Reinforcement Learning: Assignment '22

Jan Willi
17-923-053
jan.willi@uzh.ch

Raffael Mogicato
18-742-767
raffael.mogicato@uzh.ch

I. Introduction

The goal of this assignment was to create a deep reinforcement learning model that is able to tackle the task of giving checkmate in a simplified chess game. The environment of this problem consists of a 4x4 chessboard, with three pieces on it: A black king, a white king and a white queen. The pieces can move according to the normal chess rules, i.e. in all eight directions, the kings however only one step at a time. The black king is controlled by the computer and behaves randomly, which means it moves to a random, legal square during each turn. The white pieces try to give checkmate to the black king: This is achieved when the king is in check and has no legal moves left. If the black king is unable to move without being in check, the game results in a draw.

To build model that can reliably checkmate the opponent, we made a few design decisions: Firstly, we must decide on a policy to decide what action we take in a state. A second decision is that of our control algorithm. In our project we used two different algorithms: One on-policy and one off-policy control algorithm, namely SARSA and Q-learning.

II. Methods

This first chapter covers the theoretical aspects of our two chosen control algorithms, SARSA and Q-learning, as well as extensions and improvements to them.

A. Control Algorithms

Control algorithms are central to any reinforcement learning process. They learn the value of the optimal state and action pairs by transitioning through states by applying an action policy [1]. In our model we choose an ϵ-greedy policy which determines action a_t at time step t based on the Q-values Q_a of that action.

$$policy(a_{allowed}, Q) = \begin{cases} \max_a(Q) & \text{with probability } 1-\epsilon \\ \text{any } a_{allowed} & \text{with probability } \epsilon \end{cases}$$

This epsilon value describes the probability of choosing a random, non-optimal value to allow for exploration. Using this policy we can now specify our on-policy control algorithm SARSA [2] which is described in Algorithm 1. The term on-policy refers to the fact that our Q-values are updated based on the action that we choose through our ϵ-greedy policy. The second control algorithm is called Q-learning described in Algorithm 2. The most significant difference is the fact that Q-learning updates Q-values based on the highest value, regardless of whether we actually take that action or not. It is therefore an off-policy algorithm. Both SARSA and Q-learning fall under the category of online learning, as the updated Q-value estimates are directly available in the next state [3].

As we employ a two-layer neural network to predict the future Q-values, we either adapt our weights based on the action we take (SARSA) or based on the best action available (Q-learning). Note that if an episode is over, either through stalemate or check mate, the network is updated on only the reward, as no next state is available. A depiction of this deep algorithm variation can be found in Algorithm 3 for Q-learning, for the moment ignoring the blue parts. Deep SARSA follows a similar scheme with a slightly adjusted loss computation.

Algorithm 1 SARSA control algorithm

```
Initialize: $Q(s,a), \forall s \in S, a \in A(s)$, arbitrarily
        and $Q(\text{terminal-state}, .) = 0$
for each episode do
    Initialize $S, a_{allowed}$
    Choose $a$ using $policy(a_{allowed}, Q)$
    for each step of the episode do
        Take action $a$, observe $R$, $S'$, $a'_{allowed}$
        Choose $a'$ from $S'$ using $policy(a'_{allowed}, Q)$
        $Q(S,a) \leftarrow Q(S,a) + \alpha[R + \gamma Q(S',a') - Q(S,a)]$
        $S \leftarrow S', a \leftarrow a', a_{allowed} \leftarrow a'_{allowed}$
    end for
end for
```

Algorithm 2 Q-learning control algorithm

```
Initialize: $Q(s,a), \forall s \in S, a \in A(s)$, arbitrarily
        and $Q(\text{terminal-state}, .) = 0$
for each episode do
    Initialize $S, a_{allowed}$
    for each step of the episode do
        Choose $a$ using $policy(a_{allowed}, Q)$
        Take action $a$, observe $R$, $S'$, $a'_{allowed}$
        $Q(S,a) \leftarrow Q(S,a)$
                $+ \alpha[R + \gamma \max_a Q(S',a') - Q(S,a)]$
        $S \leftarrow S', a_{allowed} \leftarrow a'_{allowed}$
    end for
end for
```

B. Experience Replay

In literature, the idea of combining classical reinforcement algorithms such as Q-learning with neural networks is widespread. One critical component of such deep Q-learning (DQN) algorithms is experience replay [4]. In this approach, an experience replay buffer is maintained, storing the most recent transitions $(S, a, R, S', Done)$. Instead of online learning after each transition, Q-learning is applied after a predefined number of steps to a random sample of the buffer and the neural network is trained in a mini-batch fashion [5]. Besides the basic uniform sampling, advanced strategies such as prioritized [6] or distributed experience replay [7] have been proposed [8]. The general DQN algorithm with the experience replay addition in blue is described in Algorithm 3.

Algorithm 3 Deep Q-learning with Experience Replay

```
Initialize Q-network $Q$ with random weights
Initialize target network $Q_{target}$ with weights from $Q$
Initialize replay memory $D$ with capacity $N$
for each episode do
    Initialize $S, a_{allowed}$
    for each step of the episode do
        Choose $a$ using $policy(a_{allowed}, Q)$
        Take action $a$, observe $R$, $S', Done$
        Store $(S, a, R, S', Done)$ in $D$
        for each mini-batch sample of $D$ do
            Set $y_a \leftarrow \begin{cases} R & \text{if } Done \\ R + \gamma \max'_a Q_{target}(S', a') & \text{otherwise} \end{cases}$
            Perform backpropagation on $y_a - Q(S, a)$
        end for
    end for
    Regularly update $Q_{target}$ with weights from $Q$
end for
```

Mnih et al. [5] identify several advantages of utilizing experience replay over online learning. Most importantly, online learning is inefficient due to strong correlations between consecutive transitions. Sampling from the buffer breaks these correlations and avoids fluctuations by averaging over multiple previous states, which improves the stability of the network during training [5, 8]. Additionally, as transitions are kept in the buffer, the algorithm can learn from each transition multiple times, increasing the data efficiency, which is especially important in applications with scarce data at hand. Another modification, which directly aids experience replay is the use of a secondary network. This target network is regularly updated from the main network and used to generate Q-learning targets. This again combats oscillations and improves stability [9].

With the added complexity of experience replay, new hyperparameters need careful considerations. Besides the mini-batch size, these include the size of the replay buffer, the age of the oldest transition in the buffer, as well as the ratio at which a learning step is done with respect to transition steps [8]. Generally, performance increases when increasing replay capacity or reducing replay age, these two factors are however directly contradicting each other, making it impossible to find a general optimal setting [8].

C. Exploding Gradients

A general problem that is encountered in gradient descent optimization is that of exploding gradients. Exploding gradients occur when several large weights are multiplied together, leading to a huge gradient. This leads to a (too) large step, especially when we have a fixed learning rate [10]. Exploding gradients generally lead to unstable and low-performing networks. There are several approaches to this issue, such as gradient clipping or Root Mean Squared Propagation (RMSProp). Our choice to address this problem fell on RMSprop, as Goodfellow et al. [10] call this an an effective and practical optimization algorithm. RMSprop uses a decaying average of past and current squared gradients $\mathbf{g}$, where the weighting of past and current gradients is determined by an additional hyperparameter ρ. This average $\mathbf{r}$ is used to determine the adjusted step size based on the global step size η. With that we can calculate the new weights at time step t as

$$\mathbf{r}_t = \rho \cdot \mathbf{r}_{t-1} + (1 - \rho) \cdot \mathbf{g}^2$$
$$\mathbf{W}_t = \mathbf{W}_{t-1} + \mathbf{W}_{t-1} \cdot \frac{\eta}{\sqrt{\epsilon + \mathbf{r}}}.$$

As $r_0 = \mathbf{0}$, we add $\epsilon = 10^{-6}$ to avoid division by 0. We chose 0.9 as a value for ρ to avoid exploding gradients by giving past gradients a higher weight in the average. At the same time values of the extreme past decay with each time step, allowing for fast convergence.[10]

III. Results

The discussed algorithms and technique were implemented in Python. We then experimented with various combinations of control algorithms, hyperparameters and rewards. To give a better overview of our results, we structure this section into several subsections, each focusing on the comparison of only a few aspects. Each combination is run five times using five predefined seeds to account for the randomness of our models, such as the initialization of the weights. This should allow for a fair comparison and additionally shows the variance of the model. To reduce noise, all visualizations employ an exponential moving average ($\alpha = 0.001$).

A. Q-learning vs SARSA

Firstly, we compare the two control algorithms. As hyperparameters we use the following values: The epsilon-greedy policy uses an initial $\epsilon_0 = 0.2$ which decays with a rate of $\beta = 5 \cdot 10^{-5}$ per episode. The learning rate is fixed as $\eta = 0.0035$ and future rewards are discounted through multiplication with $\gamma = 0.85$. Each run consists of 10'000 episodes. Both approaches use vanilla gradient descent as a backpropagation algorithm. The relevant code snippets are found in Listing 1 and 2 in Appendix B. The results are depicted in Figure 1: We can observe that both control algorithms are similar in overall performance. SARSA initially performs worse in earlier episodes, taking more moves and achieving a lower reward than the Q-learning algorithm. However, after a few thousand episodes, SARSA overtakes Q-learning performance wise, achieving both a higher award and taking fewer moves on average after 10'000 episodes. A further observation is that the performance between the best and worst run is rather large, especially for the Q-learning algorithm, indicating a possible issue with gradients. Generally, the performance is far from optimal, in the next steps we show what changes we made to improve the performance.

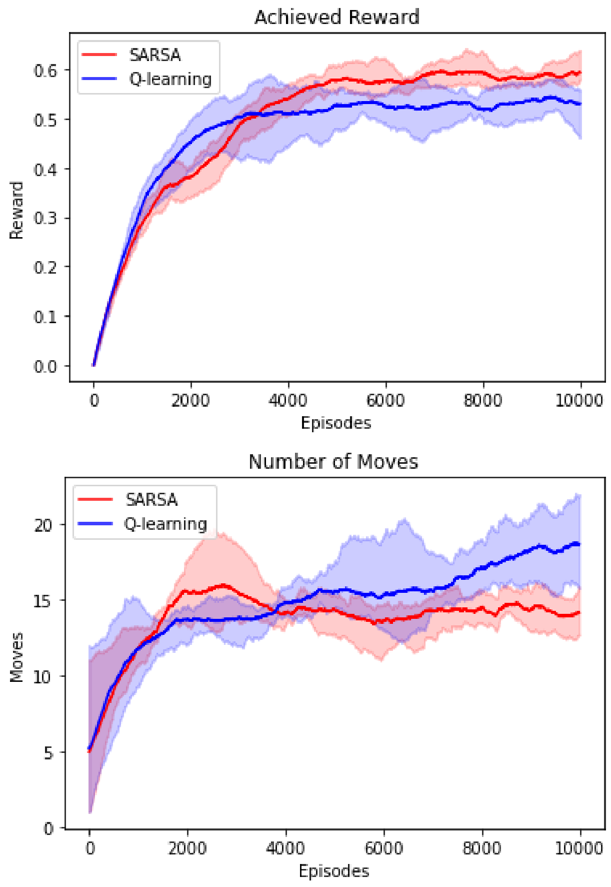

Fig. 1: Comparing the reward and moves of Q-learning and SARSA: The two lines show the average of 5 runs, the filled out space depicts the difference between the best and worst performing run.

B. Q-learning with Experience Replay

Another interesting comparison is the one of online Q-learning with its experience replay version. Again, the same hyperparameters as in the comparison to SARSA above were used: $\epsilon_0 = 0.2$, $\beta = 5 \cdot 10^{-5}$, $\eta = 0.0035$ and $\gamma = 0.85$. For the experience replay algorithm, we further use a batch size and learning ratio of 24 and update the target model every 100 steps. The code of the experience replay implementation is found in Listing 3 and 4 in Appendix B. Figure 2 shows the outcomes of online Q-learning and Q-learning with an experience replay buffer of length 100 and of length 10'000. Concerning the rewards, experience replay negatively impacts the learning speed, as the model learns from older, potentially less optimal transitions. After some time however, the experience replay means catch up and ultimately surpass the normal model. Another observation is that normal Q-learning is less susceptible to differences between runs, as the shaded area is narrower. With experience replay, some runs are able to achieve a much higher reward of about 0.7. This could be attributed to the fact that sampling the replay buffer introduces another aspect of randomness. Looking at the moves, a similarly great variance between runs is observable. Somewhat surprising is that a buffer length of 100 leads to more moves than a 10'000 elements long buffer, with the latter converging towards the normal model. The former however shows a tendency to decrease again in the final episodes, a longer training time could therefore improve on this aspect.

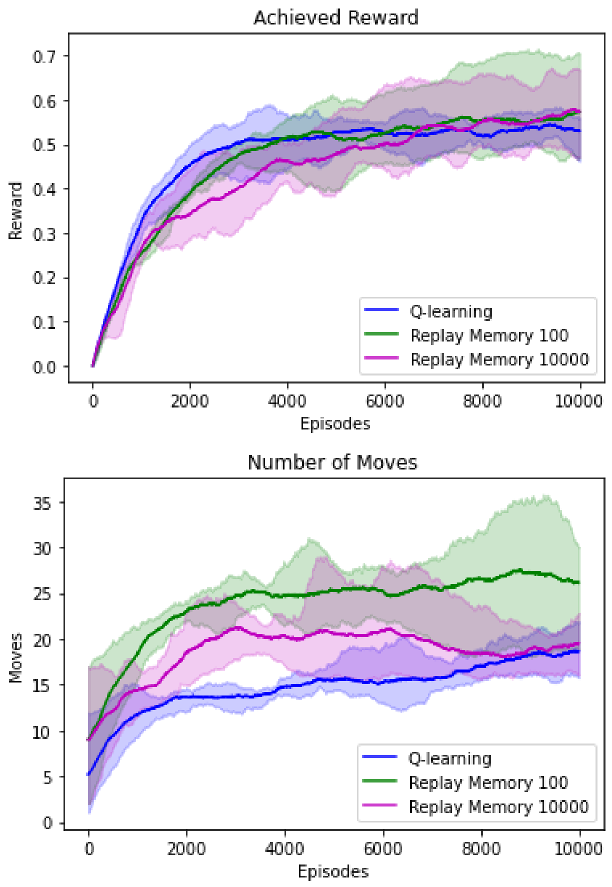

Fig. 2: Comparison of standard Q-learning to experience replay with different buffer sizes. The lines show the average of 5 runs, the filled out space depicts the difference between the best and worst performing run.

C. Q-learning Hyperparameter Variation

To analyze the effects of the hyperparameters on the model, we conducted experiments with varying the discount factor γ and the decay speed β of ϵ.

1) Discount Factor γ: The discount factor determines the impact future rewards have on the current value [2]. A factor close to 0 minimizes the significance of future rewards, whereas with a factor close to 1, the agent is willing to wait for long-term rewards [11]. These aspects can be compared to our results in Figure 3, where the default discount factor of 0.85 is compared to a factor of 0.1 and 0.99, respectively. With the high gamma, the agent is more farsighted and therefore accepts more moves to achieve checkmate. On the contrary, a myoptic agent with a low discount factor is interested in more immediate rewards and therefore takes fewer moves. While the moves greatly differ, the achieved rewards after the full 10'000 training iterations are fairly equivalent, though too high or low gammas seem to negatively impact the learning speed.

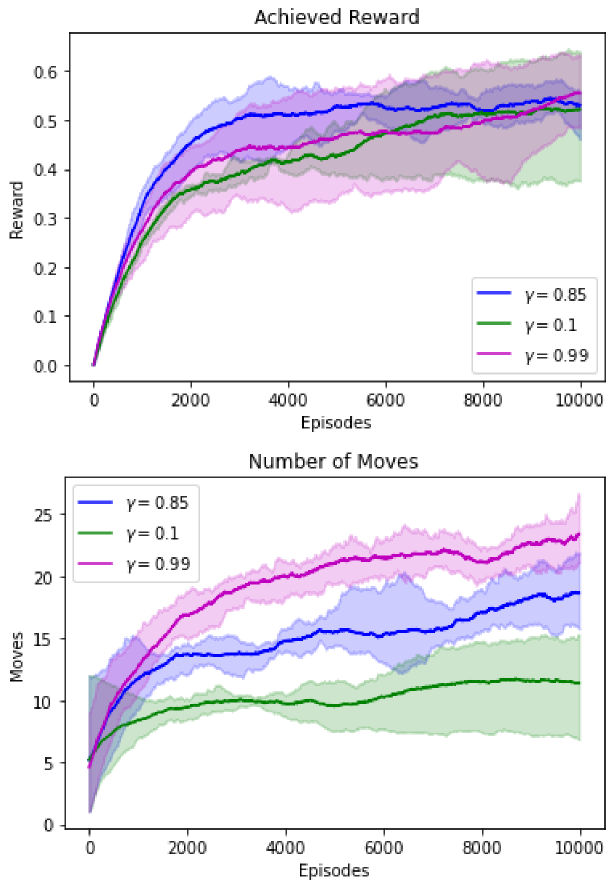

Fig. 3: Comparison of different discount factors in Q-learning. The lines show the average of 5 runs, the filled out space depicts the difference between the runs.

2) Epsilon Decay β: Depending on the outcome of the ϵ-greedy policy, the agent either exploits or explores the environment. In exploitation, actions are chosen greedily to maximize the reward, whereas in exploration, not necessarily the best actions are followed [12]. Usually, exploration is most important in the beginning of the learning process, when the agent has not much knowledge about the environment, but can then be decayed to boost the exploitation of the accumulated knowledge. In our case, this decay follows the formula

$$\epsilon = \frac{\epsilon_0}{1 + \beta n},$$

where n is the current training episode. The effects, which various β have on the model, are observable in Figure 4. Most noticeably, a too large decay leads to a very large number of moves. This could be attributed to the fact that with less exploration, less knowledge about the opponent is collected, but rather the agent chases the enemy king in circles until existing knowledge allows the finishing move. Increasing the decay even more would quickly reduce ϵ close to 0 and the games would start to become infinitely long and only finish by chance. Various decays seem to not impact the average achievable reward greatly, although the data indicates that a high decay results in greater fluctuations between runs. This is reasonable, as with less exploration, the random start greatly determines the success at later stages. Different runs therefore yield different final rewards.

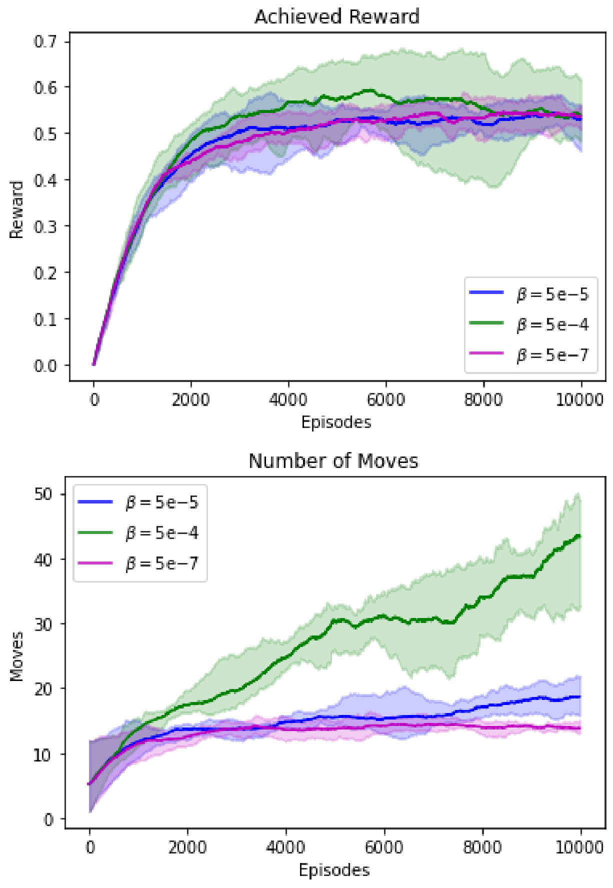

Fig. 4: Comparison of different decay speeds of ϵ in Q-learning. The colored area shows the differences in the 5 runs, with the line indicating the mean.

D. Vanishing and Exploding Gradients: RMSprop vs. Vanilla Gradient Descent

To address the issue of vanishing and exploding gradients we implemented RMSprop and keep track of gradients when using vanilla gradient descent with Q-learning. Both runs use the same hyperparameters and seed. To get a good visualization of the gradient, we take the average of the L2 norm of the gradient during each episode for each of the four gradients we have in our neural network. As vanilla gradient descent uses a fixed step size and RMSprop scales the gradients by an exponential average, we calculate the L2 Norm after scaling it with the two aforementioned parameters.

In Figure 5 we can see a direct comparison of the average L2 norms. The most notable difference is that the gradients from vanilla gradient descent change to a lesser degree than those of RMSprop, where all gradients decrease in size over the number of episodes. Furthermore, all the gradients are larger in RMSprop, both in absolute value and when compared to the largest of the four gradients.

Goodfellow et al. writes that: ”Many practitioners attribute nearly all difficulty with neural network optimization to local minima. We encourage practitioners to carefully test for specific problems. A test that can rule out local minima as the problem is plotting the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is neither local minima nor any other kind of critical point.” [10, p. 282] . The number of near-zero gradients indicates that vanilla Q-learning is hindered by vanishing gradients, unlike RMSprop where sufficiently large gradients exist to efficiently train the network.

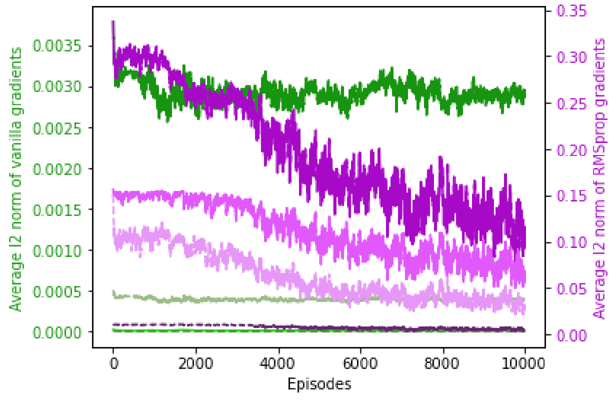

Fig. 5: Comparison of the euclidean norm of the gradients from two backpropagation approaches: The purple lines are the gradients of RMSprop and the green lines those of vanilla gradient descent.

To confirm that the gradients are indeed a problem with vanilla Q-learning, we measured the performance of the two gradient descent techniques using the same five run approach with the same hyperparameters as above. As can be seen in Figure 6, the RMSprop greatly improves the performance. For the reward of RMSprop, the initialisation influences the results, as visible in the upper bound. However, even the lowest performing run outperforms vanilla Q-learning by a considerable amount. A further noteworthy observation is that RMSprop keeps lowering the number of moves while the number of moves continually grows in vanilla Q-learning. This indicates that RMSprop continues to improve its performance even after 10’000 moves. An example of this can be found in Figure 7, where the RMSprop agent was trained twice as long. Eventually, just 3-4 moves on average are required to finish a game. Note that the reward stagnates at just over 0.8 per episode after approximately 10’000 episodes.

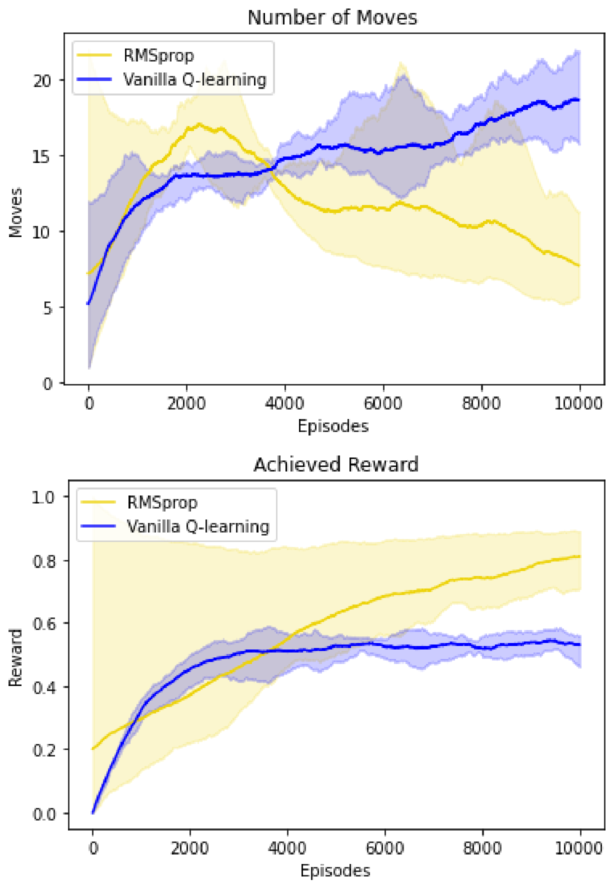

Fig. 6: Comparison of different vanilla Q-learning and RMSprop Q-learning. The translucent area depicts the difference between the highest and lowest performing run.

E. Changes in Reward Administration

As a final experiment we look at how the administration of the reward effects the performance. In all previous settings, a checkmate received a reward of 1 and a draw a reward of 0. One possibility is to introduce a punishment for unwanted behavior, instead of only rewarding wanted behavior. This is why we now introduce two further versions for administering the reward: In one version, we punish draws with a reward of -1, while still rewarding check mates with a reward of 1. In the second version, we take this concept even further and punish draws with a reward of -100, while again keeping

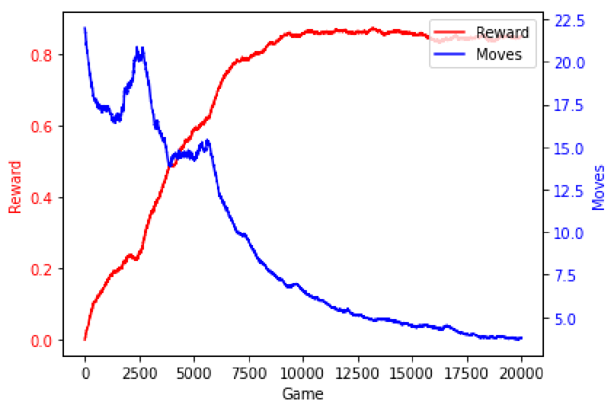

Fig. 7: Performance of a RMSprop run with 20'000 episodes. The red line indicates the achieved award, labeled on the left y-axis and the blue line the number of moves used, labeled on the right.

the reward of 1 for checkmates. In Figure 8 we scaled the resulting reward data using min-max normalization to allow for better visual comparison. The original version without explicit punishment still achieves the highest reward on average, however, it is closely followed by a punishment of -1 for draws. When looking at the number of moves, this version even slightly outperforms the original. In all cases, the version with a reward of -100 for draws performs the worst, though it also has the lowest difference between best and worst run. We can further conclude that a punishment for draws keeps the number of moves low during earlier episodes, resulting in a slightly improved run time.

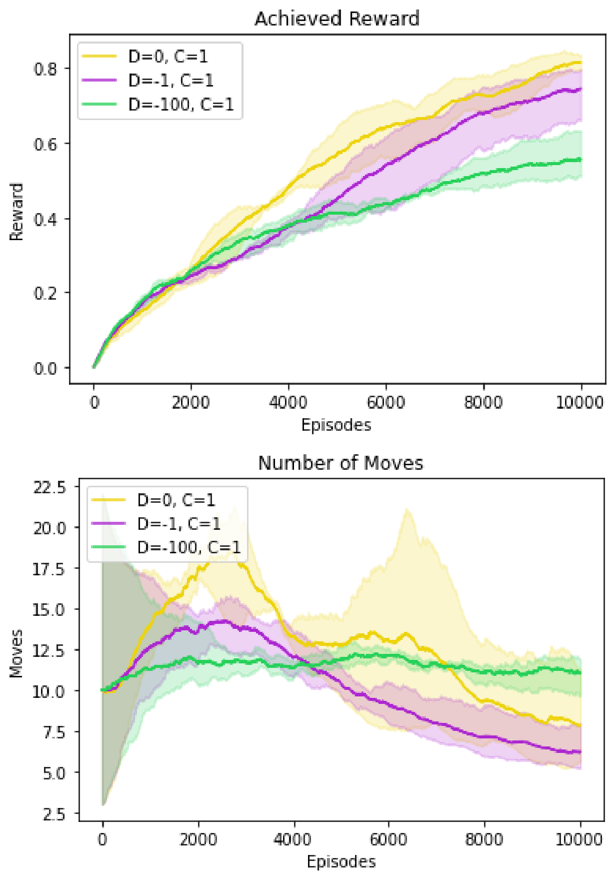

Fig. 8: Comparison of different types of reward administrations. D and C describe the reward in case of a draw and checkmate, respectively. The area with low opacity shows the difference between best and worst performing run. The values of the reward have been feature scaled for better comparison.

IV. Conclusion

As part of this assignment, different deep reinforcement learning algorithms and variations thereof were implemented and tested. Analyzing the achieved average reward, only small differences could be observed between SARSA, vanilla Q-learning and experience replay Q-learning, but a major reward gain was observable when using RMSprop as an optimization algorithm rather than vanilla gradient descent. On the topic of number of moves, RMSprop was again most optimal, especially when training for more episodes. Here, variation of the discount factor and the epsilon decay showed the importance of selecting the right hyperparameter, as the number of moves quickly rose when non-optimal values were chosen. Similar issues were encountered in the analysis of experience replay Q-learning with a main and target network, as the algorithm introduces additional hyperparameters that one can play with. Due to them being problem dependent and the fact that their impacts are not always predictable, additional effort would be needed to fully master the model and bring out its theoretical strengths. Furthermore, we observed that the award administration directly influences the performance depending on whether we want to optimize the number of moves or the achieved reward: By introducing a punishment for draws the algorithms naturally achieves a lower reward on average, but at the same time converges faster on and varies less regarding to the number of moves.

Overall the best performance was achieved with RMSprop as a optimization algorithm, achieving both a fairly high reward and a low number of moves. However, there most likely is room for improvement, as there is a large number of possible hyperparameter combinations that may improve the performance. A further possibility is to combine experience replay with RMSprop, which may further improve the performance of our model. Our approach still showed that Q-learning with a proper optimization algorithm achieves a considerable performance even with a relatively simple and straight forward approach to reinforcement learning.

References

[1] M. Corazza and A. Sangalli, "Q-learning and SARSA: A comparison between two intelligent stochastic control approaches for financial trading," *SSRN Electronic Journal*, 2015.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[3] R. S. Sutton and S. D. Whitehead, "Online learning with random representations," in *Proceedings of the Tenth International Conference on International Conference on Machine Learning*, ser. ICML'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, p. 314–321.

[4] L.-J. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, USA, 1992.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 11 2015.

[7] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver, "Distributed prioritized experience replay," 03 2018.

[8] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting fundamentals of experience replay," in *ICML*, 2020.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[11] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 4th ed. Pearson, 2020.

[12] A. Maroti, "Rbed: Reward based epsilon decay," *ArXiv*, vol. abs/1910.13701, 2019.

Appendix A
Run the code

The code of the assignment can be found on Github[1]. The *Assignment* notebook contains the implementation of Q-learning and SARSA, RMSprop can be activated via a boolean flag. Experience replay is implemented in its own notebook called *experience_replay*. For reproducibility, the 5 runs were each done using numpy random seeds 41 to 45. Both notebooks contain code store the results and to generate a plot to quickly visualize the training run. Visualizations found in the report were created according to the example in the *Visualization* notebook, which reads numpy arrays from binary `.npy` files and displays the data as shaded areas with mean lines.

[1]https://github.com/rmogicato/RL-Jan-Raffael

Appendix B
Code Snippets

Listing 1: Main training loop, where the blue and red lines correspond to Q-learning and SARSA, respectively.

```
for n in (range(N_episodes):

   epsilon_f = epsilon_0 / (1 + beta * n)
   Done, i, R = 0, 1, 0
   _, X, allowed_a = env.Initialise_game()

   while not Done:
      x1, q = predict(X, model)
      a = policy(allowed_a, q, epsilon_f)
      _, Xn, allowed_an, R, Done = env.OneStep(a)

      q_err = np.zeros(N_a)
      if Done:
         R_save[n] = R
         N_moves_save[n] = i
         q_err[a] = R - q[a]
         model = backprop(R, q, q_err, a, X, x1,
             ↪ model)
         break

      _, qn = predict(Xn, W1, W2, bias_W1, bias_W2)

      q_err[a] = R + gamma * np.max(qn) - q[a]

      an = policy(allowed_an, qn, epsilon_f)
      q_err[a] = R + gamma * qn[an] - q[a]

      model = backprop(R, q, q_err, a, X, x1, model)

      X, allowed_a = Xn, allowed_an
      a = an
      i += 1
```

Listing 2: Neural network Q-value prediction and error back-propagation. The green parts indicate the RMSprop variant.

```
def predict(x0, model):
   h1 = np.dot(model.W1, x0) + model.bias_W1
   x1 = 1 / (1 + np.exp(-h1))
   h2 = np.dot(model.W2, x1) + model.bias_W2
   x2 = 1 / (1 + np.exp(-h2))
   return x1, x2

def backprop(R, q, q_err, a, x0, x1, model):
   delta2 = q*(1-q) * q_err
   dW2 = np.outer(delta2, x1)
   delta1 = x1*(1-x1) * np.dot(model.W2.T, delta2)
   dW1 = np.outer(delta1, x0)

   rms_beta = 0.9
   # weighted average of previous squared gradient
       ↪ and current squared gradient
   dW1_squared = rms_beta * squared_gradients[0] +
       ↪ (1-rms_beta) * np.square(dW1)
   dW2_squared = rms_beta * squared_gradients[1] +
       ↪ (1-rms_beta) * np.square(dW2)
   alpha1 = eta / (1e-6 + np.sqrt(dW1_squared))
   alpha2 = eta / (1e-6 + np.sqrt(dW2_squared))

   db1_squared = rms_beta * squared_gradients[2] +
       ↪ (1-rms_beta) * np.square(delta1)
   db2_squared = rms_beta * squared_gradients[3] +
       ↪ (1-rms_beta) * np.square(delta2)
```

```
    alpha1b = eta / (1e-6 + np.sqrt(db1_squared))
    alpha2b = eta / (1e-6 + np.sqrt(db2_squared))


    model.W1 += eta * dW1
    model.W2 += eta * dW2
    model.bias_W1 += eta * delta1
    model.bias_W2 += eta * delta2

    new_squared_gradients = np.array([dW1_squared,
        ↪ dW2_squared, db1_squared, db2_squared],
        ↪ dtype=object)

    return model, new_squared_gradients
```

Listing 3: Main training loop of DQN with experience replay.

```
replay_memory = deque(maxlen=memory_length)
for n in range(N_episodes):

    epsilon_f = epsilon_0 / (1 + beta * n)
    Done, i = 0, 1
    _, X, allowed_a = env.Initialise_game()

    while not Done:
        steps_to_update += 1
        x1, q = predict(X, main_model)
        a = policy(allowed_a, q, epsilon_f)
        _, Xn, allowed_an, R, Done = env.OneStep(a)
        replay_memory.append([X, a, R, Xn, Done])

        if steps_to_update % train_rate == 0 or Done:
            main_model = train(batch_size,
                ↪ replay_memory, Done, main_model,
                ↪ target_model)

        X, allowed_a = Xn, allowed_an
        i += 1

    R_save[n] = R
    N_moves_save[n] = i

    # Update target model with main model
    if Done and steps_to_update >=
        ↪ target_model_update_rate:
        target_model.set_weights(main_model.
            ↪ get_weights())
        steps_to_update = 0
```

Listing 4: Train logic of DQN with experience replay.

```
def train(batch_size, replay_memory, Done,
    ↪ main_model, target_model):
    if len(replay_memory) < batch_size:
        return main_model

    mini_batch = random.sample(replay_memory,
        ↪ batch_size)
    current_states, intermediate_states = [], []

    q_errs = []
    for X, a, R, Xn, Done in (mini_batch):
        x1, q = predict(X, main_model)
        current_states.append(X)
        intermediate_states.append(x1)
        if Done:
            target = R
        else:
            _, qn = predict(Xn, target_model)
            target = R + gamma * np.max(qn)

        old_q = q.copy()
        q[a] = target
        q_errs.append(q - old_q)

    return backprop_batch(batch_size, current_states,
        ↪ intermediate_states, q_errs, main_model)
```

Appendix C
Meeting Logs and Contributions

In the first phase of the project, we explored the topic each on our own and both tried to implement a basic first version. For this purpose, we both had our own Github branch. After that, we shared our insights to get a working implementation that made sense to us. This then allowed us to build on top of it, where Jan focused on experience replay and Raffael got SARSA to work and tackled the exploding gradient problem with RMSprop. During implementation, we tried to keep the report up to date by continuously writing down the topic in focus. At this point, we continued on Jan's branch to ensure that we run the exact same code. As it was hard to split the results and analysis of the project, we completed this part in an iterative fashion, where we both alternatingly continued to generate data and analyze it. In the end, we merged our branches into `main` and finished up the report with a conclusion.

We met each week to discuss progress and plan the next week. All other communication was done via WhatsApp.

Date	Content
22.02	Discussed: Objective and some structural decisions Target: Individually try to implement Q-learning
01.03	Discussed: First try of the Q-learning algorithm, the problems that we encountered (mostly regarding backpropagation), possible solutions Target: Individually finish Q-learning implementation
08.03	Discussed: Comparison of our branches to see how our implementations differ, how to move forward with open tasks Target: merge our implementation, then focus on experience replay (Jan) and RMSprop / SARSA (Raffael)
5.03	Discussed: Implementation results, the structure of the result section of the report Target: write analysis as far as possible by keeping each other up to date about the status and just continue where the other paused
23.03	Discussed: The results we got so far and how they can be interpreted. Target: finish analysis of results and write conclusion, proofread report and clean up / comment code as far as possible
28.03	Discussed: Wrapping up / finishing conclusion Target: Submit project