Adaptive Intelligence Assignment 2: Reinforcement Learning with simple neuron network

Wenjie Zhang
The University of Sheffield,Sheffield,United Kingdom

***Abstract*—This report focuses on the implementation of a robot performing 'homming' task to return a particular reward location in 2D environment. The approach has been used in this assignment is SARSA(λ) algorithm with simple neuron network/perceptron. The policy used in SARSA algorithm is ϵ-greedy policy. Also, an eligibility trace is implemented to record the trajectory, then it is used when updating weight.**

***Index Terms*—SARSA(λ) algorithm, Perceptron, Reinforcement Learning, Eligibility trace, ϵ-greedy policy**

I. Question 1

There are three basic paradigms in area of machine learning – reinforcement learning, supervised learning and unsupervised learning. Their relationship is shown as Fig 1. Reinforcement learning is a biological inspired type of learning concerned with how intelligent agent ought to take actions in an environment in order to maximize the notion of cumulative reward. In another words, reinforcement learning trains agent so that the agent is able to predict the value of actions at given state, hence maximize the total reward. The tasks being considered in this assignment is how a random placed robot reaches a particular reward location (goal) with minimum steps in a 2D environment. (note: there are no explicit landmarks, the robot has some internal representation of its own position but no explicit memory of the reward location)

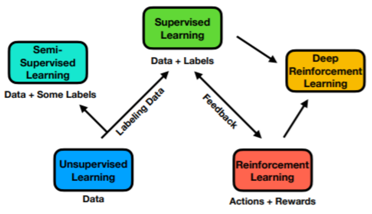


Fig. 1. Relationship.

There are two kind of update rules — one is for delta Q as shown in Fig 4 and another one is for delta W as shown in Fig 3. W means weight, Q means Q value at state s take action a. In this assignment, update rule we used is delta W, since we are using reinforcement learning with artificial neural network(perceptron). As normal artificial neural network, the error E equals square of 1/2 (reward – expected reward). dE/dW = - (reward – expected reward) * dQ/dW. Since Q is our output neuron which is equal to weight w times input x, dE/dW = learning rate * (reward – expected reward) * input x. If we define a vector containing all network outputs y where only the neuron of the selected action is active, then the delta W becomes Fig 3. It belongs to Hebbian family, to be more specific, it is Hebbian & reward signal. More details about how we implement the algorithm in this assignment is described as following.

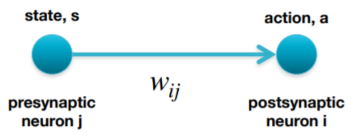


Fig. 2. Formula.

$$\Delta \mathbf{w} = \eta \left(r - Q(s,a) \right) \mathbf{y}\mathbf{x}^T$$

$$\Delta \mathbf{w} = \eta \left(\text{reward} - [\text{expected reward}] \right) \mathbf{y}\mathbf{x}^T$$

Fig. 3. Formula.

$$\Delta Q(s,a) = \eta \left[r - \left(Q(s,a) - \gamma Q(s',a') \right) \right]$$

Fig. 4. Formula.

Description of the algorithm: The algorithm initializes the task with some basic parameters. It sets a 10*10 2D environment and allows 4 actions top, down, left, right. The goal always set to (1,1) and use np.ravel-multi-index function to 'flatten' the coordinate into vector form to input to neural network. The robot is randomly placed in the environment. For this algorithm, we implement reinforcement learning with artificial neural network (perceptron). We treat the state as presynaptic neuron/input layer and actions as postsynaptic neuron/output layer. Q-value is computed by weight * input vector, then according to policy used to choose action and move robot. And update our weight matrix (n-actions,n-state) according to dE/dW = learning rate * (reward – Q-old-vlaue + gamma * Q(s,a)) * yx . Once the robot reaches the goal, the reward will become one, this process will be repeated n-trials. Since we are using SARSA which is on-policy, we can choose different policy when updating, such as soft-max, Greedy, epsilon-Greedy and optimistic Greedy.

II. Question 2

In general, exploration is important because it could give more information about the environment to the agent. If the agent has learned all the information about the environment, the agent is able to find the best strategy to maximize total reward by even just simulating brute-force, let alone many other smart approaches. However, exploitation is also important, without enough exploitation, the agent cannot complete our reward optimization task. Thus, we need a way to balance exploration and exploitation. Comparing the policies we mentioned above, soft-max policy using exponential form which is easy to increase the difference between probability of exploration and probability of exploitation. Optimistic greedy start with high Q-value then Greedy policy, which ensure we explore enough environment but the learning curve will convergence slowly. (means more run time). Greedy policy always choose action with greatest reward, which means not enough exploration. Hence, we may not be able to find best solution.

That is why we implement epsilon greedy policy to aids explorations. Greedy and epsilon-greedy policy is shown as Fig 7. For epsilon-Greedy policy, the epsilon value means the probability of random action been taken at each state, which means exploration. There are 1-epsilon probability that the agent chooses Greedy policy. The epsilon parameter in program controls it, if we set epsilon to 0.3 as shown in Fig 6 and 0 as shown in Fig 5. We can easily see 0 epsilon shows quick convergence/learning speed. When epsilon = 0 the learning curve shows convergence at 600 but for epsilon = 0.3 it shows convergence at 900. Also, the fluctuations of larger epsilon is greater than 0 epsilon. Both aspects shows robot explored more states when larger epsilon.

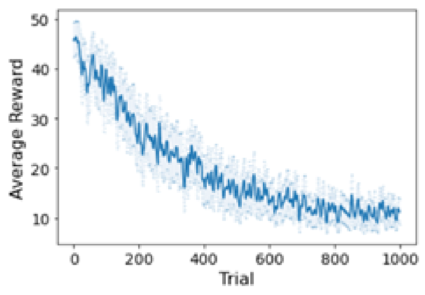


Fig. 5. Epsilon = 0

III. Question 3

We compute the number of extra steps over the optimal path per trial averaged over the run. And run 10 times, averaged again over 10 runs in all experiment. The error bar shows in grey.

First experiment, learning rate and exploration factor epsilon is controlled. Only vary discount factor from 0.5 – 0.9. The diagrams is shown in Fig 8. The result shows the greater

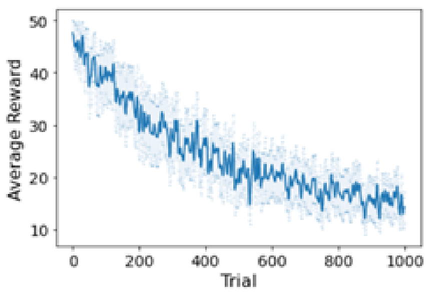
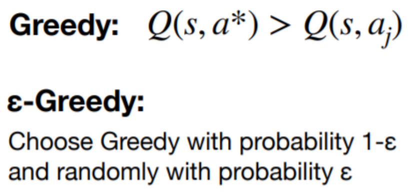


Fig. 6. Epsilon = 0.3



Greedy: $Q(s, a^*) > Q(s, a_j)$

ε-Greedy:

Choose Greedy with probability 1-ε
and randomly with probability ε

Fig. 7. Epsilon = 0

discount factor the fewer extra steps over the optimal path. This result can be explained by sometimes local optimal reward action is not globally optimal reward action. For example, action 1 reward is 0.1 with future reward 0.9 and action 2 reward is 0.5 with future reward 0. The globally optimal reward is action 1 but local optimal reward is action 2. Thus, when the robot considers more about future reward it will spend less extra step over the optimal path.

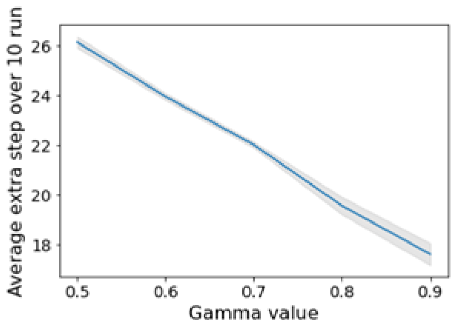


Fig. 8. Gamma value

Second experiment, learning rate and discount factor gamma is controlled. Only vary exploration factor factor from 0.1 – 0.5. The diagrams is shown in Fig 9. The result shows the greater exploration factor epsilon the greater extra steps over the optimal path. The reason for this is greater exploration factor epsilon means greater chance to choose explore action. This is a trade off between exploration and exploitation, the robot gets more information about environment by exploration. Correspondingly, the robot will spend more extra steps over the optimal path. However, this exploration could ensure that the current optimal path is globally optimal path, in another words, ensure maximum rewards.

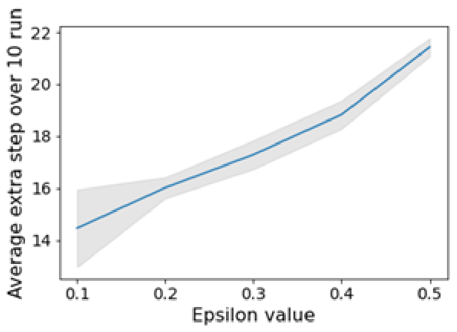


Fig. 9. Epsilon value

The last experiment, exploration factor epsilon and discount factor gamma is controlled. Only vary learning rate from 0.1 – 0.5. The diagrams is shown in Fig 10. The result shows the greater learning rate the fewer extra steps over the optimal path. The reason for this is the higher learning rate the robot learns faster to find optimal path. However, there is an obvious disadvantage that the optimal path that has been found may not be the globally optimal path. This means the robot may get stuck in local maximum when high learning rate.

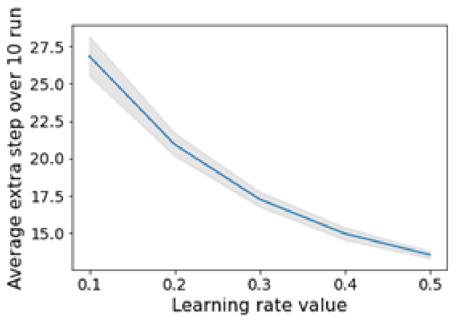


Fig. 10. Learning rate value

According to the discussion above, the final learning rate we choose 0.3, discount factor gamma we choose 0.9, exploration factor epsilon we choose 0.1.

IV. Question 4

When we are using SARSA algorithm, there is a problem — we only considered future reward but did not consider the past actions that lead the robot to the goal. In order to consider past actions, SARSA(λ) algorithm has been created. To introducing SARSA(λ), basically it is same as SARSA. Except it uses an eligibility trace to record the trajectory, with an exponential decrease for older actions, which can be used when updating the Q-values or weights. (in the code, we use it when updating weight) λ controls the rate of decay for the trace, the result of different λ value is shown in Fig 11. The gragh shows decreasing in extra step over the optimal path as the λ value increasing. However, the past actions may be unreasonable updated when using large λ value, which means the optimal path found may not be the globally optimal path. In another words, when lambda gets large then state-actions that didn't directly lead to the reward may be being rewarded despite not really contributing. Considering this reason, we choose to set λ =0.2.

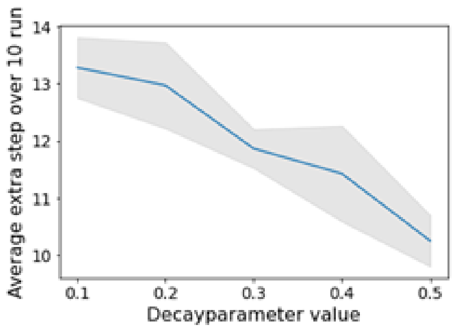


Fig. 11. Lambda value

Comparing SARSA(λ) algorithm Fig 11 and results in question 2 Fig 5, we can easily see that the learning curve of SARSA(λ) algorithm λ=0.2 convergent earlier than results in question 2. As SARSA(λ) algorithm considered both past action and future rewards, the learning speed of SARSA(λ) algorithm is faster than SARSA algorithm, hence convergent quicker.

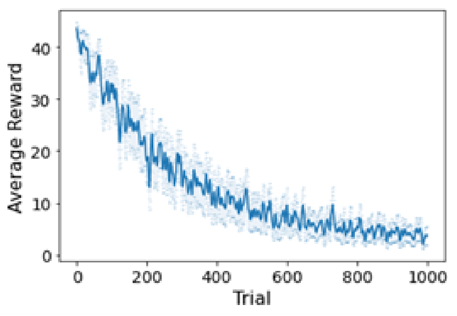


Fig. 12. Learning curve

V. Question 5

In order to investigate the preferred direction stored in the weights, we used matplotlib.axes.Axes.quiver function to show the preferred direction. As the Fig 13,14,15 shows, we can see that as the training number increase the preferred direction directly point to our goal. However, there are some strong arrow at top right. My guess is that since we've add an eligibility trace, for start at top right, it always been rewarded among multiple path. So the preferred direction stored in the weights is strong and little bit off-direction. For the edges in the graph, i think it is lack of training. If we continue to train more, it will directly point to our goal.

VI. Question 6

For bigger space need to be explored, and each edge of the square is composed by N=1000 parts. There are many difficulties, such as hard to design a suitable reward function.

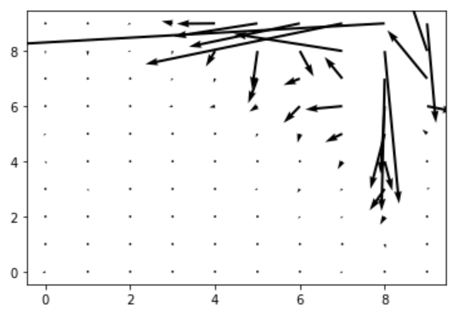


Fig. 13. preferred direction stored in the weights

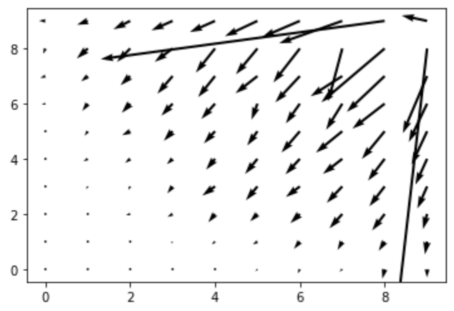


Fig. 14. preferred direction stored in the weights

AI needs to complete a relatively long action sequence t get the final reward (that is too sparse), AI cannot be wel trained. Since eligibility trace is an exponential decrease fo old actions, it is hard to reward or punish the earliest actions. Also, in action sequence (a1, a2, ... aN) there are some good actions and bad actions. We don't know which action is good and which action is bad, the reward or punishment is given to all actions in the action sequence with an exponential decrease. We cannot separately give reward or punishment to the actions along the action sequence. Also, SARSA algorithm only considered one more step as future reward, but the environment is big. This may cause the action robot chosen is not the global optimal action. In addition, there is a problem if we use reinforcement learning with deep neuron network. For example, action sequence ABCDE has a big reward, but action sequency ABC has a negative reward whilst ADE has a positive reward. Due to the gradient descendent, it is hard for the robot to explore in direction of action sequence ABCDE. Thus, the bigger reward can never be reached, which means the optimal solution the robot found may not be the globally optimal solution.

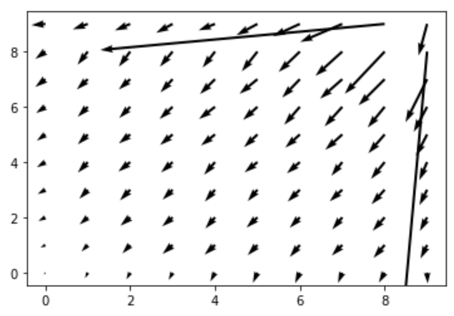


Fig. 15. preferred direction stored in the weights

There are some possible solutions to this problem. One is to cut the large task, such as reach the goal, into small tasks, such as reach some region. This solution will be like a reinforcement learning inside a reinforcement learning, the outer reinforcement learning is responsible for dividing tasks and getting current region whilst the inner reinforcement learning just do the same thing as in this assignment, to reach some points. There is a theory published by Google, as shown in Fig 16

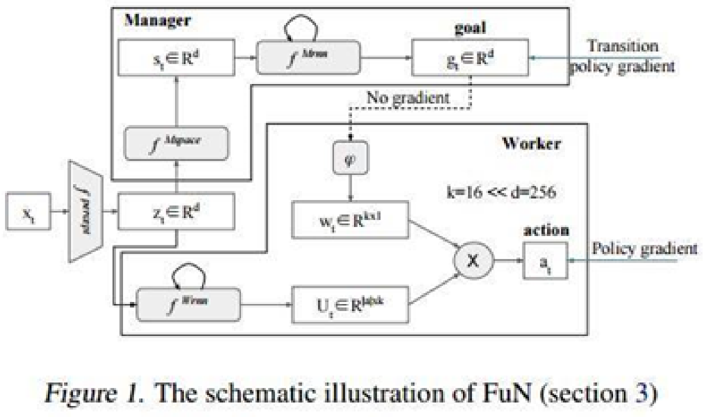


Figure 1. The schematic illustration of FuN (section 3)

Fig. 16. Learning curve

Another candidate is imitation learning which means learn from expert's action sequence. But this one will be more like a supervised learning. Essentially, this method is adding prior possibility. The examples are Unsupervised Perceptual Rewards for Imitation Learning by google and Generative Adversarial Imitation Learning by OpenAI.

VII. Question 7

The preferred direction of each square is shown as Fig 17. The wall is part of column 7,4 and 2. There is one thing we need to considered, since we set epsilon to 0.1, there is chance that robot hit wall when exploring. As we can see that in most state the robot has learnt how to navigate to the goal, however, there are some state that lack of training, such as right bottom and top left. The reason for it is the robot never get chance to explore that area due to the wall, except randomly placed in that area from start.

Acknowledgment

This report used part of code from lab8.There is 5 code files in total for different questions

References

[1] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, David Meger, "Deep Reinforcement Learning that Matters"

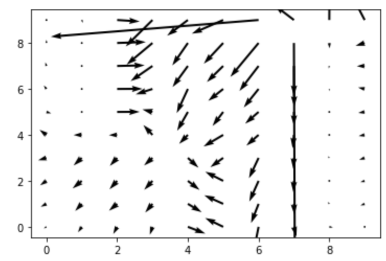


Fig. 17. result

[2] Pierre Sermanet, Kelvin Xu, Sergey Levine, "Unsupervised Perceptual Rewards for Imitation Learning", Google.

[3] Jonathan Ho, Stefano Ermon, "Generative Adversarial Imitation Learning" OPENAI.

APPENDIX

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter1d

def homing_nn(n_trials, n_steps,
learning_rate, eps, gamma, l=0.0):
    ## Definition of the environment
    N = 10
#height of the gridworld ---> number of rows
    M = 10
#length of the gridworld ---> number of columns
    N_states = N * M
#total number of states
    states_matrix = np.eye(N_states)
    N_actions = 4
#number of possible actions in each state:
    1->N 2->E 3->S 4->W
    action_row_change = np.array([-1,0,+1,0])
#number of cell shifted in vertical
    as a function of the action
    action_col_change = np.array([0,+1,0,-1])
#number of cell shifted in horizontal
    as a function of the action
    End = np.array([1, 1])
#terminal state--->reward
    s_end = np.ravel_multi_index(End,dims=(N,M),order='F')
#terminal state. Conversion in single index

    ## Rewards
    R = 1
#only when the robot reaches the charger, sited in End state

    ## Variables
    weights = np.random.rand(N_actions,N_states)
    learning_curve = np.zeros((n_trials))

    #elig_old = np.zeros((N_actions, N_states))

    ## SARSA

    # Start trials
    for trial in range(n_trials):

        # Initialization
        Start = np.array([np.random.randint(N),np.
#random start
        s_start = np.ravel_multi_index(Start,dims=
#conversion in single index
        state = Start
#set current state
        s_index = s_start
#conversion in single index
        step = 0

        # Start steps
        while s_index != s_end and step <= n_steps

            step += 1
            learning_curve[trial] = step

            input_vector = states_matrix[:,s_index
#convert the state into an input vector

            #compute Qvalues. Qvalue=logsig(weight
            Q = 1 / ( 1 + np.exp( - weights.dot(in
#Qvalue is 2x1 implementation of logsig

            # Note it is possible to remove the ac
            #Q = weights.dot(input_vector)

            #eps-greedy policy implementation
            greedy = (np.random.rand() > eps)
#1--->greedy action 0--->non-greedy action
            if greedy:
                action = np.argmax(Q)
#pick best action
            else:
                action = np.random.randint(N_actio
#pick random action


            state_new = np.array([0,0])
            #move into a new state
            state_new[0] = state[0] + action_row_c
            state_new[1] = state[1] + action_col_c

            #put the robot back in grid if it goes
            if state_new[0] < 0:
                state_new[0] = 0
            if state_new[0] >= N:
                state_new[0] = N-1
            if state_new[1] < 0:
                state_new[1] = 0
            if state_new[1] >= M:
                state_new[1] = M-1
            s_index_new = np.ravel_multi_index(state_new, dims=(N,M), order='F')
#conversion in a single index

            # Update Qvalues. Only if is not the first step
            if step > 1:
                # Update weights
                # Note we are updating the old values for the state and action from the previ
                dw = learningRate * (r_old - Q_old + gamma * Q[action]) * output_old.dot(input_
                weights += dw

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

            #update variables
            input_old = input_vector
            output_old = output
            Q_old = Q[action]
            r_old = 0
            # To do:
            calculate eligibility trace
            # elig_old =

            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:
                # Update weights
                dw = learningRate * (R - Q_old) * output_old*input_old.T
                weights += dw
                pass



    return learning_curve

# Parameter setup
nrepetitions = 10;  # number of runs for the algorithm
nTrials = 1000      # should be integer >0
nSteps = 50;         # maximum number of allowed steps
learningRate = 0.3; # should be real, Greater than 0
epsilon = 0.3;      # should be real, Greater or Equal to 0; epsion=0 Greedy, otherwise epsilon-
gamma = 0.9;         # should be real, positive, smaller than 1

learning_curve = np.zeros((nrepetitions, nTrials))


for i in range(nrepetitions):
    learning_curve[i] = homing_nn(nTrials, nSteps, learningRate, epsilon, gamma)

means = np.mean( learning_curve, axis=0)
errors = np.std(learning_curve, axis = 0) / np.sqrt(nrepetitions) # errorbars are equal to twi
```

```
smooth_means = gaussian_filter1d(means, 2)
smooth_errors = gaussian_filter1d(errors, 2)

plt.errorbar(np.arange(nTrials), smooth_means, smo
plt.plot(smooth_means, 'tab:blue') # Plot the mean

plt.xlabel('Trial', fontsize=16)
plt.ylabel('Average Reward', fontsize=16)
plt.tick_params(axis = 'both', which='major', labe
plt.savefig('Sarsa.png', dpi=300)
plt.show()
```

In order to reproduce the results, please run separate code files.