Exploiting Q-learning and SARSA to Checkmate a Simplified Chess Game

1st Yifei Liu
Department of Informatics
University of Zurich
Zurich, Switzerland,
yifei.liu@uzh.ch

2nd Ye Zhou
Department of Informatics
University of Zurich
Zurich, Switzerland,
ye.zhou@uzh.ch

3rd Luca Manneschi
Department of Computer Science
University of Sheffield
Sheffield, UK,
lmanneschi1@sheffield.ac.uk

4th Eleni Vasilaki
Department of Computer Science
University of Sheffield
Sheffield, UK,
e.vasilaki@sheffield.ac.uk

***Abstract*—This document is a report for the Chess Game in the Lecture: Introduction to Reinforcement Learning, in 2022 Spring at University of Zurich. In this report, we discuss the differences between Q-learning and SARSA, the experience replay technique, and use both learning strategies to play with a simplified chess game. We explore the effect of different hyperparameters and representation of rewards, and the effect of RMSprop and AdamW.**

***Index Terms*—Q-leaning, SARSA, reinforcement Learning, artificial neural network**

I. Introduction

Reinforcement Learning aims to learn how to act in an unknown Markov Decision Process (MDP) [8], and can be touched with either model-based algorithms, which first learn the underlying MDP and then optimize policy based on estimated MDP, or model-free algorithms, which directly estimate the Value function or the Value-Action pair function, i.e. Q function. In this report we focus on two model-free algorithms: Q-learning [9] and SARSA [6]. Sarsa is an on-policy algorithm, which evaluate and improve the same policy being used to select actions, whilst Q-learning does not estimate any policy but the optimal one.

Describe the algorithms Q-learning and SARSA: Given a state S, SARSA takes an action A according to its current policy (e.g, $epsilon$-greedy policy), receives an immediate reward R and observes a new state S', and then decides the next action A'. Then SARSA updates the Q function:

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A)) \quad (1)$$

But for Q-learning, given a state S, it takes an action A, receives an immediate reward R and observes a new state S'. Then Q-learning updates the Q function:

$$Q(S,A) \leftarrow Q(S,A)+\alpha(R+\gamma max_a Q(S',a)-Q(S,A)) \quad (2)$$

Both algorithms repeat their above respective equations for every step in a episode. Note that for SARSA, the action A for the current step should be the same as the A' in the previous step.

Explain the differences and the possible advantages/disadvantages between the two methods: As is seen in (1), SARSA is on-policy in the sense that it uses A' which is derived from the ongoing policy to update Q function. Q-learning is off-policy, because it does not use the ongoing policy to update as seen in (2). The advantage of Q-learning is clear, it can learn from a collected dataset since Q-learning updates do not rely on the policy being used to make actions. Moreover, Q-learning directly estimates the optimal state-action pairs, whilst SARSA learns the ϵ-greedy optimal policy, so if SARSA is used for finding the exact optimal policy, a ϵ decay strategy needs to be designed, which is an additional hyperparameter to tune. However, SARSA also has advantages that the ongoing policy will be more and more optimal, and the learning will converge faster [1]. Another point is that Q-learning may be too extreme about the rewards since it pursues the optimal policy, and thus become less robust, while SARSA estimate ϵ-greedy optimal policy which are easier to get punished than optimal policy, so the agent must learn to stay further away form dangerous areas, and thus leave more safety room for errors which can lead to large punishment.

The Q function can also be approximated by neural networks. However, the direct generalization of SARSA and Q-learning algorithm described above are inefficient in that experiences obtained are utilized by the network only once and then thrown away. This is wasteful, since some experience may be rare and some are costly to obtain. One way of reusing experiences is called **experience replay** [3]. It has two advantages. First, the process of blame propagation is sped up, because the previous bad experience can be used more than once. Second, in RL the data distribution changes as the agent evolves, which is problematic for deep learning methods. By using the experience replay mechanism which randomly samples previous experiences, the correlated data and non-stationary distribution problem can be alleviated.

Different variations of experience replay exist. [5] uniformly samples from the replay memory, while [7] samples more important transitions more frequently to learn more efficiently. **The uniformly sampling experience replay algorithm can be found in [5], and is duplicated here for readers' convenience:** First, initialize replay memory D with capacity N. Then, after producing every transition (s_t, a_t, r_t, s_{t+1}), store it in D. Then uniformly sample a minibatch of transitions from D and use it to perform a batch gradient descent update.

Identify applicable funding agency here. If none, delete this.

The undescribed parts are the normal parts, i.e., choosing an action and receiving new states and rewards. Note that experience replay is off-policy because the current parameters are different from those used to generate the sample, so Q-learning is suitable but not typical SARSA.

We use Q-learning and SARSA to learn to checkmate in a simplified chess game. Code is available at https://github.com/kaikai23/ToyChessGame.

II. Methods

First we take some notations to enable a proper discussion.

A chess game consists of a N x N grid with M pieces, and the state space S has size $O(N^{2M})$ considering that each square has m different ways to be occupied. At each time step t, our agent can observes full states $s_t \in S$ presented on the chessboard. We encode s_t by an isomorphism ϕ into a f-dimensional indicator feature $x_t = \phi(s_t) \subseteq \{0,1\}^f$. Note that f is $O(N^2 \cdot M)$ because each indicator indicates whether a certain piece is on a certain square. We define a feature x_t is feasible iff $\phi^{-1}(x_t) \in S$, i.e. no pieces on the board are occupying the same position or occupying any position out of bounds. This induces feasible feature space $X = \{x \in \{0,1\}^f | \ \phi^{-1}(x) \in S\}$. Note that $\phi(S) = X$ as ϕ is isomorphism.

We define a action is allowable iff the next state derived from it is feasible (note that we can say so because chess is a deterministic environment). A agent has a fixed number of pieces including dead ones, and each piece can choose from a fixed number of actions at each step including non-allowable ones, so an agent also has a fixed total number of actions to choose from at each time step. Let's suppose there are in total n actions for the agent to choose at each step, then the allowable actions can be represented by $a_t \in \{0,1\}^n$, where the i-th component decides if action i is allowable at step t. It is easy to notice that actually $a_t(s_t)$ is a deterministic and known function of the state s_t, so once s_t is given, a_t is instantly known.

To learn the best action for each state, we use a neural network to estimate Q values. For each step t, we give as input x_t and a_t to the neural network. The output is $q \in \mathbb{R}^n$ which estimate the Q values for each action. Then we can take the action $i = \underset{i}{\operatorname{argmax}}(q_i)$ to perform a greedy policy or ϵ-greedy policy.

A. Structure of Neural Network

The network is a 2-layer perceptron which has an input layer, a hidden layer and an output layer with f, h, n neurons respectively. No activation is applied to the output layer, and we set h=200 in our experiments.

For the activation function, we do not use ReLU because it restricts the output in $\mathbb{R}_+$. Instead we use Tanh described in (3), as it is nonlinear and produces both positive and negative values, which are then linearly combined to estimate arbitrary Q values.

$$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \tag{3}$$

A pitfall until now is that the output is any $q \in \mathbb{R}^n$, and a direct $\underset{i}{\operatorname{argmax}}(q_i)$ could lead to an action i in collision with a_t, especially in the early stages during training when estimations are filled with noise. To avoid this collision, we use hadamard product $\tilde{q} = q \odot a_t$ to denote the final estimate.

Note that $\tilde{q}$ is also a differentiable function of the state, see (4), so error can be properly backpropagated through the neural network.

$$\tilde{q}(s_t) = q(s_t) \odot a_t(s_t) = W_2 tanh(W_1 \phi(s_t)) \odot a_t(s_t) \tag{4}$$

, where W_1 and W_2 are neural network wights.

B. Loss Function

We denote a transition by (s, i, r, s'), where $s, s' \in S, i \in \{0, 1, ..., n-1\}$, and $r \in \mathbb{R}$. We use stochastic gradient descent for every single transition to update the neural network at each time step.

For Q-learning, a transition is (s, i, r, s') and the loss is

$$L = \frac{1}{2}(r + \gamma \max_j \tilde{q}_j(s') - \tilde{q}_i(s))^2 \tag{5}$$

For SARSA, a transition is (s, i, r, s', j) and the loss is

$$L = \frac{1}{2}(r + \gamma \tilde{q}_j(s') - \tilde{q}_i(s))^2 \tag{6}$$

, where γ is the discount factor.

C. Reward

If the game is not done, the reward is 0 for each step. If our agent checkmate, the reward is 1. If the game ends in a draw, the reward is 0.

For distinguishing purpose, we clarify that when we say reward per game, it means the last step reward above, while the algorithms are maximizing the accumulated discounted reward, which is different from the last step reward.

III. Experiments

We set up a series of ablation experiments to study the effect of each part of our architecture. The baseline has the default values for hyperparameters. Learning rate: $lr = 0.0035$, discount factor: $\gamma = 0.85$, decay speed of ϵ: $\beta = 5e^{-5}$, ϵ-greedy policy: $\epsilon = 0.2$, number of episodes: $N_{episodes} = 100000$.

A. Q-Learning vs. SARSA

Fixing all hyperparameters to default values, we compare Q-Learning and SARSA by training both algorithms from scratch.

Fig. 1 and Fig. 2 show the rewards and number of moves during training for Q learning and SARSA. Both Q learning and SARSA got similar rewards close to 1 and number of moves in the end, although Q learning converges a bit faster for number of moves. This may be because the state space is too simple and problem is too easy, so Q learning performs a little bit better as it directly estimates the optimal policy.

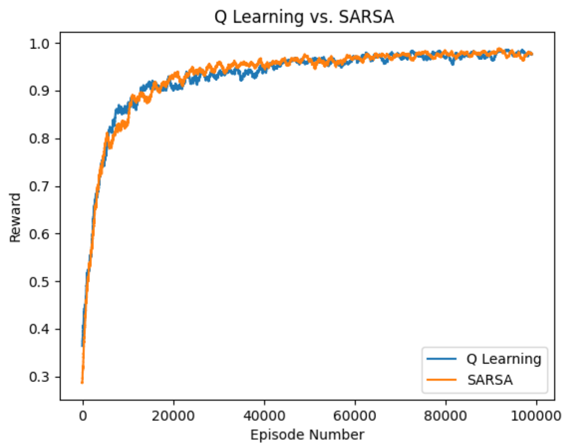

Fig. 1: Reward per game vs. training time for Q Learning and SARSA. All hyperparameters are default.

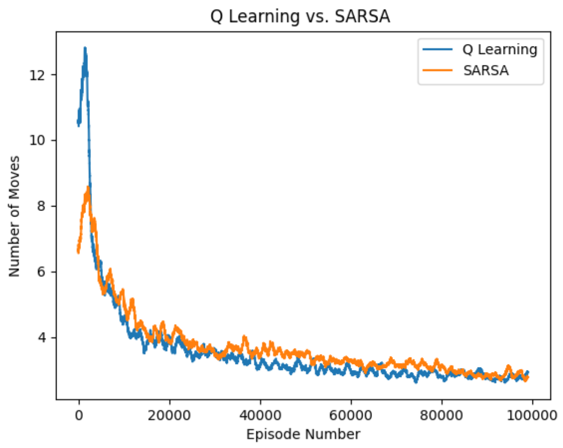

Fig. 2: Number of moves per game vs. training time for Q Learning and SARSA. All hyperparameters are default.

B. Discount Factor Study

Fixing other hyperparameters to default values, we experimented with different discount factor values γ = 0.75, 0.85 (default) and 0.95. The results are shown in Fig. 3.

We can see from plots in the right column that a lower discount factor encourages faster end for the game. This is intuitive because the agent tries to maximize the accumulated discounted reward, which is discounted more if the agent takes more steps, so the agent wants to checkmate as soon as possible. When γ is smaller, the agent is in a more hurry to checkmate. Accordingly, from the plots in the left column we see that a smaller discount factor leads to a bit smaller reward per game, because the agent is in a rush and has lower probability to checkmate.

C. Decay Speed Factor Study

Fixing other hyperparameters to default values, we experimented with different decay speed factor values $\beta = 5e^{-4}$,

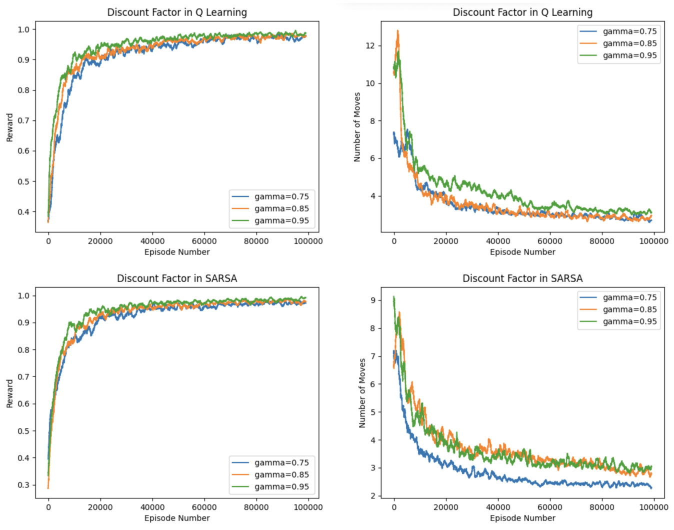

Fig. 3: Reward per game and Number of moves per game vs. training time for Q Learning and SARSA. All hyperparameters are default except for discount factor γ.

$5e^{-5}$ (default) and $5e^{-6}$. ϵ decays according to (7), where l is the current episode number.

$$\epsilon_{t+1} = \frac{\epsilon_t}{1 + \beta * l} \tag{7}$$

The results are shown in Fig. 4. As we see from plots in

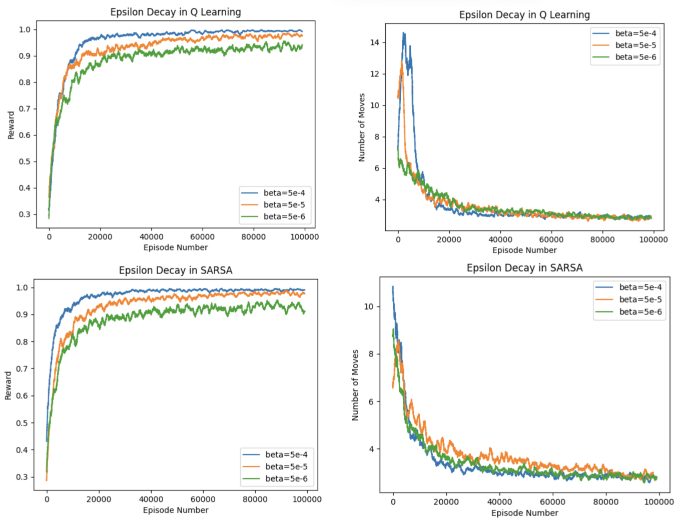

Fig. 4: Reward per game and Number of moves per game vs. training time for Q Learning and SARSA. All hyperparameters are default except for decay speed factor β.

the right column, the decay speed do not have an apparent influence to number of moves per game. But a smaller decay speed β makes the reward per game drop greatly. This is because a small β would cause a large ϵ and thus favors exploration more than exploitation. Even if the Q values can be estimated perfectly, a large ϵ still would still let the agent make wrong actions, and thus the reward per game is far from 1 in such cases.

D. SGD vs. RMSprop sv. AdamW

The reason why we do this experiment is to answer the question about gradient exploding. We think the best way to avoid gradient exploding is to make the neural network model simple if possible. Fortunately, the simplified chess game has limited number of states and is simple enough for allowing a simple neural network such as ours (4). So we are glad we have solved the problem of gradient exploding in the first place.

But for learning purpose, we still investigate about why RMSprop and Adam help prevent gradient exploding problem. RMSprop is an unpublished but very well-known optimization algorithm. In our experiment, RMSprop works as follows:

```
input : $\alpha$ (alpha), $\gamma$ (lr), $\theta$ (params), $f(\theta)$ (objective)
initialize : $v_0 \leftarrow 0$ (square average)

for $t = 1$ to $\ldots$ do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $v_t \leftarrow \alpha v_{t-1} + (1-\alpha) g_t^2$
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t / \left(\sqrt{v_t} + \epsilon\right)$

return $\theta_t$
```

In RMSprop, instead of directly using the gradient of the loss g_t to update weights, a weighted moving average of the squared gradient, i.e. v_t, is used to scale g_t. When g_t is large, v_t also gets large, so $\frac{g_t}{\sqrt{v_t}+\epsilon}$ becomes smaller than g_t, and this is how RMSprop alleviate gradient exploding problem. However, RMSprop could lead to instabilities in training and even diverge during training [2], so Adam [2] is introduced. Adam performs equal or better than RMSprop regardless of hyper-parameter setting, by using $\frac{m_t}{\sqrt{v_t}+\epsilon}$ to update weights, where m_t is a weighted moving average of the gradient. Noticing that $\sqrt{v_t}$ is the magnitude of accumulated gradients, we can see m_t is also scaled and thus can alleviate the gradient exploding problem. Finally, AdamW [4] fixes the weight decay of Adam to improve the generalization performance.

We can see from Fig. 5 that the training for RMSprop diverges, just as what we have discussed before. We can also see that AdamW is a little weaker than SGD for this specific experiment setting, which is normal because SGD has better generalization performance, and there are still many hyperparameters for AdamW to be tuned.

E. Change State representation

In Section. II we introduced the encoding of state $x_t = \phi(s_t)$, but did not explain what exact ϕ we use. Now we discuss more about it.

For all experiments above, we used a 58-dimensional indicator vector: the first 3 x 16 dimensions represents position for 3 pieces in the 4 x 4 grid, the next 2 dimensions represent

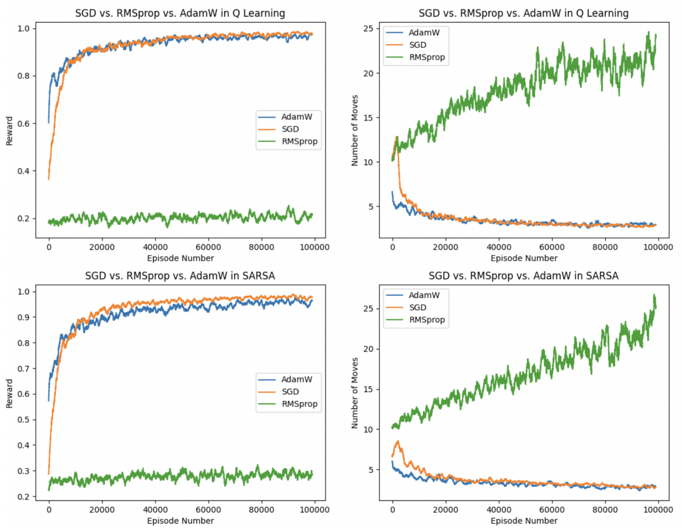

Fig. 5: Reward per game and Number of moves per game vs. training time for Q Learning and SARSA. All hyperparameters are default except for the type of optimizer. Learning rate for SGD, RMSprop and AdamW are 0.0035, 0.01, 0.001, respectively

whether the opponent's king is in check, and the last 8 dimensions is a one-hot vector representing how many actions the opponent's king can make.

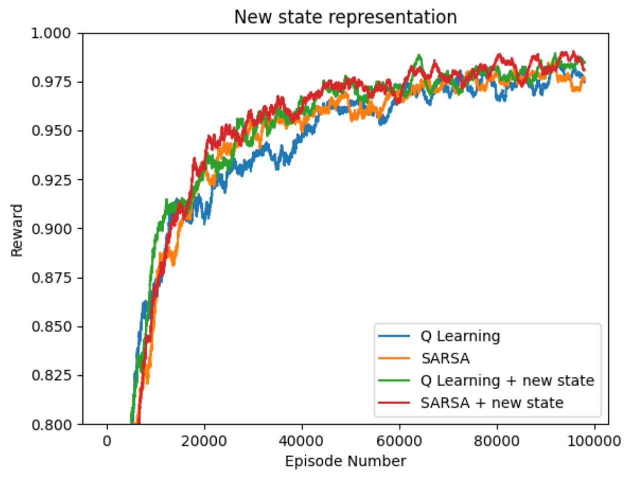

Fig. 6: Reward per game vs. training time for Q Learning, SARSA old state representation and new state representation. All hyperparameters are default vallues.

We notice that the above ϕ has a deficiency that the last 8 dimensions constitute a one-hot vector which is not very informative. Although theoretically the neural network can learn to recover any representation and information from the bare positions of pieces , we still think it is beneficial to explicitly have more intuitive encoded features. So we change the last 8 dimension into an indicator vector which is also 8 dimension. The indicator vector indicates whether each of the 8 actions can be take, thus is more informative than the only number of allowable actions. We use this new ϕ' to form features to be fed into the neural networks, and make a comparison with old representation ϕ.

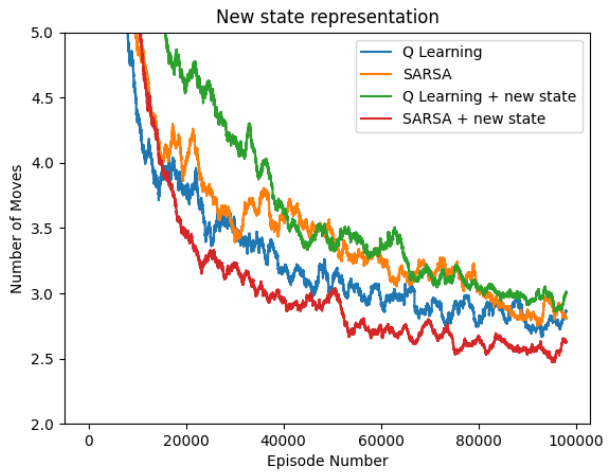

Fig. 7: Number of moves vs. training time for Q Learning, SARSA old state representation and new state representation. All hyperparameters are default vallues.

From Fig. 6 and Fig. 7, we summarize the ranking of different combinations of state representations and algorithms in Table I, thus verifying that our new representation is superior to old representation.

	#1	#2	#3	#4
Reward	SARSA + new	Q learn + new	Q learn	SARSA
Moves	SARSA + new	Q learn	SARSA	Q learn + new

TABLE I: Ranking for combinations of different states representations and algorithms

IV. Conclusion

By properly abstracting the chess game, we utilized Q learning and SARSA algorithms to learn to checkmate in a simplified chess game. Our objective was never searching for the best model in this specific simplified application, so we did not search any combination of more than 2 hyperparameters. Instead, we focus on the ablation study for every single hyperparameter in concern. We summarize our conclusion as follows:

- Q learning and SARSA with decayed ϵ-greedy policy have similar performance in a simple state space.
- A small discount factor will urge the agent to win fast, and drops the last step reward, which we care about.
- If the decay speed for ϵ is not enough, the agent will not adopt optimal policy in the end.
- SGD is a simple yet effective optimizer and outperforms RMSprop and AdamW in this chess game setting.
- Our simple neural network structure helps us avoid gradient exploding.
- Our proposed new state representation is superior to the old one.

Acknowledgment

We thank Luca Manneschi and Eleni Vasilaki for providing the bone code structure and the chess environment.

References

[1] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. A comparison of reinforcement learning techniques for fuzzy cloud autoscaling. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 64–73, 2017.
[2] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[3] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, USA, 1992. UMI Order No. GAX93-22750.
[4] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.
[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.
[6] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.
[7] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.
[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[9] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

Appendix A
Code Snippet

The code snippets referring to algorithm implementation is the **Agent** class, which is very self-explanatory.

```
class Agent:
    def __init__(self, env):
        self.env = env
        self.Q_Net = Q_Net(in_dim=58, hidden_dim=200, out_dim=32)
        self.eta = 0.0035
        self.gamma = 0.85
        self.beta = 0.00005
        self.epsilon_0 = 0.2
        self.N_episodes = 100000
        self.R_save = np.zeros([self.N_episodes, 1])
        self.N_moves_save = np.zeros([self.N_episodes, 1])
        self.optimizer = torch.optim.SGD(self.Q_Net.parameters(), lr=self.eta)
        self.optimizer_name = 'SGD'

    def reset_Network_Parameters(self):
        self.Q_Net = Q_Net(in_dim=58, hidden_dim=200, out_dim=32)
        if self.optimizer_name == 'SGD':
            self.optimizer = torch.optim.SGD(self.Q_Net.parameters(), lr=self.eta)
        elif self.optimizer_name == 'AdamW':
            self.optimizer = torch.optim.AdamW(self.Q_Net.parameters(), lr=self.eta)

    def reset_R_and_N_moves(self):
        self.R_save = np.zeros([self.N_episodes, 1])
        self.N_moves_save = np.zeros([self.N_episodes, 1])

    def set_N_episodes(self, N):
        self.N_episodes = N

    def set_discount_factor(self, gamma):
        self.gamma = gamma

    def set_decay_speed(self, beta):
        self.beta = beta

    def set_optimizer(self, lr=0.0035, name='SGD'):
        self.eta = lr
        self.optimizer_name = name
        if name == 'SGD':
            self.optimizer = torch.optim.SGD(self.Q_Net.parameters(), lr=lr)
        elif name == 'AdamW':
            self.optimizer = torch.optim.AdamW(self.Q_Net.parameters(), lr=lr)
        elif name == 'RMSprop':
            self.optimizer = torch.optim.RMSprop(self.Q_Net.parameters(), lr=lr)
        else:
            raise 'ERROR: Method Not Implemented'

    def reset_all(self):
        self.Q_Net = Q_Net(in_dim=58, hidden_dim=200, out_dim=32)
        self.optimizer = torch.optim.SGD(self.Q_Net.parameters(), lr=self.eta)
        self.eta = 0.0035
        self.gamma = 0.85
        self.beta = 0.00005
        self.epsilon_0 = 0.2
        self.N_episodes = 100000
        self.R_save = np.zeros([self.N_episodes, 1])
        self.N_moves_save = np.zeros([self.N_episodes, 1])

    def train_q_learning(self):
        self.reset_R_and_N_moves()
        self.reset_Network_Parameters()
        print("=========== Starting to train ============ ")
        print("----------- Method: Q Learning ----------- ")
        print(f'number of episodes: {self.N_episodes}\ndicount factor: {self.gamma}\ndecay factor: {self.
                                                              beta}')
        for n in tqdm(range(self.N_episodes)):
            epsilon_f = self.epsilon_0 / (1 + self.beta * n)
            Done = 0
            i = 1
            S, X, allowed_a = self.env.Initialise_game()
            while Done == 0:  ## START THE EPISODE
                Q_values = self.Q_Net(torch.from_numpy(X).float(), torch.from_numpy(allowed_a).squeeze())
                a = EpsilonGreedy_Policy(Q_values, allowed_a, epsilon_f)
                S_next, X_next, allowed_a_next, R, Done = self.env.OneStep(a)
                if Done == 1:
                    self.R_save[n] = np.copy(R)
                    self.N_moves_save[n] = np.copy(i)
                    self.optimizer.zero_grad()
                    loss = 0.5 * (Q_values[a] - R) ** 2
                    loss.backward()
                    self.optimizer.step()
                    break
                else:
                    Q_values_next = self.Q_Net(torch.from_numpy(X_next).float(), torch.from_numpy(
                                                                                allowed_a_next).squeeze())
                    self.optimizer.zero_grad()
                    loss = 0.5 * (Q_values[a] - R - self.gamma * Q_values_next.max()) ** 2
                    loss.backward()
                    self.optimizer.step()
                X = np.copy(X_next)
                allowed_a = np.copy(allowed_a_next)
                i += 1
        print('=========== Finished training ============\n ')
        return self.R_save, self.N_moves_save

    def train_sarsa(self):
        self.reset_R_and_N_moves()
        self.reset_Network_Parameters()
        print("=========== Starting to train ============ ")
        print("----------- Method: SARSA ----------- ")
        print(f'number of episodes: {self.N_episodes}\ndicount factor: {self.gamma}\ndecay factor: {self.
                                                                        beta}')
        for n in tqdm(range(self.N_episodes)):
            epsilon_f = self.epsilon_0 / (1 + self.beta * n)
            Done = 0
            i = 1
            S, X, allowed_a = self.env.Initialise_game()
            Q_values = self.Q_Net(torch.from_numpy(X).float(), torch.from_numpy(allowed_a).squeeze())
            a = EpsilonGreedy_Policy(Q_values, allowed_a, epsilon_f)
            while Done == 0:
                Q_values = self.Q_Net(torch.from_numpy(X).float(), torch.from_numpy(allowed_a).squeeze())
                S_next, X_next, allowed_a_next, R, Done = self.env.OneStep(a)
                if Done ==1:
                    self.R_save[n] = np.copy(R)
                    self.N_moves_save[n] = np.copy(i)
                    self.optimizer.zero_grad()
                    loss = 0.5 * (Q_values[a] - R) ** 2
                    loss.backward()
                    self.optimizer.step()
                    break
                Q_values_next = self.Q_Net(torch.from_numpy(X_next).float(), torch.from_numpy(
                                                                            allowed_a_next).squeeze())
                a_next = EpsilonGreedy_Policy(Q_values_next, allowed_a_next, epsilon_f)
                self.optimizer.zero_grad()
                loss = 0.5 * (Q_values[a] - R - self.gamma * Q_values_next[a_next]) ** 2
                loss.backward()
                self.optimizer.step()
                X = np.copy(X_next)
                allowed_a = np.copy(allowed_a_next)
                a = a_next
                i += 1
        print('=========== Finished training ============\n ')
        return self.R_save, self.N_moves_save
```

APPENDIX B
REPRODUCE EXPERIMENTS

All experiments can be rerun by a single click!
If you want to use the stored data and visualize plots, click run button for **Visualize.py**
If you want to run all the experiments in one go from scratch, click run button for **PlayGame.py**

Appendix C
Try your new experiment

Any new experiment can be run by just a few lines! Suppose we want to run Q learning with discount factor = 0.99, decay speed = 0.0005 and 20000 episodes, **all you need is just creating a new python file with following code:**

```python
from chess_env import Chess_Env
from PlayGame import Agent
env = Chess_Env(4)
agent = Agent(env)
agent.set_discount_factor(0.99)
agent.set_decay_speed(0.0005)
agent.set_N_episodes(20000)
R, N = agent.train_q_learning()
```

and visualize in whatever way you want with R and N!