Reinforcement Learning On a 4x4 Chess Board with 3 Pieces

Aslihan Ilgin Okan - aca19aio - Assignment Repo
The University Of Sheffield
COM3240 - Prof. Eleni Vasilaki

1 Introduction

Deep reinforcement learning refers to goal-oriented algorithms that are learning to achieve a complex goal over a series of actions. These algorithms map states and actions within an environment to rewards, applying unison of function approximation and target optimisation [5]. There are two different reinforcement learning (RL) types; on-policy and off-policy. In RL terms, behaviour of the algorithm is defined in terms of policies. On-policy RL evaluates and improves policy according to the results of the actions taken, whereas off-policy figures out the optimal action regardless of agent's motivation [7] (which could be avoiding poor results). Therefore, on-policy RL updates its behaviour during training to get to the target using the best $Q(s, a)$ pairs [6], which is the expected reward of being in a state after an action taken; and off-policy RL keeps behaviour for reaching the target the same. Q-Learning could be given as an example to an off-policy and SARSA to an on-policy RL algorithm.

In this project, the mentioned algorithms SARSA and Q-Learning will be explored in depth through a reduced chess game environment. The game environment has a 4x4 chess board with three pieces: a King, a Queen, and the Opponent's King. During the game, the player's aim is to checkmate the opponent and the opponent's King will be moving to a random position where it is not checked when it's their turn. The two algorithms that will be explored in this project, Q-Learning and SARSA, are examined within this environment and discussed according to it.

2 Q-Learning vs. SARSA

Both SARSA and Q-Learning algorithms are largely similar in the way they execute, however, they have aspects that cause them to differ (refer to Figures 1 and 2 for their pseudocodes). Both algorithms consider two steps: the current step and the step that will be taken after that. Q-Learning has a target policy that is always greedy, meaning it will always choose the action that results in the maximum $Q(s, a)'$ value, which is the subsequent action that is in the optimal path to the goal. Therefore, Q-learning learns from optimal actions that are succeeding at reaching the goal but will not learn much from wrong actions taken since it will always choose next actions with a greedy policy. This behaviour of Q-learning is what makes it "off-policy", meaning that it does not learn from wrong actions and aims to reach the target as efficiently as possible.

Algorithm 1 Deep Learning SARSA algorithm

```
epsilonValue ←decayingEpsilon
Done ←0
stepCounter ← 1
S, X, allowed_a ←Initialise game
while Done do
    initialisedGradients ←0
    QValues ←Forward propagate
    Action ←Epsilon greedy policy
    S',X', allowed'_a ←Take a step in the environment
    if Done is 1 then
        errorSignal ←Target QValues, QValues
        updatedBiases, updatedWeights ←Back propagate
        Rewards, Moves ←Save reward and total number of moves
        break
    end if
    QValues' ←Forward propagate
    Action' ←Greedy policy
    errorSignal ←Target QValues, QValues
    updatedBiases, updatedWeights ←Back propagate
    S, X, allowed_a ← S', X', allowed'_a
    stepCounter ← stepCounter + 1
end while
```

Figure 1: Deep Q-Learning pseudocode. Ref: [2]

On the other hand, SARSA is an on-policy reinforcement learning algorithm that updates its strategy to reach the target during training. SARSA's target policy is the same as its behaviour policy [6], meaning instead of considering the step that will get the agent to the goal fastest in the exploration of the second step (like how Q-Learning does), it chooses the next action with a probability of exploring a random or high $Q(s, a)$ value action. If the algorithm receives a reward that indicates a wrong action during training, it will try to avoid actions that will lead the agent to earn that bad reward in the later stages. This allows the agent to take safer paths rather than optimal ones that Q-learning will be likely to take. SARSA is useful when the aim is to optimise the value of the exploring agent since SARSA allows for exploration of the environment instead of focusing on reaching the goal the fastest like Q-learning does. SARSA is more conservative than Q-Learning, meaning if there is a risk within the optimal path to the target, SARSA will tend to avoid this path [8].

To find the most efficient and safest way to the goal, the chosen deep learning algorithm will be different according to the problem at hand. Comparing and contrasting these two algorithms provide a comprehensive understanding of their strengths and limitations in the context of deep reinforcement learning. It is important to consider the trade-offs between their exploration-exploitation policies and update strategies. If exploration of different paths is more important, the obvious choice will be SARSA; however, if efficiency regardless of risks possible is more important, the choice will be Q-Learning. Ideally, if existed, an algorithm that is a mixture of both

SARSA and Q-Learning would be used. However, this could be mimicked by modifying the hyperparameters the algorithms use to influence their behaviour which will be discussed more in detail in Section 3.1.

Algorithm 1 Deep Learning SARSA algorithm

```
epsilonValue ←decayingEpsilon
Done ←0
stepCounter ← 1
S, X, allowed_a ←Initialise game
while Done do
    initialisedGradients ←0
    QValues ←Forward propagate
    Action ←Epsilon greedy policy
    S',X', allowed'_a ←Take a step in the environment
    if Done is 1 then
        errorSignal ←Target QValues, QValues
        updatedBiases, updatedWeights ←Back propagate
        Rewards, Moves ←Save reward and total number of moves
        break
    end if
    QValues' ←Forward propagate
    Action' ←Epsilon greedy policy
    errorSignal ←Target QValues, QValues
    updatedBiases, updatedWeights ←Back propagate
    S, X, allowed_a ← S', X', allowed'_a
    stepCounter ← stepCounter + 1
end while
```

Figure 2: Deep SARSA pseudocode

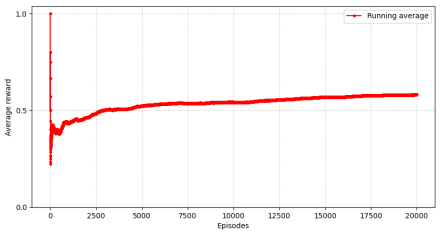


Figure 3: Rolling average of rewards per episode.

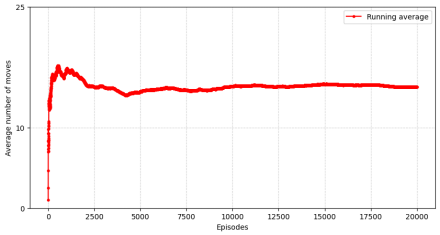


Figure 4: Rolling average of moves per episode. For initial indicative values given. (Refer to Figure 5)

3 Deep SARSA Implementation

SARSA has been chosen as the first algorithm to be implemented using a deep neural network with a sigmoid activation function. SARSA makes use of the board's new state S' and explores the surrounding Q values with a policy named Epsilon-Greedy (see Figure 6). SARSA neural network (which can be called "the agent" interchangeably in this context) chooses the action to perform next by looking "one-step-ahead" without the obligation of taking that explored action in the next step, however, it uses the $Q(s', a')$ value to update the learning weights. This is important in a chess game since to have more chance to win the game, the agent needs to consider its next steps. Figure 2 illustrates the steps of how SARSA within the concept of deep RL has been implemented. Since every step of the game is treated as a learning opportunity for the agent, regardless of the reward being good or bad, the network will update its weights in every step according to the reward.

As the game progresses, the network will start to learn the actions that it needs to take in certain situations as a result of the agent earning a good reward when it takes a good action. Figure 3 plots the progression of the rolling average of the reward the agent gains per episode and Figure 4 plots the progression of the rolling average of moves the agent makes per episode. Evidently, as the episodes progress the agent learns from its actions and improves the average number of rewards that it gets, demonstrating that it attains a greater number of wins over time. The absence of convergence in Figure 3 suggests that at episode 20k the model is still learning and has yet to achieve convergence.

The sudden peak of average reward values at the beginning of the training solely indicates that the agent has started the training progress by winning some games, hence earning rewards per a small number of games that results in a high ratio. In the meantime, due to the fact that the network does not have a lot of training in the earlier games, it will lose games with a small total number of moves. However, over the course of training, this average decreases as the network starts to learn sensible actions and reduces the number of wrong actions taken. Although the average number of moves appears to have stabilised within 20k episodes, considering the plots 3 and 4 together will reveal that the average number of moves will decrease albeit at a slow pace.

However, the performance of the agent is not solely contingent on the learning method itself but on how the rewards are presented to it as well. This "how" is controlled by a few parameters that the network is set up with.

3.1 Varying Hyperparameters

There are a couple of hyperparameters that can be modified to explore different learning approaches. These hyperparameters are ϵ (epsilon), β (beta), γ (gamma), and η (eta/learning rate) which correspond to particular aspects of how the neural network learns. Figure 5 shows the initial given values of these hyperparameters. It is important to have the right values for the hyperparameters ϵ, β, γ, and η to train the neural network as efficiently as possible. Several different combinations of these hyperparameter values have been explored using the SARSA algorithm as well as their effect on the training as individual parameters. Figures 12, 13, 14 (ref: Appendix) demonstrate the

outcomes of the hyperparameter value modifications. The following paragraphs will discuss how modifying each of these parameters will affect the learning of the neural network.

```
epsilon_0 = 0.2
beta = 0.00005
gamma = 0.85
eta = 0.0035
```

Figure 5: Initial values provided for neural network parameters.

3.1.1 Varying ϵ

The ϵ value indicates the extent of importance the neural network gives to exploration and/or exploitation. ϵ values range between 1 and 0. A higher value indicates a higher rate of exploration, meaning the agent will be more likely to take random actions, preventing the network from exploiting the information the model has so far[1]. Whereas, a lower value indicates the opposite where the agent will have a fixed exploration rate[3] to decide if it is going to choose a random action (exploration) or take a greedy action (exploitation). The Epsilon-Greedy policy (see Figure 6 for pseudocode) implements this decision-making process between a probabilistic exploration or a greedy action that was just mentioned. The epsilon value should be decided according to the environment the neural network is getting trained for as well as the analysis intentions of the network. Different values for the starting ϵ value have also been explored to observe the effect it has on the training and it has been derived that having the starting ϵ value of 0.2 is the most optimal choice to allow other hyperparameters to influence ϵ itself during training. An alternative ϵ value setting approach is using a method called decaying-epsilon. Decaying-epsilon is a way of diminishing the value of epsilon as the training is performed over n number of games which will decrease the probability of exploration over time.

Algorithm 2: Epsilon-Greedy Action Selection

Data: Q: Q-table generated so far, : a small number, S: current state
Result: Selected action
Function *SELECT-ACTION(Q, S, ϵ)* **is**
  n ← uniform random number between 0 and 1;
  if $n < \epsilon$ **then**
    A ← random action from the action space;
  else
    A ← maxQ(S,.);
  end
  return selected action A;
end

Figure 6: Pseudocode for EpsilonGreedy algorithm. (Ref:[1])

3.1.2 Varying β

The formula for decaying epsilon decreases the value of ϵ to be used in each game with a rate of β times number of games played so far (n). β value regulates how fast the epsilon value is decaying. In Figure 7, $epsilon_0$ value represents the starting value of epsilon and $epsilon_f$ represents the epsilon value that is going to be considered for the Epsilon-Greedy policy. Therefore, a higher value of β will result in ϵ value decaying faster, discarding the option to explore random states sooner, as it was mentioned earlier that a lower epsilon value leads the agent to take more greedy actions. Respectively, a lower value of beta will result in slower decay of the ϵ value which preserves the high probability of the agent exploring random states within the environment for longer. From Figure 12 it can be concluded that the optimal value for β is any value between 0.00005 and 0.0005 when keeping the other hyperparameter values in their initial values. The other values for β result in the convergence of the average reward value much earlier than others, therefore, are not desirable to allow the network to train efficiently.

$$epsilon_f = epsilon_0/(1 + \beta * n)$$

Figure 7: Decaying epsilon formula.

3.1.3 Varying η

The learning rate, η, also plays a significant role in setting up the neural network to produce efficient results. The learning rate is related to the loss value within training. Not finding an appropriate value for this hyperparameter results in divergent behaviours where with a high η value the network constantly misses the optimal solutions or substantially increases the learning time with a low η value[4].

3.1.4 Varying γ

The hyperparameter γ regulates the importance the agent gives to immediate and future rewards. A value of 1 for γ will make every reward weigh the same as every other reward in the eyes of the agent. This is not effective since the agent will not explore new paths or take risks but will prefer to take safe paths in which it is certain to receive good rewards. Likewise, if γ is closer to zero, the agent will tend to consider only immediate rewards[3]. From the initial analysis of graphs in Figure 13, the optimal value for γ, while keeping the other hyperparameters in their initial values, is 0.50.

In light of the discussion made in this section, the combination of ϵ=0.2, $\beta = 0.0005$, $\gamma = 0.5$, and $\eta = 0.005$ values are the optimal choices from the various permutations of hyperparameters that have been

experimented with that allows the network to train as efficiently as possible.

4 Deep Q-Learning Implementation

As a second algorithm, deep Q-Learning has been implemented within the same environment to observe the differences it displays compared to the SARSA algorithm. In reference to the discussion in Chapter 2, Q-Learning shares some common traits with SARSA, however, differs in how it chooses an action to be carried out next. Figure 1 contains the pseudocode of how deep Q-Learning algorithm has been implemented.

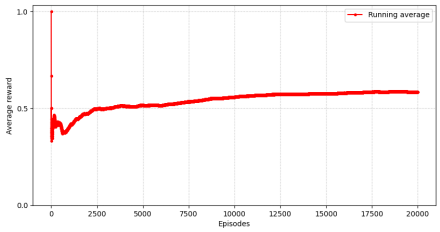


Figure 8: Rolling average of rewards per episode.

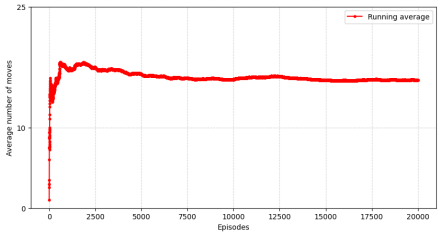


Figure 9: Rolling average of moves per episode.

Figure 10

When the best hyperparameter values are plugged in to both SARSA and Q-Learning algorithms, it is observable in Figure 11 that SARSA performs better than Q-Learning. In accordance with the earlier discussion of these algorithms in Section 2, this outcome is consistent with previous research about how each of the algorithms explore and choose future actions. Since a chess game requires the analysis of foreseeable moves to beat the opponent, it is reasonable to deduce that SARSA will perform better over Q-Learning. Furthermore, it is evident that for the reduced chess environment Q-Learning is tending towards converging earlier than SARSA, indicating that using Q-Learning for this environment will yield a slower and less efficiently trained model to beat an opponent.

In chess, there are many ways of winning a game, the fastest methods having eight moves or less[9]. However, small number of total moves in a game is not always an indicator of an efficient game since if an agent is untrained, it will lose a game with small number of total moves in the beginning of training. Therefore, when analysing the average number of moves per episode plots, the average values should follow the trend of increasing-stabilising or increasing-decreasing-stabilising depending on the given board. Figure 4 and 15a compare the average number of moves per episode for the SARSA algorithm; Figure 9 and 15b compare the average number of moves per episode for Q-Learning, both with initial indicative and most optimal hyperparameter values respectively. Based on the analysis of these figures, it can be concluded that the total number of moves decrease as the agent trains.

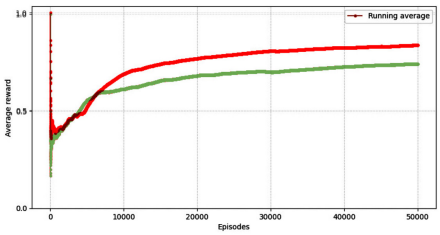
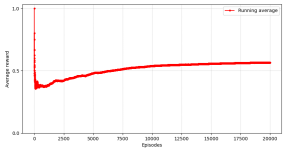


Figure 11: Average rewards for SARSA vs Q-Learn algorithms. $Red : SARSA, Green : Q - Learning$.

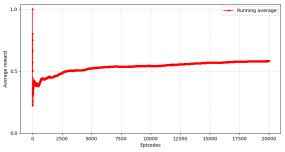
5 Conclusions

In this report, two algorithms, SARSA and Q-Learning, has been explored to observe their effects on training a deep neural network within a reduced chess environment. The algorithms have been compared according to their implementation, strengths, and weaknesses. It was mentioned that choosing one or the other depends on the training needs and environment. It has been discussed how the hyperparameters ϵ, β, γ, and η could affect the way a neural network learns. These parameters need to be tuned in accordance with each other due to each of them affecting different aspects of the learning. The effect of these hyperparameters have been analysed on SARSA and discussed afterwards. From modifying these hyperparameters on SARSA network, it has been revealed that they influence the efficiency of the network notably. It has been concluded that the values ϵ=0.2, $\beta = 0.0005$, $\gamma = 0.5$, and $\eta = 0.005$ are the most optimal. Q-Learning has been analysed using these optimal values to provide network training comparison. This analysis has been made through average number of rewards and moves per episode. It has been noticed that SARSA achieved better results in the chess game environment. All in all, it has been concluded that deep learning algorithms perform differently according to the environment in question and the tuning of their hyperparameters in relation to the training objective.

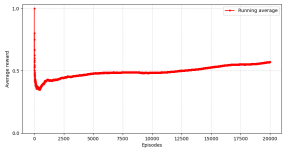
6 Appendix



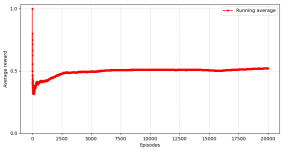
(a) β value 0.000005



(b) β value 0.00005

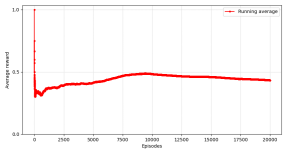


(c) β value 0.0005

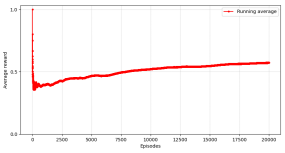


(d) β value 0.005

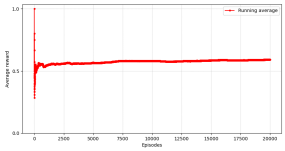
Figure 12: Average rewards per episode plotted with different β values.



(a) γ value 0.10

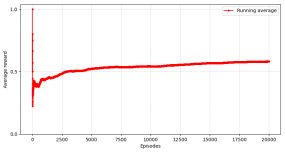


(b) γ value 0.50

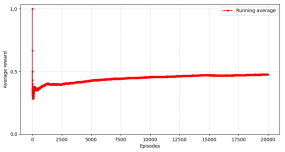


(c) γ value 0.95

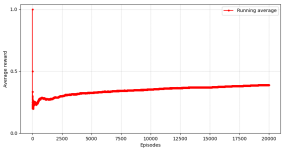
Figure 13: Average rewards per episode plotted with different γ values.



(a) ϵ value 0.2

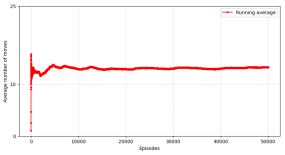


(b) ϵ value 0.5

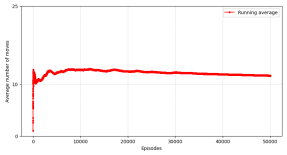


(c) ϵ value 0.8

Figure 14: Average rewards per episode plotted with different ϵ values.



(a) SARSA. Refer to Figure 4 for initial indications values plot.



(b) Q-Learning.

Figure 15: Rolling average number of moves per episode for SARSA and Q-Learning.

7 Additional Notes

7.1 Exemplar Code Snippets

This section includes novelty introduced to the given code. Figure 16 demonstrates how SARSA chooses and action using Epsilon-Greedy policy to explore future steps and their Q values.

Figure 17 demonstrates the implementation of plotting the rolling average of rewards per episode.

7.2 Reproducing Results

Packages used while development:

Library	Version
numpy	1.23.5
matplotlib	3.7.1

There is no need to set a seed value within *numpy* as this has been done within the files *assignment.py/Assignment.ipynb*, *generate_game.py*, and *Chess_env.py*.

There are two possible ways to run the code:

Using Jupyter Notebook

Run each cell within Assignment.ipynb. Choose one of the deep learning algorithms to run and skip the other one.

Running assignment.py

This is a Python file that consists of cells that you can run within VSCode. Alternatively, you can run the code as normal as follows:

```
python assignment.py
```

Make sure you comment out one of the deep learning algorithms before running the program.

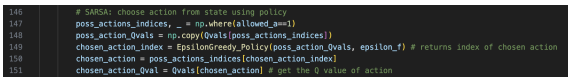


Figure 16: How SARSA chooses an action and finds the Q value associated.

Figure 17: Plotting the rolling average of rewards per episode.

References

[1] Baeldung. Epsilon-greedy q-learning — baeldung on computer science. https://www.baeldung.com/cs/epsilon-greedy-q-learning, 2020.

[2] Tim Eden, Anthony Knittel, and Raphael van Uffelen. Reinforcement learning - algorithms. https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html, 2019.

[3] Shreyas Gite. Practical reinforcement learning — 02 getting started with q-learning. https://towardsdatascience.com/practical-reinforcement-learning-02-getting-started-with-q-learning-582f63e4acd9#:~:text=Gamma%20varies%20from%200%20to, 04 2017.

[4] Jeremy Jordan. Setting the learning rate of your neural network. https://www.jeremyjordan.me/nn-learning-rate/, 03 2018.

[5] Chris V. Nicholson. A beginner's guide to deep reinforcement learning. https://wiki.pathmind.com/deep-reinforcement-learning.

[6] Pankaj Kumar Porwal. Sarsa (state action reward state action) learning - reinforcement learning - machine learning. https://www.youtube.com/watch?v=FhSaHuC0u2M, 04 2020.

[7] Ram Sagar. On-policy vs off-policy reinforcement learning. https://analyticsindiamag.com/reinforcement-learning-policy/, 04 2020.

[8] Neil Slater. When to choose sarsa vs. q learning. https://stats.stackexchange.com/questions/326788/when-to-choose-sarsa-vs-q-learning, 02 2018.

[9] Colin Stapczynski. 10 fastest checkmates. https://www.chess.com/article/view/fastest-chess-checkmates, 03 2022.