COM3240: Adaptive Intelligence
Assignment 2: Reinforcement Learning

Alexandra-Cristina Butoi
The University of Sheffield

***Abstract*— This paper describes a Deep Reinforcement Learning algorithm for learning how to checkmate the opponent in a simplified game of Chess. The learning curve of the deep neural network is analysed when 2 algorithms are used, namely Q-Learning and SARSA, both using the $\epsilon - greedy$ policy. Also, the performance of the network is compared when different values for the discount factor γ and the decaying trend β of ϵ are used. Finally, the Gradient Descent optimisation algorithm is replaced with RMSprop in the attempt to solve the problem of exploding gradients.**

I. INTRODUCTION

Reinforcement Learning is a Machine Learning paradigm in which a software agent learns how to behave in some environment in order to maximise the reward. The agent follows a stochastic process in order to explore the space of possible actions and find the actions that maximise the long-term reward [1]. The agent uses the collected rewards as feedback by which it measures the success of its actions. For each pair of state s and action a, the agent estimates the expected reward $Q(s, a)$ that will be obtained by taking action a from state s. The Q-values estimate the quality of action a taken in state s.

The algorithm by which an agent chooses its actions is called a *policy*. $\epsilon - greedy$ (Appendix B, Figure 10) is a frequently-used policy in which the agent explores with probability ϵ and it exploits with probability $1 - \epsilon$. The value of ϵ decreases over time so that the agent explores more at the beginning by performing random actions and observing the collected rewards. As it starts learning, the agent exploits the actions that are expected to give the highest reward (i.e. highest $Q(s, a)$). This problem (whether to explore in the attempt to find the best possible action or exploit the best action given what the agent has learned so far) is called the "Exploration vs. Exploitation" problem.

In Deep Reinforcement Learning, a deep neural network is used for learning the Q-values. The input layer encodes information about the environment in a particular state (e.g. the position of the agent) while the output layer maps each action with its Q-value. As the agent learns, the weights are adjusted so that it improves the estimates of the Q-values. The error is the difference between the expected reward and the actual reward received when performing a particular action [2]. The errors are backpropagated through the network like in Supervised Learning.

II. TASK DESCRIPTION

The goal of this research work is implementing a two-layer neural network that is capable of learning how to checkmate the opponent in a simplified game of Chess. The Chess board is a 4x4 grid on which there are only 3 pieces: the King, the Queen and the opponent's King. The agent must choose actions using its King and Queen that are expected to perform checkmate to the opponent's King while the opponent chooses random actions that do not threaten its King. The game can end in 2 states: checkmate (when the opponent's King is threatened and there isn't any move that can make it escape this threat) or stalemate (when the opponent's King is not currently threatened but any action will make it become threatened), resulting in a draw. The agent receives a very small reward when the game ends in a draw, thus the network must learn how to avoid the moves that are expected to lead to this state.

III. METHODOLOGY

We use a two-layer neural network to learn how to end the game in a checkmate. The network takes as input the position of each of the 3 pieces (as a binary matrix where 1 represents the position of the piece), the degree of freedom of the opponent King and a binary variable indicating whether the opponent King is checked, thus the input layer contains $3 \times 4 \times 4 + 2$ nodes. We use the ReLU activation function and optimise the network using Q-learning/SARSA with backpropagation (Appendix B, Figure 8).

We use an Exponential Moving Average with $\alpha = 1/10000$ for plotting the reward per episode and number of moves per episode. The average reward R' at episode t is calculated using the following formula:

$$R'_t = (1 - \alpha)R'_{t-1} + \alpha R_t$$

A similar formula is used for the number of moves per episode. Figure 1 and Figure 2 show the reward per episode and the number of moves per episode using Q-Learning. As more episodes are played, the agent performs a checkmate more frequently, using fewer moves.

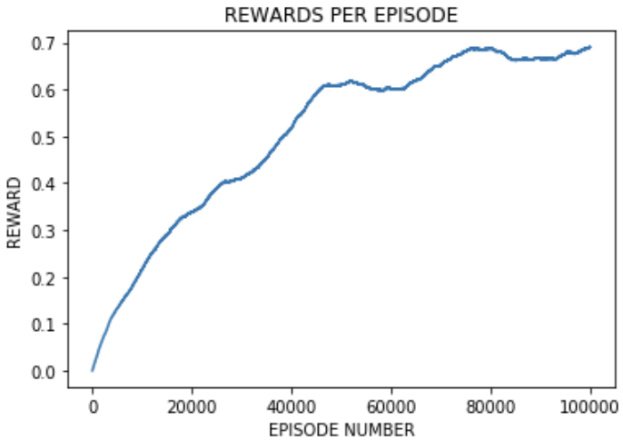

Fig. 1. Reward per episode using Q-learning.

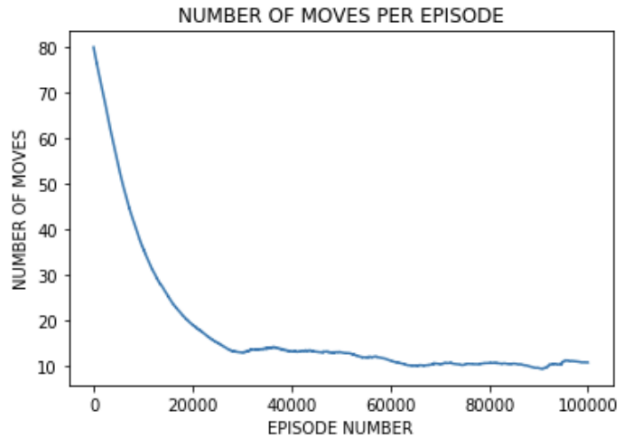

Fig. 2. Number of moves per episode using Q-learning.

IV. REINFORCEMENT LEARNING ALGORITHMS

A. Q-Learning

Q-learning is an off-policy reinforcement learning algorithm that uses the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

where γ is a discount factor which indicates the importance of future rewards. With a small γ, the agent will tend to consider only current rewards while with very high γ the agent will try to obtain a high long-term reward.

In Q-Learning, the estimates are updated using the maximum Q value from the possible actions available in the next state, irrespective of the action that should be taken based on the policy (Appendix B, Figure 9). For this reason, it is an off-policy algorithm. The policy is used only to determine what action the agent should take next [3].

B. SARSA

Unlike Q-Learning, SARSA is an on-policy algorithm, whose name is given by the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which is used for updating the Q values. SARSA uses the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

The agent uses the policy to choose what action to perform and then uses the same action to update the estimates (Appendix B, Figure 9). Q-Learning learns directly the optimal solution, but it is more likely to make errors while learning. SARSA is more conservative, learning a safer but sub-optimal solution. When using SARSA, the agent will tend to avoid the dangerous optimal solution and will learn how to use it when it starts exploring less frequently [4].

The "Cliff Walking" problem best illustrates the difference between Q-learning and SARSA. In the Cliff Walking problem, the agent needs to move in a grid from the "start" state to the "goal" state without stepping in a region marked as the cliff (see Figure 3).

When using Q-Learning, the agent will learn the optimal path which is right along the edge of the cliff. Occasionally, the agent will jump off the cliff while following this path. SARSA, on the other hand, will learn the longer but safer path, moving far from the edge of the cliff. When training a robot in a real environment, SARSA would be preferable as it is more conservative and it would try to avoid the risk of damaging the robot during training.

Figure 4 illustrates the reward per episode using both SARSA and Q-Learning. Although Q-Learning learns

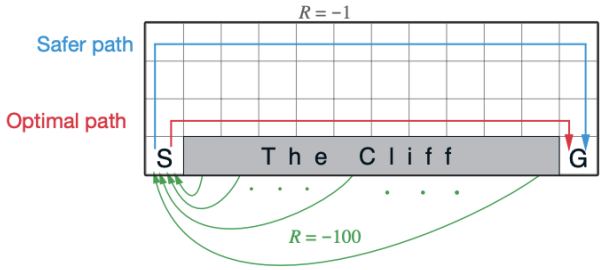

Fig. 3. The environment in the "Cliff Walking" problem. Source: [2]

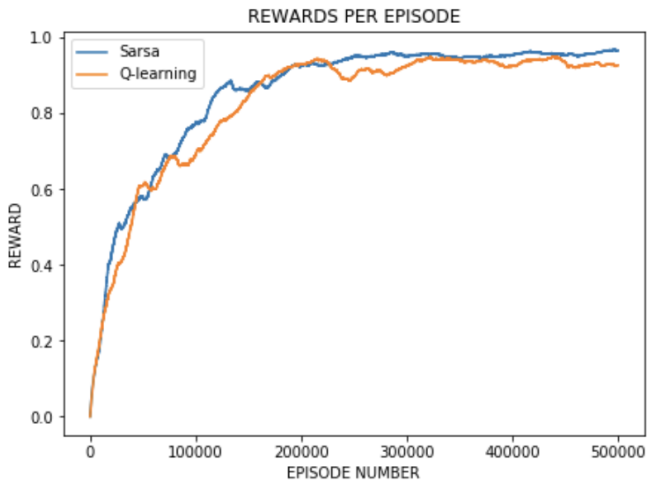

Fig. 4. Reward per episode using Q-Learning and SARSA.

faster at the beginning, SARSA proves to perform better over a large number of episodes, achieving a higher reward per episode. When the agent uses the SARSA algorithm, it takes the longer path in order to avoid ending the game in a draw and receiving only a small reward.

V. DISCOUNT FACTOR γ

As immediate rewards are preferable to distant rewards, we want to give higher weight to near rewards than rewards received further in the future. The discount factor γ indicates the importance of future rewards. If γ is small the agent will consider only rewards obtained in the near future. If γ is large, the agent will look further into the future for a higher long-term reward.

Figure 5 shows the reward per episode for different values of γ. The highest reward per game is obtained for the largest γ (0.95) which indicates that it is preferable to have a high long-term reward rather than a smaller immediate reward. In our simplified game of Chess it is better to play more actions in order to perform a checkmate rather than ending the game in a draw using fewer moves.

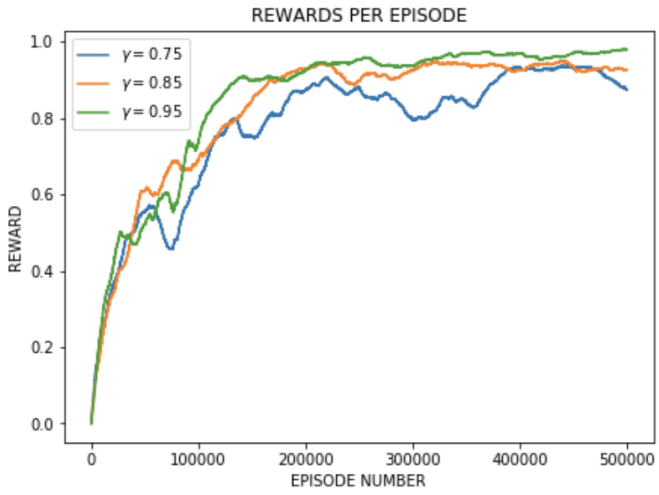

Fig. 5. Reward per episode using different values for γ.

VI. SPEED β OF THE DECAYING TREND OF ϵ

In the $\epsilon - greedy$ policy, the agent explores with probability ϵ and exploits with probability $1-\epsilon$. At the beginning, the agent should explore with a high probability, but as it learns more by playing more episodes, the probability of exploration should decrease. In order to achieve this, ϵ is discounted in each iteration in order to have a smaller probability to explore:

$$\epsilon_n \leftarrow \frac{\epsilon_0}{1+\beta \times n}$$

where n is the number of the current episode and ϵ_0 is the initial ϵ. A large value for β makes ϵ decay faster, thus the agent explores less as the episode number increases.

Figure 6 shows the reward per episode for different values of β. With small values for β (e.g. $\beta = 5 \times 10^{-6}$), the agent learns faster at the beginning because it explores more but in the long run in performs worse because it performs random, sub-optimal actions more frequently. With large values for β (e.g. $\beta = 5 \times 10^{-4}$), ϵ will decay very fast, thus the agent will explore only with a very small probability. As the agent does not explore enough, the chances that it finds the optimal solution are smaller. The agent achieved the best performance when $\beta = 5 \times 10^{-5}$ as ϵ decreases neither too fast nor too slow, having a good balance between exploration and exploitation.

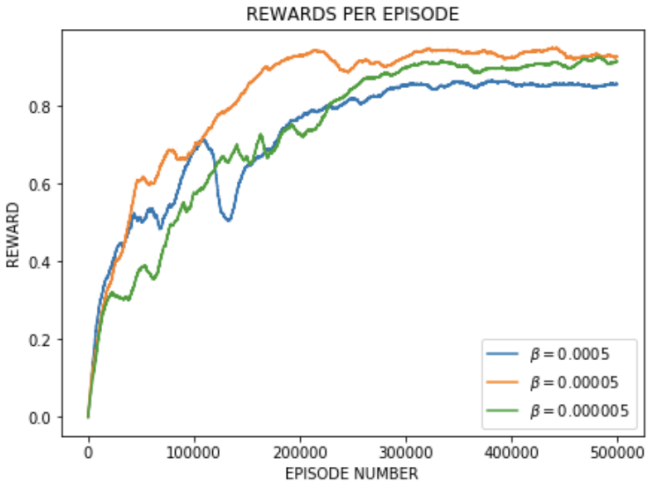

Fig. 6. Reward per episode using different values for β.

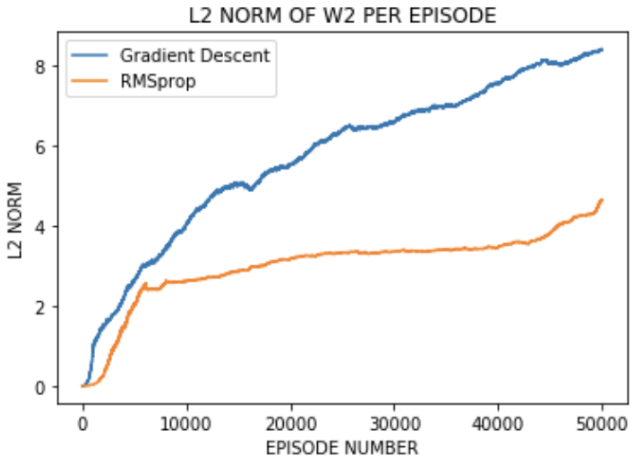

Fig. 7. Growth of L2 norm of W2 using Gradient Descent and RMSprop.

VII. RMSprop

When training a deep neural network, the errors can accumulate and lead to very large gradients. When these gradients are used to update the weights, the network becomes unstable and the weights can grow exponentially, resulting in numerical overflow. One possible solution to this problem is using the RMSprop optimisation algorithm. In RMSprop, each weight is updated with a different learning rate based on the magnitude of the gradients. Instead of using only the current gradient for a weight, we take an exponential moving average of the squared gradients from the previous time steps [5].

$$S_{dW} \leftarrow \beta S_{dW} + (1-\beta) dW^2$$

When the weights are updated, they are scaled by the square root of the moving average.

$$W \leftarrow W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$

Thus, the learning rate is decreased for large gradients in order to avoid exploding.

Figure 7 shows the growth of $W2$ for both Gradient Descent and RMSprop over 50,000 episodes. The L2 norm is computed in order to determine the length of the weight matrix $W2$. Even though the weights increase in both cases, they increase at a much lower rate when RMSprop is used.

VIII. Conclusion

This work implemented 2 different Reinforcement Learning algorithms, namely Q-Learning and SARSA, for learning how to perform a checkmate in a simplified game of Chess. Even though the agent learns faster at the beginning when it uses Q-Learning, SARSA performs better over a large number of episodes. Experiments show that a large value for the discount factor γ leads to a higher reward per game as the agent looks further into the future for a high reward. When using the $\epsilon - greedy$ policy, it is important that ϵ does not decay too fast or too slow as the agent must have a good balance between exploration and exploitation. RMSprop is a better alternative to Gradient Descent as it adjusts the learning rate independently for each parameter. For large gradients, the step size is very small, stopping the gradients from exploding.

References

[1] Hertz, J., Krogh, A. and Palmer, R. (1991). Introduction to the theory of neural computation. Redwood City: Addison-Wesley.

[2] Skymind. (2019). A Beginner's Guide to Deep Reinforcement Learning. [online] Available at: https://skymind.ai/wiki/deep-reinforcement-learning [Accessed 8 May 2019].

[3] Sutton, R. and Barto, A. (1998). Reinforcement learning. 2nd ed. MIT Press.

[4] Learning, W. and Slater, N. (2019). When to choose SARSA vs. Q Learning. [online] Cross Validated. Available at: https://stats.stackexchange.com/questions/326788/when-to-choose-sarsa-vs-q-learning/326802#326802 [Accessed 8 May 2019].

[5] Coursera. (2019). RMSprop - Optimization algorithms — Coursera. [online] Available at: https://www.coursera.org/lecture/deep-neural-network/rmsprop-BhJlm [Accessed 11 May 2019].

APPENDIX A

The results can be reproduced by following the instructions from the file *README.md*. The number of moves per episode and reward per episode for the experiments that were conducted are saved in the corresponding *pickle* files and the plots can be seen by running the Jupyter notebook *Plots.ipynb*.

APPENDIX B

```
W2_delta = (R + gamma * next_Q_value - Q[a_agent]) * np.heaviside(Q, 0)
W1_delta = np.heaviside(out1, 0) * np.dot(W2.T, W2_delta)

W2 += eta * np.outer(W2_delta, out1)
bias_W2 += eta * W2_delta

W1 += eta * np.outer(W1_delta, x)
bias_W1 += eta * W1_delta
```

Fig. 8. Q-Learning/SARSA with backpropagation.

```
if sarsa:
    # if SARSA, choose next action based on policy
    next_Q_value = Q_next[epsilon_greedy(epsilon_f, Q_next, allowed_a)]
else:
    # if Q-learning choose action with maximum Q value
    next_Q_value = max(Q_next[allowed_a])
```

Fig. 9. Next action for SARSA and Q-Learning.

```
def epsilon_greedy(epsilon, Q, allowed_a):
    """
    Epsilon greedy policy
    """
    if np.random.rand() < epsilon:
        # sample random action from a_allowed
        a_agent = np.random.choice(allowed_a)
    else:
        # select action with maximum Q value
        a_agent = allowed_a[0]
        for action in allowed_a:
            if Q[action] > Q[a_agent]:
                a_agent = action

    return a_agent
```

Fig. 10. The $\epsilon - greedy$ policy.