Adaptive Intelligence COM3240*

1st Ethan Barker
dept. Computer Science
University of Sheffield
Sheffield, United Kingdom
ebarker4@sheffield.ac.uk

GitHub Repo: https://github.com/EthanBarker/COM3240-Reinforcement-Learning-Assignment

I. Introduction

This document sets out to answer four questions for the module: Adaptive Intelligence COM3240. We break this paper up into five sections with the following four dedicated to answering a single question. The first question we answer surrounds describing the Q-Learning Algorithm and SARSA algorithm in the context of Deep Reinforcement Learning. We will explain the differences and the possible advantages and disadvantages of each algorithm. After this, our next section will show our plots produced by the Q-learning algorithm as this was the first one we chose to implement. We will show two plots that show the reward per game and the number of moves per game vs training time. On top of this, we will also explain what these results tell us. In the section after we will change the discount factor γ and the speed β of the decaying trend of ϵ and explain how this affects our results. Then we will show my results yielded by the SARSA algorithm and explain how these differ from the Q-learning algorithm. We will compare the results against each other and say if they performed how we predicted they would. Finally, we will end this paper with a conclusion summarising the main points in a concise form.

II. Describe the algorithms Q-learning and SARSA

In this part of the paper we describe the two algorithms Q-learning and SARSA which stands for "State Action Reward State Action". After explaining the two algorithms we will state the advantages and disadvantages of each one. It is important to note that both of these algorithms were used within the context of Deep Reinforcement Learning as in the implementation we used a Deep Neural Network to approximate the $Q(s, a)$.

A. Q-learning

Q-learning is an example of an off-policy reinforcement learning algorithm. Off-policy algorithms are algorithms which have the goal of developing their own policy. Using this knowledge we can say that the Q-Learning algorithm will find its own approach to a given problem. This algorithm learns the optimal Q value function defined as $Q(s, a)$ which is an expected cumulative reward. From this definition, it can be said that the reward is calculated by taking the action a in a state s. In Q-learning the algorithm finds the best policy by adding its current state reward to its maximum reward from its future states. Q-Learning updates via the following equation:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$$

B. SARSA

SARSA on the other hand, which stands for "State Action Reward State Action", is an example of an on-policy reinforcement learning algorithm. This means that this algorithm differs from Q-Learning as it uses a decision policy to update Q values. The policy which we use in our implementation is Epsilon greedy. SARSA updates the Q-function via this equation:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$$

C. Advantages and Disadvantages of Q-Learning vs SARSA

One key difference which provides insight into the advantages and disadvantages of the algorithms is how the update rule is calculated. Q-Learning is an off-policy algorithm and SARSA is an on-policy algorithm. As a result of this Q-Learning has an advantage as it can learn from a collected dataset and directly estimates the optimal state-action pairs. Compared to this SARSA learns via the epsilon greedy policy which can provide an advantage as it should converge faster than Q-Learning. On top of this, it is suggested that the SARSA algorithm is more robust than that of the Q-Learning algorithm as it forces the agent to explore the environment at least at the start. Hence, SARSA could learn to avoid actions which result in smaller amounts of rewards

III. Implementation of Deep Q-Learning

In this part of the report, we show the results yielded by our implementation of the Deep Q-Learning algorithm. We will first show screenshots of our code explaining what it does and then show plots produced which show us the reward per game and the number of moves per game vs training time.

A. Epsilon Greedy Policy

Before we implemented the Q-Learning algorithm with a neural network we understood that we would first need to implement a policy which would dictate how the algorithm

manages exploration vs exploitation. We opted for using the Epsilon greedy policy as this was the one we were most familiar with from the previous lab sessions and we also knew that this policy would help prevent the agent from getting stuck in suboptimal solutions by encouraging further exploration instead of exploitation.
In our solution, we created a function named "EpGreedPol" which took in three inputs and produced a chosen action a. The inputs to this function were Q which represents the Q-values for all possible actions, ϵ which represents the probability of a random action being chosen and $allowed_a$ which is an array of all of the accepted actions which can be performed in that state.

B. Neural Network

For the neural network, it was suggested that we use a network with one hidden layer and a size of 200. The purpose of using a neural network was to learn a Q-function that maps states to action values in our Q-Learning algorithm. Our neural network consisted of two layers. The first of these was the hidden layer with contained our activation function. The choice of activation function we used was ReLU and this was made as we felt most comfortable coding this. The second layer we had was our output layer with a linear activation function.

C. Training Loop

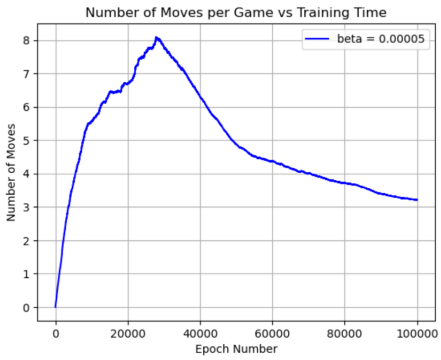
Finally, for our training loop, we implemented a solution which iterated over 100,000 episodes to train the agent to play chess. The purpose of this training loop was to learn the optimal policy for the agent to maximise its rewards. At each iteration, we first initialise the environment. From here we use an Epsilon greedy policy to explore and exploit the environment. As our agent goes through a training episode it uses its current reward and Q-values to update the Q-value estimates.

D. Results

We produced two plots to show the results of our work. The first was a plot showing the reward per game against the epoch version. The second plot we produced shows the number of moves per game vs training time. We first ran these tests with the hyperparameters which were provided. These were:

1) ϵ = 0.2
2) β = 0.00005
3) γ = 0.85
4) eta = 0.0035

This yielded the following two plots after 100,000 epochs.



The results we got matched our expectations, the first graph shows that over time to 100000 epochs our Q-Learning algorithm ends at a point of getting a reward of approximately 0.92 in each game. This suggests to us that our algorithm has been successful as over time the reward has continuously gone up. The second graph shows us that until around 30000 epochs the number of moves was increasing to a peak of 8 but after 30000 epochs the number of moves decreased first sharply to 4.2 at 50000. After this, the number of moves slowly declined to around 3.3 over the rest of the training time.

IV. Changing the γ, β and ϵ values

Once we had created the initial two plots using the initial variables provided, our next task was to change the values of γ, β and ϵ. Before showing the results we explain each hyperparameter, what it does in relation to the code and how we believe changing it will affect our results. We will then state if our predictions are correct after showing plots with different values.

A. γ "Gamma"

The first hyperparameter we will discuss is the γ hyperparameter also known as the "discount factor". This hyperparameter adjusts the importance of future rewards against immediate rewards. This hyperparameter holds a value between 0 and 1. If the value is closer to 1 then this means that the agent will put a larger importance on looking for higher future rewards. This would then result in the agent considering the longer-term consequences. On the other hand, if the value is closer to 0 then the agent will mainly focus on immediate rewards. If this occurs then the agent could miss out on higher future rewards.

B. ϵ "Epsilon"

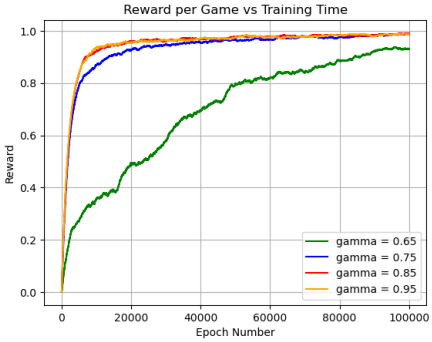
The second hyperparameter we will explain is the ϵ value. This is the starting value of the Epsilon greedy policy which is used to balance exploration and exploitation. This hyperparameter also holds a value between 0 and 1. If the value is higher then the agent is more likely to explore as it takes more random actions. This is especially useful to have when the agent begins as it is a good way for the agent to gain lots of information about what works and what doesn't work. If the value is then lower the agent will exploit by choosing actions with the highest Q-values. Exploitation works best when the agent has good knowledge of the environment it is in and it will lead to more deterministic behaviour overall.

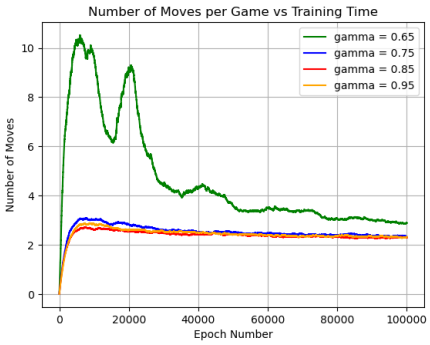
C. β "Beta"

The final hyperparameter we will discuss is that of the β value. In our solution, this is represented as the speed of the rate at which epsilon decays over time. As mentioned before the epsilon value is used to balance exploration and exploitation. The beta value chosen decides how fast the agent transforms from exploration to exploitation. A higher value for beta would result in a faster decay of epsilon. This would mean that the agent would quickly switch from exploration to exploitation. Hence, a lower value for the beta would result in a slower transition from exploration and exploitation meaning the agent will explore for a longer amount of time.

D. Results

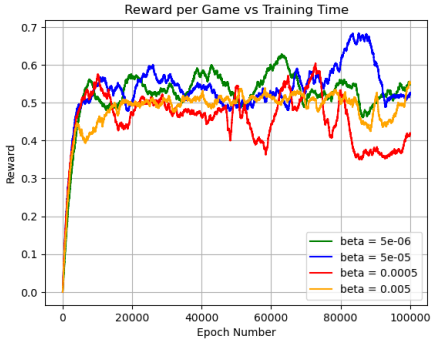
When testing how these hyperparameters affected our results we ran two sets of tests.
The first set of tests we ran was changing the γ "Gamma" value. We chose to use gamma values of 0.65, 0.75, 0.85 and 0.95. We believed that these values had a good range so that we could see how the agent performs when it has to choose between prioritising long-term vs short-term rewards. As we had before we created two plots the first of these showing reward per game vs training time and the second of these showing the number of moves per game vs training time. We predicted that for our lower gamma values the agent will focus on short-term rewards and not focus as much on long-term rewards. We believed that as we incremented this value from 0.65 to 0.95 the agent would shift to focus on long-term rewards as the gamma value increased.
The images next show our results after 100,000 epochs. We used gamma values of 0.65, 0.75, 0.85 and 0.95.

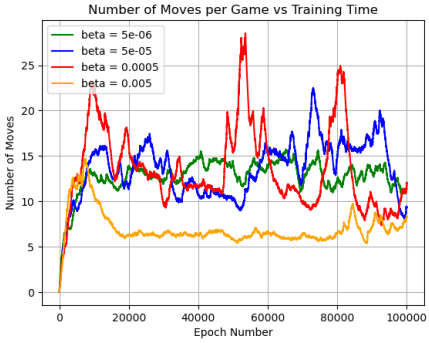




These results showed us that a higher gamma value leads to a higher reward in a shorter number of time when compared to a lower gamma value. This is because a higher gamma value leads the value to long-term consequences earlier on than lower gamma values. We can also see from the second graph that a higher gamma value will usually lead to a smaller number of moves per game vs training time.

The final set of tests we ran involved changing the β "Beta" values which represented the rate of decay. For these tests, we used beta values of 0.000005, 0.00005, 0.0005 and 0.005. We also believed that this was an acceptable range as results from these tests would provide us with insight into how the beta value affects the output. We predicted using our knowledge from before that low beta values such as 0.000005 will allow the agent to explore for longer but as this value changes to a higher value like 0.005 then the agent will begin to focus on exploitation faster.
These images show our results after 100,000 epochs. We used beta values of 0.000005, 0.00005, 0.0005 and 0.005.





From these two graphs, we can see that both are noisy diagrams. What we expected to see was that for higher beta values the rate of decay is faster. We predicted this could lead

to a faster convergence for the higher gamma values but this can not be seen in our graphs. We also predicted there would be less exploration and this can be seen in the second graph we produced. The number of moves is lower for the highest beta value of 0.005.

V. Implementation of SARSA

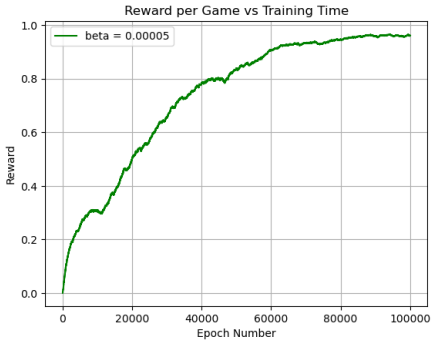
In this section of our report, we discuss our implementation of the SARSA algorithm. We show results yielded from tests we performed and then compared these results against the Q-Learning algorithm. In this section, we discuss how we expect these algorithms to differ and state if our predictions were correct or not.

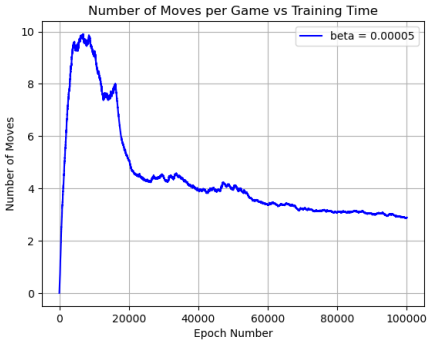
The SARSA algorithm uses the same epsilon greedy policy and Neural Network that our Q-Learning algorithm uses. The main difference between the algorithms can be found in the training loop. This difference is only small and resides in how the update step is chosen. For SARSA an action is chosen based on the Epsilon greedy policy during this update step which shows us that this is an example of an on-policy algorithm. On the other hand, Q-Learning as mentioned before is an off-policy algorithm so the action with the largest Q-value is chosen during the update step. This is only a small change in our implementation but allows the SARSA agent to learn by taking into account the actions which are more likely to work best based on the current exploration strategy.

We ran the same set of tests as we had previously for Q-Learning. We used the same hyperparameters as before which were:

1) ϵ = 0.2
2) β = 0.00005
3) γ = 0.85
4) eta = 0.0035

This provided us with the following two results.





Comparing SARSA to the Q-Learning algorithm we found that after 100,000 epochs both algorithms performed very similarly in terms of the reward at the end of 100000 epochs. However, we did see a slight difference in conversion with the Q-Learning algorithm converging at a smoother and faster rate than SARSA. We also saw a difference between the two algorithms in the number of moves vs training time. We saw both finished at a similar value of moves per game but the SARSA algorithm started to take fewer turns per game faster than the Q-Learning algorithm. From this, we can see the advantages and disadvantages of both algorithms. For Q-Learning the reward increases faster vs training time and for SARSA the number of moves per game decreases faster vs training time.

VI. Conclusion

To conclude this report set out to answer the questions for the module: Adaptive Intelligence COM3240. This included defining the two algorithms which we wanted to implement and then explaining the advantages and disadvantages of each one. From here we moved on to our implementation of the Q-Learning algorithm and explained how we created this. After, we explained how each hyperparameter affects the results and ran tests to enforce these explanations. Finally, we stated the key difference between our implementation of the SARSA algorithm and the Q-Learning algorithm whilst saying what this means for our results.

VII. Repeatability

We made all of our code repeatable by using a seed with the number 90. This ensures that someone else could run my code using this seed and recieve the same results as we did.

```
#Load a random seed for repeatability
np.random.seed(90)
```

APPENDIX

The code below shows our implementation of the Neural Network we used in the Q-Learning and SARSA algorithms .

```
# INITIALISE THE PARAMETERS OF YOUR NEURAL NETWORK AND...
# PLEASE CONSIDER TO USE A MASK OF ONE FOR THE ACTION MADE AND ZERO OTHERWISE IF YOU ARE NOT USING VANILLA GRADIENT DESCENT..
# WE SUGGEST A NETWORK WITH ONE HIDDEN LAYER WITH SIZE 200.

#Initialize the neural network with random weights and biases
def initialize_NN(N_in, N_h, N_a):
    W1 = np.random.randn(N_h, N_in) * 0.01
    b1 = np.random.randn(N_h, 1) * 0.01
    W2 = np.random.randn(N_a, N_h) * 0.01
    b2 = np.random.randn(N_a, 1) * 0.01
    return W1, b1, W2, b2
#Carry out forward propagation in the NN
def forward_prop(X, W1, b1, W2, b2):
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = Z2
    return Z1. A1. Z2. A2
#Carry out backward propagation in the NN
def backward_prop(X, Y, Z1, A1, Z2, A2, W1, W2):
    m = X.shape[1]

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = relu_backward(dA1, Z1)
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m

    return dW1, db1, dW2, db2

    return dW1, db1, dW2, db2
#Update weights/ bias parameters using gradients and learning rate
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, eta):
    W1 -= eta * dW1
    b1 -= eta * db1
    W2 -= eta * dW2
    b2 -= eta * db2
    return W1, b1, W2, b2

# Relu activation function
def relu(Z):
    return np.maximum(0, Z)
#Gradient of Relu
def relu_backward(dA, Z):
    return dZ
#Initialise game
S, X, allowed_a = env.Initialise_game()
N_a = np.shape(allowed_a)[0]   # TOTAL NUMBER OF POSSIBLE ACTIONS

N_in = np.shape(X)[0]    ## INPUT SIZE
N_h = 200                ## NUMBER OF HIDDEN NODES

## INITALISE YOUR NEURAL NETWORK...
W1, b1, W2, b2 = initialize_NN(N_in, N_h, N_a)

# HYPERPARAMETERS SUGGESTED (FOR A GRID SIZE OF 4)

epsilon_0 = 0.2     # STARTING VALUE OF EPSILON FOR THE EPSILON-GREEDY POLICY
beta = 0.00005      # THE PARAMETER SETS HOW QUICKLY THE VALUE OF EPSILON IS DECAYING (SEE epsilon_f BELOW)
gamma = 0.85        # THE DISCOUNT FACTOR
eta = 0.0035        # THE LEARNING RATE

N_episodes = 100000 # THE NUMBER OF GAMES TO BE PLAYED

# SAVING VARIABLES
R_save = np.zeros([N_episodes, 1])
N_moves_save = np.zeros([N_episodes, 1])
```

APPENDIX

The code below shows our implementation of the training loop we used in the Q-Learning algorithm. Both algorithms share the same Neural Network. .

```
#Training Loop
for n in range(N_episodes):
    epsilon_f = epsilon_0 / (1 + beta * n)   ## DECAYING EPSILON
    Done=0                                   ## SET DONE TO ZERO (BEGINNING OF THE EPISODE)
    i = 1                                    ## COUNTER FOR NUMBER OF ACTIONS

    S,X,allowed_a=env.Initialise_game()      ## INITIALISE GAME

    while Done==0:                           ## START THE EPISODE
        #Forward prop for Q-values
        _, _, _, Qvaluesall = forward_prop(X.reshape(N_in, 1), W1, b1, W2, b2)
        #Call epislon greedy to make an action
        a_agent = EpGreedPol(Qvaluesall, epsilon_f, allowed_a)
        #Execute action and look at the next state, reward this action and then flag it as done
        S_next, X_next, allowed_a_next, R, Done = env.OneStep(a_agent)

        if Done==1:
            #Terminate episode and save number of moves
            R_save[n] = np.copy(R)
            N_moves_save[n] = np.copy(i)
            #Update QValue for action using reward
            Qvaluesall[a_agent] = R
            #Call back prop to update parameters
            dW1, db1, dW2, db2 = backward_prop(X.reshape(N_in, 1), Qvaluesall, *forward_prop(X.reshape(N_in, 1), W1, b1, W2,
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, eta)
            break
        else:
            #Forward Prop for the next state
            _, _, _, Qvaluesall_next = forward_prop(X_next.reshape(N_in, 1), W1, b1, W2, b2)
            #Run on greedy policy where epislon = 0
            a_agent_next = EpGreedPol(Qvaluesall_next, 0, allowed_a_next)
            #Update QValue by action observed reward and discounted Qvalue of next action
            Qvaluesall[a_agent] = R + gamma * Qvaluesall_next[a_agent_next]
            #backprop and update params
            dW1, db1, dW2, db2 = backward_prop(X.reshape(N_in, 1), Qvaluesall, *forward_prop(X.reshape(N_in, 1), W1, b1, W2,
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, eta)

        # NEXT STATE AND CO. BECOME ACTUAL STATE...
        S=np.copy(S_next)
        X=np.copy(X_next)
        allowed_a=np.copy(allowed_a_next)

        i += 1  # UPDATE COUNTER FOR NUMBER OF ACTIONS
    if n % 200 == 0:
        print(f"{n} Epochs Complete - Reward: {R_save[n]} - Number of Moves: {N_moves_save[n]}")
```

APPENDIX

The code below shows our implementation of the training loop we used in the SARSA algorithm. Both algorithms share the same Neural Network. .

```
#Training Loop
for n in range(N_episodes):
    epsilon_f = epsilon_0 / (1 + beta * n)   ## DECAYING EPSILON
    Done=0                                   ## SET DONE TO ZERO (BEGINNING OF THE EPISODE)
    i = 1                                    ## COUNTER FOR NUMBER OF ACTIONS

    S,X,allowed_a=env.Initialise_game()      ## INITIALISE GAME

    while Done==0:                           ## START THE EPISODE
        #Forward prop for Q-values
        _, _, _, Qvaluesall = forward_prop(X.reshape(N_in, 1), W1, b1, W2, b2)
        #Call epislon greedy to make an action
        a_agent = EpGreedPol(Qvaluesall, epsilon_f, allowed_a)
        #Execute action and look at the next state, reward this action and then flag it as done
        S_next, X_next, allowed_a_next, R, Done = env.OneStep(a_agent)

        if Done==1:
            #Terminate episode and save number of moves
            R_save[n] = np.copy(R)
            N_moves_save[n] = np.copy(i)
            #Update QValue for action using reward
            Qvaluesall[a_agent] = R
            #Call back prop to update parameters
            dW1, db1, dW2, db2 = backward_prop(X.reshape(N_in, 1), Qvaluesall, *forward_prop(X.reshape(N_in, 1), W1, b1, W2,
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, eta)
            break
        else:
            #Forward Prop for the next state
            _, _, _, Qvaluesall_next = forward_prop(X_next.reshape(N_in, 1), W1, b1, W2, b2)
            #Use epsilon_f instead of 0 for SARSA
            a_agent_next = EpGreedPol(Qvaluesall_next, epsilon_f, allowed_a_next)
            #Update QValue by action observed reward and discounted Qvalue of next action
            Qvaluesall[a_agent] = R + gamma * Qvaluesall_next[a_agent_next]
            #backprop and update params
            dW1, db1, dW2, db2 = backward_prop(X.reshape(N_in, 1), Qvaluesall, *forward_prop(X.reshape(N_in, 1), W1, b1, W2,
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, eta)

        # NEXT STATE AND CO. BECOME ACTUAL STATE...
        S=np.copy(S_next)
        X=np.copy(X_next)
        allowed_a=np.copy(allowed_a_next)

        i += 1  # UPDATE COUNTER FOR NUMBER OF ACTIONS
    if n % 200 == 0:
        print(f"{n} Epochs Complete - Reward: {R_save[n]} - Number of Moves: {N_moves_save[n]}")
```