Introduction to Reinforcement Learning Assignment Report

Hongjie Guan
department of Informatics
University of Zurich
Zurich, Switzerland
hongjie.guan@uzh.ch

***Abstract*—This is the final code for the assignment of the course, Introduction to Reinforcement Learning. In this assignment, I have implemented a deep neural network capable of deep learning to perform checkmate in a 4x4 chessboard. The network is first built with Q-learning algorithm and then with State–action–reward–state–action (SARSA) algorithm. Through the comparison, we can see the differences between these two major algorithms in value-based reinforcement learning. Meanwhile, I also explored the effect of multiple parameters, such as the discount factor γ and the epsilon decaying factor β, on neural network training.**

***Index Terms*—Deep Q network, SARSA, Discount factor, Reinforcement Learning**

I. Introduction

Reinforcement learning is an important paradigms of machine learning based on training an agent with incentives and punishment. There are two most commonly used value-based algorithms, Q-learning and SARSA, to decide the policy by estimating the associated value of the policy. In the following Section II, I will first describe these two algorithms and discuss their differences as well as the advantages and disadvantages between the two methods from the theoretical perspective. In the third Experiments section, I will first show how the deep Q networks(DQN) is implemented , and how the discount factor γ and the epsilon decaying factor β affect the training. Then I will show the experiments with Deep SARSA. Through its comparison with DQN, We can understand their difference from a more empirical perspective.

II. Q-learning and SARSA

In Reinforcement learning, an agent starts with only the estimation of the reward instead of the exact knowledge of rewards. The agent has to explore the environment to update its own estimation or knowledge of the rewards(or punishment). So We first assumed that a agent employs an ϵ-greedy strategy which is important for exploration.

During each episode, from a current state **s**, the agent takes an action **a** based on its estimation of the values of taking these actions, **Q(s,a)**. Through a chosen action **a**, the agent will observe a reward **R** and enter another new state **s'**. Next, the agent will update its estimation **Q(s,a)** with the reward **R** and the estimation of values at the new state **Q(s',a')**. The Q-learning and SARSA are different algorithms to update Q values.

A. Q-learning

The Q-learning algorithm updates **Q(s,a)** with the maximized **Q(s',a')**:

$$Q(s,a) = Q(s,a) + \alpha[R + \gamma \arg\max_u Q(s,u) - Q(s,a)]$$

When the agent reach stats **s'**, it gets new information(the rewards **R**). The agent updates the Q value estimation with a learning rate α times a error, which equals the reward **R** minus the estimated Q value and plus the estimated future Q value at **s'** which is discounted by the discount rate γ. Q-learning is called off-policy because the new **a'** is chosen on a greedy strategy instead of a current ϵ-greedy strategy, even if the agent will stick to the ϵ-greedy strategy at **s'** later. After this update, the agent will take **s'** as the new **s** and continue the process until the end state is reached.

B. SARSA

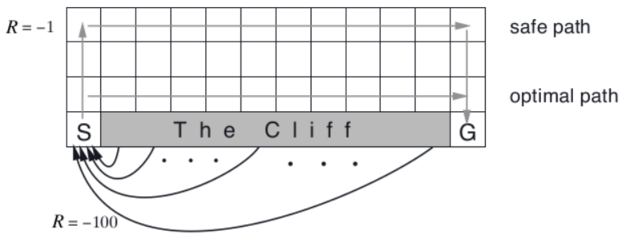
The SARSA algorithm updates **Q(s,a)** with the a **Q(s',a')** chosen by the on-going $epsilon$-greedy strategy:

$$Q(s,a) = Q(s,a) + \alpha[R + \gamma Q(s,a') - Q(s,a)]$$

SARSA has a similar Q updating function with learning rate α and discount factor γ. The only difference is that in SARSA the new **a'** is chosen with the on-going ϵ-greedy strategy, which is the reason why SARSA is called on policy. After this update, the agent will also take **s'** as the new **s** and continue the process until the end state is reached.

C. Comparison

The advantages of Q-learning over SARSA is that Q-learning learns directly the optimal policy while SARSA learns a near-optimal one. If the SARSA wants to learning towards the optimal policy it needs to add a decaying-ϵ parameter(β in this assignment). When the epsilon goes to zero as training time increase, as ϵ-greedy policy will converge to greedy policy, SARSA will converge to Q-learning.This comparison can be illustrated by this commonly used Cliff example:



Q learning will learn the optimal path while SARSA will tends to train the safe path.

Hence, the advantages of SARSA over Q-learning is that SARSA is more conservative. SARSA will approach convergence considering possible penalties from exploratory moves, whilst Q-learning will ignore them. Using the above Cliff example, the optimal path gives the highest Q values to both methods, the SARSA doesn't always take it while Q-learning will always take it to update its Q estimation. However, when the agent is actually on the optimal path(since it still use ϵ-greedy strategy), it can possibly fall down from the cliff(observing a high penalty). The SARSA, on the other hand, is less likely to step into this danger zone.

This leads to another advantage of SARSA: SARSA tends to have a lower per-sample variance than Q-learning especially in an environment of "big reward comes with big risk". Q-learning will be exposed to higher risk or observe more penalties because it learns the optimal strategy. In our assignment, this advantage of SARSA is not obvious because there are no dangerous moves that can bring back big returns in our game.

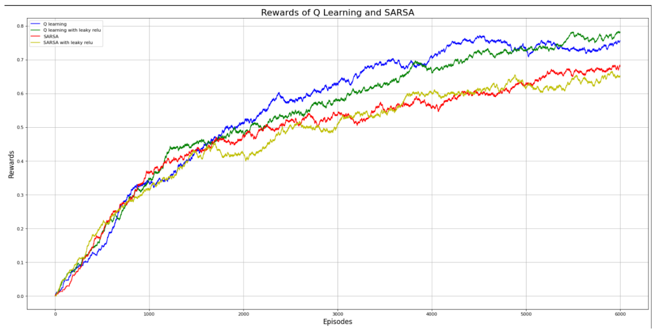
III. Experiments

In our Experiments, I first build a deep Q network with 1 hidden layer with 200 neurons and then the deep SARSA with identical network structure. I also explore effect of discount rate γ on the deep Q network training and the epsilon decaying factor β on both the deep Q network training and deep SARSA network. During the training, I observed that "dying ReLU" problem might happen so I created two leaky ReLU versions of the network as well for Q-learning and SARSA. The effect of leaky ReLU function is to allow a small, positive gradient for the the inactive unit. to let them contribute to the output layers as well.

Important Note: The experiment result can be reproduce by setting a random seed of 6.

A. Deep Q Network and Deep SARSA

Both of the network are trained for one hundred thousand episodes. Their rewards both converge to the reward of checkmate and the number of steps of them both converge to 2-3. The following graph shows the experiments result for the first sixty thousand episodes. The full result for one hundred thousand episodes can be found in the code deliverable.



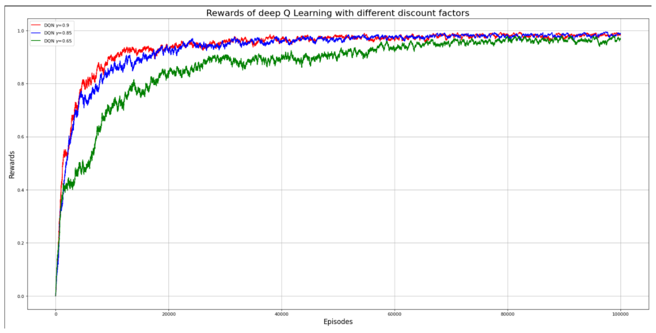
The graph shows that the two Q-learning network converges quicker than the SARSAs. This observation is consistent with the former analysis from theoretical perspective as Q-learn learns toward the option strategy(to perform checkmate) while SARSA only learns the strategy of "performing checkmate under a 1-ϵ probability". In the early training episode, this difference is obvious but in the later episode, as the β decays the ϵ, ϵ becomes smaller so this difference will disappear.

B. Deep Q Network with Leaky ReLU activation function

In some experiments, I observed that using ReLU activation function makes many neurons inactive (output 0), which makes training process inefficient. To solve this problem, I changed ReLU with leaky ReLU functions to speed up the training. However, as I changed a random seed, the "dying ReLU" problem doesn't show up again. Hence, the effect is not obvious.See Appendix for more details of implementing leaky ReLU.

C. Deep Q Network with a change of γ

The discount factor γ is a 0 to 1 parameter to discount the future rewards to the present state. For example, if γ=0.8, 1 unit of reward from the next state will only equal to 0.8 unit of reward from the current state. As γ increases, the future reward is going to have an increasing effect on Q updating. The following picture has shown the experiment result of a deep Q Network with different γs.



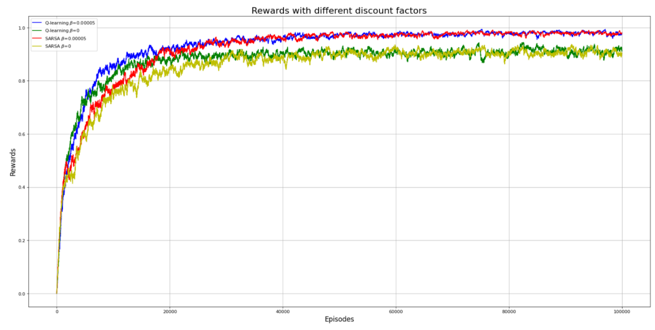
As we can see, the result is consistent with our theory. As γ increases, the reward of checkmate in the future will have a larger effect on training, so the reward converges to 1 quicker. Similarly effects should be able to be observed in case of SARSA.

D. β'effects on Deep Q Network and Deep SARSA

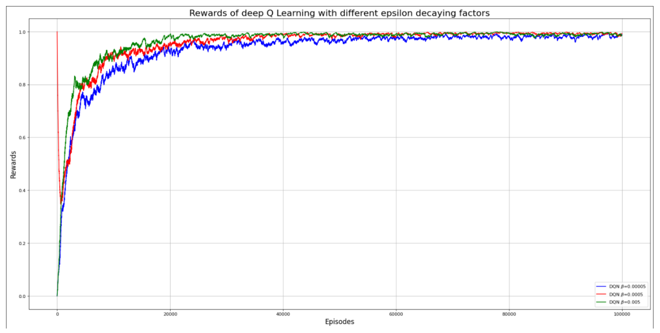
According to the definition of the ϵ decaying factor β, the larger the β is, the faster ϵ decays as training time increases.

According to the definition of ϵ-greedy policy, greedy policy is identical to 0-greedy policy(when ϵ equals 0). Consequently, as the ϵ decays, the ϵ-greedy policy will converge to greedy policy.

The following graph shows that when the β equals 0,ϵ is unchanged during training process. In the end, both Q-learning and SARSA can only converge to an ϵ-optimal level(a near-optimal level) instead of the optimal level since they are both using ϵ-greedy policy.When β doesn't equal 0, both of the curve can gradually converges to 1, the optimal level.



This part might seem counter-intuitive, because Q-learning learns the optimal strategy but it still converges to only the ϵ-optimal level. The reason is that when β equals 0, in state a, both methods will still use the ϵ-greedy policy with the unchanged ϵ. Even if model is well-trained already, the agent with such a policy won't always choose the optimal action, so the rewards can't always be optimal. Now, we consider how different β influences the training.



A experiment with 3 different βs shows that the higher the β, the earlier rewards converges to the optimal level. It is worth noticing that these three βs are all in a acceptable range for this environment case. If the environment is more complicated and the β is large enough to make the ϵ decayed too fast, the agent will employ exploitation without enough exploration of the environment. A early exploitation will result in low average rewards

IV. Conclusion

The experiment shows that in such a not complicated binary-rewarding environment:

1. Q-learning converges faster than SARSA in this case.

2. The higher the γ, the faster the algorithm converges to the optimal level.

3. The higher the β, the faster the algorithm converge to the optimal level (once the agent has enough exploration).Without the decaying-epsilon strategy, both the Q-learning and SARSA will converges to an ϵ-optimal level in stead of the optional level

V. Appendix

A. GitHub link for the code

GitHub Link

B. Implement leaky ReLU

The ReLU function and its derivative:

$$f(x) = \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases} \tag{1}$$

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x <= 0 \end{cases} \tag{2}$$

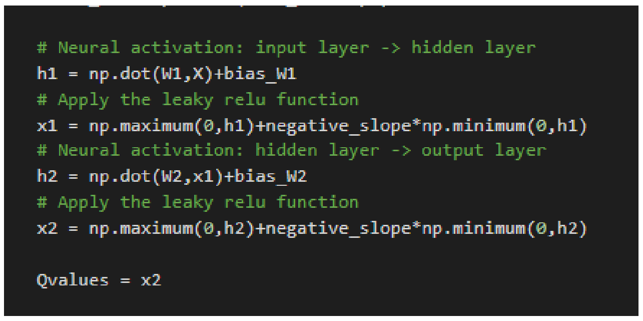
The leaky ReLU function and its derivative:

$$f(x) = \begin{cases} x & x > 0 \\ k * x & x <= 0 \end{cases} \tag{3}$$

$$f'(x) = \begin{cases} 1 & x > 0 \\ k & x <= 0 \end{cases} \tag{4}$$

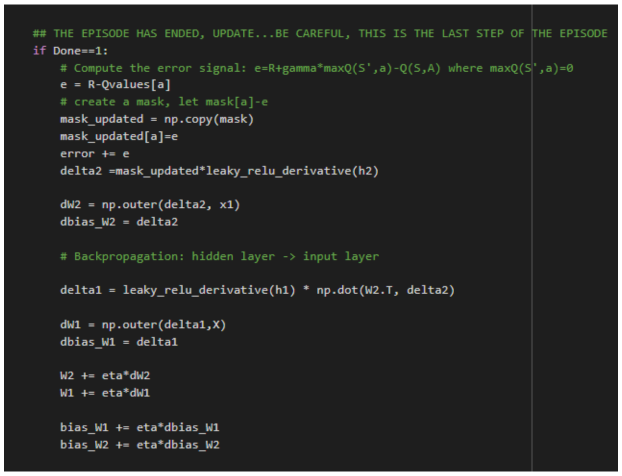
Here we assume k=0.01.

The code is implemented as the following snippets:

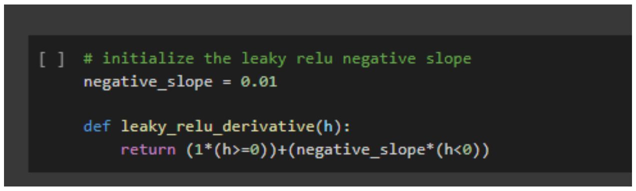


```
# Neural activation: input layer -> hidden layer
h1 = np.dot(W1,X)+bias_W1
# Apply the leaky relu function
x1 = np.maximum(0,h1)+negative_slope*np.minimum(0,h1)
# Neural activation: hidden layer -> output layer
h2 = np.dot(W2,x1)+bias_W2
# Apply the leaky relu function
x2 = np.maximum(0,h2)+negative_slope*np.minimum(0,h2)

Qvalues = x2
```

During the forward propagating process, the activation function output equals the input if the input is larger than 0 and k if the input is smaller or equal to 0.



```
## THE EPISODE HAS ENDED, UPDATE...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
if Done==1:
    # Compute the error signal: e=R+gamma*maxQ(S',a)-Q(S,A) where maxQ(S',a)=0
    e = R-Qvalues[a]
    # create a mask, let mask[a]-e
    mask_updated = np.copy(mask)
    mask_updated[a]=e
    error += e
    delta2 =mask_updated*leaky_relu_derivative(h2)

    dW2 = np.outer(delta2, x1)
    dbias_W2 = delta2

    # Backpropagation: hidden layer -> input layer

    delta1 = leaky_relu_derivative(h1) * np.dot(W2.T, delta2)

    dW1 = np.outer(delta1,X)
    dbias_W1 = delta1

    W2 += eta*dW2
    W1 += eta*dW1

    bias_W1 += eta*dbias_W1
    bias_W2 += eta*dbias_W2
```



```
[ ]  # initialize the leaky relu negative slope
     negative_slope = 0.01

     def leaky_relu_derivative(h):
         return (1*(h>=0))+(negative_slope*(h<0))
```

During the backward propagating process, we will use the function derivative, which returns 1 for inputs larger than 0 and returns k if the input is smaller or equal to 0.