Deep SARSA and Deep Q-Learning

Yehia Ragab

I. Introduction

The link to the GitHub Repository is here.

Reinforcement Learning is a sub-field of Machine Learning where an agent learns to make optimal decisions by interacting with a defined environment, and observing the feedback, in the form of rewards and punishments, and using that feedback to improve future actions that maximise rewards and minimise punishments. Deep Reinforcement Learning represents a significant breakthrough in the field of Reinforcement Learning, as it leverages Neural Networks to enhance traditional Reinforcement Learning. This report focuses on Deep SARSA (State-Action-Reward-State-Action) and Deep Q-Learning. We will implement and test a Deep Q-Learning and a Deep SARSA agent and compare their algorithmic implementations and their performances in a controlled environment.

In the realm of Reinforcement Learning, Chess represents a huge challenge as it is a very complex game with many rules and permutations that it is impossible to solve through brute force. However, Chess endgames with 7 pieces or less have already been solved with optimal play, and the tablebase, called Szygy Endgames tablebase, is publicly available. For this assignment, we were tasked with solving a chess endgame in a 4x4 board involving 3 pieces, the 2 kings and a queen. The goal of this assignment is to analyse the differences in performance and time to convergence of Deep SARSA and Deep Q-Learning.

In this report, we will explain what Deep Q-Learning and Deep SARSA are, explain our methodology for our implementation, and present and analyse our results.

II. Methodology

A. Deep SARSA and Deep Q-Learning

Deep SARSA and Deep Q-Learning are deep reinforcement learning algorithms that are very similar in concept. We will first discuss the similarities, then we will explain the differences.

1) Similarities: Both algorithms involve initialising the weights and biases of the neural network with random numbers between 0 and 1. The dimensions of the weight of the neural network are as follows: (i, h), (h, o), where i is the dimension of the input, in our case, the features X, h is the number of hidden nodes, and o the dimensions of the output, in our case the number of possible actions for the agent king and queen.

The environment is initialised for each episode, which is defined by making the agent play the game once. The initial state of the environment is defined as s. We run forward propagation on our neural network with s as our input. The output of the neural network is an array of **Q-values**, which are estimations of future rewards for each corresponding action. Using our chosen policy, which in our case is **ϵ-greedy**, we pick the first action a. ϵ defines the percentage of times the algorithm will pick the best possible action based on the **Q-values**, compared to the percentage of times the algorithm will pick a random action based on all possible actions. The agent performs that action, and observes R, the reward, and s', the new state. We define the **target Q-values** as a copy of the **Q-values** that were output by the neural network. If the s' state is an end-state, meaning either the agent won, or drew, we set the **a-th** value of target **Q-values** to the reward R observed. Otherwise, we run forward propagation on s' and select the optimal action a' based on a policy, which is different for Q-Learning and SARSA. We get the predicted Q-Value of a', defined as **Q-value**[a'] and set the **a-th** value of **target Q-values** to $R + \gamma$ **Q-value**[a']. We then compute the **error**, defined as **target Q-values - Q-values**. We then run backward propagation and update the weights and biases of the neural network. We repeat the process until the game reaches an endstate, with a' becoming a and s' becoming s.

2) Exploration vs Exploitation: The difference between Q-Learning and SARSA, as explained before, is in the policy used to pick the best possible action for all states other than the initial state s. For SARSA, we use an **ϵ-greedy** policy to choose the action we will use when computing the error, which means that we don't always use the action that maximises the **Q-values** that were output by the neural network for backpropagation. For Q-Learning, after making the initial action a, we always pick the optimal action corresponding to the highest **Q-value** for backpropagation, meaning we use a greedy policy. Deep SARSA is said to be an on-policy method, meaning it uses the chosen initial policy to dictate the future action for each state in an episode, while Q-Learning is said to be an off-policy algorithm, meaning that after making the initial action a, the algorithm picks the best possible action based on a greedy policy.

The main advantage of Q-learning is that it has the potential to converge more quickly than SARSA, as it explores less than SARSA. However, the disadvantage is that it may converge to a local optimum instead of a global optimum.

III. Implementation details

A. Neural Network

Our neural network used the suggested dimensions, that is to say there is 1 hidden layer with 200 hidden nodes.

B. Forward propagation

For our forward propagation function, we used ReLU activation. We computed the dot product of X, our features, and $W1$, the weights of the first layer, and added the biases $b1$. We then applied the ReLU activation function **h_activation**, then computed the **Q-values** by getting the dot product of **h_activation** and $W2$, then added $b2$.

C. Backward propagation

For our forward propagation function, we used X, the **error** obtained, and **h_activation**. We then computed the outer product of **h_activation** and **delta2**, (the error) , which yield **dW2**. We updated the **W2** and **b2**., then applied the inverse of the ReLU function, which involves taking the heaviside of **h_activation** and multiplying it by the dot product of **W2** and **delta2**, which yields **delta1**. We then compute the outer product of **X** and **delta1** to get **dW1**. We then update the .

D. Rotating the board

In chess, there is always at least 1 square between both kings, both horizontally and vertically. With our chess board being 4x4, that means that if we divide the board into 4 2x2 squares, the kings will always be on different quarters. Furthermore, all our pieces are able to move in all directions in equal measure, so the orientation of the board doesn't change anything. The optimal move in a board with the state s is the mirror move of the optimal move of a board with the state s_{mirror}. The same logic applies to a rotated or transposed board. For this reason, we implemented as function that ensures the agent's king is always in the top left quarter of the board by rotating the board until this requirement is satisfied. Since the enemy king can't be in the same quarter, the king will be in one of the remaining quarters. We can ensure that enemy king will be on one of the quarters on the right (excluding the coordinate (3,2)) by transposing the board if it isn't already the case. This makes it so that the boards are all normalised and the number of possible states drastically decreases. This rotating function is applied after every move is made 8.

E. Features

After normalising the board, there are 4 possible spots for the agent's king, and 7 possible spots for the enemy king, which means we can reduce the number of features from the original 58 to 37, with the first 4 numbers indicating the position of the agent's king, the next 16 numbers indicating the position of the agent's queen, and the next 7 numbers indicating the position of the enemy king. We removed the last 10 numbers which indicated whether the enemy king was in check and the allowed moves the enemy kings can make.

F. Optimizer

For our optimizer, which is used to update the weights and biases of the neural network, we used a modified version of the Adam optimizer 9 that was in the lab, with $beta1 = 0.9$, $beta2 = 0.999$, and $\epsilon = 1 \times 10^{-8}$.

G. Code organisation

The Agent is defined in a class called Agent. The class is initialised with all the necessary parameters 10. This class works for both the Q-Learning implementation and the SARSA implementation. The class has the following functions:

- **get_action**: This function takes the state, the policy and allowed_a as inputs and returns an action that depends on the policy and all the possible actions. The functions returns the chosen action, the Q-values, of the state, and h_activation 11.
- **forward_prop**: This function takes the state as input and returns the Q-values and h_activation 11.
- **backward_prop**: This function rakes in the state, the error, and h_activation as input and updates the weights 11.
- **episode**: This function runs an episode of the training loop and returns the reward and number of steps of the episode. It also prints the average reward and average number of steps over the previous 100 episodes which happens every 100 episodes 12.
- **run_training_loop**: This function takes in the number of episodes one would like to run the training loop for and runs the training loop. It also returns the average reward and average number of steps of the training loop.
- **testing_episode**: This function runs 1 testing episode, which is an episode where epsilon is set to 0 and there is no backpropagation.
- **test**: This function takes in the number of episodes one would like to test, and prints the average Reward and number of steps of the trained agent.

H. Parameters

1) ϵ_f and β: ϵ_f is the value of ϵ that is chosen for the ϵ-greedy policy. It is set to decay for every episode that is run so that over time, the agent reduces the amount of exploration it does in favour of exploitation. The formula for ϵ_f is:

$$\epsilon_f = \frac{\epsilon}{1 + (\beta * n)} \tag{1}$$

where ϵ is the initial value chosen, β being the decaying rate, and n being the number of the episode. Increasing β would reduce the amount of exploration the agent does over time, as ϵ_f would decay quicker. Although it might converge quicker, it also make the agent more prone to being stuck in a local optimum. Decreasing β would increase the amount of exploration compared to smaller β over time which decreases the likelihood of being stuck in a local optimum, but it also increases the amount of episodes needed to find the global optimum, which is the goal of the agent.

2) γ: γ, also known as the discount factor, plays a crucial role in the Bellman equation, which is used to calculate the error. The Bellman equation used in the code if the environment isn't in an endstate is as follows:

$$TargetQvalue[action] = R + \gamma \times Qvalues_{next}[action_{next}] \tag{2}$$

A small γ makes the agent more concerned with immediate rewards compared to future rewards, while a big γ makes the agent more concerned with future rewards compared to immediate rewards. In many cases, a bigger γ tends to increase variance and make the learning process unstable.

I. Plots

All plots were created using matplotlib. For each plot, 3 different agents with the same parameters were trained separately, then final rewards were the average of the reward at each episode, and the exponential moving average function from pandas was used on the averaged rewards. The alpha chosen was 0.0025, which is the value that struck the best balance between reducing noise and showing the trend. All agents were trained on 10,000 episodes, and in every plot, the blue line represents the Q-Learning agent and the green line represents the SARSA agent.

IV. Results and Discussion

A. Initial Implementation

The initial implementation for this assignment was Deep Q-Learning and Deep SARSA with the suggested parameters which are:

- $hidden\_layers = 1$
- $hidden\_nodes = 200$
- $\epsilon_0 = 0.2$
- $\beta = 0.00005$
- $\gamma = 0.85$
- $learning\_rate = 0.0035$

As we can see from the plots 1, the performance and convergence of both agents is very similar. The main difference here is that the SARSA agent makes fewer moves per game on average at the start.

B. Effects of changing β and γ

We tested the agent with 3 different values of β and 3 different values of γ. The values of γ we tested were 0.75, 0.85 (base), and 0.95. The values of β we tested were 0.00005 (base), 0.0001, and 0.00025.

1) β: The features implemented decrease the number of episodes needed for convergence, which makes the default β value a bottleneck in our implementation. For this reason, we decided to only test higher β values. For both plots 2 and 3, the agent converges quicker than the agents with the base parameters. The average number of moves on average is lower for SARSA in both cases.

2) γ: The average reward for the Q-Learning agent is higher than the average reward for the SARSA agent for the first 5,000 episodes when we lower γ to 0.75, as seen in 4. This can be because with a lower γ, the SARSA agent has fewer opportunities to update the Q-Values for the optimal action, meaning it converges slower.

When we increase γ to 0.95, the inverse happens. The average reward for the SARSA agent is higher than the average reward for the SARSA agent for the first 5,000 episodes, as seen in 5. The average number of moves is also lower for the SARSA agent.

However, in both cases, the difference isn't considerable when comparing them to the agents trained with the base parameters and when comparing both agents with each other.

C. Additional Experiments

1) Original Features vs Rotated board: The plots 6 show that the altered features perform significantly better than the original features. Each episode also involves less moves. Additionally, the altered features run in half the time of the original features due to the considerably smaller input vector (27 vs 58). These results were expected because the agent has fewer permutations to learn from, and is therefore able to find the optimal solution faster.

2) Finding a better agent: We decided to experiment further to try and find a better performing agent. To do so, we tried to increase β to 0.0005, the learning rate to 0.005, and decreased ϵ to 0.175. We were able to get the best performing agent both in terms of convergence speed and average reward 7.

V. Conclusion

In this environment, the performance of Deep Q-Learning and Deep SARSA is very similar. This might be due to the simplicity of the task at hand. We were able to train agents that converge in a small number of episodes and reach the optimal reward. We looked at the effect of changing certain parameters and changing the shape of the environment and input features.

Appendix A: Plots

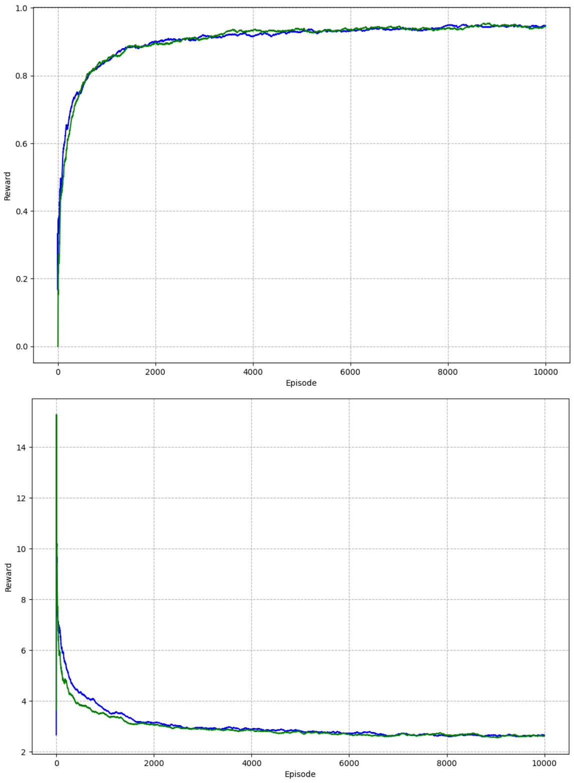


Fig. 1. Exponential Moving Average Reward and Number of steps for the agents with the base parameters.

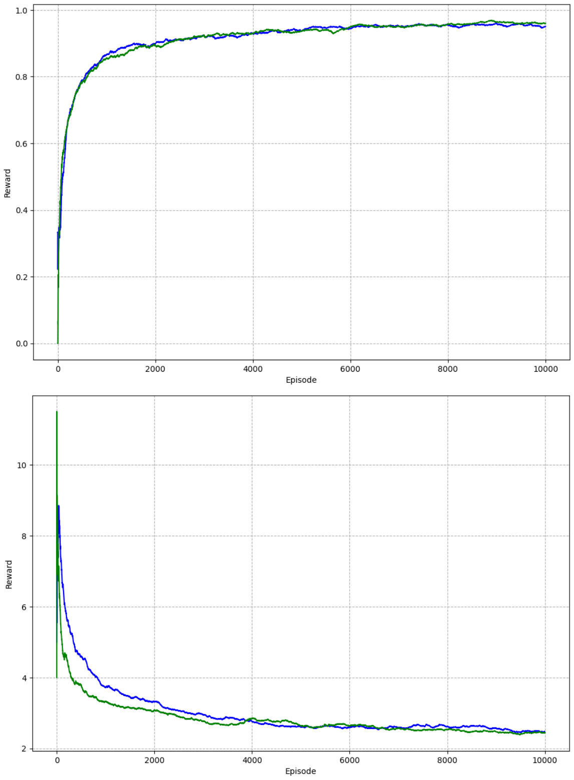


Fig. 2. Exponential Moving Average Reward and Number of steps for the agents with the base parameters with $\beta = 0.0001$.

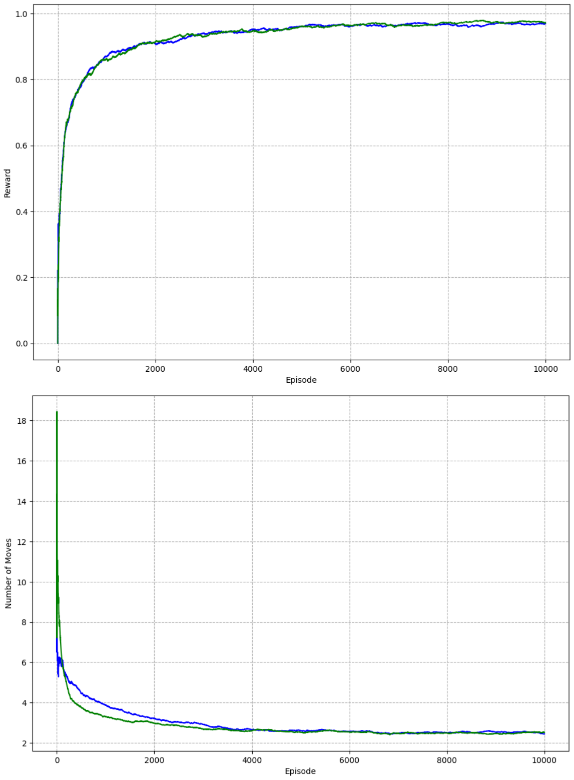


Fig. 3. Exponential Moving Average Reward and Number of steps for the agents with the base parameters with $\beta = 0.00025$.

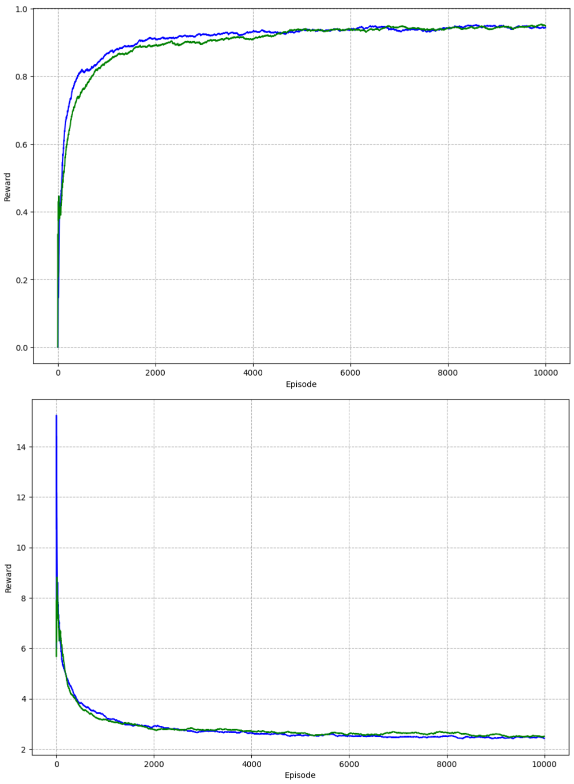


Fig. 4. Exponential Moving Average Reward and Number of steps for the agents with the base parameters with $\gamma = 0.75$.

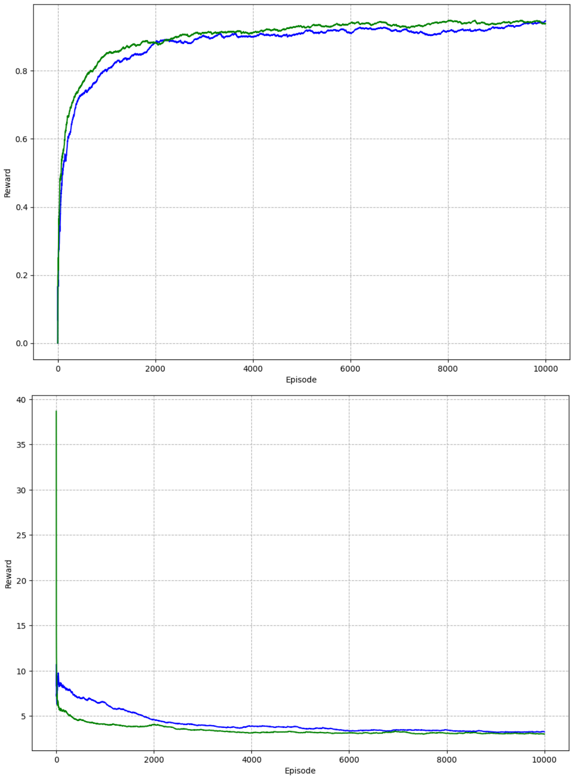


Fig. 5. Exponential Moving Average Reward and Number of steps for the agents with the base parameters with $\gamma = 0.95$.

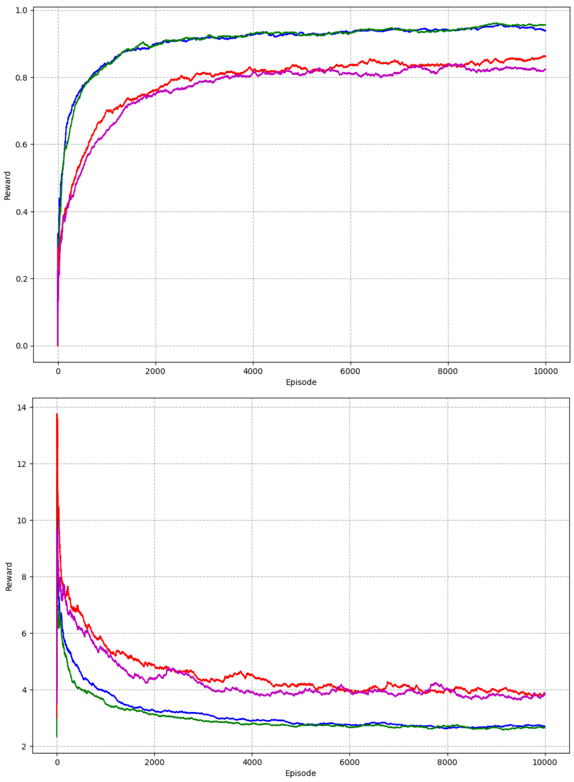


Fig. 6. Comparison between original features and altered features with rotated board. Red represents the Q-Learning Agent with the original features and magenta represents the SARSA agent with the original features.

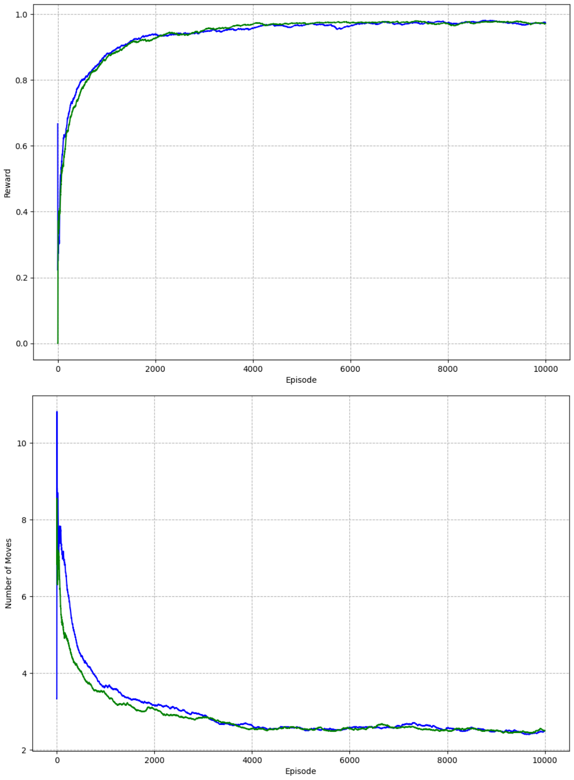
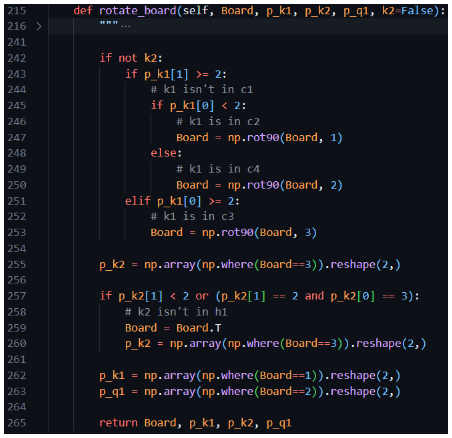


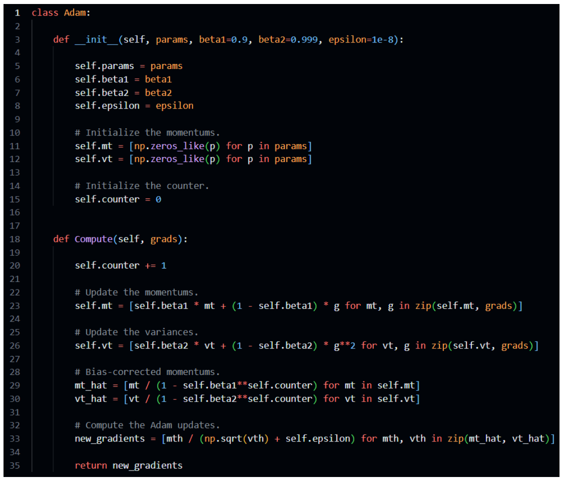
Fig. 7. The best agent found.

APPENDIX B: CODE



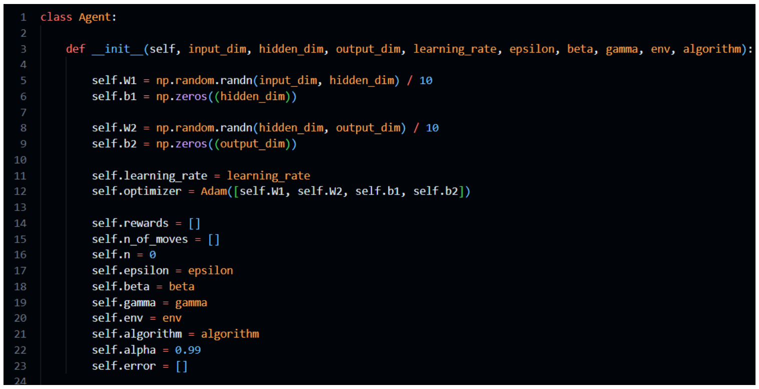
```python
    def rotate_board(self, Board, p_k1, p_k2, p_q1, k2=False):
        """...

        if not k2:
            if p_k1[1] >= 2:
                # k1 isn't in c1
                if p_k1[0] < 2:
                    # k1 is in c2
                    Board = np.rot90(Board, 1)
                else:
                    # k1 is in c4
                    Board = np.rot90(Board, 2)
            elif p_k1[0] >= 2:
                # k1 is in c3
                Board = np.rot90(Board, 3)

        p_k2 = np.array(np.where(Board==3)).reshape(2,)

        if p_k2[1] < 2 or (p_k2[1] == 2 and p_k2[0] == 3):
            # k2 isn't in h1
            Board = Board.T
            p_k2 = np.array(np.where(Board==3)).reshape(2,)

        p_k1 = np.array(np.where(Board==1)).reshape(2,)
        p_q1 = np.array(np.where(Board==2)).reshape(2,)

        return Board, p_k1, p_k2, p_q1
```

Fig. 8. The function that rotates the board.



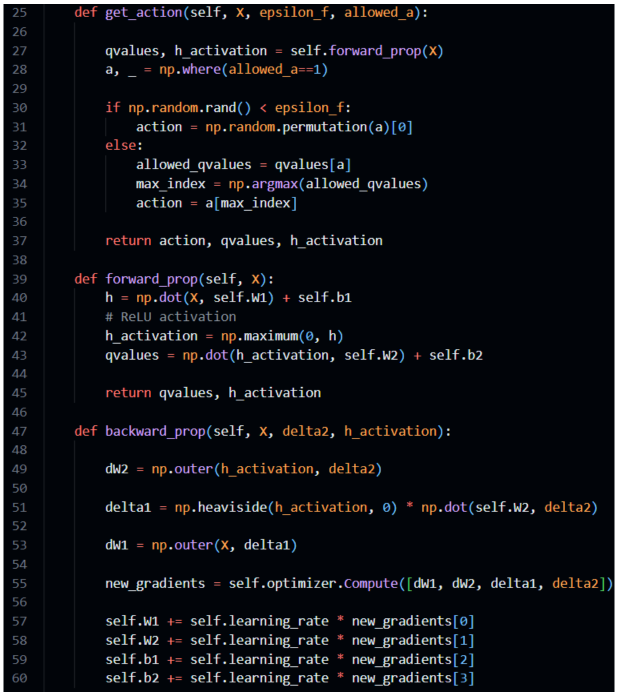
```python
class Adam:

    def __init__(self, params, beta1=0.9, beta2=0.999, epsilon=1e-8):

        self.params = params
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon

        # Initialize the momentums.
        self.mt = [np.zeros_like(p) for p in params]
        self.vt = [np.zeros_like(p) for p in params]

        # Initialize the counter.
        self.counter = 0


    def Compute(self, grads):

        self.counter += 1

        # Update the momentums.
        self.mt = [self.beta1 * mt + (1 - self.beta1) * g for mt, g in zip(self.mt, grads)]

        # Update the variances.
        self.vt = [self.beta2 * vt + (1 - self.beta2) * g**2 for vt, g in zip(self.vt, grads)]

        # Bias-corrected momentums.
        mt_hat = [mt / (1 - self.beta1**self.counter) for mt in self.mt]
        vt_hat = [vt / (1 - self.beta2**self.counter) for vt in self.vt]

        # Compute the Adam updates.
        new_gradients = [mth / (np.sqrt(vth) + self.epsilon) for mth, vth in zip(mt_hat, vt_hat)]

        return new_gradients
```

Fig. 9. The implementation of Adam.



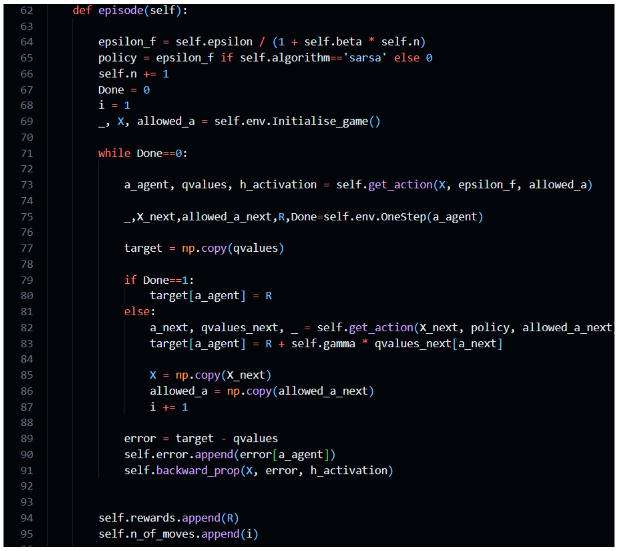
```python
class Agent:

    def __init__(self, input_dim, hidden_dim, output_dim, learning_rate, epsilon, beta, gamma, env, algorithm):

        self.W1 = np.random.randn(input_dim, hidden_dim) / 10
        self.b1 = np.zeros((hidden_dim))

        self.W2 = np.random.randn(hidden_dim, output_dim) / 10
        self.b2 = np.zeros((output_dim))

        self.learning_rate = learning_rate
        self.optimizer = Adam([self.W1, self.W2, self.b1, self.b2])

        self.rewards = []
        self.n_of_moves = []
        self.n = 0
        self.epsilon = epsilon
        self.beta = beta
        self.gamma = gamma
        self.env = env
        self.algorithm = algorithm
        self.alpha = 0.99
        self.error = []
```

Fig. 10. Initialising an Agent.



```python
    def get_action(self, X, epsilon_f, allowed_a):

        qvalues, h_activation = self.forward_prop(X)
        a, _ = np.where(allowed_a==1)

        if np.random.rand() < epsilon_f:
            action = np.random.permutation(a)[0]
        else:
            allowed_qvalues = qvalues[a]
            max_index = np.argmax(allowed_qvalues)
            action = a[max_index]

        return action, qvalues, h_activation

    def forward_prop(self, X):
        h = np.dot(X, self.W1) + self.b1
        # ReLU activation
        h_activation = np.maximum(0, h)
        qvalues = np.dot(h_activation, self.W2) + self.b2

        return qvalues, h_activation

    def backward_prop(self, X, delta2, h_activation):

        dW2 = np.outer(h_activation, delta2)

        delta1 = np.heaviside(h_activation, 0) * np.dot(self.W2, delta2)

        dW1 = np.outer(X, delta1)

        new_gradients = self.optimizer.Compute([dW1, dW2, delta1, delta2])

        self.W1 += self.learning_rate * new_gradients[0]
        self.W2 += self.learning_rate * new_gradients[1]
        self.b1 += self.learning_rate * new_gradients[2]
        self.b2 += self.learning_rate * new_gradients[3]
```

Fig. 11. get_action, froward_prop, and backward_prop.



```python
    def episode(self):

        epsilon_f = self.epsilon / (1 + self.beta * self.n)
        policy = epsilon_f if self.algorithm=='sarsa' else 0
        self.n += 1
        Done = 0
        i = 1
        _, X, allowed_a = self.env.Initialise_game()

        while Done==0:

            a_agent, qvalues, h_activation = self.get_action(X, epsilon_f, allowed_a)

            _,X_next,allowed_a_next,R,Done=self.env.OneStep(a_agent)

            target = np.copy(qvalues)

            if Done==1:
                target[a_agent] = R
            else:
                a_next, qvalues_next, _ = self.get_action(X_next, policy, allowed_a_next
                target[a_agent] = R + self.gamma * qvalues_next[a_next]

                X = np.copy(X_next)
                allowed_a = np.copy(allowed_a_next)
                i += 1

            error = target - qvalues
            self.error.append(error[a_agent])
            self.backward_prop(X, error, h_activation)


        self.rewards.append(R)
        self.n_of_moves.append(i)
```

Fig. 12. Training episode.