Report of assignment B

by Ruining Zhang

Abstract: Reinforcement learning is a widespread used algorithm for machine learning, which provides a method for the learner to learn what to do without any instructions, the object is not told which actions should be taken but will receive the positive reward when choose the correct one(negative reward otherwise). Thus the object will keep trying and discover which actions yield the best reward. In this report, a issue of robot exploration is going to be implemented and some improved methods will be discussed as well (such as eligibility trace) according to the trials' outcome. All questions were answered in below.

1. Introduction

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem. The interesting thing here is we not only care about the immediate reward but also the future reward, since the learner is more likely to reach the global optimization, otherwise it may be a local optimization point. Return to the assignment, a robot was set in a random start position of a square grid world (7×7) with four possible actions, North, South, East and West, and it asked to explore the space to find out the terminal location with the shortest path. In order to be there, the robot is free to move in the map and reward (defined as charger in this case) is only given when it reaches the segment. In other words, the robot will not stop until receive the reward or reach the upper limitation of the total steps.

2. SARSA algorithm on ANN

The State Action Reward State Action (SARSA) is an another "State-action" algorithm based on Temporal-Difference (TD) learning, which aims to learn a Markov decision process policy. The system exploring according to the policy and updating the value of each step. Figure 1 illustrates the process of SARSA, where s_t is the current state, a_t is the action of state s_t.

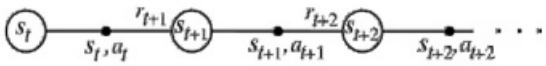


Figure.1. The exploration process of robot(Sutton et al).

The object take action a_t at state s_t based on the policy and observe reward and next state s_{t+1}. Moving to next state and updating $Q(s,a)$, then repeat the above procedure till the end. The update rule is here:

$$\Delta Q(s, a) = \eta[\text{-} Q(s, a) + Q(s', a')]$$

where $Q(s,a), Q(s',a')$ are current reward and future reward respectively. Furthermore, SARSA on artificial neuron network (ANN) is the main part that will be discussed. The framework of that is shown in figure 2.

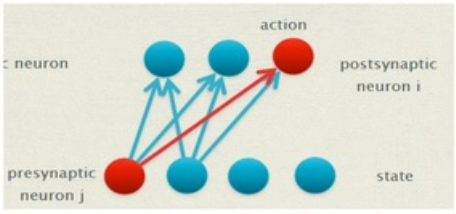


Figure.2. The network of SARSA(https://staffwww.dcs.shef.ac.uk/people/E.Vasilaki/campus_only/COM3240/Lecture8_RL2.pdf)

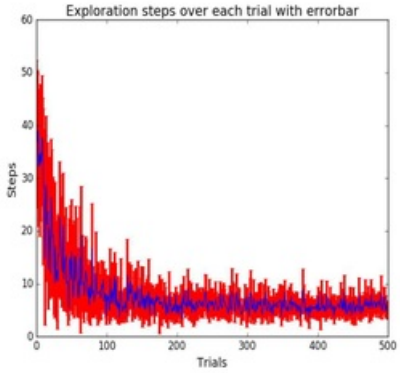
The state presents the presynaptic neuron and the action refers to the postsynaptic neuron. Meanwhile, the update rule is change as well.

$$\delta = [\, r_{t+1} - (Q(i, j) - \gamma Q(i', j'))] \quad (2)$$

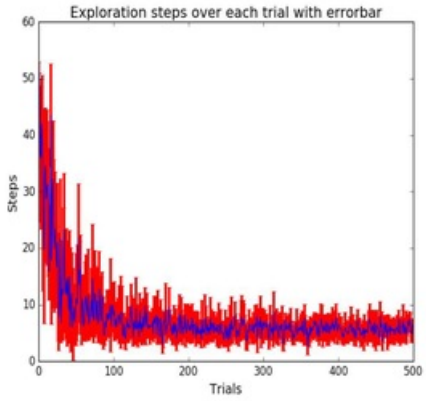
$$\Delta w = \eta \delta x_i x_j \quad (3)$$

3. Questions answering

3.1 The algorithm was compiled according to the above instructions, which shown in appendix code part 1.The procedure that the robot's movement from start to the end is called one run of the algorithm. Considering each ten times of running as one group and executing it twice, and two different average learning curves are displayed as figure 3.



(a)



(b)

Figure.3.(a)(b) The average learning curve for two trials with error bars.

With respect to the figure3, we can see these two curves have a similar shape but not a entire one. There are two random variable parameters in the initialization, the weights matrix and the start location of the robot. It is solely impossible to give same value to them for every trial, and this is the reason why that two distinct figures are produced.

3.2 Eligibility trace (ET) is implemented to propagate information faster comparing with the normal SARSA algorithm. The key structure of this method is updating all Q-values of the state and action pair and was described in below equations which are similar in SARSA. $e_{ij} = \gamma \lambda e_{ij} + x_i x_j$ (3) where γ, λ are discounting factors, and $x_i x_j$ corresponds to output and input, $x_i, x_j = 1$ for active neuron, else 0. The relevant code is in the appendix,original code, part 3. Figures with eligibility trace and without are illustrated in figure 4 in the following.

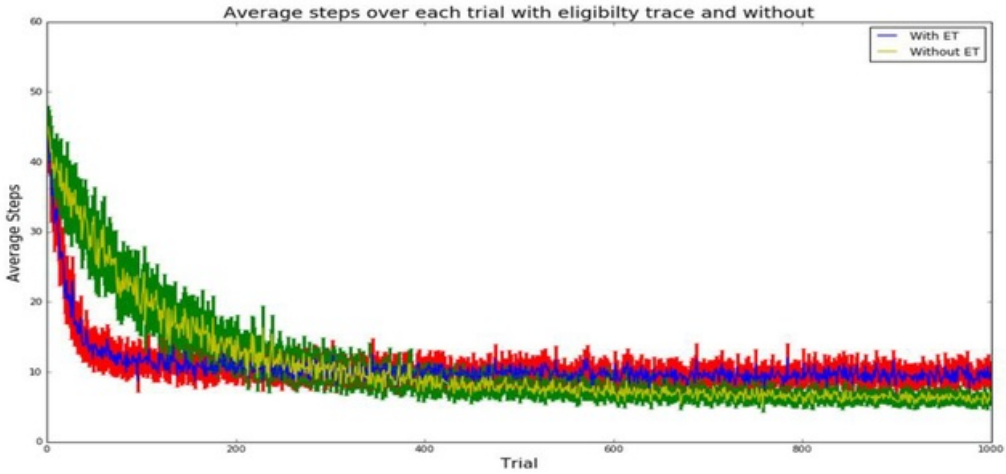


Figure.4. The average steps with eligibility and without eligibility

From the above figure, on the one hand, the average steps of eligibility trace method converges quickly than the normal TD, that is, the robot learns more fast in this case. On the other hand, ET takes more steps contrasting to DT. In sum, ET algorithm converges to the minimum steps but the optimal solution is not good as the DT algorithm.

3.3 The control variable method is employed here to find the best epsilon(ε), learning rate (α) and discounting factor(γ). Only one of the above three parameter was set to be a variable each time and plot them in figure 5.

It can be observed that alpha and gamma are fall down with steps but epsilon slope up steeply. With respect to the figure, the lowest steps corresponding to the optimal value, and there are 0.01, 0.99, 0.99 which refer to epsilon, alpha, and gamma respectively.

After that, we plot the previous TD figure in Question 2 (epsilon,gamma,alpha =0.5) and the updated TD figure with optimal parameters, which shown in figure 6. The optimized figure is better not only from the stable and learning speed. High learning rate drive the robot learning fast, and gamma means how much the robot care about the future reward in every states, if it is high, the robot will have a long-term version and it is better. As for epsilon, high value refers to low probability to be greedy. However, it is more likely to take a better action when the robot be greedy, thus lower value of epsilon is recommended.

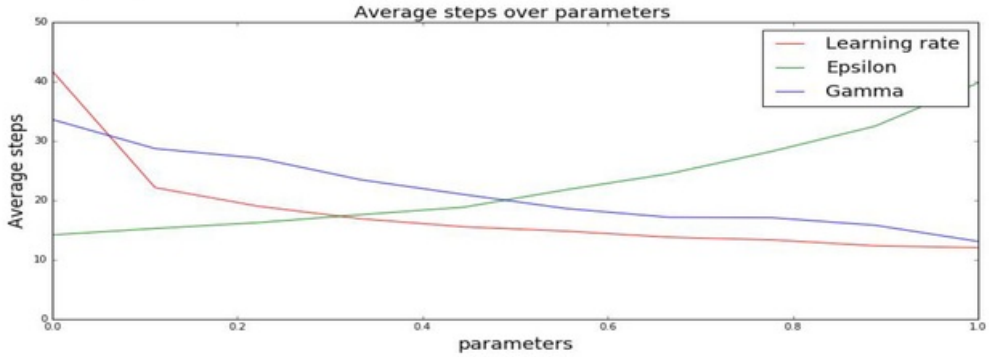


Figure.5. Three parameters over steps

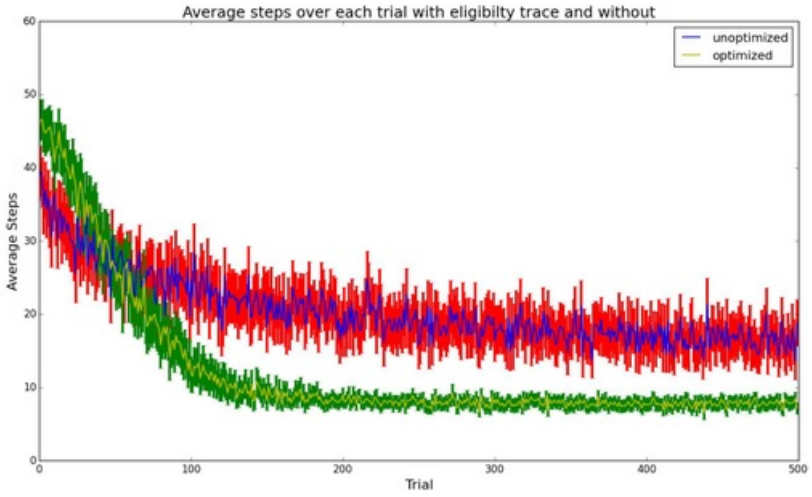
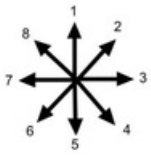


Figure.6. Non-optimized and optimized figures in TD

3.4 The 8 actions are displayed as the figure 7 and the corresponding program is in the forth parts of original code of appendix.

Figure.7. Arrows of eight actions

Figure 8 draws two learning cure, blue-red one and yellow-green one refer to four actions and eight actions respectively.



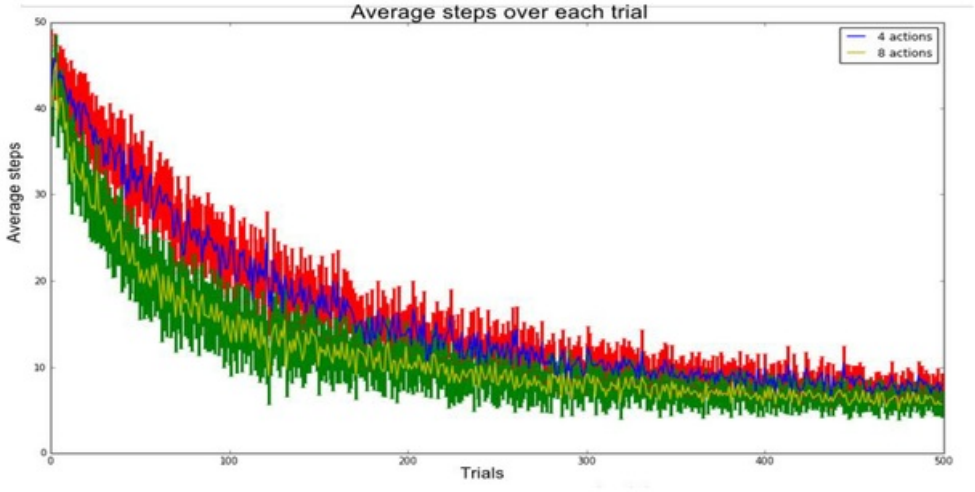


Figure.8. Average steps over trials under four and eight

Increasing the number of actions will not only improve the learning efficiency of the robot but also decrease the average steps needed in the exploration. Since with increased number of states, the robot is able to move approximately along the line that connect the start and end locations directly, and it could save time and develop accuracy.

3.5 As we all know, the world is a continuous states set. However, in order to simulate the reinforcement learning system, the continuous variables are often been discretized. The issue following as well, if we propose a bad assumption of discretization of state-action pairs, the robot may not learn from the optimal policy and meet hidden-states.

One possible solution is raise the number of states and actions. According to the solution in Question 4, the increased number of actions improved the trial's performance apparently. There is another example:
A robot move in a 1-dimensional space shown in figure 9.

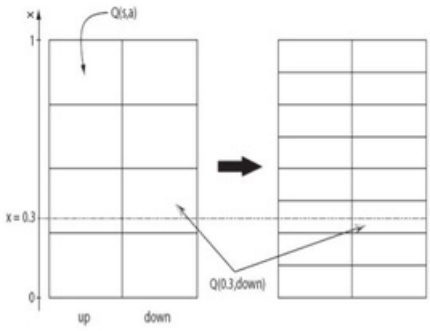
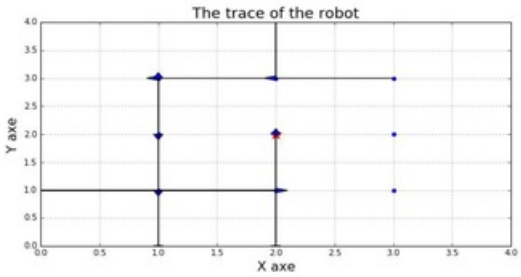


Figure.9. The continuous 1-dimensional spaces

Discretize these two spaces into 4 and 8 steps and the speed of the robot has to be fixed regardless the discretization scheme. It is easy to compute that the robot move fast in 8 discretization scheme. In sum, increasing the number of states and actions will improve the performance.

In contrast, it is impossible to store all information and the running time of a large number of actions and states grows exponentially in the trial. Some new algorithm were presented and going to solve it, such as the Hedger Prediction Algorithm.

3.6 Eligibility trace is aims to store the trajectory information of the robot. In this case, we modified the update rule of ET and the new one is like: $e_{ij} = e_{ij} + 1$ for the specific action and state, then update eligibility trace by $e_{ij} = \gamma\lambda e_{ij}$ for all actions and states. Once this rule is implemented, at each recycle of the main part of code, only actions which be chose from the states have the corresponding value in the eligibility trace matrix. It means the states that pass though by the robot and their actions have nonzero values and rest of the parameters of the state-action is none. Thus we can plot the path of the robot based on this theory, which shown in figure 10.



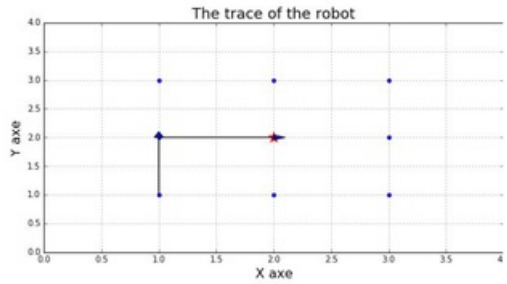


Figure.10. The trajectory of the first trial (upper) and 500th (lower).

In order to improve the efficiency of the program running, the grid world is decreased to 3 by 3 points. In the figure, the arrow illustrates the direction of the robot movement and the red star is the terminal state. Obviously, the robot wastes many steps to find the end point. However, the robot learning with raised trial times and reach the end location more straightly.

Appendix

References

1. Sutton, R.S. and Barto, A.G., 1998. Reinforcement learning: An introduction. MIT press.
2. Assignment B notebook: https://staffwww.dcs.shef.ac.uk/people/E.Vasilaki/campus_only/COM3240/assignmentB.pdf
3. Smart, W.D. and Kaelbling, L.P., 2000, June. Practical reinforcement learning in continuous spaces. In *ICML* (pp. 903-910).

Original code

```
#####   Part 1 Question1 #######
## TODO update Qvalues. Only if is not the first step
if step > 1:
# Update weights
      dw = learning_rate * (r_old - Q_old + gamma * Q[action]) * output_old.dot(input_old.T)
      weights += dw

#store variables for sarsa computation in the next step
      output = np.zeros((N_actions,1))
      output[action] = 1

      #update variables
      output_old = output
      input_old = input_vector
      Q_old = Q[action]
      r_old = 0

      state[0] = state_new[0]
      state[1] = state_new[1]
      s_index = s_index_new

      ## TODO: check if state is terminal and update the weights consequently
      if s_index == s_end:
            #pass     #pass means doing nothing

      # Update weights for the terminal state
            dw = learning_rate * (R - Q_old) * output_old.dot(input_old.T)
            weights += dw
```

```
#####    Part 2 Question2 #######
# Initialse eligibilty matrix for the first step
if step ==1:
    e[action,s_index] = e[action,s_index] + 1

    ## TODO update Qvalues. Only if is not the first step
if step > 1:
    # Update weights
    dw = learning_rate * (r_old - Q_old + gamma * Q[action])*e
    weights += dw
    e = gamma*lamb*e + output_old.dot(input_old.T)
    e[action,s_index]+=1
```

```
#####    Part 3 Question3 #######
##plot average learning curve##
import matplotlib.pyplot as plt
#%matplotlib inline
import numpy as np
n_trials = 500
eps = 0.01
gamma = 0.99                    #curve full down very quickly when applying high gamma
learning_rate = 0.99
repetitions = 10

total_op = np.zeros((repetitions,n_trials))
for i in range(repetitions):
    total_op[i,:] =    homing_nn_TD(n_trials,learning_rate,eps,gamma)

means_op = np.mean(total_op,axis=0)
errors_op = 2 * np.std(total_op, axis = 0) / np.sqrt(repetitions)

fig = plt.figure(figsize=(16,9))
ax = fig.add_subplot(111)
# add labels
ax.errorbar(np.arange(n_trials),means, errors, 0, fmt='-',ecolor='r', elinewidth = 3)
ln1=ax.plot(np.arange(n_trials),means,'b',linewidth = 1.5, label='unoptimized')
ax.errorbar(np.arange(n_trials),means_op, errors_op,fmt='-',ecolor='g',elinewidth = 3)
ln2=ax.plot(np.arange(n_trials),means_op,'y',linewidth = 1.5, label='optimized')
```

```
#####   Part 4 Question4 #######
N_actions = 8       #number of possible actions in each state
#number of cell shifted in vertical as a function of the action
action_row_change=np.array([-1,-1,0,+1,+1,+1,0,-1])
action_col_change =np.array([0,+1,+1,+1,0,-1,-1,-1])
```

```
#####   Part 5 Question6 #######
dy = np.array([+1,0,-1,0])          #action matrix
dx = np.array([0,+1,0,-1])

fig, ax = plt.subplots(figsize=(10,5))
e_index = homing_nn_q6(n_trials,learning_rate,eps,gamma,lamb)

#find the nonzero parameters in Eligibility matrix
for i in range(N):
      for j in range(M):
            e_ravel = np.array([i,j])
           #transform the current state to ravel model
            e_index_ravel =np.ravel_multi_index(e_ravel,dims=(N,M),order='F')
             #search the nonzero one in k_th column of the new ravel matrix
            for k in range(4):
                  if e_index[k,e_index_ravel] !=0:
                        #define the start and length of the arrow
                        ax.arrow(j+1,N-i, dx[k], dy[k],head_width=0.08, head_length=0.1)
```

Report of assignment B

GRADEMARK REPORT

FINAL GRADE

67 /100

GENERAL COMMENTS

Instructor

"Presentation Style: good. Some margins are too wide.

RL: description: missing R in the formula update and Q(s,a) and Q(s',a') are not (current/future) reward. they are expected reward for the specified state-action pair.
needs definition of notation (x_i, x_j , etc)

etRL: description lacking depth but good formalism
figure and discussion are good

optimisation missing the single plots. the cumulative figure is showing a control variable method instead of a full search

8 act: this increases the number of actions not the number of states (maybe it is just a typo)

discretisation: description is confusing and figure caption is wrong. proposed solution (increase states) is not a solution. Hedger Pred Algo is missing in the references

No direction preference plot. Eligibility traces are shown instead."

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9