# Reinforcement Learning

## I. Introduction

This paper discusses two different approaches to reinforcement learning; Q learning and Sarsa. These two approaches are tested using a deep neural network that is capable of learning how to perform check mate in a game of chess. The chess game itself consists of a 4x4 board with three pieces, a white queen and king and a black king that are placed randomly each time on the board. The aim of the game is for the white player to perform checkmate on it's opponents king. A standard game progresses in the following manner:

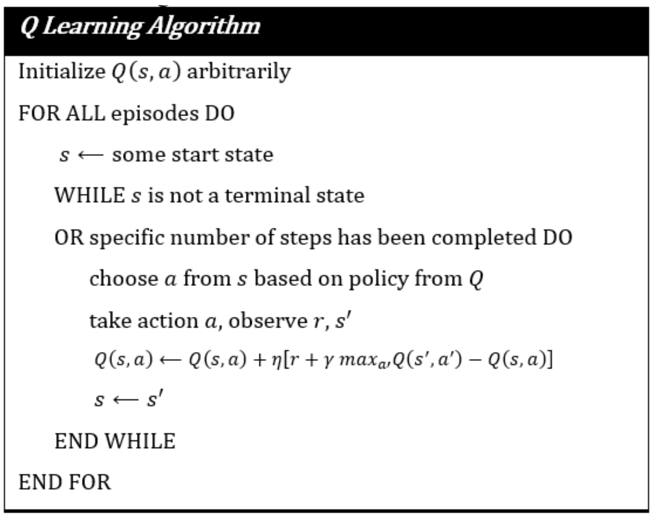
1. Player 1 performs an action and moves the King or the Queen aiming for a checkmate.
2. Player 2 performs a move by randomly moving the Opponent's King to a position so that it is not threatened by Player 1's King or Queen.
3. These two steps are repeated until:
   - It is the turn of Player 2 and the Opponent's King is currently threatened and any action will not make it avoid the threat (checkmate).
   - It is the turn of Player 2 and the Opponent's King is not currently threatened but any action will cause threat to it (draw).

## II. Q Learning and SARSA

Reinforcement learning makes use of functions that are widely known as value functions. These value functions are state action pairs that give an indication to how successful a given action will be from a given state. They can be described using the notation Q(s,a) where s represents the state, a the action and Q the Q value. The size of the Q value indicates how successful action a is when in states.

Both Q learning and Sarsa are temporal difference algorithms that allow an agent to learn from each action it takes within its environment. An agent's environment can be described as a world consisting of states, actions and rewards. Where each action observed for a given state results in a reward. Temporal difference learning methods work by calculating an estimation of the final reward for each step and subsequently updating a state-action value. They can broadly be categorized as either on-policy methods and off-policy methods. On policy methods allow agents to learn the value of the policy that is used to make a decision, whereas off policy methods allow agents to learn different policies for behavior and estimation.

Q learning is an example of an off-policy algorithm that uses an update rule to evaluate the Q values of a given action and choose the action in a greedy fashion. Figure 1 describes the process of Q learning. Before the process of learning begins Q is initialized to an arbitrarily fixed value. After this at each time step the agents selects an action a and observes a reward r before entering a new state s+1. During each iteration the Q value is updated using the weighted average of the old Q values and the new observed information. The learning rate $\eta$ is a value set between 0 and 1 and describes to what extent the Q values are updated at each iteration i.e. a factor of *0* means the agent learns nothing. The discount factor $\gamma$ is responsible for determining the importance of future rewards and again is set between 0 and 1. A higher discount factor means the agent will look for Q values that result in long term success. However, discount factors close to 1 can result in propagation of errors and instabilities that can lead to exploding weights. The variable *maxa* simply describes he maximum reward that is attainable in the state following the current one.



```
Q Learning Algorithm
Initialize $Q(s,a)$ arbitrarily
FOR ALL episodes DO
    $s \longleftarrow$ some start state
    WHILE $s$ is not a terminal state
    OR specific number of steps has been completed DO
        choose $a$ from $s$ based on policy from $Q$
        take action $a$, observe $r$, $s'$
        $Q(s,a) \longleftarrow Q(s,a) + \eta[r + \gamma\, max_{a'} Q(s',a') - Q(s,a)]$
        $s \longleftarrow s'$
    END WHILE
END FOR
```

*Figure 1: Pseudocode for Q Learning Algorithm*

Sarsa is an example of an on-policy algorithm that updates a policy based on actions that have been taken and is described in figure 2. Since SARSA does not explore new actions it is usually implemented alongside the ϵ-greedy policy. This policy allows SARSA to choose the action with the highest expected reward at first, and then have a probability of following the same action or another, random one, the rest of the time. The value of epsilon determines the exploration-exploitation of the agent, a large value means that a random action is less likely to be taken and so there is less exploration. Alternatively, a low epsilon values means the agent will choose random actions more frequently allowing it to explore its environment more. Typically, the epsilon policy is implemented with a discount factor, so that overtime there is less exploration allowing the Q values to converge to optimal values

```
SARSA Learning Algorithm
Initialize $Q(s,a)$ arbitrarily
FOR ALL episodes DO
    $s \longleftarrow$ some start state
    choose $a$ from $s$ based on policy from $Q$
    WHILE $s$ is not a terminal state
    OR specific number of steps has been completed DO
        take action $a$, observe $r, s'$
        choose $a'$ from $s'$ based on policy from Q
        $Q(s,a) \longleftarrow Q(s,a) + \eta[r + \gamma\, Q(s',a') - Q(s,a)]$
        $s \longleftarrow s'$
        $a \longleftarrow a'$
    END WHILE
END FOR
```

*Figure 2: Pseudocode for Sarsa Algorithm*

The only difference therefore between SARSA and Q learning then is that SARSA takes the next action based on the current policy whereas Q learning takes the action which maximises the utility of the following state. Q learning has the advantages that it learns the optimal policy when compared to SARSA since SARSA does not explore new actions, however this can be rectified using the aforementioned epsilon greedy strategy. There are some disadvantages though of using Q learning when compared to Sarsa. One of these is that Q learning has a higher per sample variance than SARSA which means that it may have difficulties converging. Another disadvantage is that Q learning can trigger unwanted rewards close to the optimal path when exploring whereas SARSA is more conservative and will approach convergence easier as it allows for possible penalties received from exploratory moves. To conclude SARSA is therefore the learning algorithm of choice when it comes to training an agent in an iterative environment.

## III. Q Learning Results

To run the Q learning algorithm on the chess problem, the number of neurons for the input layer, hidden layer and output layer needed to be specified first as backpropagation was required. The number of neurons for the input layer was 50, which is ((3N*N) +2), they represent the states. The output layer neurons were set to 32 as this specified the number of actions. This is 8*(N-1) for the queen and 8 for the king, where N is the size of the board (4). Following this the weights for the network were initialized using a uniform distribution and the biases were also initialized with zeros (see Appendix 1).

After the initializing of the network was complete, the Q learning algorithm was implemented alongside a method for computing the Q values (see Appendix 2 and 4). To increase exploration during the q learning the epsilon greedy policy was implemented which can be seen in Appendix 3. This policy was set up so that overtime the chance for a random move would decrease by using an epsilon discount factor.

In order to analyses the results of running the Q learning algorithm two graphs were plotted, one showing an exponential moving average of the reward per game and one showing a running average of the number of moves per game (Figure 3 and Figure 4). The calculation of these averages can be observed in Appendix 5. It was important that for the calculation of the exponential moving average that an alpha value close to 0 was implemented (0.0001) in order to smooth out the plot. The results showed that the rewards increased up to about 30000 episodes but from there remained fairly stable, this might have been because of the exploding gradients problem (see section 6). The running average of the moves per game steadily increased per game, this was because the epsilon discount factor was fairly low

(0.00005) and so exploration throughout the algorithm played a factor.

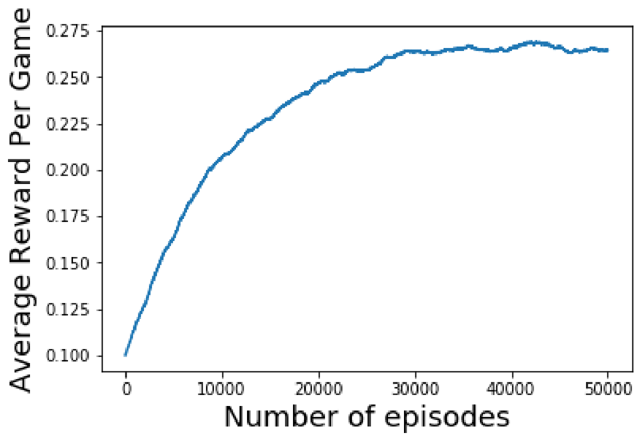


*Figure 3: Exponential moving average of rewards per game for Q learning*

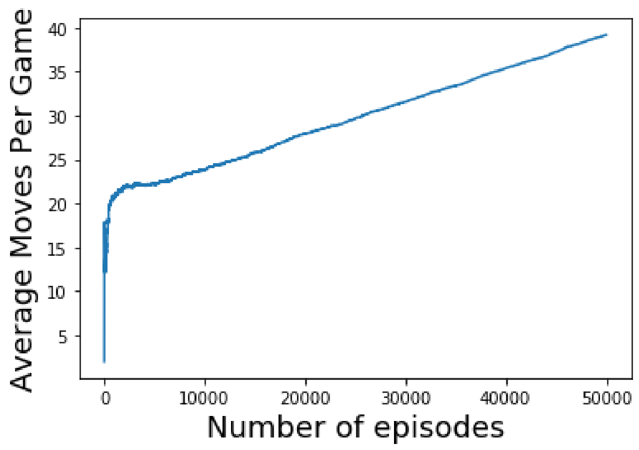


*Figure 4: Running average of moves per game*

## IV. Q LEARNING RESULTS AFTER CHANGING DISCOUNT FACTOR AND SPEED OF EPSILON GREEDY DECAY

Figures 5 and 6 show the results of running the Q learning algorithm again on the same initial setup but with a different discount factor and epsilon discount factor. The discount factor was decreased from and 0.85 to 0.1 and the epsilon discount factor increased to 0.0001 from 0.00005. As can be seen from figure 5 the rewards average looks similar to that witnessed previously, this implies that the network is not learning properly. This is because a lower discount factor should result in a graph that converges more slowly as only short-term rewards are considered. Increasing the epsilon discount factor meant that over time the network was selecting less random actions and as such the running average of moves per game is lower than that witnessed previously.

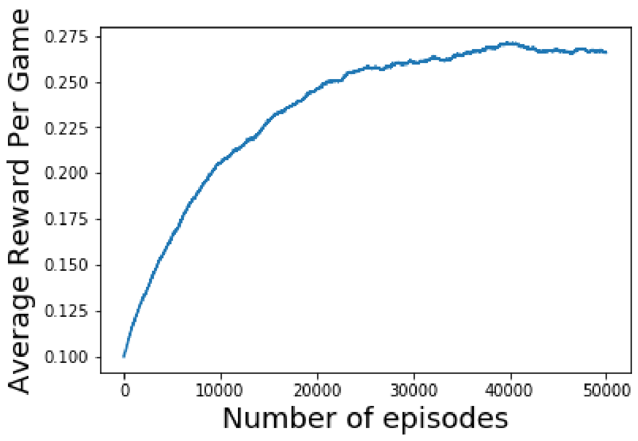


*Figure 5: Exponential moving average of rewards per game for Q learning*

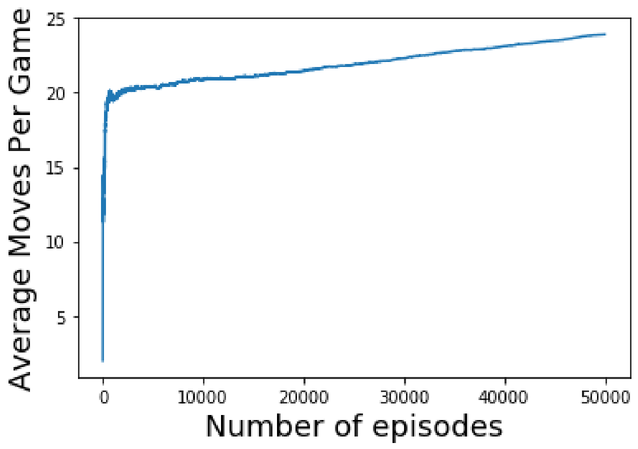


*Figure 6: Running average of moves per game*

## V. SARSA RESULTS

The SARSA version of the algorithm was implemented based on the pseudocode demonstrated by Figure 2. This algorithm was implemented in the same way as that for Q learning except the target (R+gamma*max( Q_{next})) was changed to reflect choosing Q_{next} by applying a policy, in this case epsilon-greedy (Appendix 7). The results of using SARSA can be shown by Figure 7. These results indicate that the SARSA implementation was not functioning correctly as it should be the case that SARSA converges quicker than Q learning towards rewards (see section 2). This would be easily demonstratable using a learning curve that compares the two.

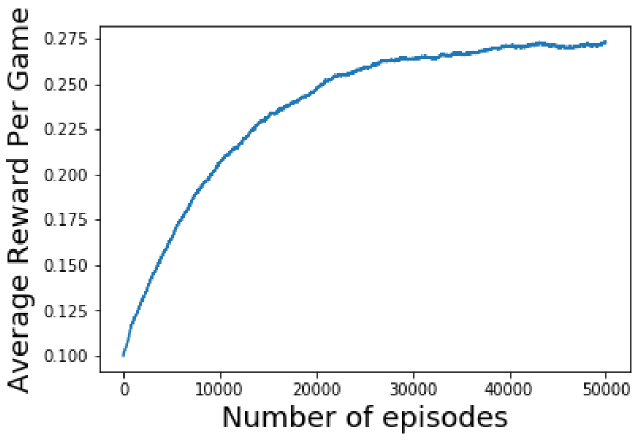


*Figure 7: Exponential moving average of rewards per game for SARSA learning*

## VI. RMSprop and Exploding Gradients

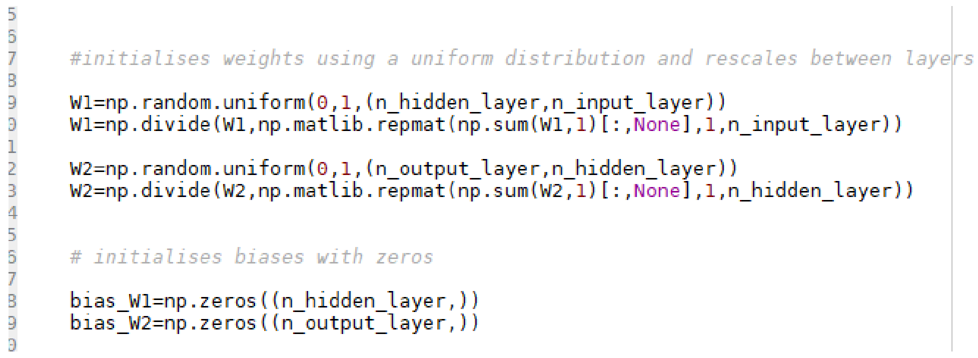
Exploding gradients are a problem that involves the weights during learning to become excessively high during training. This has the effect of making the learning unstable resulting in the agent not being able to effectively learn optimal paths. The explosion of weights is a result of the gradients being propagated through time and having to go through continued matrix multiplications causing them to grow exponentially. This eventually causes the gradients to explode as a result of numerical overflow resulting in NaN values, which hence crash the learning process. The exploding gradients problem is typically associated with artificial neural networks that involve gradient based learning methods and backpropagation such as this one.

A possible for the exploding gradients problem is RMSprop. This algorithm consists of a moving average of squared gradients that has the effect of decreasing the step size during gradient updates. It increases the step size for large gradients so as to avoid exploding and decreases the stop size for small gradients to avoid vanishing gradients (the opposite of the exploding gradients problem). A plot of the error loss function over time would be an appropriate way to show how using RMSprop solves the issue of exploding gradients.

## VII. Reproducing the Results

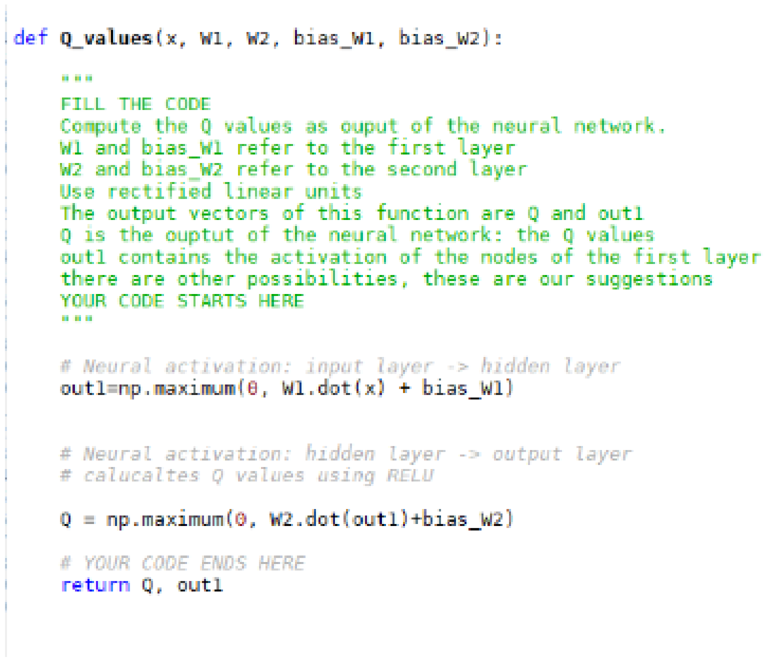
In order to reproduce results, NumPy seed value have been used. To mirror the results for the Q learning run the algorithm on seed 10 with a discount factor of 0.85 and epsilon discount factor of 0.00005, leave the rest unchanged. To reproduce the subsequent Q learning results change the discount factor to 0.1 and the epsilon discount factor to 0.0001 and re-run the code. The SARSA results can be obtained by running the chess_student_SARSA code unchanged.

# Appendices



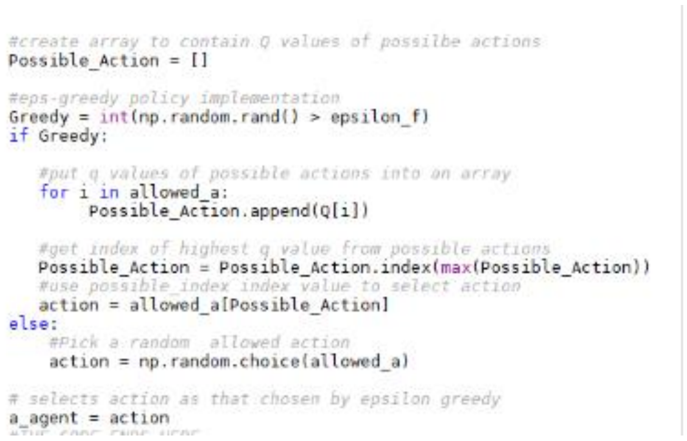
```
#initialises weights using a uniform distribution and rescales between layers

W1=np.random.uniform(0,1,(n_hidden_layer,n_input_layer))
W1=np.divide(W1,np.matlib.repmat(np.sum(W1,1)[:,None],1,n_input_layer))

W2=np.random.uniform(0,1,(n_output_layer,n_hidden_layer))
W2=np.divide(W2,np.matlib.repmat(np.sum(W2,1)[:,None],1,n_hidden_layer))


# initialises biases with zeros

bias_W1=np.zeros((n_hidden_layer,))
bias_W2=np.zeros((n_output_layer,))
```

*Appendix 1: Initialization of weights and biases*



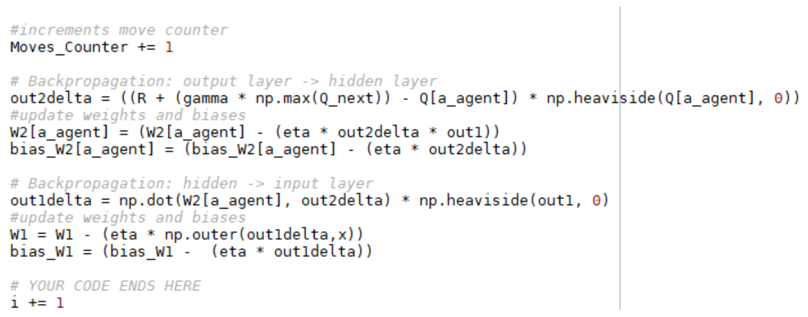
```
def Q_values(x, W1, W2, bias_W1, bias_W2):

    """
    FILL THE CODE
    Compute the Q values as ouput of the neural network.
    W1 and bias_W1 refer to the first layer
    W2 and bias_W2 refer to the second layer
    Use rectified linear units
    The output vectors of this function are Q and out1
    Q is the ouptut of the neural network: the Q values
    out1 contains the activation of the nodes of the first layer
    there are other possibilities, these are our suggestions
    YOUR CODE STARTS HERE
    """

    # Neural activation: input layer -> hidden layer
    out1=np.maximum(0, W1.dot(x) + bias_W1)


    # Neural activation: hidden layer -> output layer
    # calucaltes Q values using RELU

    Q = np.maximum(0, W2.dot(out1)+bias_W2)

    # YOUR CODE ENDS HERE
    return Q, out1
```

*Appendix 2: Q Values implementation*



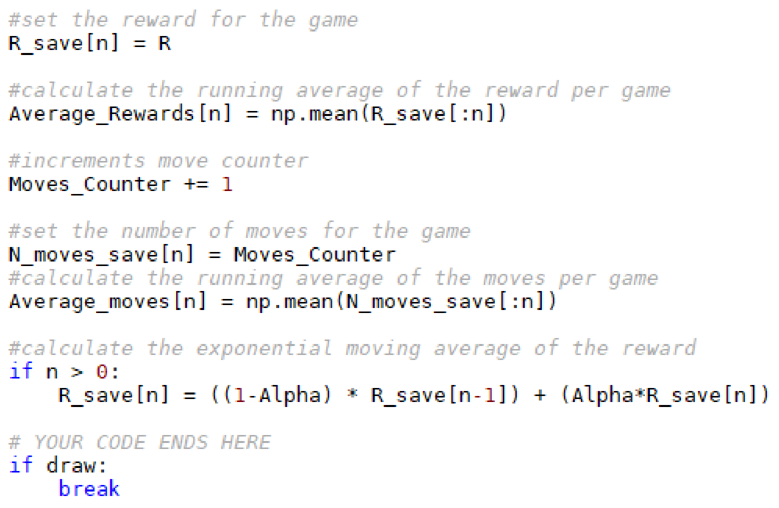
```
#create array to contain Q values of possilbe actions
Possible_Action = []

#eps-greedy policy implementation
Greedy = int(np.random.rand() > epsilon_f)
if Greedy:

    #put q values of possible actions into an array
    for i in allowed_a:
            Possible_Action.append(Q[i])

    #get index of highest q value from possible actions
    Possible_Action = Possible_Action.index(max(Possible_Action))
    #use possible_index index value to select action
    action = allowed_a[Possible_Action]
else:
     #Pick a random  allowed action
     action = np.random.choice(allowed_a)

# selects action as that chosen by epsilon greedy
a_agent = action
```

*Appendix 3: Epsilon Greedy implementation*



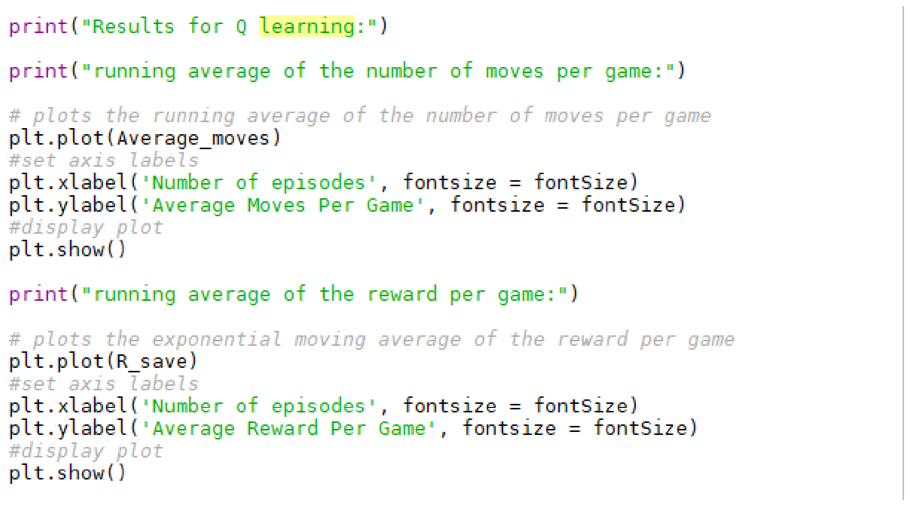
```
#increments move counter
Moves_Counter += 1

# Backpropagation: output layer -> hidden layer
out2delta = ((R + (gamma * np.max(Q_next)) - Q[a_agent]) * np.heaviside(Q[a_agent], 0))
#update weights and biases
W2[a_agent] = (W2[a_agent] - (eta * out2delta * out1))
bias_W2[a_agent] = (bias_W2[a_agent] - (eta * out2delta))

# Backpropagation: hidden -> input layer
out1delta = np.dot(W2[a_agent], out2delta) * np.heaviside(out1, 0)
#update weights and biases
W1 = W1 - (eta * np.outer(out1delta,x))
bias_W1 = (bias_W1 -  (eta * out1delta))

# YOUR CODE ENDS HERE
i += 1
```

*Appendix 4: Backpropagation implementation*



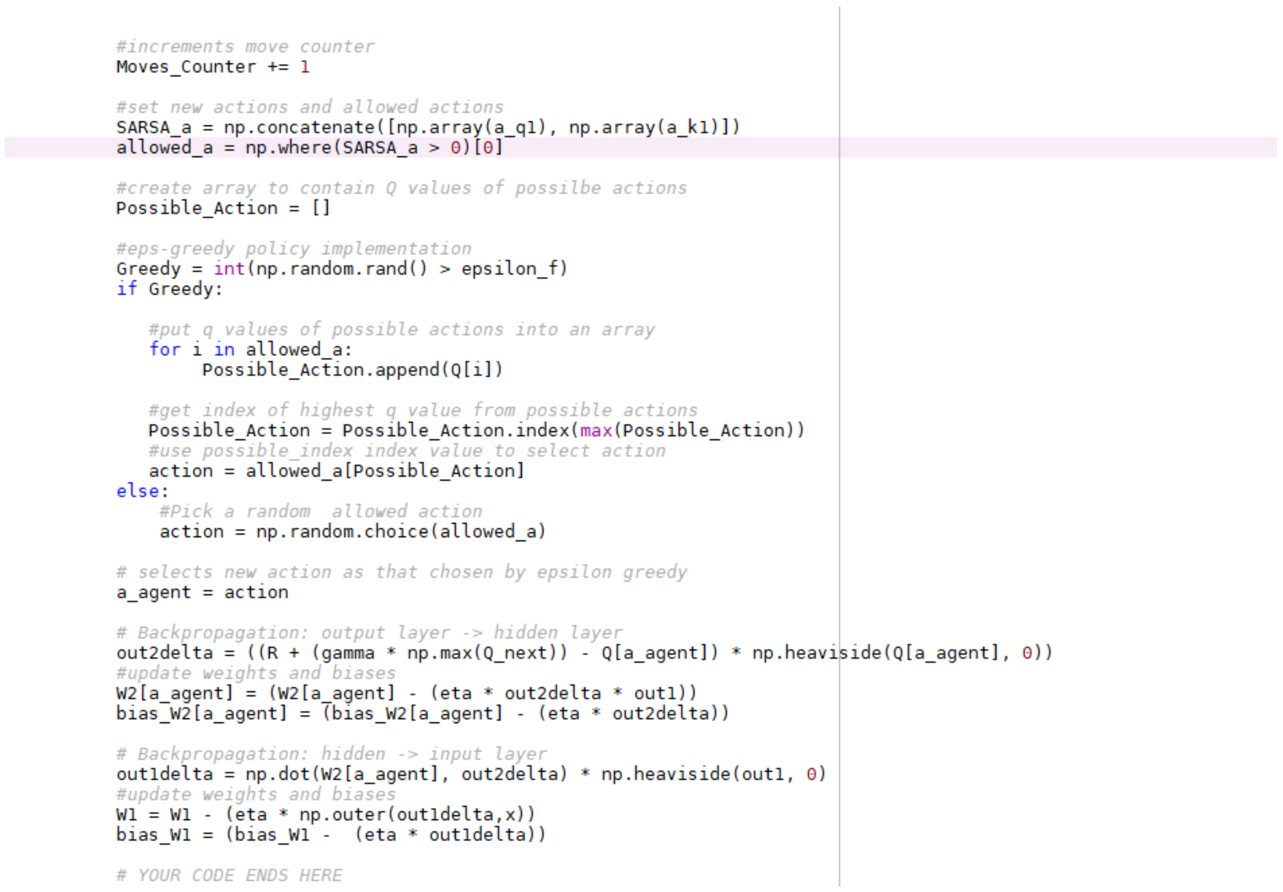
```
#set the reward for the game
R_save[n] = R

#calculate the running average of the reward per game
Average_Rewards[n] = np.mean(R_save[:n])

#increments move counter
Moves_Counter += 1

#set the number of moves for the game
N_moves_save[n] = Moves_Counter
#calculate the running average of the moves per game
Average_moves[n] = np.mean(N_moves_save[:n])

#calculate the exponential moving average of the reward
if n > 0:
    R_save[n] = ((1-Alpha) * R_save[n-1]) + (Alpha*R_save[n])

# YOUR CODE ENDS HERE
if draw:
    break
```

*Appendix 5: Calculation of average moves/rewards per game*



```
print("Results for Q learning:")

print("running average of the number of moves per game:")

# plots the running average of the number of moves per game
plt.plot(Average_moves)
#set axis labels
plt.xlabel('Number of episodes', fontsize = fontSize)
plt.ylabel('Average Moves Per Game', fontsize = fontSize)
#display plot
plt.show()

print("running average of the reward per game:")

# plots the exponential moving average of the reward per game
plt.plot(R_save)
#set axis labels
plt.xlabel('Number of episodes', fontsize = fontSize)
plt.ylabel('Average Reward Per Game', fontsize = fontSize)
#display plot
plt.show()
```

*Appendix 6: Plotting of running averages*



```
#increments move counter
Moves_Counter += 1

#set new actions and allowed actions
SARSA_a = np.concatenate([np.array(a_q1), np.array(a_k1)])
allowed_a = np.where(SARSA_a > 0)[0]

#create array to contain Q values of possilbe actions
Possible_Action = []

#eps-greedy policy implementation
Greedy = int(np.random.rand() > epsilon_f)
if Greedy:

    #put q values of possible actions into an array
    for i in allowed_a:
         Possible_Action.append(Q[i])

    #get index of highest q value from possible actions
    Possible_Action = Possible_Action.index(max(Possible_Action))
    #use possible_index index value to select action
    action = allowed_a[Possible_Action]
else:
     #Pick a random  allowed action
     action = np.random.choice(allowed_a)

# selects new action as that chosen by epsilon greedy
a_agent = action

# Backpropagation: output layer -> hidden layer
out2delta = ((R + (gamma * np.max(Q_next)) - Q[a_agent]) * np.heaviside(Q[a_agent], 0))
#update weights and biases
W2[a_agent] = (W2[a_agent] - (eta * out2delta * out1))
bias_W2[a_agent] = (bias_W2[a_agent] - (eta * out2delta))

# Backpropagation: hidden -> input layer
out1delta = np.dot(W2[a_agent], out2delta) * np.heaviside(out1, 0)
#update weights and biases
W1 = W1 - (eta * np.outer(out1delta,x))
bias_W1 = (bias_W1 -  (eta * out1delta))

# YOUR CODE ENDS HERE
```

*Appendix 7: SARSA implementation*