Using reinforcement learning for Minesweeper game

School of Computing, Engineering and Digital Technologies

Module: Artificial Intelligence Foundations

Author Name: Keyhan Azarjoo

Email: keyhanazarjoo@gmail.com

Jan 2023

Contents

Abstract

In this report, we implemented a Q-learning (Reinforcement Learning) on Minesweeper game. We checked Q-learning on this game in two different situations. First, we fixed the mines in the game board and train our model 10,000 times. After that we check the numbers of the wins and loses during the training and the result was great. Almost 95% of the times agent wins the game. But the problem was that if we change the game board, agent cannot win, and it is a big problem. In the second model, we try to use different game board each time and we put the mines randomly in the bord. Then we run the model with two different parameters. First by acting based on Q-table and second by acting randomly. We wanted to put our priority on filling the Q-table and the result shows that by increasing the exploration we get about 50% wins in most of the situation.

By Keyhan Azarjoo

Introduction

Minesweeper is a game based on decision-making with un insufficient information. It is a single play game that a n*m grid has been given to the player and there are K mine in this grid. These mines randomly have been put in the grid and the player don't know about the location of the mines. the player should uncover all the boxes in the grid and shouldn't chose a mine. Decisions are based on the numbers next to the mines which show the number of mines next to that block.

The first action is randomly but the rest of the game is based on 8 blocks around each block.

Analysing the patterns and situation is the significant part of this game, the player should find the safest block that can be uncovered during the game, there for, the player can use a flag for a block if he or she find that it is a mine.

However, the player can lose even if he/she decide a block as the lowest probability of being a mine and it is computationally difficult to find the best decision without considering all other option and blocks [1].

We decided to use Q-learning in reinforcement learning to train the AI to play this game. For doing that, first we should design a Q-Table and giving appropriate rewards to the actions based on their position.

We implemented two different methods in this research, and we plan to improve it and use neural networks in future work.

Background

Improvement of computer and Artificial intelligence (AI), surprised human in this area. Reinforcement Learning (RL) is an important part of AI which is about interaction between an agent and environment. RL acting based on giving reward or punishing the agent during it is acting. In RL agent learn step by step and continue to learn any time. Q-learning is a model in RL which the agent records any reward in a table as Q-table. During the running algorithm the agent fill the Q-table based on a formula and explore the environment based on the exploration rate. Deep Learning (DL) and Deep Neural Network have been used with RL in gaming, robotic etc, especially during the last decades [15].

Using AI in some games like GO [2] and Chess [3] are some examples of using AI in games.

In these two games, AI learned how to play and then they win the champion of this games.

Minesweeper is a popular game which can be put in difficult and NP-hard class problems [4] and there are a lot of research about this game in AI field.

There are different ways to implement this game using AI. In one strategy which called single-point strategy one uncovered instance can be consider as the next block and it should be from id immediate neighbours, but this strategy is good for lower mine dimension [5].

In [6] the writer used CSP as one of the easiest ways to implement this game and in [7] David Becerra used CSP to solve the minesweeper game and he got that this method work effectively on this game.

Dechter and Forst in [8] avoid unimportant movement in the search space ant they could get a better result in CSP method.

Wei and Nakov used POMDO (Partially Observable Markov Decision Problem) and Sequential decision-making problem in minesweeper game in [9]. In addition to that, they decrease the state space as well.

In [10], Castillo and Wrobel search and use a method for learning a strategy to learn how to play by giving the state and finding the best guess which reduce the risk of losing. Moreover, they use ILP (Induction of logic program) like Macros greedy.

Research have been done by Sebag and Teytaud about using POMDP and in this research author simplify the problem for optimizing the problem. They use Upper Confidence Bounds (UCB) in Monte-Carlo Trees [11].

According to Legendre research [12], researchers have utilised heuristics with computing probability from CSP called HCSP to analyse the effects of the initial move and to solve the challenge of choosing the next block to uncover for complicated Minesweeper grids based on various techniques. They achieved better results for different sizes of Minesweeper matrices by favouring the blocks that are closer to the border in the event of a tie on the probability of mines.

In another research by Buffet and his group in [13] they use UCB in [11] and HCSP in [12] to improve the UCB. The reason for that was they wanted to use the algorithm for small and big boards.

Gardea in [14] used Machine Learning and Artificial Intelligence methods like Reinforcement Learning to solve this game and the results were quite good.

Moreover, for solving MDPs, Q-Learning [15] which is a reinforcement learning has been successfully used in [5] and [16] and we use Q-learning in a part of our research.

Research Questions

1. How can we write a program to play the Minesweeper game by its own choses and win the game?
2. How can we fill the Q-table and give the reward to the actions?

Methodology

Getting data

As Minesweeper is an agent-based game and we don't have any data table for this game, we should create our data first and fill the Q-table based on those data. In this research we implement this game and fill our Q-table by using Q-Learning in reinforcement learning.

Then we run some new game and masur the result for both training and testing part of the research.

Experiment

We use quantitative methods by creating an algorithm to play this game for thousands of times and fill the rewards table(Q-table) based on actions and rewards. We use different method and measure the rewards in different ways.

First, we implement the game with one unchangeable game and train our RL algorithm and fill Q-table. Then in the second part we train our AI algorithm by running a different environment. The different of these two methods is them rewarding methods.

Two methods have been used in this research:

1. By giving only one game board. In this method we fixed the mines in special blocks and by each time playing we just reset the game and covered all the boxes.
   The rewarding system was simple and by giving +50 point to wining, +10 point to choosing a box that is not mine and -50 by clicking on a bomb.

2. Putting bombs in random places each time that the algorithm plays the game. In this method we did not gave special reward to each action though we create the reward based on the box and probability of being mine.

The most significant difference between these two methods are the rewarding system and Q-table.

First method

Explanation

As we mentioned above, in this method, we put 4 mines in specific boxes and during the game we don't change the mines places. And by resetting the game we cover all boxes.

We play the game for 10,000 time during the training and gave reward to each action.

The game board was created based on N*N which in this research the N is 5 and the board game is 5 * 5.

And we put K bombs on the game which K is 4.

So, we have 25 states and 25 action that the AI can chose.

Based on this number we create the Q-table 25 * 25.

Implementing the game

For making everything easier we use OOP (Object oriented programming) in our programming.

So, at the first step we wrote a class with different functions for creating, running, rewarding, resetting and doing other functions.

As you can see in (figure 1) a function has been written to create the game and putting the mines in the game board.

```
Minesweeper.action_space_size = n * n
Minesweeper.state_space_size = n * n
for i in range(n):
    for j in range(n):
        Minesweeper.arr_sam[i][j] = '-'

for num in range(k):
    r = random.randint(0,Minesweeper.N-1)
    c = random.randint(0,Minesweeper.N-1)
    if(Minesweeper.arr[r][c] == 'X'):
        r = random.randint(0,Minesweeper.N-1)
        c = random.randint(0,Minesweeper.N-1)
    Minesweeper.arr[r][c] = 'X'

for c in range(n):
    for r in range(n):
        if Minesweeper.arr[r][c] == 'X':

            if (c != Minesweeper.N-1 and Minesweeper.arr[r][c+1] != 'X'):
                Minesweeper.arr[r][c+1] += 1 # center right

            if (c != 0 and Minesweeper.arr[r][c-1] != 'X'):
                Minesweeper.arr[r][c-1] += 1 # center left

            if (r != 0 and Minesweeper.arr[r-1][c] != 'X'):
                Minesweeper.arr[r-1][c] += 1 # center top

            if (r != Minesweeper.N-1 and Minesweeper.arr[r+1][c] != 'X'):
                Minesweeper.arr[r+1][c] += 1 # center bottom


            if (c != Minesweeper.N-1 and r != 0 and Minesweeper.arr[r-1][c+1] != 'X'):
                Minesweeper.arr[r-1][c+1] += 1 # top right

            if (c != 0 and r != 0 and Minesweeper.arr[r-1][c-1] != 'X'):
                Minesweeper.arr[r-1][c-1] += 1 # top left

            if (c != Minesweeper.N-1 and r != Minesweeper.N-1 and Minesweeper.arr[r+1][c+1] != 'X'):
                Minesweeper.arr[r+1][c+1] += 1 # bottom right

            if (c != 0 and r != Minesweeper.N-1 and Minesweeper.arr[r+1][c-1] != 'X'):
                Minesweeper.arr[r+1][c-1] += 1 # bottom left5
return Minesweeper.arr
```

Figure 1

Figure 2

```
M = Minesweeper
n = 5
k = 4
arr = M.Create_minesweeper(n,k)
M.Show_Map()
```

1	1	2	X	1
1	X	3	3	2
1	2	X	2	X
0	1	1	2	1
0	0	0	0	0

Figure 3

Here we put the mines in the first part of the code and increase the number of mines around each box. like the numbers and mines in (figure 2) and we implemented this part of the code by the last part of the (figure 1).

In the figure 3 you can see an example of how this part of code create the game board.

Then we create the Q-table in (figure 4) which both of state space size and action space size are 5*5. In other word q_table is in the size of 25*25

```
q_table = np.zeros((state_space_size, action_space_size))
```

Figure 4

We decide to create our Q-table in [n*n] [n*n] dimension for showing where was the last box, which the algorithm uncovered and based on that which box has the highest reward. We put the agent in the last block which is covered and made the decision based on the status. The reason for this is as Q table does not change during the test, the action will be the same in every game, so we use this method for making the game more real.

Furthermore, we gave a number based on the location of the box to make the process easier to understand

For instance, we give number 20 for box in location [5][1] in (figure 3).

Implementing the Model

In Q-learning, we have some parameters as you can see in (figure 5).

In reinforcement learning, we have two ways of acting. Exploration and Exploitation. The difference between this two is acting based on the Q-table and acting randomly. In (figure 5) we gave exploration rate to the algorithm.

```
num_episodes = 10000
max_steps_per_episode = 200

learning_rate = 0.1
discount_rate = 0.99

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01
exploration_decay_rate = 0.001

rewards_all_episodes = []
```

Figure 5

```
# Q-learning algorithm
win_list = []
CW, Q_counter, random_counter, win_Counter = 0
for episode in range(num_episodes):
    M.reset()
    clear_output(wait=True)
    print('Play : ' , episode)
    state = M.action_space_sample()
    done = False
    rewards_current_episode = 0

    for step in range(max_steps_per_episode):

        # Exploration-exploitation trade-off
        exploration_rate_threshold = random.uniform(0, 1)
        if exploration_rate_threshold > exploration_rate:
            action = np.argmax(q_table[state,:])
            Q_counter += 1
        else:
            action = M.action_space_sample()
            random_counter += 1
        M.run_Steps_Game(action)
        reward = Minesweeper.Rewards
        new_state = M.action_space_sample()
        done = M.Done
        # Update Q-table for Q(s,a)
        q_table[state, action] = q_table[state, action] * (1 - learning_rate) + learning_rate * (reward + discount_rate * np.max(q_table[new_state, :]))
        state = new_state
        rewards_current_episode += reward
        if done == True :
            CW = M.Check_win(n,k)
            break
    win_list.append(CW)
    if CW == 1:
        win_Counter+=1
    # Exploration rate decay
    exploration_rate = min_exploration_rate + (max_exploration_rate - min_exploration_rate) * np.exp(-exploration_decay_rate*episode)

    rewards_all_episodes.append(rewards_current_episode)
    print('Win : ',win_Counter)
Play :  9999
Win :  7207
```

Figure 6

In the training and filling part, as you can see in (figure 6) we fill the Q-table based on the first formula and we change the exploration rate based on the second formula.

In this figure, you can see the result and after 10,000 time playing, the algorithm wins 7,207 times.

Based on (figure 7), during the training the algorithm make 303,663 decisions based on Q-table and 17,216 decisions randomly for exploring the environment.

```
print('decision based on Q-table : ' , Q_counter)
print('decision randomly : ', random_counter)
```

```
decision based on Q-table :  303663
decision randomly :  17216
```

Figure 7

Then we calculate the average reward per 1000 episode and the result is shown in (figure 8). As you can see in this algorithm almost after 6,000 time playing the game the reward stays almost stable and the algorithm couldn't learn more than that.

```
# Calculate and print the average reward per thousand episodes

rewards_per_thosand_episodes = np.split(np.array(rewards_all_episodes), num_episodes/1000)
count = 1000
print("********Average reward per thousand episodes********\n")
for r in rewards_per_thosand_episodes:
    print(count, ':', str(sum(r/1000)))
    count += 1000
# Print updated Q-table
print("\n\n********Q-table********\n")
print(q_table)
```

```
********Average reward per thousand episodes********

1000 : 8.06700000000001
2000 : 66.04700000000001
3000 : 106.1790000000003
4000 : 134.57500000000064
5000 : 144.7780000000003
6000 : 152.65399999999988
7000 : 150.7060000000001
8000 : 153.17000000000007
9000 : 154.1469999999997
10000 : 150.63700000000034
```

Figure 8

For observing how algorithm learned and acted in each episode we put the winning time in a linear graph in (figure 9). In this graph you can see after almost 4,000 time playing the game the result was about 90% wining.

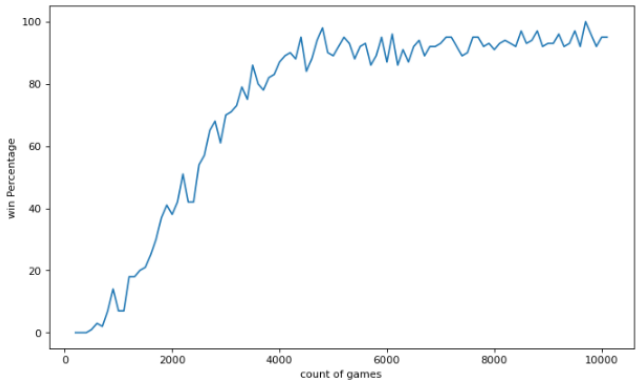

Figure 9

Second method

Explanation

In this method, we observe the whole board and we put the state as 1.

As we should give a reward for each action in Q-learning and we do not have different states, only one, we decide to create our Q-table in shape [1] [n * n * 100].

Our game is a 5(rows) * 5(columns) matrix. So, we have 100 states for each box and consequently we have 2500 state in all.

Number 1 is because we see the board not in special block but in out of the board.

N * N is based on the number of boxes which in our scenario we have 25 boxes.

Number 100 is based on the probability of each box. First, we gave to each box 1 to 100 percent reward and the reward is based on the probability of being mine. The lower probability, the higher reward. By this mechanism we have a floating score.

Then we gave 100 columns to each block.

Block with 1% probability until block with 100% probability and we fill the columns based on their probability and rewards.

For instance, block number 12 with 83% probability will go to column number:

12 * 100 + 83 = 1283

Imagen, we want to calculate the probability of red and green box in (figure 10). For finding the probability of the green block, we should find how many percentages of the 1 next to the green box is for green box. We should do that calculation for all boxes next to the green box.

As you can see in (figure 10) the probability of having a mine in next to the black box is 0.5 and 0.2 for each of them.

For doing that, first we find the number of covered boxes around the number then we put it in the below formula:

Number in box / covered boxes.

Figure 10

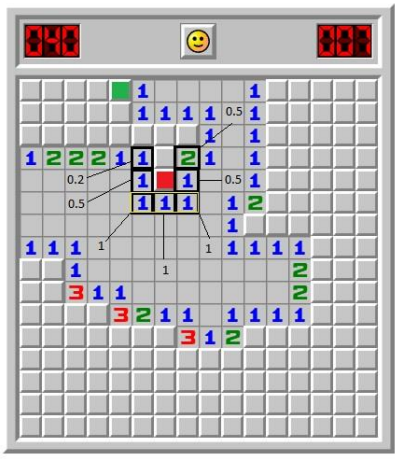

Figure 11

For the number at right of the green box we calculate it as:

1/2 = 0.5

And for number at right down of the green box we have:

1/5 = 0.2

So, we give 0.5+0.2 = 0.7 to the green box.

We put the probability of 10 for when we 100% sure about a mine. Now we should reverse this number and as the highest probability is 10, we use this formula for reversing the number.

(10 - prob) / 10. So, the probability became between 0 and 1. Now the probability of the green box is:

(10 – 0.7) / 10 = 0.93

So, we give 0.93 as reward to the green box.

Now we do the same for the red box in (figure 11).

For finding the probability of 100% being mine, in (figure 11) we check that is the count of the covered boxes is the same as the number inside the uncovered box?

If yes, we gave the probability of 10 from 10.

So here in 3 yellow boxes the result will be 10 for each of them so the probability will be more than 10. Then the reward will be (10 – 10) / 10 = 0

Implementing the Model

For getting the probability of boxes with number we use the code in (figure 9). And for getting the probability of covered boxes we used function in (figure 13).

```
def find_probabelity_around(num):
    prob = 0
    box = 0
    r,c = Minesweeper.Convert_StepNumber_To_Address(num)
    if Minesweeper.arr_sam[r][c] == 0:
        return 0
    if Minesweeper.arr_sam[r][c] != '-' :
        if (c != Minesweeper.N-1 and Minesweeper.arr_sam[r][c+1] == '-'):
            box += 1 # center right

        if (c != 0 and Minesweeper.arr_sam[r][c-1] == '-'):
            box += 1 # center left

        if (r != 0 and Minesweeper.arr_sam[r-1][c] == '-'):
            box += 1 # center top

        if (r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c] == '-'):
            box += 1  # center bottom

        if (c != Minesweeper.N-1 and r != 0 and Minesweeper.arr_sam[r-1][c+1] == '-'):
            box += 1 # top right

        if (c != 0 and r != 0 and Minesweeper.arr_sam[r-1][c-1] == '-'):
            box += 1 # top left

        if (c != Minesweeper.N-1 and r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c+1] == '-'):
            box += 1 # bottom right

        if (c != 0 and r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c-1] == '-'):
            box += 1 # bottom left

        probability = Minesweeper.arr_sam[r][c] / box

        if Minesweeper.arr_sam[r][c] == box:
            probability = 10

        return probability #
```

Figure 13

```
def find_probabelity(num):
    r,c = Minesweeper.Convert_StepNumber_To_Address(num)
    prob = 0
    if Minesweeper.arr_sam[r][c] == '-' :
        if (c != Minesweeper.N-1 and Minesweeper.arr_sam[r][c+1] != '-'):
            # center right
            num_T = Minesweeper.Convert_Address_To_StepNumber(r,c+1)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (c != 0 and Minesweeper.arr_sam[r][c-1] != '-'):
            # center left
            num_T = Minesweeper.Convert_Address_To_StepNumber(r,c-1)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (r != 0 and Minesweeper.arr_sam[r-1][c] != '-'):
            # center top
            num_T = Minesweeper.Convert_Address_To_StepNumber(r-1,c)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c] != '-'):
            # center bottom
            num_T = Minesweeper.Convert_Address_To_StepNumber(r+1,c)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (c != Minesweeper.N-1 and r != 0 and Minesweeper.arr_sam[r-1][c+1] != '-'):
            # top right
            num_T = Minesweeper.Convert_Address_To_StepNumber(r-1,c+1)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (c != 0 and r != 0 and Minesweeper.arr_sam[r-1][c-1] != '-'):
            # top left
            num_T = Minesweeper.Convert_Address_To_StepNumber(r-1,c-1)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (c != Minesweeper.N-1 and r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c+1] != '-'):
            # bottom right
            num_T = Minesweeper.Convert_Address_To_StepNumber(r+1,c+1)
            prob += Minesweeper.find_probabelity_around(num_T)
        if (c != 0 and r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c-1] != '-'):
            # bottom left
            num_T = Minesweeper.Convert_Address_To_StepNumber(r+1,c-1)
            prob += Minesweeper.find_probabelity_around(num_T)
        if prob >= 10:
            prob = 10
    return prob
```

Figure 14

As the finding reward is based on the probability of being mine, we checked the probability of being mine in all covered boxes around an uncovered box. For doing that, we use the function in (figure 15)

```
def find_rewards(num):
    Counter = 0
    Free_box = 0
    r,c = Minesweeper.Convert_StepNumber_To_Address(num)

    if (c != Minesweeper.N-1 and Minesweeper.arr_sam[r][c+1] != '-'):
        Counter += Minesweeper.arr_sam[r][c+1] # center right
        Free_box += 1

    if (c != 0 and Minesweeper.arr_sam[r][c-1] != '-'):
        Counter += Minesweeper.arr_sam[r][c-1] # center left
        Free_box += 1

    if (r != 0 and Minesweeper.arr_sam[r-1][c] != '-'):
        Counter += Minesweeper.arr_sam[r-1][c] # center top
        Free_box += 1

    if (r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c] != '-'):
        Counter += Minesweeper.arr_sam[r+1][c] # center bottom
        Free_box += 1

    if (c != Minesweeper.N-1 and r != 0 and Minesweeper.arr_sam[r-1][c+1] != '-'):
        Counter += Minesweeper.arr_sam[r-1][c+1] # top right
        Free_box += 1

    if (c != 0 and r != 0 and Minesweeper.arr_sam[r-1][c-1] != '-'):
        Counter += Minesweeper.arr_sam[r-1][c-1] # top left
        Free_box += 1

    if (c != Minesweeper.N-1 and r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c+1] != '-'):
        Counter += Minesweeper.arr_sam[r+1][c+1] # bottom right
        Free_box += 1

    if (c != 0 and r != Minesweeper.N-1 and Minesweeper.arr_sam[r+1][c-1] != '-'):
        Counter += Minesweeper.arr_sam[r+1][c-1] # bottom left
        Free_box += 1
    if Counter == 0 and Free_box == 0:
        return 0

    prob = Minesweeper.find_probabelity(num)
    return (10 - prob) / 10
```

Figure 15

In this function we found the count of free boxes and the count of numbers inside of the boxes around the “num” (the box that we want to find the probability of it). If both were 0 it means that the “num” is covered by uncovered boxes.

Otherwise, it reverses the probability and put it between 0 and 1.

Furthermore, for finding the best chose we should find the edges and chose a box among those edges, edges boxes are those boxes which they have an uncovered box around them which you can see in (figure 16). By finding the edge boxes we avoid choosing randomly.

During the training and testing we clear our Q-table based on the edge boxes.

We find the edge boxes first then we find the reward for each of the boxes. So, we have some column which are the boxes with their probability and rewards then we delete unused boxes for giving the better score and deleting random blocks, in addition to that we delete the boxes with probability of 100 being mine or 0 rewards.

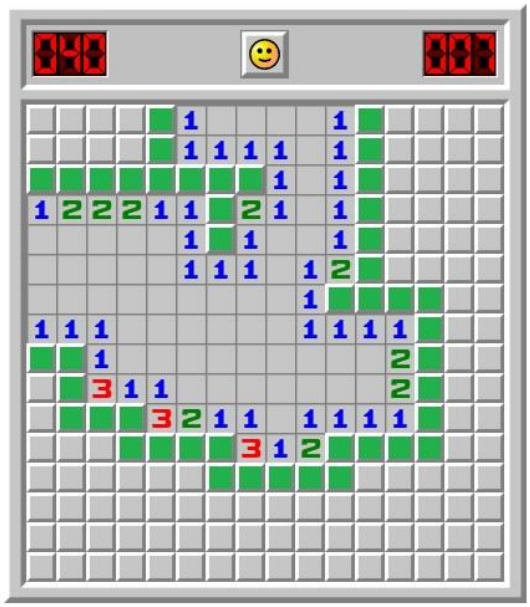

Figure 16

For running the algorithm, we should define the factors of the Q-learning and give the algorithm the number of episodes and the maximum number of acting in each episode. The exploration and other factors have been given to the algorithm as well (figure 17).

```
num_episodes = 10000
max_steps_per_episode = 100

learning_rate = 0.1
discount_rate = 0.99

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.1
exploration_decay_rate = 0.01

rewards_all_episodes = []
```

Figure 17

Now we run the algorithm for 10,000 time and we train our model. During the training we fill the Q-table based the given information in (figure 15). Then we wrote the code in (figure 18) for training our model and filling the Q-table.

As you can see, the first action is always randomly but after that all the action will chose based on Q-table or randomly again.

As you can see in (figure 18) after playing 10,000 times the games, the algorithm wins only 2826 times and based on (figure 19) after running the code, model chose 26444 times based on Q-table and 5728 times randomly (figure 19).

```
# Q-Learning algorithm
win_Counter, random_counter, first_acction_counter, Q_counter, CW = 0,0,0,0,0
win_list = []
for episode in range(num_episodes):
    arr = M.Create_minesweeper(My_N,My_K)
    clear_output(wait=True)
    print(episode)
    rewards_current_episode , state = 0,0
    done = False
    first_action = M.action_space_sample(-1)
    first_action = int(first_action / 100)
    for step in range(max_steps_per_episode):
        q_table_Templet = np.copy(q_table)
        q_table_Templet = Clear_Q_Table(q_table_Templet)
        if first_action != -1:
            action = first_action
            first_acction_counter += 1
        else:
            exploration_rate_threshold = random.uniform(0, 1)
            if exploration_rate_threshold > exploration_rate:
                action_Num = np.argmax(q_table_Templet[state,: ])
                action = int(action_Num / 100)
                Q_counter += 1
            else:
                random_counter += 1
                action = M.action_space_sample(0)
                action_Num = action
                action = int(action_Num / 100)
        M.run_Steps_Game(action)
        reward = Minesweeper.Rewards
        done = M.Done
        # Update Q-table for Q(s,a)
        if first_action == -1:
            q_table[state, action_Num] = q_table[state, action_Num] * (1 - learning_rate) + learning_rate * (reward + discount_rate * np.max(q_table[state, :]))
        else:
            first_action = -1
        rewards_current_episode += reward
        if done == True :
            CW = M.Check_win(My_N,My_K)
            break
    win_list.append(CW)
    if CW == 1:
        win_Counter+=1
    # Exploration rate decay
    exploration_rate = min_exploration_rate + (max_exploration_rate - min_exploration_rate) * np.exp(-exploration_decay_rate*episode)
    rewards_all_episodes.append(rewards_current_episode)
    print(win_Counter)
print(win_Counter*3)
```

```
9999
2826
```

Figure 18

```
print('decision based on Q-table : ' , Q_counter)
print('decision randomly : ', random_counter)
```

```
decision based on Q-table :  26444
decision randomly :  5728
```

Figure 19

During the training we record the winning and losing in each episode and in (figure 20) we plot it to have a better understanding about the model.

As you can see in this liner graph, at the beginning it wins only around 20 games per 1000 play but after training more the result became better, and the model win more than 100 games among 1000 games.

It means about 350 wins among 1000 plays.

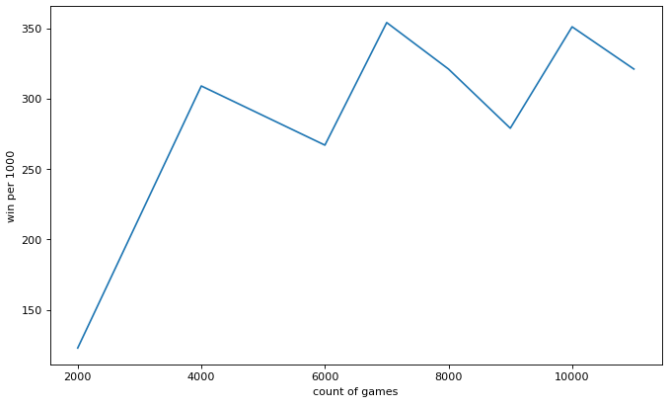

Figure 20

In (figure 21) you can see the reward that have been taken during the training process.

After training we try to play the game 10,000 time again but this time, we run the algorithm totally based on the Q-table.

As you can see the wining rate is as (figure 21). The algorithm wins between 380 and 460 when we use only Q-table for finding the best action.

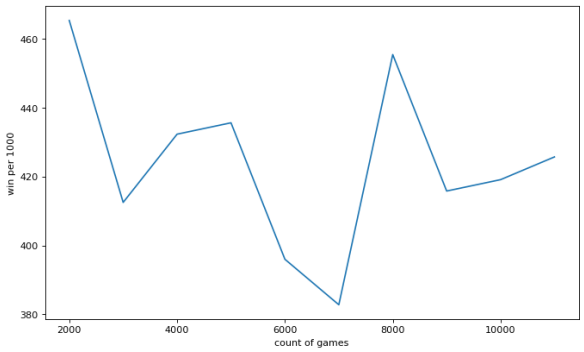

Figure 21

Now we changed the exploration_decay_rate from 0.01 to 0.0001. the reason for this change is that we want the algorithm act randomly more to fill the Q-table better.

After the change we run the model again and the results shows that this time, the algorithm act 9,060 times based on Q-table and 17,501 times randomly (figure 22).

It means that the model tries to learn more rather than acting based on previous knowledge.

```
print('decision based on Q-table : ' , Q_counter)
print('decision randomly : ', random_counter)
```

```
decision based on Q-table :  9060
decision randomly :  17501
```

Figure 22

However, this time the winning rate was as (picture 23). The result was not good enough because it acts randomly more than previous time.

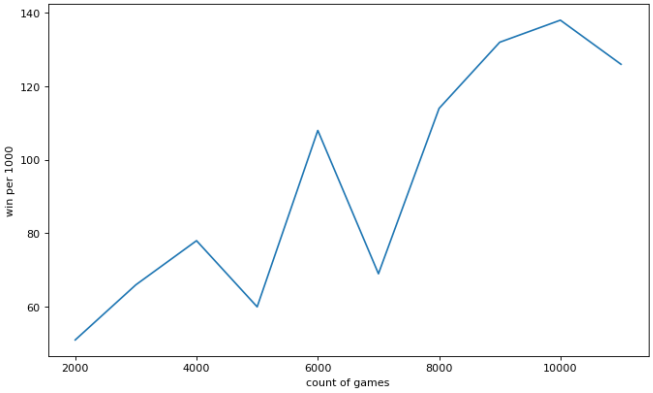

Figure 23

Then we try to run the process 10,000 times but this time it acts exactly based on the Q-table and the result is as (figure 24). In this method you can see it reached to 70% once and id was fluctuating between 20 and 60.

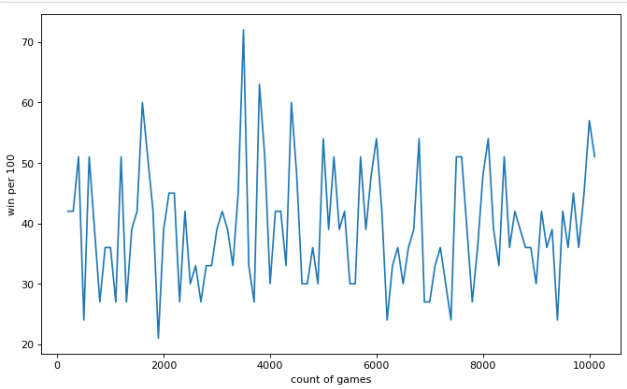

Figure 24

These models are good for this kind of games, but the problem is that most of the rewards are close to each other. For example, most of the rewards are between 80 to 100 and distinguishing between the rewards in Q-Learning formula make it hard to get the appropriate reward to the act and in long time, it makes it hard to detect the best action and following that the result will not be the best possible result.

Future work

As you can see, the result was not good enough in the second method and the reason is for un appropriate reward. Furthermore, the number of playing during the learning were only 10,000 times, need a better system, and it were not enough. In the future work we want to use a more efficient rewarding system and categorize the probabilities and use neural network by recording the states, actions, probability, and result.

Then we will use different neural networks models for predicting the results and actions. By using these methods, we can get a better result and predict the actions that have the most reward and probability of not being mine.

Conclusion

By growing AI technology, programmer and companies start using AI and reinforcement learning in their game products.

In this case study, a research has been done about the minesweeper game and we use two different methods for implementing this game by Q-table in reinforcement learning.

First method is that the game board stay stable, and the mines position don't change during the game and after a thousand time playing the game, the model act good and it win the games 95% times in testing process.

But the problem with this method is that if we change the game board, the result will be bad.

However, In the second method, we use different boards during the training and filling our Q-table. We use special calculation for giving the reward to each action, by finding the probability of being mine and converting it to reward.

In this method, we use two training, the difference between these two models is the parameters.

At the first one, we gave 0.01 to the exploration decay rate and the result was about 35 win among 100 game.

But in the second model, we changed the exploration decay rate to 0.0001. this change cause the acting the agent more randomly instead of acting based on Q-Learning. As we wanted to just fill the Q-table, we tried to act randomly and get reward as much as we can.

As you can see, the result for this model was around 60 wins among 100 games and it shows that the AI win about 50% as an average.

Limitation

The hardest part of this research was implementing the rewarding system. Although it is easy for a human to win this game, Reinforcement Learning needs to get rewards to understand which action is better. As in Minesweeper game we have a lot of probabilities, we should find a good way to calculate those probabilities and following the rewarding calculation. For doing that, we use different formula and none of them were good enough as rewarding system.

After a lot of experiment, we find a good rewarding system. However, even after all those tries, we did for finding this formula, it did not give us a perfect result.

This is the reason that we should have more practice on this research in the future works.

References

1. Kaye, R., 2000. Minesweeper is NP-complete. Mathematical Intelligencer, 22(2), pp.9-15.
2. Koch, C., 2016. How the computer beat the go player. Scientific American Mind, 27(4), pp.20-23.
3. Hsu, F.H., 2002. Behind Deep Blue: Building the computer that defeated the world chess champion. Princeton University Press.
4. Sajjad, M.H., 2022. Neural Network Learner for Minesweeper. arXiv preprint arXiv:2212.10446.
5. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.
6. Becerra, D.J., 2015. Algorithmic approaches to playing minesweeper (Doctoral dissertation).
7. Studholme, C., 2000. Minesweeper as a constraint satisfaction problem. Unpublished project report.
8. Dechter, R. and Frost, D., 2002. Backjump-based backtracking for constraint satisfaction problems. Artificial Intelligence, 136(2), pp.147-188.
9. Nakov, P. and Wei, Z., 2003. Minesweeper,# Minesweeper. Unpublished Manuscript, Available at: http://www. minesweeper. info/articles/Minesweeper (Nakov, Wei). pdf.
10. Castillo, L.P. and Wrobel, S., 2003, August. Learning minesweeper with multirelational learning. In INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (Vol. 18, pp. 533-540). LAWRENCE ERLBAUM ASSOCIATES LTD.
11. Sebag, M. and Teytaud, O., 2012. Combining myopic optimization and tree search: Application to minesweeper. In LION6, Learning and Intelligent Optimization (Vol. 7219, pp. 222-236). Sringer Verlag.
12. Legendre, M., Hollard, K., Buffet, O. and Dutech, A., 2012. MineSweeper: Where to probe? (Doctoral dissertation, INRIA).
13. Buffet, O., Lee, C.S., Lin, W.T. and Teytuad, O., 2013. Optimistic heuristics for Minesweeper. In Advances in Intelligent Systems and Applications-Volume 1 (pp. 199-207). Springer, Berlin, Heidelberg.
14. Gardea, L., Koontz, G. and Silva, R., 2015. Training a minesweeper solver. An Autumn CS, 229.
15. Christopher, J.C.H., 1992. Watkins and peter dayan. Q-Learning. Machine Learning, 8(3), pp.279-292.
16. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. nature, 518(7540), pp.529-533.