Reinforcement Learning

Stefanos Konstantinou[1]

[1]Department of Informatics, University of Zurich, Switzerland

Abstract In this writing, two Deep reinforcement learning techniques will be used in order to train an agent to checkmate an opponent in a game of chess. The two techniques that are incorporated are SARSA and Q-Learning. Additionally for SARSA, a comparison is to be made between two different sets of parameter values (β & γ). The analysis will be drawn using the rewards accumulated and moves taken to reach the reward, for each agent.

Introduction Reinforcement Learning entails the idea of learning through reinforcement methodology. This requires an agent learning through a mapping of actions and the consequences that follow based on its environment optimally in order to maximize future reward, [Sutton and Barto, 2018]. Moreover, in reinforcement learning it is important to highlight that there is no supervision or known outcomes at the beginning. Therefore, the agent that is being trained, learns through its own decision making and it is rewarded when the desired outcome is achieved. This process over various attempts of the agent shapes the decision that it takes based on the state or situation it finds itself in.

In order to make the agent to experiment various decisions there needs to be a desire to explore. Exploration means that the agent in each trial during the training process, attempts to take a random action instead of the action that is already known to be preferred and already knows where it will lead it to. This exploration aims for the agent to discover new ways to reach the reward, requiring less amount of actions. On the other hand, an agent needs to also make use of existing knowledge (exploitation). This means that exploration suits the purpose of enriching an agent's knowledge particularly at the beginning stages of its learning process.

Therefore, exploration and exploitation balance should always be carefully considered so that the agent is trained to aim for maximum rewards while keeping the amount of actions as low as possible.

In this project, the task is to implement a deep neural network capable of learning to perform checkmate in a game of Chess. The point is to evaluate two reinforcement learning approaches (SARSA & Q-Learning) based on reward collection and amount of actions taken.

Set-up The world of the chess board is made of a 4×4 grid size. Three pieces, 1× King, 1× Queen and 1× Opponent's King, are placed in a random location of the board. In the initial positions, the pieces are not causing any threats. The aim is for the agent to move its King or its Queen to checkmate the opponent's King. Also the opponent's king is moved randomly to a position where it is not threatened during each turn. There are 2 outcomes: **checkmate**, where the opponent's King has no squares that it can move but is threatened & **draw**, where the opponent's king has no moves but is not threatened. Checkmate gives a reward of 1 and draw a reward of 0.1. (see code in Appendix E)

Methodology Deep Reinforcement Learning is incorporated in this project in order to train the agent. Deep reinforcement learning combines the general architecture of Neural Networks and the components of Reinforcement learning. Essentially, Reinforcement learning is used as a universal approximator [Li, 2017]: meaning that the Neural network is going to be used as an approximator for the Reinforcement learning components such as the optimal policy policy, reward signal & value function (Q-values) components.
Q-Values:

$$Q(s, a; \theta) \tag{1}$$

where s is the state, a is the action and θ is the weights of the neural network. The weights are updated during learning so that it can best adjust to the appropriate Q-values of the task. This is done using backpropagation which is performed through the network's layers using the difference of the expected reward and the actual reward achieved after an action. In this project, stochastic gradient descent will be used to update the weights. Additionally, the activation function to be used is RELU. (see code in Appendix A)

Temporal Difference algorithms to be used The first algorithm to be used is **SARSA**, an on-policy control method to find the optimal policy [Li, 2017]. The update rule for SARSA is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{2}$$

where α is the learning rate, γ is the discount factor. It must be noted that γ controls the importance of future rewards, so the smaller the value, the less future rewards are considered. Also Q(s',a') is the Q-values of the next step.

The second algorithm is **Q-learning**, an off-policy control method to find the optimal policy [Li, 2017]. The update rule of Q-learning is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)] \tag{3}$$

The off-policy characteristic of Q-learning is expressed by the max operation used in its update rule.

This means that it ignores the policy used and chooses only the highest reward of the next state i.e.:

$$max_{a'}Q(s', a')$$

(see code in Appendix B)

In more conceptual terms, Q-learning only considers the best case that you can get in the nest time step. Consequently, the agent especially in the beginning, will make more errors since the currently known optimal solution (at that particular time-frame) might not be truly optimal. On the other hand, SARSA keeps the policy that is used, and tends to follow a safer approach (in terms of errors).

Exploration Vs Exploitation The method that will be used to balance the exploration and exploitation, is ϵ-greedy. This process is carried simply by a random probability (p).

$$\text{Action at time(t)} \leftarrow \begin{cases} \max_a Q(s_t, a), & \text{if } p > \epsilon \\ \text{random action}, & \text{otherwise} \end{cases}$$

But the value of ϵ will vary depending on the amount of episodes that passed. For this the speed of the decaying trend of ϵ is introduced, β:

$$\epsilon \leftarrow \frac{\epsilon_0}{1 + \beta \times n} \tag{4}$$

where ϵ_0 is the initialised value of ϵ, and n is the current episode. The smaller the value of β, then the smaller decay of ϵ and thus, the agent incorporates exploration for more episodes. (see code in Appendix C)

Experimentation Set-up The aim of this paper is to give a comparison of the 2 algorithms mentioned. Also, a further comparison is to be made on SARSA by changing the default values of the discount factor γ and β the speed of decaying trend of ϵ. It must be noted that Q-learning model has the same parameters as the Default Parameters. (see values of β & γ values in Appendix D)

For evaluation, two plots will be produced per each comparison that show the reward per game and the number of moves per game vs training time. Since the plots are expected to be noisy, exponential moving average (EMA) will be used.
EMA to denote the reward plotted at time t (R_t) by using reward R collected at t episode:

$$R_t \leftarrow (1 - \alpha)R_{t-1} + \alpha R \tag{5}$$

where α is the smoothing constant.

Results and Discussion

In the first experiment, 2 parameter combinations was tested for the SARSA algorithm, which can be illustrated in figure 1. It can be clearly seen that SARSA with default parameters performed better. SARSA with Non-Default parameters had a lower β and the impact can be seen by the delayed growth of Rewards at the first 100000 Episodes. Moreover, γ was also lower, and due to the agent considering less future rewards, it performed worse than the agent with default parameters. On the other hand, the Number of moves resulted to have the same value at the end, but the agent with default parameters experienced episodes with higher peaks and this can mean that it was able to balance the exploitation and exploration better.

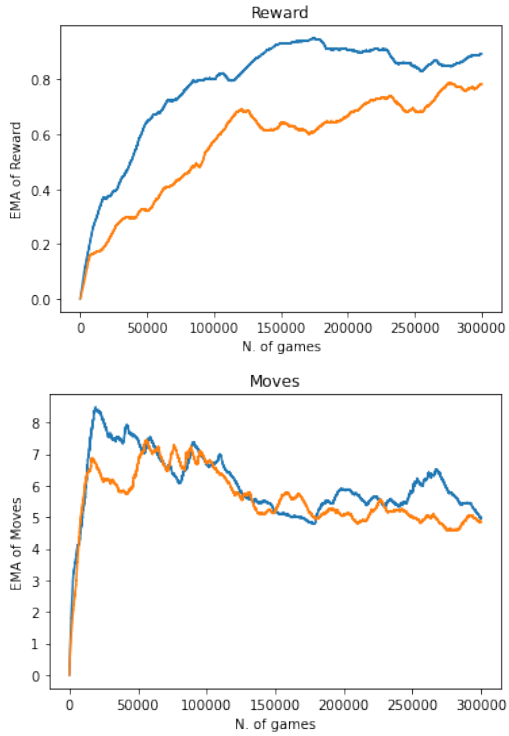

Figure 1: SARSA Default Params (blue) Vs SARSA Non-Default Params (orange)

In the second experiment, SARSA and Q-learning algorithms were compared, which can be illustrated in figure 2. It can be clearly seen that it was a close tie, but in the end Q-learning achieved a better reward result. For the majority of the episodes, SARSA had more reward value but also at least higher number of moves per episode. This is due to the exploratory characteristic of SARSA. Because of Q-learning's optimal solution strategy, it had a slower growth of reward and optimally kept its moves lower than SARSA; a direct result of its greediness towards future rewards.

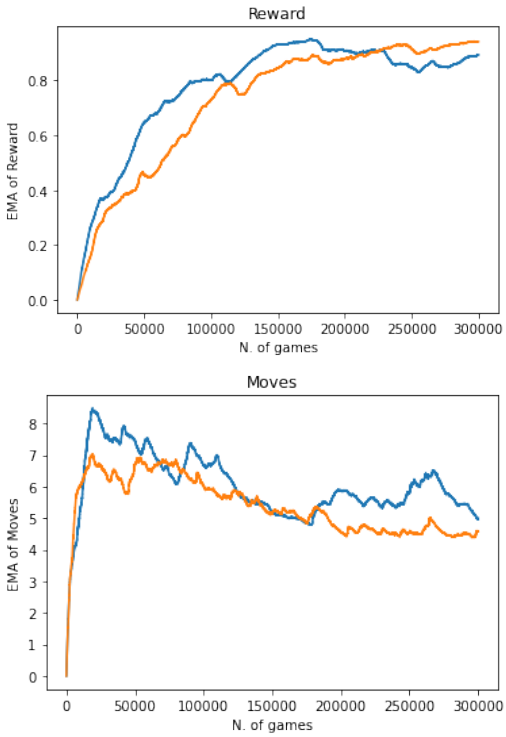

Figure 2: SARSA Default Params (blue) Vs Q-Learning (orange)

Conclusions To conclude, in this paper it can be seen that the proper parameterisation of a model such as SARSA, can greatly influence performance. Moreover, SARSA and Q-learning was compared and the results of these 2 algorithms were very close. But, it is worth noting that the different nature of the algorithms could be illustrated by tracking their progress over training.

How to reproduce To see the results or run the project, use the link offered below in this section and open the jupyter notebook file "Chess RL.ipynb". `https://github.com/acd17sk/DRL-Chess`

References

[Li, 2017] Li, Y. (2017). Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Appendices

Appendix A

```
# input to h
x1 = np.dot(W1, transform_input(X)) + bias_W1
h1 = np.maximum(x1, 0)

# h to  output layer
x2 = np.dot(W2, h1) + bias_W2
Q = np.maximum(x2, 0)
```

Figure 3: Forward propagation

```
# back-probagation
deltaw2 = (targ_rew - Q[a_agent]) * np.heaviside(Q, 0)
deltaw1 = np.heaviside(h1, 0) * np.dot(W2.T, deltaw2)

W2 += eta * np.outer(deltaw2, h1)
bias_W2 += eta * deltaw2

W1 += eta * np.outer(deltaw1, X)
bias_W1 += eta * deltaw1
```

Figure 4: backpropagation

Appendix B

```
    a_agent_next = EpsilonGreedy_Policy(Q_next, a_next, epsilon_f)

    # For Sarsa
    targ_rew = R + gamma * Q_next[a_agent_next]


# back-probagation
deltaw2 = (targ_rew - Q[a_agent]) * np.heaviside(Q, 0)
```

Figure 5: Update for SARSA

```
    # For Q-learning
    targ_rew = R + gamma * max(Q_next[a_next])


#back-probagation
deltaw2 = (targ_rew - Q[a_agent]) * np.heaviside(Q, 0)
```

Figure 6: Update for Q-learning

Appendix C

```
epsilon_f = epsilon_0 / (1 + beta * n)   ## DECAYING EPSILON
```

Figure 7: Decay of ϵ

```
def EpsilonGreedy_Policy(Qs, actions, eps):

    N_ac = np.shape(actions)[0]

    rand_value = np.random.uniform(0,1)

    if rand_value < eps:
        a = actions[np.random.randint(0, N_ac)]
    else:
        a = actions[np.argmax(Qs[actions])]
    return a
```

Figure 8: ϵ-greedy policy

Appendix D

	beta	gamma
Default Params	0.00005	0.85
Non-Default Params	0.000005	0.7

Table 1: Two combinations of Parameters

Appendix E

```
# CASE OF CHECKMATE
if np.sum(self.dfk2_constrain) == 0 and self.dfq1[

    # King 2 has no freedom and it is checked
    # Checkmate and collect reward
    Done = 1        # The epsiode ends
    R = 1           # Reward for checkmate
    allowed_a=[]    # Allowed_a set to nothing (en
    X=[]            # Features set to nothing (end

# CASE OF DRAW
elif np.sum(self.dfk2_constrain) == 0 and self.dfq

    # King 2 has no freedom but it is not checked
    Done = 1         # The epsiode ends
    R = 0.1        # Reward for draw
    allowed_a=[]     # Allowed_a set to nothing (e
    X=[]             # Features set to nothing (en
```

Figure 9: Case of Win and Draw Code