COMP 4106 Honours Project Report

Q Networks Solve Driving Game

Tianqiang Liu (100975095)

Tianqiangliu@cmail.carleton.ca

Course Code: COMP4106

School of Computer Science

Carleton University

2020/4/10

TABLE OF CONTENTS

INTRODUCTION

The maze problems are a classical question for AI, Machine Learning, and even Math areas. It also can be considered as efficient test tool for different algorisms. It required AI move one item from on certain location to other specific location. In addition, there will be barriers on the ground, so AI need to find the way to go around those barriers. What is more, the maze problems can convert to driving problems. The free blocks can be considered as road, while barriers are be considered as item on the road such as other vehicles, wall, or trees. In driving problems, the AI need to find the right path to reach the goal. Also, AI need to avoid hits on item on the road. In the driving game, the AI are required to move vehicle to the destination as soon as possible without hitting any item. Every hitting will reduce scores on final score. Ideally, after learning the game, the AI should be able to avoid the damage and driving to the destination. The driving problems are not the focusing of the project. Learning and implementing Deep Q Networks are the most important for this project.

Deep Q Network was first reinforcement learning proposed by DeepMind in 2013. Deep Q Network combines Deep Neural Network and Reinforcement Learning to avoid the weakness of Q-learning. The Q-learning is a very powerful algorithm. However, it does not have ability to provide action with unseen states. So DQN use a neural network to estimate the Q-value. The neutral network will get current input and output the corresponding Q-value for each of action.

METHODOLOGY

1. Q-LEARNING

Q-Learning is a reinforcement learning algorithm that to find the best action for current state. Off-policy is one of the important attributes that Q-Learning has, as Q-Learning taking random actions so that it is not necessary to have a policy.

To implement Q-Learning to the program, you should, firstly, create a q-table in a two-dimension data structure. The q-table store states of events and actions of events so that the program can select best action based on the table, like Figure 1.

```
public class QLearning {
    private final double alpha = 0.1;
    private final double gamma = 0.9;
    private final int reward = 100;
    private final int penalty = -10;
    private int[][] rewardsTable;

    //Q-table
    private double[][] qLearningTable;

    private ArrayList<String> board;

    public void initial(ArrayList<String> board, int wide, int boardSize) {
        rewardsTable = new int[boardSize][boardSize];

        //Initial Q-table
        qLearningTable = new double[boardSize][boardSize];

        this.board = board;
```

Figure 1. Create and Innitial Q-Table

After creating q-table, there are two methods that we can use to fill q-table with useful data. They are exploiting and exploring. Exploiting method is use all information,

which the program currently has, to view all possible actions for given state. For example, the program can keep finding the maximum value and executing action to find out the best path. On the other hand, exploring method is randomly selecting actions and find out the result. This method allow program to discover more new states so that the program can find the best path. This project is using exploring method, like Figure 2. All actions that program takes are produced randomly.

```
public void qLearning() {
    Random random = new Random();

    for (int i = 0; i < 1000; i++) {

        int position = 0;

        while (!isFinished(position)) {
            int[] actions = possibleActionsFromState(position);


            int index = random.nextInt(actions.length);
            int nextState = actions[index];

            // Q(state,action)= Q(state,action) + alpha * (R(state,action) + gamma * Max(next state, all actions) - Q(state,action))
            double q = qLearningTable[position][nextState];
            double maxQ_Value = maxQ(nextState);
            int reward = rewardsTable[position][nextState];

            double value = q + alpha * (reward + gamma * maxQ_Value - q);
            qLearningTable[position][nextState] = value;

            position = nextState;
        }
    }
}
```

Figure 2. Q-Table Learning and Update

Furthermore, the program is required to update q-table after receiving rewards or penalty. The update rule for this project is `Q[state, action] = Q[state, action] + lr * (reward + gamma * np.max(Q[new_state, :]) − Q[state, action])`[1]. The lr is referred learning rate, which defined how much the program accepts new value, or how fast the program learns. The gamma is a constant parameter to balance reward.

What is more, after tests, I use 0.9 as gamma, as it balances the reward, produces accurate result, and spend less time.

2. TSETLIN

This project additionally was implemented Tsetlin automaton. The Tsetlin implement is similar as assignment 3 of Comp4106 2020Winter. However, since every states need to have different actions, Arraylist is used to store states for each position.

```
public Integer goWhere(int position) {

    if(states.get(position)<=nInteger) {
        return 0;
    }else if(states.get(position)<=2*nInteger) {
        return 1;
    }else {
        System.out.println(states);
    }
    return -1;

}
```

Figure 3. Tsetlin Action Decision

```
    public void rewardSystem(int reward, ArrayList<Integer> arrayList) {
        for (int i = 0; i < arrayList.size(); i++) {
            int position = arrayList.get(i);
            int state = states.get(position);
            if(reward == 0) {
                if(state!=1&&state!=nInteger+1) {

                    state=state-1;
                }
            }else {
                if(state!=nInteger&&state!=(2*nInteger)
                        ) {

                    state=state+1;
                }
                else if(state==nInteger) {
                    state = 2*nInteger;
                }
                else if(state==2*nInteger) {
                    state = nInteger;
                }
            }
//          System.out.println("position:"+position+"\nstate:"+state);
            states.set(position, state);
        }

    }
```

Figure 4. Tsetlin Reward System

RESULT

The program has one thousand times experiment on Q-Learning automaton, which will play the game for one thousand times. The result is 100% correct if there is a correct path. The Tsetlin automaton, on the other hand, has 99.99% accurate if there is a correct path. Hence, the Q-Learning is more accurate than Tsetlin automaton in this project. However, the time cost for Q-learning is higher that Tsetlin automation.

There are many enhancements that can be implemented to this project. For example, Artificial Neural Network can be used with Q-Learning so that programs do not have to spend much memory to store q-table. Moreover, as Q-learning does not store unseen data, Artificial Neural Network can help program to handle unseen data by estimating it to a seen q-value.

REFERENCE

[1] A. Violante, *Simple Reinforcement Learning: Q-learning*, Mar 18, 2019. Accessed on April 10, 2020. [Online]. Available: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

[2] J.Oommen, *A Summary of Some Topics: Learning Automata*, Mar 14, 2020. Accessed on April 10, 2020. [Online]. Available: http://people.scs.carleton.ca/~oommen/Courses/COMP4106Winter20/BriefSummaryLA.pdf

[3] V. Mnih, *Playing Atari with Deep Reinforcement Learning*, Dec 19, 2013. Accessed on April 10, 2020. [Online]. Available: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

APPENDIX

Running Instruction:

a) Cd ../ FinalProject/target
b) java -jar CarDrive.jar 5 t (t for Tsetlin automaton, 5 for board width)
/ java -jar CarDrive.jar 5 q (for Q-Learning automaton, 5 for board width)