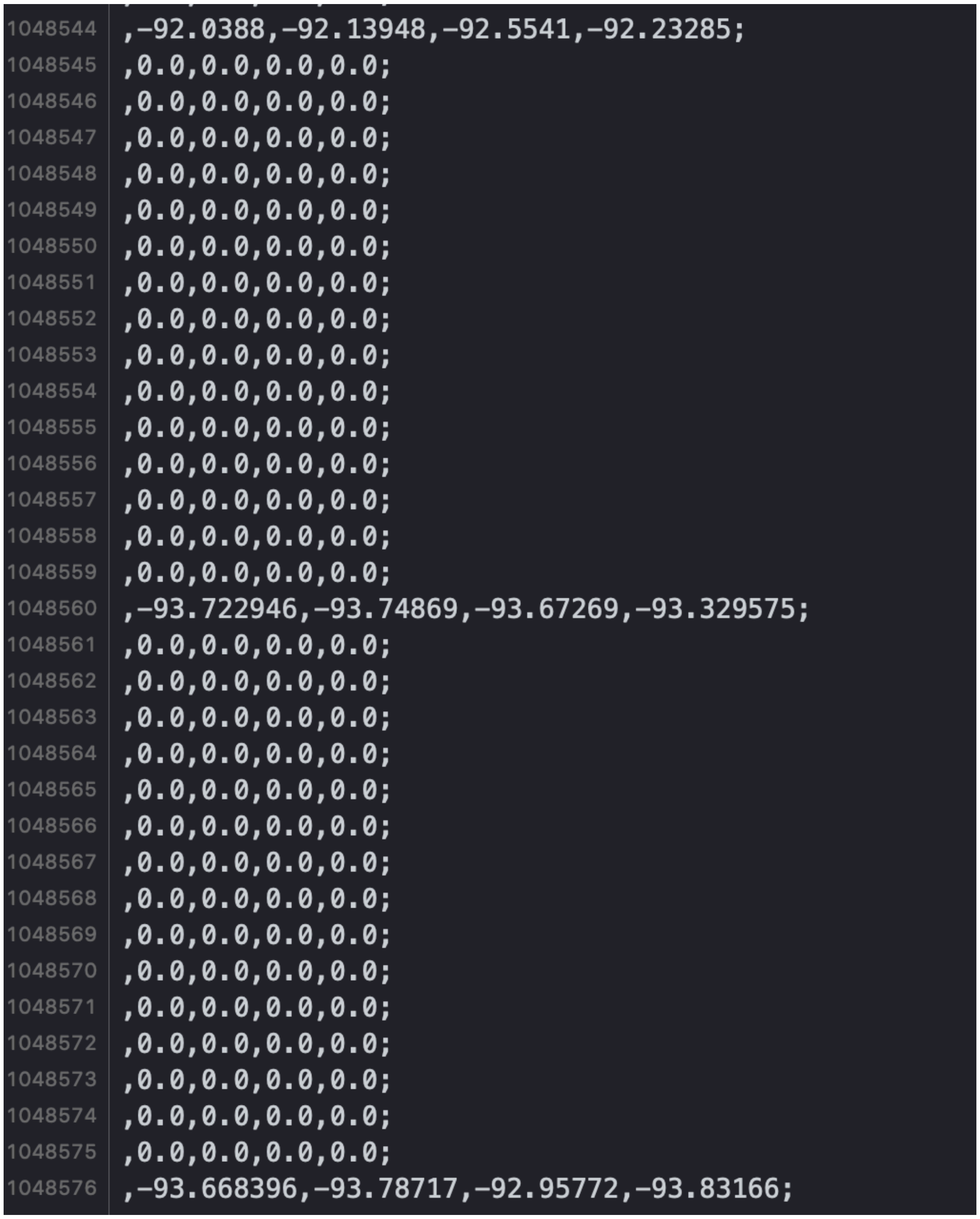
As can be seen, the limitations for the basic Q-learning algorithm is apparent as state space increases, the reason is mostly because q-table is basically a discrete function that store's all the values of all the possible states and actions, unlike a neural network, which is really good at handling unknown states, the Q-learning algorithm can only handle the past experience it has learned. Therefore, state abstractions become really important. The way I do state abstraction is pretty basic, there are still a lot of different states that can be grouped together to perform similar actions and to learn faster. And based on my q-table, I have also noticed that a lot of the states are impossible to get into, for example this is what my q-table looks like after 5 hours of training with a total of 70,000 death:

Looking at the graph, you will notice that a lot of the states have 0 q-values, my assumption is that those states are never possible to get into in the first place.



```
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,6.8898854,0.0;
,0.0,9.837959,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
```

And some states have very similar Q-values, which might be better to group them together in the first place:



```
,-92.0388,-92.13948,-92.5541,-92.23285;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,-93.722946,-93.74869,-93.67269,-93.329575;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,0.0,0.0,0.0,0.0;
,-93.668396,-93.78717,-92.95772,-93.83166;
```

There are some techniques to overcome its limitation of course, one of which is called Deep Q-Learning, which is combining the idea of q-learning and neural networks. Another way to overcome this is to do state abstractions via a neural network while still keep the q-table.

Source
Since this is an implementation project, there wasn't really any academic paper or books I read, I just watched a bunch of tutorials and read some blogs, but I did make my project public on the github:

Link to the project: https://github.com/ShiyiDu/Q_Learning_Java

In which you can also find the LearningData.txt, which is the record of how well the agent is learning. You can also find a bunch of training data file in the training_date folder, which is basically the q-table.

In the Main.java, you can run one of the following lines to see the training in action:

```
initAppleFinderGame();
initSnakeGame();
initSnakeGame2();
```