Mastering Simplified Chess Endgames with Reinforcement Learning

Source code available at https://github.com/timZurichLoaf/Reinforcement_learning_chess_endgame

Zheng Luo
Faculty of Business, Economics and Informatics
University of Zurich
Zurich, Switzerland
zheng.luo@uzh.ch | UZH ID: 21-738-901

***Abstract*—When we reflect on how an infant learns to play, walk and not touch a hot radiator, there is no explicit teacher. It reveals the very nature of learning process, to learn by interacting with the environment. Reinforcement learning is a computational approach mimicking the interactive learning. This study will review the two mainstream Temporal Difference (TD) learning methods, Q-learning and State–action–reward–state–action (SARSA), along with experience replay technique from a theoretical perspective. A simplified chess endgame will be then used as a sample task to examine the TD learning methods and the dominant role of goal, or the administration of rewards, in shaping the behavior of an agent. Deep reinforcement learning algorithm and the effect of hyper-parameters will also be analyzed in detail.**

I. Introduction

A TD agent relies on the estimation of rewards of state-action pairs (Q values) to decide its moves. As it explores the environment by trial-and-error, the estimation is updated based on its observation. Conventional TD methods face the sequential dependency. The experience replay technique first studied by [Lin92] is a successful attempt to cope with it. In the simplified 'over-killing' endgame on a 4-by-4 chessboard, a reinforcement learning agent learns to checkmate. Despite the small chessboard, all conventional chess rules specified in [Sch03] apply. SARSA and Q-learning are used in training a neural network to approximate Q-values. As an agent plays, moves per game and rewards received in each episode are recorded to evaluate the models. The effect of hyper-parameters is explored theoretically and empirically with SARSA in light of the trade-off between exploration and exploitation. Finally, the administration of reward is modified to train an impatient agent eager to end a game as soon as possible, by punishing long games and neutralizing the outcome.

II. Methods

A. Theoretical Review of TD methods, Q-learning and SARSA[i]

Q-learning and SARSA are two TD learning methods, which combine the idea of Monte Carlo and that of dynamic programming (DP). TD resembles Monte Carlo method in a way that an agent learns directly from raw experience without much a posteriori knowledge or a given model of the environment's dynamics. Similar to DP, TD updates estimations based on existing learning outcomes, meaning that neither of them waits until the final reward is available.

B. Comparison between Q-learning and SARSA[i]

Both as TD methods, the primary difference between Q-learning and SARSA lies in the approaches to estimate Q values, namely on-policy and off-policy. SARSA as an on-policy method, estimates $q_\pi(s,a)$ of current state-action pair (s,a) by its immediate reward and the subsequent $q_\pi(s',a')$ following a given policy π. Intuitively, a SARSA agent guesses the reward of current state-action pair based on its guess of the next. Q-learning, on the other hand, as an off-policy method, estimates $q_\pi(s,a)$ by considering the optimal outcome of all possible actions $max_a q_\pi(s',a)$, given a following state s'. Thus, Q-learning directly approximates the optimal action-value function $q*$, regardless of the policy. Besides, whenever a rewarding move has been discovered, the information propagates fast to its preceding state-action pairs.

Later in this study, an over-killing chess endgame experiment shows that a Q-learning agent accumulates fewer rewards and takes more moves per game than a SARSA agent, as a rewarding move may drive it to explore some costly intermediate moves. However, in the demonstration game, the Q-learning agent adopts an optimal strategy, while SARSA sticks to a cautious time-consuming approach.

C. Introduction of Experience Replay[ii]

Experience replay was first studied by [Lin92]. When the agent executes each action, a quadruple $(S_t, A_t, R_{t+1}, S_{t+1})$ is stored in the replay memory of a fixed size to accumulate experiences over time. At each time step, Q-learning updates a mini-batch by sampling uniformly from the replay memory

[i]Answer to Task 1: Describe Q-learning, SARSA and explain their differences.

[ii]Answer to Task 2: Describe the experience replay technique and cite relevant papers.

and then proceeds according to a given policy. The experience replay complements the conventional Q-learning by making good use of its off-policy nature, which is independent of connected trajectories. Uncorrelated successive updates contribute to a reduction of the variance. Nevertheless, [Lin92] points out that experience replay is applicable to a static environment, where the rewards almost don't change. Otherwise, past experience may become irrelevant or even harmful.

[Mni+13; Mni+15] extend the idea of [Lin92] by introducing a second network and holding the second network's weights constant for a certain number C of updates. The output of this duplicate network with temporarily constant weights serves as the Q-learning target during C updates and leads to the following expression,

$$\begin{aligned} w_{t+1} = w_t + \alpha[R_{t+1} + \gamma \max_a Q^n(S_{t+1}, a, \tilde{w_t}) \\ - Q^n(S_t, A_t, w_t)]\nabla Q^n(S_t, A_t, w_t) \end{aligned} \tag{1}$$

Thus experience replay brings Q-learning closer to simpler supervised learning and speeds up the learning process.

D. Implementation of SARSA and Q-learning for 'over-killing' endgames

1) Environment setup: In the 'over-killing' endgames on a 4-by-4 chessboard, two white pieces, a king and a queen, are controlled by the agent and a black king by a random agent. Each episode of the game is initialized in a state where the opponent king is not threatened, as demonstrated in Figure 1.

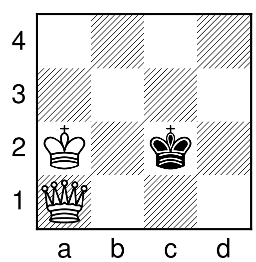

Fig. 1. An example of the initial state of an episode, where the opponent king (black) is not threatened.

2) Artificial Neural Network: An artificial neural network (ANN) with 1 hidden layer of 200 units is used to approximate the true Q-value functions. A default learning rate η is set to 0.0035 to avoid overshooting in gradient descent of back-propagation. To combat vanishing gradient, Rectified Linear Unit (ReLU) is applied to both the hidden layer and the output layer. The default architecture of the ANN is demonstrated in Table I.

TABLE I
DEFAULT ARCHITECTURE OF THE ANN

Layer	*Input*	*Hidden*		*Output*	
Feature	#dimension	#neurons	Activation	#dimension	Activation
Value	58	200	ReLU	32	ReLU

3) SARSA: In the course of learning, an ϵ-greedy policy is adopted with $\epsilon = 0.2$ to encourage early exploration. A decay factor $\beta = 5 \cdot 10^{-5}$ reduces the actual ϵ_n used in the n^{th} episode to $\frac{\epsilon}{1+\beta \cdot n}$ to facilitate exploitation of accumulated knowledge in the late stage.

Given a current state-action pair (s, a), a SARSA agent anticipates the next move a' according to the specified policy and trains the ANN with the observed reward r and the Q-value of the next move $q(a')$. If s is a terminal stage of the episode, then the agent considers only r. A reward $r = 1$ is given in case of checkmate, otherwise, $r = 0$.

4) Q-learning: Unlike SARSA, in non-terminal scenarios, a Q-learning agent speculates on the seemingly most rewarding next move a^* among all feasible moves and updates the neural network with the observed reward r and Q-value of this anticipated best move $q(a^*)$. However, a^* may not be the move to take. The Q-learning agent takes the next move a' according to the same ϵ-greedy policy as SARSA and proceeds to an immediate subsequent state-action pair (s', a'), which is the core logic of the off-policy learning method.

To curb the exploding weights of the neural network, Sigmoid activation is applied to the output layer. ReLU activation remains in use for the hidden layer.

5) Change of the administration of reward: With the default game setup and SARSA learning method, a negative unit reward $r = -1$ is imposed on each move taken, but no reward is granted at the terminal state, regardless of the result.

III. RESULTS

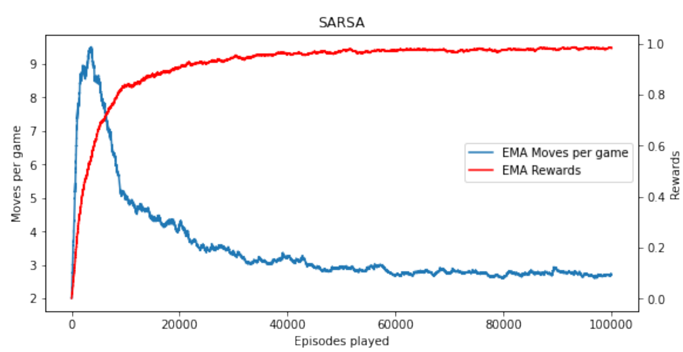

Fig. 2. Exponential Moving Average (EMA) moves per game and rewards received by a SARSA agent over 100k episodes.

A. Learning with SARSA[iii]

As a SARSA agent learns to checkmate without much knowledge of the game, except for the basic rules, it proceeds randomly at the beginning and may find itself in limbo performing certain inefficient moves. Figure 2 shows a spike of exponential moving average (EMA) moves per game in the early phase of the training. As the training progresses, with better knowledge, the agent checkmates more often and

[iii]Solution to Task 3: Implement SARSA and produce plots that show the reward per game and the number of moves per game vs training time with exponential moving average.

faster, which leads to a steep hike in rewards and a drop in moves per game before either reaches a plateau, where playing more episodes does not improve the performance significantly. Having trained for 100k episodes, the agent accumulates an reward of 0.92779 by finishing a game in 3.54345 moves on average.

B. Hyper-parameter analysis with SARSA[iv]

With a SARSA agent, hyper-parameters are scrutinized in a grid-search fashion for $\beta \in \{5\cdot10^{-7}, 3\cdot10^{-5}, 5\cdot10^{-5}, 7\cdot10^{-5}\}$ and $\gamma \in \{0.2, 0.85, 0.99\}$. As reinforcement learning relies on the agent-environment interaction, meaning no separation between training and testing, the average moves per game and rewards over 100k episodes are used as the performance measures. The discount factor γ dominates how much the agent deducts from a future reward while considering an immediate move. The smaller γ is, the more myopic an agent is. β regulates the exploration behavior of an agent. Given the exploration rate ϵ, in the n^{th} episode, the actual exploration rate is given by $\epsilon_n = \frac{\epsilon}{\beta \cdot n}$. The larger β is, the less often an agent explores as the training progresses.

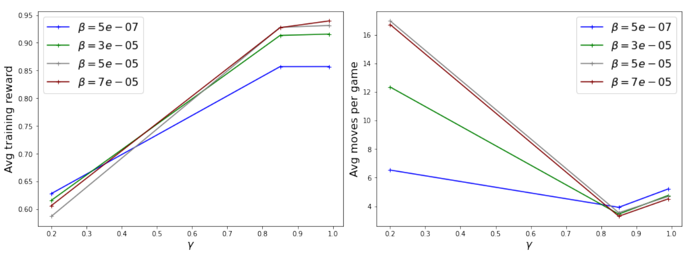

Fig. 3. Average moves per game and rewards received by a SARSA agent after 100k episodes with different combinations of discount factor γ and decay factor β.

As Figure 3 shows, the results of the experiment agree with the intuition that a rather small discount factor γ encourages an agent to focus only on the immediate effect of a move rather than the subsequent states where it might be better off, it therefore fails to explore efficient strategies. An increasing γ leads to more rewards and fewer moves per game in general. However, games last longer when γ goes from 0.85 to 0.99. One explanation is that by discounting future rewards very slightly, an agent might take unnecessary moves to where it is not significantly better than its current state.

The exploration decay factor β has a say in the trade-off between exploration and exploitation. Large β works better by allowing the agent to deviate from its sub-optimal strategy when the agent fails to learn decent strategies with a small discount factor γ.

With $\gamma = 0.99$ and $\beta = 7 \cdot 10^{-5}$, the SARSA agent accumulates the highest average reward of 0.93943 and finishes a game within 4.50880 moves on average over 100k episodes.

[iv]Solution to Task 4: Change the discount factor γ and the speed β of the decaying trend of ϵ and analyse the results.

C. Q-learning[v]

As illustrated in Figure 4, a Q-learning agent has bumpier curves compared with SARSA. The Q-learning agent achieves lower average rewards of 0.86206 and higher average moves per game of 11.23885 after 100k episodes. However, in spite of its sub-ideal performance measures, Q-learning might have explored a more efficient approach to tackle the game, which will be discussed later with a demonstration game. Besides, as neither of the curves reaches a plateau, one might argue that further training could yield a better outcome.

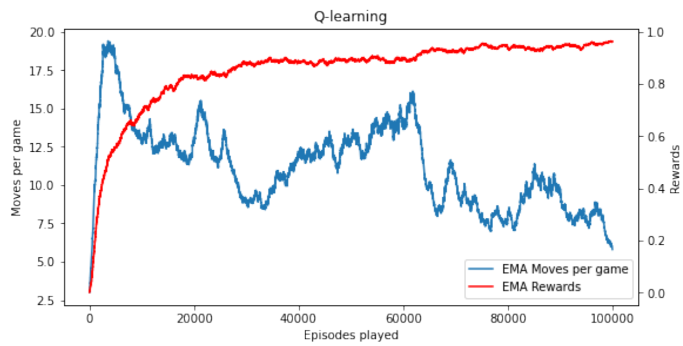

Fig. 4. EMA moves per game and rewards received by a Q-learning agent over 100k episodes.

D. An impatient agent[vi]

By changing the administration of reward, the SARSA learning agent is now encouraged to end a game quickly, since the negative reward for each move penalizes long games. The absence of reward at a terminal state makes a checkmate and a draw indifferent for the agent, thus neutralizes the motivation to win the game. Figure 5 shows that the SARSA agent learns to end a game within 2 moves on average.

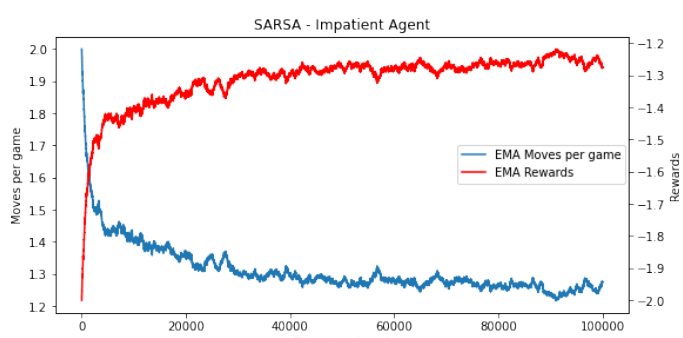

Fig. 5. EMA moves per game and rewards received by an impatient SARSA agent aiming at finishing the game ASAP over 100k episodes.

[v]Solution to Task 5: Implement Q Learning and compare the new results with the previous results.

[vi]Solution to Task 6: Change the administration of reward and interpret the results

E. A demonstration game

Figure 1 visualizes the initial setup of a demonstration game, with **Qa1Ka2 Kc2**.

A SARSA agent checkmates within 6 moves. The first 3 moves in Figure 6 are analyzed. The first move **Qd4 Kc1** is optimal, but the following **Qb2 Kd1** is not. The agent doesn't proceed aggressively to keep the opponent king in check. Instead, it backs off with **Qb3 Kd2** to avoid a stalemate.

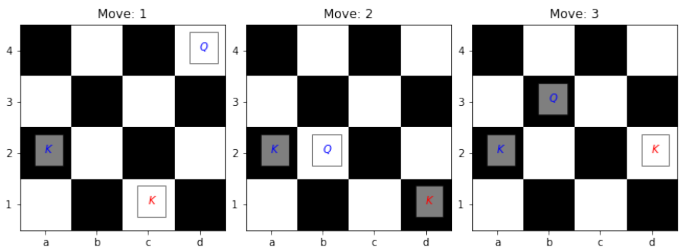

Fig. 6. Sub-optimal moves of a SARSA agent

With the same initial setup and identical first move, Figure 7 shows that a Q-learning agent pursues a more aggressive par-human approach by taking **Kb3 Kb2** to expand the squares covered by its king and leave only one possible option for the opponent. With **Qb2#**, it checkmates in 3 moves.

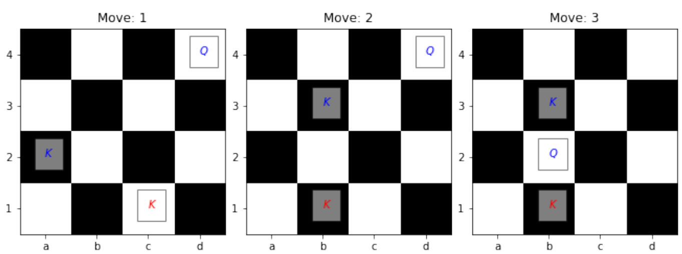

Fig. 7. Par-human optimal moves of a Q-learning agent

As demonstrated in Figure 8, an impatient SARSA agent doesn't care about the outcome but only wants to end the game as soon as possible. **Qb2 Kd3** leaves the opponent king in a stalemate. With **Kb3**, the game ends in a draw.

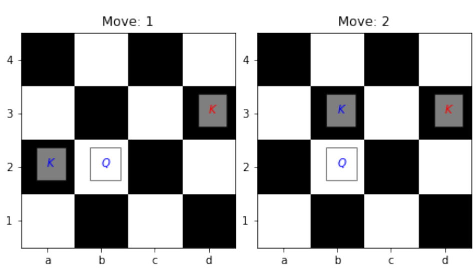

Fig. 8. Impatient agent only wants to end the game ASAP

IV. Conclusion

ANN is a powerful tool to approximate Q-values, so deep reinforcement learning can attempt tasks with high-dimensional state-action space. In the simplified chess task, a SARSA agent generates relatively smooth EMA curves of moves per game and rewards, while a Q-learning agent explores more effective strategies powered by its off-policy update rule. To facilitate comparison, the number of episodes is fixed at 100k. Training the Q-learning agent for more episodes is expected to generate a better outcome.

By design, reinforcement learning aims at solving goal-directed tasks, where policy and reward structure shape the behavior of an agent.

References

[Lin92] Long-Ji Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8.3 (1992), pp. 293–321.

[Sch03] Eric Schiller. *Official Rules Of Chess*. Cardoza Publishing, 2003.

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

Appendix

Detailed implementation of TD algorithms is attached in the appendix, where several highlights are worth mentioning.

An objected-oriented Artificial Neural Network is implemented to facilitate easy access to intermediate results, such as values of hidden neurons, in forward-feed and back-propagation.

A set of functions, namely the Chess Visualization Toolkit, is developed to facilitate visual inspection of the endgames, translation from the algebraic notation to matrix indices and easy customization of the chessboard.

A. Neural Network

```
# OOD ANN implementation
class ANN: # ANN object w/ 1 hidden layer w/ 200 units by default

    def __init__(self, N_in = 58, N_a = 32, random_seed = 9, act1 = 'Relu', act2 = 'Relu'): # constructor w/ the size of input, the size of output and random_seed as optional parameters
        self.N_h=200            ## NUMBER OF HIDDEN NODES (A NETWORK WITH ONE HIDDEN LAYER WITH SIZE 200)

        ### Random seed
        np.random.seed(random_seed)

        ### Xavier initialization
```

```
        self.W1 = np.random.randn(self.N_h, N_in) * np.sqrt(1 / (N_in)) # input layer, of shape (200, 58)
        self.W2 = np.random.randn(N_a, self.N_h) * np.sqrt(1 / (self.N_h)) # hidden layer, of shape (32, 200)

        self.b1 = np.zeros((self.N_h,)) # of size 200
        self.b2 = np.zeros((N_a,)) # of size 32

        ### Parameterize the choices of activation functions
        self.act1 = 1 if act1 == 'Relu' else 2
        self.act2 = 1 if act2 == 'Relu' else 2
        self.act2 = 3 if act2 == 'Sigmoid' else self.act2

        ### Initiate neurons and activations
        self.z1 = np.zeros((self.N_h,))
        self.a1 = np.zeros((self.N_h,))
        self.z2 = np.zeros((N_a,))
        self.a2 = np.zeros((N_a,))

    def forwardfeed(self, X):
        ## Forwardfeed
        ## input -> hidden
        self.z1 = self.W1 @ X + self.b1 # of size 200

        self.a1 = self.relu(self.z1) if self.act1 == 1 else self.z1

        ## hidden -> output
        self.z2 = self.W2 @ self.a1 + self.b2 # of size 32

        self.a2 = self.relu(self.z2) if self.act2 == 1 else self.z2 # of size 32
        return self.a2

    def relu(self, x): # rectified linear unit activation to cope w/ vanishing gradient
        return (x > 0).astype(int) * x

    def backpropagation(self, X, delta, a_agent, eta):
        ## Backpropagation
        ## Gradients
        dz1, dz2 = self.calc_gradient(delta, a_agent)

        ## Descent (update)
        self.W2[a_agent, :] = self.W2[a_agent, :] + eta * dz2  * self.a1
        self.b2[a_agent] = self.b2[a_agent] + eta * dz2

        self.W1 = self.W1 + eta * np.outer(dz1, X)
        self.b1 = self.b1 + eta * dz1

        return self.W1, self.W2

    def calc_gradient(self, delta, a_agent):
        ## Gradients
        dz2 = delta * self.heavy_side(self.a2[a_agent]) if self.act2 == 1 else delta # of size 1

        dz1 = delta * self.W2[a_agent, :] * self.heavy_side(self.a1) if self.act1 == 1 else dz2 * self.W2[a_agent, :] * self.a1 # of size 200
        return dz1, dz2


    def heavy_side(self, x): # pseudo derivative of ReLu
        return (x > 0).astype(int)

    def load_weights(self, W1, W2, b1, b2): # load trained model
        self.W1 = W1
        self.W2 = W2
        self.b1 = b1.flatten()
        self.b2 = b2.flatten()
```

Listing 1. Object-oriented implementation of the Artificial Neural Network

B. SARSA

```
# TRAINING LOOP of SARSA

nn = ANN() # instantiate a Neural Network object w/ default activation functions ('Relu')

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)
```

```python
    i = 1    ## COUNTER FOR NUMBER OF ACTIONS

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

    # Choose action A (w/o taking it actually)
    ## pre-action meditation
    a,_=np.where(allowed_a==1) # find allowed actions

    ## find Q-values
    Qvalues = nn.forwardfeed(X) # generate a vector of estimated Q values by ANN
    Qvalues_of_allowed_a = Qvalues[a] # select only the Q values of allowed actions

    ## Given Q-values, pick an action in an e-Greedy way
    a_agent = a[EpsilonGreedy_Policy(Qvalues_of_allowed_a, epsilon_f)][0] # idx of the chosen action

    q = np.copy(Qvalues[a_agent]) # copy the Q value of the chosen action

    while Done==0:    ## START THE EPISODE

        # take the action a_agent & observe
        S_next,X_next,allowed_a_next,R,Done=env.OneStep(a_agent)

        ## THE EPISODE HAS ENDED, UPDATE ...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
        if Done==1:

            # Record the reward and moves taken
            R_save[n]=np.copy(R)
            N_moves_save[n]=np.copy(i)

            # Update Weights
            ## Backpropagation
            delta = (R - q) # Compute the delta
            nn.backpropagation(X, delta, a_agent, eta) # backpropagation via gradient descent

            break

        else:
            # post-action reflection (choose the next action w/o actually taking it either)
            a_next,_=np.where(allowed_a_next==1) # find allowed actions

            # Find Qvalues
            Qvalues_next = nn.forwardfeed(X_next) # generate a vector of estimated Q values by ANN

            Qvalues_of_allowed_a_next = Qvalues_next[a_next] # select only the Q values of allowed actions

            a_agent_next = a_next[EpsilonGreedy_Policy(Qvalues_of_allowed_a_next, epsilon_f)][0] # idx of the chosen action

            q_new = np.copy(Qvalues_next[a_agent_next]) # copy the Q value of the chosen action

            # Update Weights
            ## Backpropagation

            delta = (R + gamma * q_new - q) # Compute the delta

            nn.backpropagation(X, delta, a_agent, eta) # backpropagation via gradient descent


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

        i += 1  # UPDATE COUNTER FOR NUMBER OF ACTIONS


    print_progress(n, N_episodes) # odometer to show training process

print_progress(None, N_episodes) # indicate whether training is over
```

```python
print('e-Greedy agent w/ 1QN SARSA, Average reward:',np.mean(R_save),' Number of steps: ',np.mean(N_moves_save)) # print performance measures
```

Listing 2. Training logic of SARSA

C. Q-learning

```python
# TRAINING LOOP of Q-learning

nn = ANN(act2 = 'Sigmoid') # to curb the exploding weights, use sigmoid activation function in the output layer


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

        # Choose action A (w/o taking it actually)
        ## pre-action meditation
        a,_=np.where(allowed_a==1) # find allowed actions

        ## find Q-values
        Qvalues = nn.forwardfeed(X)
        Qvalues_of_allowed_a = Qvalues[a]

        ## Given Q-values, pick an action in an e-Greedy way

        a_agent = a[EpsilonGreedy_Policy(Qvalues_of_allowed_a, epsilon_f)][0] # idx of the chosen action

        q = np.copy(Qvalues[a_agent])

        # take the action a_agent & observe
        S_next,X_next,allowed_a_next,R,Done=env.OneStep(a_agent)


        ## THE EPISODE HAS ENDED, UPDATE ...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
        if Done==1:

            # Record the reward and moves taken
            R_save[n]=np.copy(R)
            N_moves_save[n]=np.copy(i)

            # Update Weights
            ## Compute the delta
            delta = (R - q)

            ## Backpropagation
            nn.backpropagation(X, delta, a_agent, eta)

            break

        else:
            # post-action reflection (choose the next action w/o actually taking it either)
            a_next,_=np.where(allowed_a_next==1) # find allowed actions

            # Find Qvalues
            Qvalues_next = nn.forwardfeed(X_next)

            Qvalues_of_allowed_a_next = Qvalues_next[a_next]
            a_agent_next = a_next[EpsilonGreedy_Policy(Qvalues_of_allowed_a_next, epsilon_f)][0] # idx of the chosen action

            q_new = np.max(Qvalues_next)

            # Update Weights
            ## Compute the delta
            delta = (R + gamma * q_new - q) # no punishment

            ## Backpropagation
            nn.backpropagation(X, delta, a_agent, 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


    print_progress(n, N_episodes)

print_progress(None, N_episodes)

print('e-Greedy agent w/ 1QN Q-learning, Average reward:',np.mean(R_save),' Number of steps: ',np.mean(N_moves_save))
```

Listing 3. Training logic of Q-learning

D. Hyper-parameter analysis

```
# TRAINING LOOP of SARSA w/ Hyper-parameter tuning

# Grid-search for hyperparameters
beta_list = [5e-07, 3e-05, 5e-05, 7e-05]     # Decay factor of EPSILON

gamma_list = [0.2, 0.85, 0.99]        # THE DISCOUNT FACTOR

performance = []

for beta in beta_list:

    for gamma in gamma_list:

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

        # instantiate a new ANN object for each combination of hyper-parameters
        nn = ANN() # instantiate a Neural Network object w/ default activation functions ('Relu')

        for n in range(N_episodes):

            '''
            ** Game initialization **

            Some code is hidden for simplicity
            Please refer to detailed source code in the main training logic of SARSA
            '''

            while Done==0:            ## START THE EPISODE

                ## THE EPISODE HAS ENDED, UPDATE...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
                if Done==1:

                    # Record the reward and moves taken
                    R_save[n]=np.copy(R)
                    N_moves_save[n]=np.copy(i)

                    '''
                    ** Episode terminates **

                    Some code is hidden for simplicity
                    Please refer to detailed source code in the main training logic of SARSA
                    '''

                    break

                else:

                    '''
                    ** Game continues **

                    Some code is hidden for simplicity
                    Please refer to detailed source code in the main training logic of SARSA
                    '''


        # Given a set of hyper-parameters, save performance metrics
        pd.DataFrame(R_save).to_csv(f'R_save_w_beta_{beta}_gamma_{gamma}_100k.csv')

        pd.DataFrame(N_moves_save).to_csv(f'N_moves_save_w_beta_{beta}_gamma_{
```

```
    gamma}_100k.csv')

        performance.append([beta, gamma, np.mean(R_save), np.mean(N_moves_save)])
```

Listing 4. Hyper-parameter analysis with SARSA

E. Change of the administration of reward

```
# TRAINING LOOP of impatient SARSA

nn = ANN(act2 = None) # use linear activation function (no activation) of the output layer to approximate negative Q-values
'''
** ONLY different implementation from SARSA is exhibited in this snippet of code **

Some code is hidden for simplicity
Please refer to detailed source code in the main training logic of SARSA
'''

for n in range(N_episodes):

    R_accu = 0 # initiate accumulative Reward of an episode

    '''
    ** Game initialization **

    Some code is hidden for simplicity
    Please refer to detailed source code in the main training logic of SARSA
    '''

    while Done==0:    ## START THE EPISODE

        # take the action A & observe
        S_next,X_next,allowed_a_next,R,Done=env.OneStep(a_agent)

        ## THE EPISODE HAS ENDED, UPDATE ...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
        if Done==1:

            # Record the reward and moves taken
            R_save[n] = R_accu - 1 # record the accumulative reward of the episode

            # Update Weights
            ## Backpropagation
            delta = (0 - q) # Neutralize the impact of result

            '''
            ** Episode terminates **

            Some code is hidden for simplicity
            Please refer to detailed source code in the main training logic of SARSA
            '''

            break

        else:
            '''
            ** Game continues **

            Some code is hidden for simplicity
            Please refer to detailed source code in the main training logic of SARSA
            '''

            # Update Weights
            ## Backpropagation

            R_accu -= 1 # immediate reward  -1 for each step the agent takes (penalize long games)

            delta = (-1 + gamma * q_new - q) # Compute the delta w/ penalty -1  for each step the agent takes

            nn.backpropagation(X, delta, a_agent, eta) # backpropagation via gradient descent
```

Listing 5. Impatient agent with SARSA

F. Chess Visualization

```
# chess_vis.py
# define a series of functions to facilitate visualization
import numpy as np
import matplotlib.pyplot as plt
from degree_freedom_queen import *
from degree_freedom_king1 import *
from degree_freedom_king2 import *
```

```python
from generate_game import *
from Chess_env import *


# define a function to translate piece location from board to the environment index system: a0 -> [3, 0]
def trans(pos):
    y = ord(pos[0]) - 97 # char c -> c* in ASCII -> c* - 97
    x = 4 - int(pos[1]) # char n -> int n - 1
    return (x, y)

# define a function to initiate the game lie the demo case
def demo_initialise_game(env, k1, q1, k2):
    # START THE GAME BY SETTING PIECIES

    size_board = 4; # 4-by-4 board

    env.Board = np.zeros([size_board, size_board], dtype=int) # initialize the board given size w/ a numpy zero matrix

    k1_x, k1_y = k1 # read the x,y coordinates of my king
    env.p_k1 = np.array([k1_x, k1_y]) # 1 for my king, pass its initial position to the env.
    env.Board[k1_x, k1_y] = 1 # update env.board w/ king's position

    q1_x, q1_y = q1 # read the x,y coordinates of my quen
    env.p_q1 = np.array([q1_x, q1_y]) # 2 for my queen, pass its initial position to the env.
    env.Board[q1_x, q1_y] = 2 # update env.board w/ queen's position

    k2_x, k2_y = k2 # read the x,y coordinates of opponent king
    env.p_k2 = np.array([k2_x, k2_y]) # 3 for oppenent king, pass its initial position to the env.
    env.Board[k2_x, k2_y] = 3 # update env.board w/ opponent king's position

    # Allowed actions for the agent's king
    env.dfk1_constrain, env.a_k1, env.dfk1 = degree_freedom_king1(env.p_k1, env.p_k2, env.p_q1, env.Board)

    # Allowed actions for the agent's queen
    env.dfq1_constrain, env.a_q1, env.dfq1 = degree_freedom_queen(env.p_k1, env.p_k2, env.p_q1, env.Board)

    # Allowed actions for the enemy's king
    env.dfk2_constrain, env.a_k2, env.check = degree_freedom_king2(env.dfk1, env.p_k2, env.dfq1, env.Board, env.p_k1)

    # ALLOWED ACTIONS FOR THE AGENT, ONE-HOT ENCODED
    allowed_a=np.concatenate([env.a_q1,env.a_k1],0)

    # FEATURES (INPUT TO NN) AT THIS POSITION
    X=env.Features()

    return env.Board, X, allowed_a


# define a function to print the chess board w/ given position of pieces
def print_board(S, num = None, figsize = (5, 5), ax = None):

        # instantiate a subplots obj w/ default figsize = (5, 5)
        if ax == None:
                fig, ax = plt.subplots(figsize = figsize)

        # create a blank board
        blank_board = np.add.outer(range(4), range(4))%2 # create grids
        ax.imshow(blank_board, cmap="binary_r") # use 2D raster to generate the board
        ax.set_xticks([0.0, 1.0, 2.0, 3.0]) # formulate x ticks
        ax.set_xticklabels(['a', 'b', 'c', 'd']) # formulate x ticks
        ax.set_yticks([0.0, 1.0, 2.0, 3.0]) # formulate y ticks
        ax.set_yticklabels([4, 3, 2, 1]) # formulate y ticks


        # put pieces on the board
        board_idx = np.array([[(x, y) for y in range(4)] for x in range(4)]) # generate a positional index table
```

```python
    pieces = ['K', 'Q', 'K'] # piece names
    colors = ['blue', 'blue', 'red'] # piece colors (blue -> player, red -> opponent)
    for i in range(3): # assign each piece
        y_loc, x_loc = board_idx[S == i + 1][0]
        ax.text(x_loc, y_loc, pieces[i], style='italic', color = colors[i],
                bbox={'facecolor': 'white', 'alpha': 0.5, 'pad': 10})

    # format title
    if num == None:
        ax.set_title(f'Overkilling Endgame')
    else:
        ax.set_title(f'Move: {num}')

    # show figure
    if ax == None:
        fig.show()
```

Listing 6. Chess Visualization Toolkit