

COM3240 Assignment Report

Jiacong Wu

I. INTRODUCTION

In this assignment, we aim to explore how the SARSA algorithm perform, compared to Q-learning, for a reinforcement learning application to a modified game of chess on a 4x4 board, where agent aims to checkmate the opponent given an initial, random, legal board state.

II. TASK 1 - Q-LEARNING AND SARSA

In Q-learning, for an agent in state s , taking an action a based on a behaviour/action selection policy, it receives reward r and moves to a new state s' . We obtain the expected reward $Q(s', a')$ from the new state and a new action a' , based on the target policy (in the case of Q-learning, this target policy is always greedy [1]), which contributes to the target Q value in the update of the Q-value of a state-action pair using equation 1

$$Q(s, a) = Q(s, a) + \eta(r + \max_{a'} \gamma Q(s', a') - Q(s, a)) \quad (1)$$

- γ is the discount factor $0 < \gamma < 1$, adjusted depending on how much importance to put on the immediate reward and future reward. For $\gamma = 0$, the agent will only consider the immediate reward, while a factor approaching 1 will make it strive for a long-term high reward [2].
- η is the learning rate/step size, which is used to control the speed of learning.

The SARSA (State-action-reward-state-action) algorithm is a slightly modified version of the Q-learning algorithm, in the difference being it uses the action a' selected by the behaviour policy to obtain the expected reward $Q(s', a')$ in the update of the Q-value of a state-action pair using equation 2, instead of using a fixed greedy target policy as with Q-learning

$$Q(s, a) = Q(s, a) + \eta(r + \gamma Q(s', a') - Q(s, a)) \quad (2)$$

And obviously, with SARSA's approach we will have the advantage of being able to use a non-greedy behaviour policy for obtaining the expected reward, which allows the algorithm to explore a range of different solutions, and not be potentially stuck in a local optimal solution as with Q-learning's greedy approach.

III. TASK2 - IMPLEMENTING SARSA

We wrote the code to forward inference of the neural network to obtain the Q-value for each action, shown in Figure 6 (implementation details/code are listed in the Appendix section). The RELU activation function for the output weighted sums from the hidden layers, instead of a sigmoid function.

The advantage of using RELU is that it reduces the likelihood of the vanishing gradient problem [3], arising from gradient-based such as backpropagation which is what we used to train the neural network [4]. The vanishing gradient problem is when the gradient of the loss function approaches 0, causing the weights to not be updated, and therefore the neural network will fail to learn anything.

We proceed to implement the behaviour/action policy in Figure 7. The behaviour policy is epsilon-greedy, where the agent will take a random action `a_agent` from the array of legal actions `a`, with probability ϵ , and take the greedy action with probability $1 - \epsilon$. If after taking the action, we haven't reached the terminal state, another action `a_agent2` will be taken from the list of next legal actions, resulted taking the previous action as shown in Figure 8. This is chosen according to a target policy, which is the same as the behaviour policy in the case of SARSA.

A factor of the Q-value corresponding to the next action is added to the target reward to incorporate future reward. In the case of choosing an action that reaches a terminal state (checkmate or stalemate), no future action can be taken, therefore no future is incorporated. An error signal, which is the difference between the target and actual output, is computed in order to perform backpropagation, implemented in Figure 9. The weights are updated with the ADAM optimizer, which is a gradient descent optimization algorithm that helps us achieve convergence much quicker. When updating the weights for the last layer `w2`, we made sure we are only updating the weights corresponding to the action taken and not all the weights. We also implemented a function in Figure 10 for computing the exponential moving averages for the data we collected, for the accumulated reward and number of moves taken during each episode, using a window size of 1000.

I found out that using a learning rate η of 0.00015 instead of the default suggested value of 0.0035, gave me a much higher average reward of around 0.85, as well as having a stable convergence trend at the end of 40000 episodes, shown in Figure 1.

IV. TASK 3 - VARYING γ AND ϵ VIA β

A. Varying γ

γ is the discount factor controlling the importance of future rewards. A higher γ value will make the agent strive for a long-term high reward, while a lower γ value will make the agent focus on the immediate reward. In the case of the modified chess game, a higher γ value will make the agent focus on the long-term reward of checkmating the opponent and therefore incorporating long-term planning behaviour. While a lower γ value will make the agent focus on immediate rewards such as

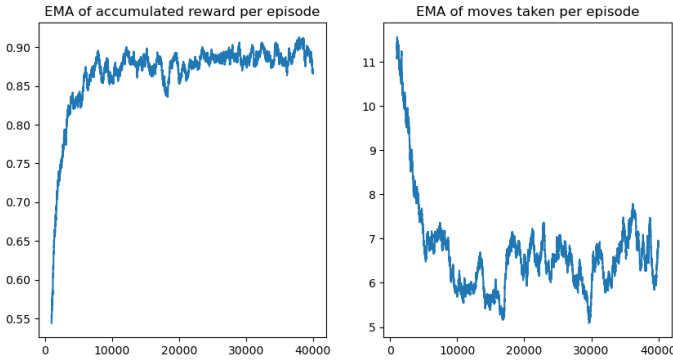


Fig. 1. EMA of data recorded for SARSA, with $\eta = 0.00015$, $\gamma = 0.85$, $N_{\text{episodes}} = 40000$

threatening the opponent's king. I found using a γ of 0.75 gave a much more smoother convergence trend (Figure 2) compared to other values. Increasing the value of γ , the agent was able to obtain.

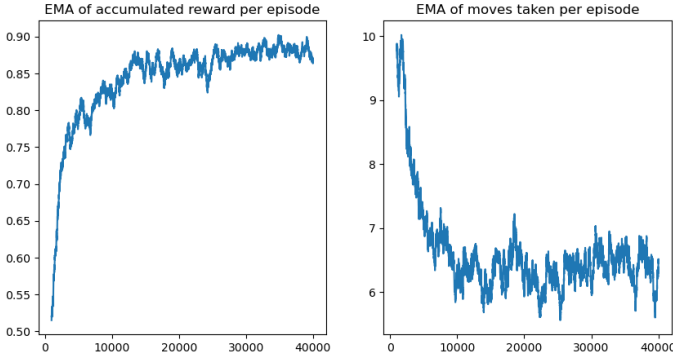


Fig. 2. EMA of data recorded for SARSA, with $\eta = 0.00015$, $\gamma = 0.75$, $N_{\text{episodes}} = 40000$

B. Varying ϵ via β

ϵ is the probability of the agent taking a random action, instead of a greedy action by choosing the action that corresponds to the highest Q-value. A lower ϵ value will make the agent more likely to take the greedy action, while a higher value will make the agent more likely to take a random action, this will allow the agent to explore different solutions rather than always exploiting what's currently best.

β is a parameter that sets how quickly the value of ϵ is decaying. After each episode, ϵ is set using equation 3, where ϵ_0 is the initial probability and n is the episode number.

$$\epsilon_n = \frac{\epsilon_0}{1 + \beta * n} \quad (3)$$

Using a lower β value seems to allow us to reach convergence much quicker shown in Figure 3, however the value converged to seems to be lower than what using a higher β value of 0.00005 (Figure 4) would be, in the long run.

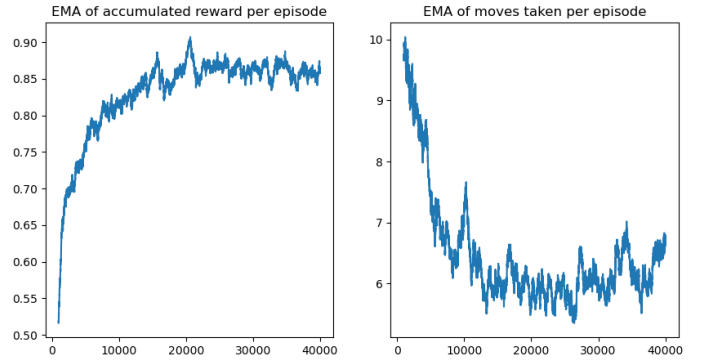


Fig. 3. EMA of data recorded for SARSA, with $\eta = 0.00015$, $\gamma = 0.75$, $\beta = 0.0000005$, $N_{\text{episodes}} = 40000$

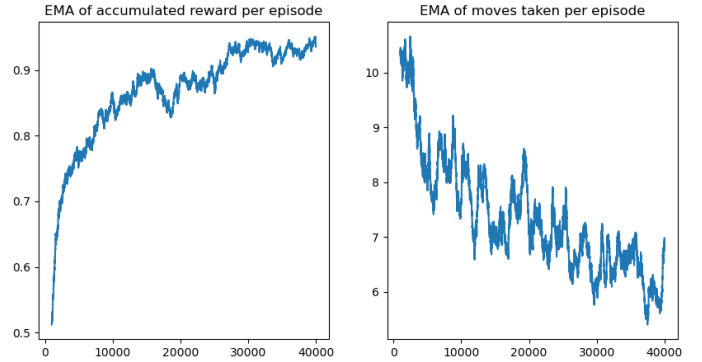


Fig. 4. EMA of data recorded for SARSA, with $\eta = 0.00015$, $\gamma = 0.75$, $\beta = 0.00005$, $N_{\text{episodes}} = 40000$

V. TASK 4 - IMPLEMENTING Q-LEARNING

Again, the Q-learning and SARSA algorithms are very similar, the only difference being the target policy used to obtain the expected reward. Q-learning's target policy is always greedy, that is, it always chooses the action that gives the highest Q-value/expected reward. We simply commented out SARSA's target policy and replaced with the line of code in Figure 11 to implement Q-learning.

Comparing the results obtained from Q-learning below in Figure 5, with SARSA in Figure 2. SARSA seems to reach convergence much quicker, however that reward value seems to be lower than what Q-learning is capable of achieving in the long run, also Q-learning takes much less number of moves.

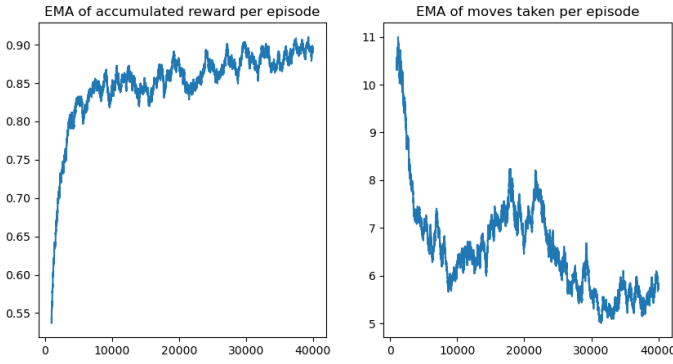


Fig. 5. EMA of data recorded for Q-learning, with $\eta = 0.00015$, $\gamma = 0.75$, $\beta = 0.00005$, $N_{\text{episodes}} = 40000$

REFERENCES

- [1] P. Porwal, "SARSA (State Action Reward State Action) Learning - Reinforcement Learning - Machine Learning" YouTube. https://www.youtube.com/watch?v=FhSaHuC0u2M&ab_channel=PankajPorwal (Accessed: April 6, 2023)
- [2] "State-action-reward-state-action", Wikipedia. <https://en.wikipedia.org/wiki/State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action> (Accessed: April 6, 2023)
- [3] DaemonMaker, "What are the advantages of ReLU over sigmoid function in deep neural networks?" Stack Overflow. <https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks> (Accessed: May 16, 2023)
- [4] "The Vanishing Gradient Problem" Great Learning. <https://www.mygreatlearning.com/blog/the-vanishing-gradient-problem/> (Accessed: May 16, 2023)

NOTES

The code for this assignment can be found at <https://github.com/jwu02/com3240-reinforcement-learning-assignment>. We used the same seed for the Numpy random number generator to ensure reproducibility of results. Furthermore, we introduced a variable to note the number of episodes trained on if we wish to discontinue training at some point, this variable can be used with the exponential moving average function to easily verify that the results are reproducible, without changing the code. Unless explicitly stated, the parameters used to generate each of the plots can be assumed to be the default, suggested hyperparameters:

- $\epsilon_0 = 0.2$
- $\beta = 0.000005$
- $\gamma = 0.85$
- $\eta = 0.0035$
- $N_{\text{episodes}} = 100000$

At least 1000 episodes must be ran to produce a plot, since that's the window size we used. For switching between the Q-learning and SARSA algorithms, we simply commented out the epsilon-greedy target policy for SARSA and replaced it with the greedy target policy for Q-learning, and vice versa.

APPENDIX

```
h1 = np.dot(w1,x)+bias_w1 # Neural activation: input layer -> hidden layer
x1 = h1*(h1>0) # Apply the RELU function
h2 = np.dot(w2,x1)+bias_w2 # Neural activation: hidden layer -> output layer
x2_Qvalues = 1/(1+np.exp(-h2)) # Apply the sigmoid function
```

Fig. 6. Forward inference/neural activation of neural network to calculate Q-value for each action

```
# Epsilon-greedy parameter
# with probability epsilon choose action at random
# if epsilon = 0 then always choose Greedy
eGreedy = int(np.random.rand() < epsilon_f)

# Implement the behaviour/action policy
if eGreedy:
    # if epsilon>0 (e-Greedy) choose action at random
    a_agent = np.random.permutation(a)[0]
else:
    # otherwise choose greedy
    # will result in action index corresponding to
    # highest Q-value out of the possible actions a
    a_agent = a[np.argmax(x2_Qvalues[a])]

S_next,X_next,allowed_a_next,R,Done=env.OneStep(a_agent)
```

Fig. 7. Implementing epsilon-greedy behaviour policy

```
a2, _ = np.where(allowed_a_next==1)

# SARSA target policy same as behaviour policy - epsilon greedy
eGreedy = int(np.random.rand() < epsilon_f)
if eGreedy:
    a_agent2 = np.random.permutation(a2)[0]
else:
    a_agent2 = a2[np.argmax(x2_Qvalues[a2])]
```

Fig. 8. Implementing epsilon-greedy target policy for SARSA

```
# Compute the error signal (target output - actual output)
e_n = (R + gamma * x2_Qvalues[a_agent2]) - x2_Qvalues[a_agent]

# Backpropagation: output layer -> hidden layer
delta2 = x2_Qvalues*(1-x2_Qvalues) * e_n
masked_delta2 = np.zeros_like(delta2)
masked_delta2[a_agent] = delta2[a_agent]
dw2 = np.outer(masked_delta2, x1)

# Backpropagation: hidden layer -> input layer
delta1 = x1*(1-x1) * np.dot(w2.T, masked_delta2)
dw1 = np.outer(delta1, x)

# update weights
w2[a_agent] += eta*Adam_w2.Compute(dw2)[a_agent]
bias_w2[a_agent] += eta*Adam_bias_w2.Compute(masked_delta2)[a_agent]
w1 += eta*Adam_w1.Compute(dw1)
bias_w1 += eta*Adam_bias_w1.Compute(delta1)
```

Fig. 9. Backpropagation and update weights with ADAM optimizer

```
def exponential_moving_average(data, window_size):
    ema_alpha = 2/(window_size+1)

    ema = np.zeros_like(data)
    # initial EMA is a simple moving average (SMA)
    initial_ema = np.mean(data[0:window_size])
    ema[window_size-1] = initial_ema
    for i in range(window_size, len(data)):
        ema[i] = ema_alpha*data[i] + (1-ema_alpha)*ema[i-1]

    return ema[window_size:]
```

Fig. 10. Function for computing exponential moving averages

```
# Q-learning target policy is always greedy
a_agent2 = a2[np.argmax(x2_Qvalues[a2])]
```

Fig. 11. Implementing greedy target policy for Q-learning