Introduction to Reinforcement Learning FS22 Chess Experiment Final Report

Euxane Vaz Pinto
18-211-391

Rupal Saxena
21-739-289

Alessandro Vanzo
20-751-558

I. Introduction

We train two deep reinforcement learning models to play a simplified version of chess, using Sarsa and Q-learning, respectively. We compare the results from each model, and look at how their performance changes when we fine-tune the parameters and experiment with variations with state representation and administration of reward.
First, we discuss Sarsa, Q-learning and some related topics that are relevant for our experiment and the course. We describe our experimental setup and the chess environment we use. Then, we describe our implementations of an artificial neural network, Sarsa and Q-learning, and outline our strategies for experimentation and fine-tuning of the parameters. Finally, we discuss the results.

II. Sarsa and Q-learning

Sarsa and Q-learning are two popular algorithms for reinforcement learning. Both can be used with tabular or deep learning. Before describing the algorithms, it is helpful to define some concepts.

A. Policy

In reinforcement learning, when we talk about policy, we refer to the strategy adopted in selecting an action at a given state. Given the expected return $Q(S_i, a_i)$ for each state of the world and each possibly action, there are several approaches to choose a strategy. Here we list some:

1) **Greedy**: the agent will choose the action maximizing the expected return.
2) ε-**greedy**: the agent will choose according to the greedy policy with probability $(1 - \varepsilon)$, and randomly with probability ε
3) **Softmax**: the agent will choose an action based on a softmax density on the expected returns.

B. Q-value

In most problems, an agent does not know the expected return of actions in its environment, as it cannot observe the underlying model. For this reason, the expected return of an action at a given state will be estimated based on observed rewards at each episode. We call the *estimate* of the expected return for a given state-action pair *Q-value*. The Q-value $Q(S, a)$ is therefore a function of a state-action pair. Estimating accurate Q-values is the key aspect of learning. Neural Networks are universal approximators and can be used to estimate the Q-values of complex models, which is what we will do in this project.

C. Sarsa

Sarsa is an on-policy learning algorithm, meaning that the agent will learn from the actions it takes at each step. The weights are initialized randomly for each state, except for the terminal states where they are set to zero. After initializing the starting state S, the agent will choose an action a_i based on the policy selected. After taking the action, the agent will go to a state S', it will observe the Q-value and it will choose an action a'. The Q-value for state S and action a is then updated as follows:

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

R is the (optional) reward obtained from playing action a at state S, and it is defined by the environment. α is the learning rate and γ is the discount rate on future rewards; these two parameters can be chosen arbitrarily. This process is repeated until the agent reaches a terminal state.

D. Q-learning

Q-learning is an off-policy learning algorithm first developed by Watkins (1989). The learning process is similar to that of Sarsa. The key difference is that the agent will not learn from the action it plays, but rather from the optimal action it could have played. This can favor a higher level of exploration depending on the chosen policy. The weights Q-values are initialized randomly for each state-action pair, and the agent starts at the starting state S. The agent picks an action a based on policy and moves to the state S'. The agent observes the state S' and updates the Q-value for state S based on the Q-value of the optimal action a' at state S'. The update rule is:

$$Q(S, a) \leftarrow Q(S, a) + \alpha[R + \gamma max_{a^*} Q(S', a^*) - Q(S, A)] \quad (2)$$

The variables R, α and γ are defined as in the previous equation. a^* is the optimal action at state S'. After updating the Q-value, the agent chooses an action a' based on the policy. Note that a' is not necessarily equal to a^*. The process is then repeated until the agent reaches a terminal state.

code: https://github.com/rupalsaxena/Chess-rl

E. Comparison of Sarsa and Q-learning

The main difference between Sarsa and Q-learning is that the former is an on-policy learning algorithm, while the latter is an off-policy learning algorithm. This means that Sarsa always learns by the action that the agent takes, whereas Q-learning will learn by the actions the agent took and by those *it could have taken*, and in particular from the optimal one. Both algorithms perform well in a variety of problems, and we could find no consensus in the literature on which one should be used when. One famous example of the different results we can get from Sarsa and Q-learning is the cliff problem, described by Sutton and Barto (2018) [2]. Sarsa develops a more "conservative" approach, leading the agent on a longer route that minimizes the likelihood of falling off the cliff. Q-learning, on the other hand, learns the most efficient route, but its agent will fall off the cliff more often.

F. Experience replay

In the previous sections, we looked at the overview of off-policy and on-policy reinforcement learning algorithms. The algorithms described above can be inefficient, since some experiences may be rare and some costly to obtain. Experiences should be reused in an effective way. In order to reuse the experiences, the experience replay technique is used [4].

Experience replay is used to train off-policy algorithms in deep reinforcement learning. With Experience replay, we first store the agents experiences at each time step in a data called replay memory. At time t, the agent's experience e_t can be defined as:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}) \tag{3}$$

where, s_t is state of environment at time t, a_t is action taken from the state at time t, r_{t+1} is reward received from previous state and action pair, and s_{t+1} is the next state of the environment.

All the agents experiences at each time steps for all episodes are stored in replay memory. We then sample the data from replay memory to train the deep network.

In practice, however, experience replay is implemented as a circular buffer, where the oldest experiences is removed to make room for a new experience to be collected. Experiences are then sampled from the buffer at fixed intervals for use in training [5].

There are several ways to sample from replay memory and replay buffer. A simple technique is uniform sampling, where each experience from the buffer is chosen with equal probability. Another commonly used technique is prioritized experience replay [6]. The idea behind prioritized experience replay is that the agent can learn more effectively from some experiences than others. Some experiences can be redundant or task-relevant. To address this, we assign priorities to each experience in the replay buffer and call it prioritized replay buffer. A variation of prioritized experience replay is distributed prioritized experience replay [7].

Should we always use Experience replay while training a reinforcement learning model? If the environment is changing rapidly, then past experiences may become irrelevant or even harmful [4]. In such cases, using experience replay is not a very good idea.

What problem is Experience replay solving? There can be some experiences which are rare. Using past experiences allows network to learn rare experience more than once. In addition, a deep Q-Network (DQN) can just be trained with the consecutive samples of experiences as they occur sequentially in the environment. However, these samples can be highly correlated since they only represent the states of environment which are very close to each other. This can lead to inefficient training. Choosing better samples from replay memory breaks this correlation and therefore lead to a better training.

III. Experimental Setup

We trained a model that learns to play a simplified version of chess. Only three pieces are used: the agent's king and queen, and the opponent's king. These pieces move on a 4x4 chess board. At the beginning of each game, the 3 pieces are distributed randomly on the board. It is important to note that the basic rules of chess are respected: two kings can't be on touching squares and a king cannot put himself deliberately in a check position. Additionally, the agent's queen will never put herself in danger. The game can end either by a checkmate of the opponent's king, if he is on a threatened square and cannot move to a safe square, or in a draw – if he is not threatened at that moment, but cannot play when it is its turn as there is no safe square he can move to.

In our simulation, the goal is to train the agent to checkmate the enemy king. An ending in a draw is not desirable.

IV. Methods

We implemented deep reinforcement learning using Sarsa and Q-learning, respectively. The q-value estimation was done using an artificial neural network that we implemented. The same neural network architecture was used for Sarsa and Q-learning; the code for the different functions (such as back propagation, the epsilon-greedy algorithm, etc.) was adapted from the Labs of the Introduction to Reinforcement Learning lecture [8].

A. Neural Network implementation

a) Feature encoding: The input layer included the position of the 3 pieces of the game (16 options each), whether the opponent's king is in checkmate or not, and his degree of freedom. These were one-hot-encoded in a binary vector of length 58.

b) Hidden layer: Our network included one hidden layer composed of 200 neurons.

c) Output layer: The output is a vector of the 32 action that the agent can take at a given state. 24 actions are for the queen, as she can go 1, 2 or 3 steps in 8 directions; and the remaining 8 for the king, which corresponds to one step possible in the 8 directions. Note that while the network outputs an estimated q-value for each action, the agent can only choose among a subset of those actions according to the rules of the game.

d) Weight initialization: One of the first decisions to make for the neural network implementation is the weight initialization. After some experimentation with different approaches, we found that the Xavier initialization was yielding better results. We chose to keep it for the final model.

e) Activation function: We chose to use a rectified linear unit activation function. The main advantages of the ReLU activation function are faster computation and avoidance of the vanishing gradient problem.

B. Sarsa implementation

The first model we implemented used Sarsa, and in particular the implementation described in Sutton and Barto (2018) [2]. For Sarsa, only the result of the action taken should be used for learning. Thus, the error corresponding to that action should be propagated specifically to the concerned weights. For the weights connecting the hidden layer and the output layer, only the weights connected to the taken action are updated. For the weights connecting the input and the hidden layer however, the input is one hot encoded which means it's the combination of the entire vector who determines a state and not just one. Therefore, all the weights connecting these two layers are updated.

C. Q-learning implementation

We implemented Q-learning after implementing Sarsa, once again following the algorithm described in Sutton and Barto(2018) [2]. It is worth noting that the implementation of the two algorithms ended up being quite similar, as the only difference is in the way the initial error signal is computed, and not in the way it is backpropagated through the network. In particular, after the reward and the expected future rewards are observed by the agent, we update the weights connected to the optimal action, which may or may not have been the one our agent chose.

D. Parameters tuning

On keeping all other parameters constant, we tried to tune β and γ. We first kept β constant and changed γ values. For the best γ values, we then changed β values. We decided the best β and γ using the best *average reward* received. We performed this experiment for Sarsa algorithm. We then tested Q-learning algorithm for same best parameters.

E. Experiments with state representation and rewards

We experimented different structures for our model mainly by varying two key factors:

- the state representation, i.e. how the information is encoded in the input vector.
- the administration of reward.

In this section, we will give a high-level overview of our experiments. The outcomes will be discussed in the results section.

1) Changes of state representation:

1) First, we modified the state representation by removing the input features concerning the degrees of freedom. Among the information concatenated in the input vector, the degree of freedom of the enemy king seemed the least important, as it could theoretically be derived from the disposition of pieces on the board. It could therefore be redundant for a well trained model. By removing it, the first experiment will then use a scarcer input vector of only 1x50.
2) The second input state change concerned the one-hot-encoding. As of now, every position (respectively degree of freedom) is represented by a 1 in that position. It is possible to remove the last number, still keeping the same information: all zero digits would correspond to the 16th square (resp. 8 degree of freedom). Similarly, the encoding of $check$, currently using two digits which can be represented with just one (0 for a draw and 1 for a checkmate). This change has a weaker impact compared to the first one. We reduced the input of 5 digits - 3 for the positions of each pieces, one for the degree of freedom and one for the checkmate - and end up with a shorter input vector of 1x53.

2) Changes in reward: We kept the original state representation to experiment with variations in the administration of reward. We experimented with 3 variations:

1) In the original setup, only the result of winning the game is rewarded. However, being able to do it in a low number of moves is arguably also important to have an efficient algorithm. Therefore, to encourage the agent to get to a checkmate position as soon as possible, we increased the reward in the case of a really low number of moves to win. It is important to note that the reward will not be increased if the game ends in a draw. We increased the reward to 2 in case of a game finishing in 5 moves or less. For a higher number of moves, we left the original reward of 1. In the case of a draw, independently of the number of moves, the reward was 0.
2) We tried administering a reward of 2 for every win, unconditional on the length of the game.
3) We tried it administering a significantly bigger reward of 10 for a wining game, unconditional on the length of the game.

F. Exploding Gradients and RMSprop

While training deep neural networks, the values of weights can increase dramatically, especially in the initial layers, as the gradient is obtained by multiplying several numbers with each other. When relatively large numbers (more than 1) are multiplied, their multiple will be even larger. Therefore, gradients can becomes really high. This phenomenon is known as exploding gradients.

In order to understand this concept practically, we tuned our network to explode its gradients. In the same neural network architecture mentioned above for Sarsa, instead of Xavier initialization, we randomly initialized our weights from 0 to 300. As a result, our gradients exploded.

In order to fix exploding gradients, we implemented a root mean squared propagation optimizer (RMSprop). RMSprop is an extension of Gradient Descent optimization. RMSprop [3] keeps a moving average of the squared gradient for each weight. Code snippet can be found in Appendix [3].

We trained the same neural network which has exploding gradients but the only difference was that optimizer is RMSprop. We found that RMSprop fixed the problem of exploding gradients, and we will discuss this in detail in the results section.

G. *Understanding code*

Our code is divided into four main parts: *agents*, *env*, *utils*, *configs*. *agents* has *Sarsa*, *Q-learning*, and *Random* agents. *env* has all the files for environment setup. *configs* has configurations corresponding to all the agents. All the parameters can be tuned by the *configs*. *utils* has utility and helping functions. It also has *NN* file which has back propagation and forward propagation implemented. In neural network, we have options to choose between ReLU and sigmoid activation functions. We also have options to choose between Gradient Descent and RMSprop optimizers. Choice can be given in *configs*. Using this, we performed several experiments and results of which can be found in Results section below. Details regarding running code is provided in README of our git repository. Default *configs* are the ones which gives the best results.

V. Results

A. *Sarsa and parameter tuning*

With Sarsa, using the original parameters (β of 0.00005 and γ of 0.85), resulted of an average reward of 0.874 needing on average 4.017 moves with *N_episodes*=100'000.

We performed parameter tuning of γ and β for Sarsa (Xavier init, ReLU activation, GD optimizer). We ran our experiment on *eta*=0.0035 and *N_episodes*=10,000. On keeping β constant and changing γ, we get the following results.

β	γ	Average Reward	Average steps
0.00005	0.75	0.5118	7.2885
0.00005	0.85	0.4648	7.5982
0.00005	**0.95**	**0.5679**	7.9358

On increasing γ, *average reward* increased. On keeping best γ constant, we then change β. Following are the results.

β	γ	Average Reward	Average steps
0.0001	0.95	0.5795	7.7267
0.001	**0.95**	**0.7103**	9.35
0.01	0.95	0.678	11.2759
0.1	0.95	0.5602	48.7148

On increasing the speed β of the decaying trend of ϵ, *Average steps* is increasing. Based on the experiments performed above, best β is 0.001 and best γ is 0.95. By running it with the total *N_episodes*=100,000, it resulted in improvement in the performance: an average reward of 0.959, needing however 4.853 steps.

Fig. 1 and Fig. 2 are *average rewards* for default parameters and best tuned parameters respectively. It can be observed that with tuned parameters, algorithm converges faster as compared to default parameters.

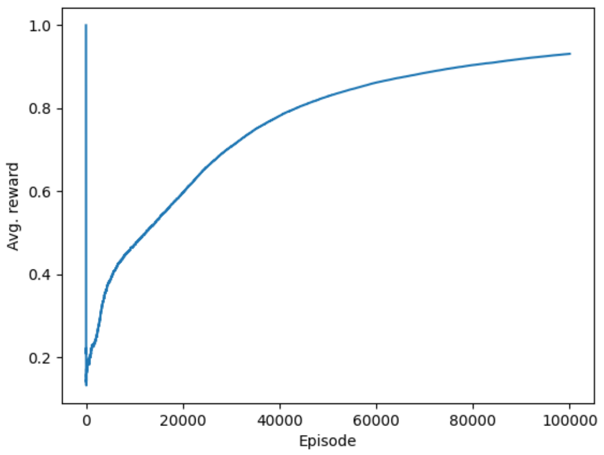


Fig. 1. Sarsa: Average Reward of Original parameters

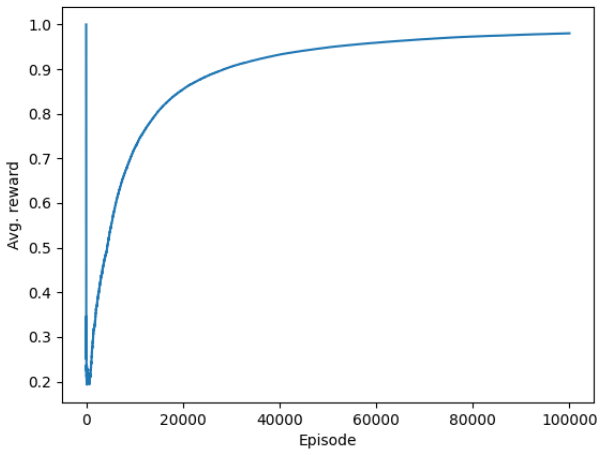


Fig. 2. Sarsa: Average Reward of Optimized parameters

B. *Q-learning*

With the original parameters, Q-learning achieved an a average reward of 0.911 with 3.757 steps.

When we used the tuned parameters β and γ from Sarsa, the average reward (shown in Fig. 3) increased to 0.955, taking on average 5.668 steps. It is worth noting that the number of required moves increased, suggesting that the model eventually learned a more prudent strategy less likely to result in a draw.

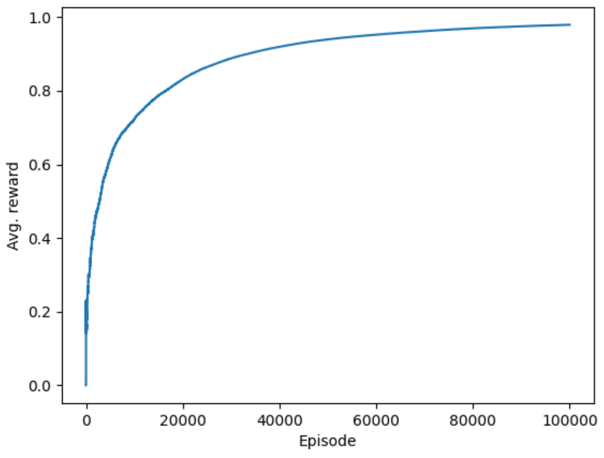


Fig. 3. Q-learning: Average Reward of Optimized Parameters

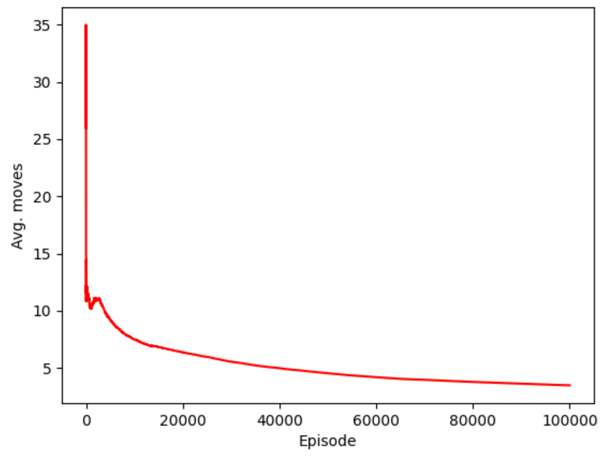


Fig. 4. Sarsa: Average Moves of Original parameters

C. Change of representation

Theses modifications have been implemented with the Sarsa agent using the optimized parameters, for 100'000 episodes. They can be reproduced either by changing the tuning parameters in $config$ or as explained in the appendix for the cases of the reduced one-hot encoding [1] and the increased reward conditional on the number of steps [2].

We will first discuss the impact of the modifications on the state representation, and then on the reward administration.

1) State representation:

Modifications	Victory Rate	Avg Steps
Original parameters	0.874	4.017
Tuned parameters	0.959	4.853
No degree of freedom	0.910	5.632
Reduced one-hot-encoding	0.942	4.611

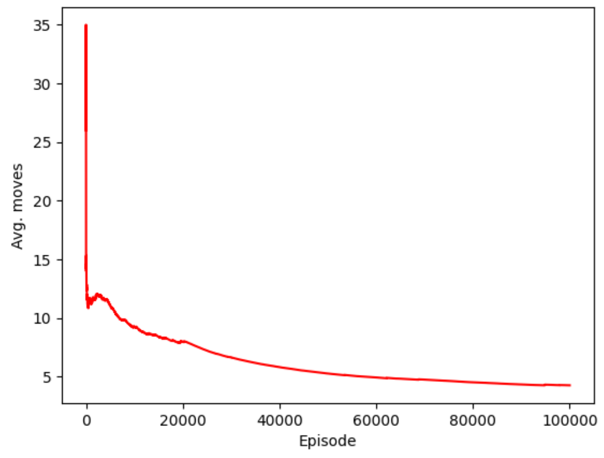


Fig. 5. Sarsa: Average Moves of Optimized Parameters. Note the reduction in the number of steps required to achieve good performance.

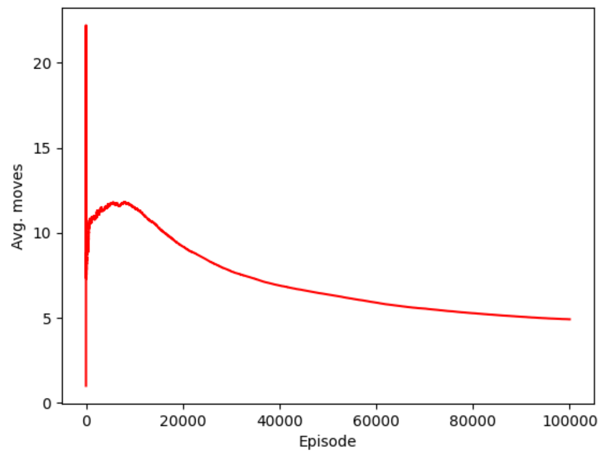


Fig. 6. Q-learning: Average Moves of Optimized Parameters

To start, the removal of the degree of freedom revealed itself to be a bad decision as both the reward and the steps only worsened. This information proved itself to be a relevant input in the neural network.

The simpler change in the one-hot encoding, removing the last digit of every information was however more counterbalanced. This time, the rewards averaged 0.942 (slightly worse) and in only 4.611 steps (slightly better). There is no significant difference with the original input.

We can therefore see that the results were not conclusive for the change in input. However, some better results were yielded by changing the representation of the reward.

2) Reward administration:

We report below the results of our experiments from our changes in state representation. Note that we discuss victory rate and not average reward, as we did in previous tables, as the reward is occasionally multiplied by 2 and 10, making it non-meaningful. The victory rate is equivalent to the average reward with a reward of 1 for winning, therefore the results in this table are comparable to the other results reported previously.

Modifications	Victory Rate	Avg Steps
Original parameters	0.874	4.017
Tuned parameters	0.959	4.853
R=2 if 5 steps or less	0.955	4.183
R=2 (unconditionally)	0.963	3.704
R=10 (unconditionally)	0.974	3.228

First, the doubled reward as payoff for a short path (i.e., 5 moves or less) allowed us to improve the efficiency of the algorithm, as it can be observed in the drop of the number of steps: only 4.183 on average. The victory rate stayed consistent: 0.955 on average, almost the same than with a normal reward attribution.

However, rewarding specifically rapid game was proved to be not what improved our neural network. With the unconditional reward of 2 – for a checkmate - both the victory rate and the average number of moves improved : the numbers of steps decreased even more, averaging 3.704, and we reached our highest average reward up to that point: 0.963. We can therefore note that is is straightforwardly the increased reward which is responsible for better results.

Following this observation, we can notice the how drastic the change that an even higher reward brings to the implementation. By going further and administrating a reward of 10 for each win, the agent achieves a victory rate of 0.974, with an average of 3.228 moves. This was our best performing model.

D. *Observations of performance*

We obtained our best result, with a victory rate of 0.974 and an average of 3.228 moves using a fine-tuned Sarsa model with a reward of 10 from winning, and 0 from draws unconditional on the number of moves required. We used a learning rate of 0.0035, a beta of 0.001, a gamma of 0.95 and trained the model for 100'000 episodes.
It is interesting to note that Sarsa performed better than Q-learning. We hypothesized that this could be due to the fact that Sarsa tends to learn more conservative strategies than Q-learning, as illustrated in the cliff example in Sutton and Barto (2018) [2], leading to fewer draws. One surprising finding is that increasing the reward for winning in less than 5 moves did decrease the average number of moves our agent took, but not as much as just increasing the reward to 10.

E. *Exploding Gradients and RMSprop*

In order to show that we fixed the problem of exploding gradients using RMSprop, we plotted weight of neural network between 10th neuron from input layer to 10th neuron on hidden layer. Weight has been chosen on random basis and has no bias from our side. In Fig. 7, it can be seen that several values are missing. The reason is those values were nan and that's why not plotted. It shows that we had exploding gradients. On using RMSprop, we plotted Fig. 8. Since it was noisy, we decided to use exponential moving average. In order to further show we had problem of exploding gradients and we have fixed it, we added 2 zip files named *exploded_weights* and *fixed_weights* in our git repository. There are csv files of weights of first layer for a few episodes. Weights in *exploded_weights* became nan after a few actions in 1st episode. Weights in *fixed_weights* are fixed.

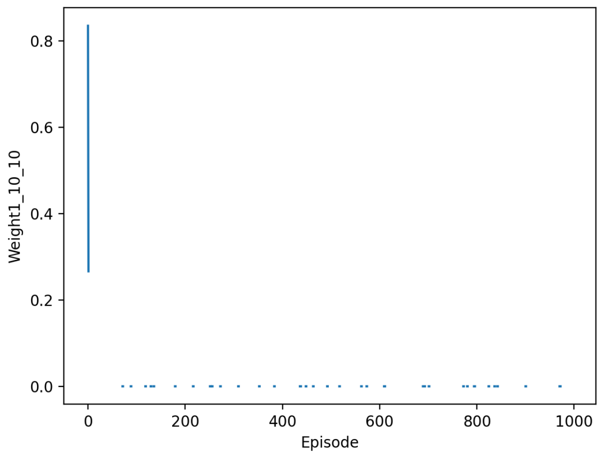


Fig. 7. Sarsa: Exploding Gradients

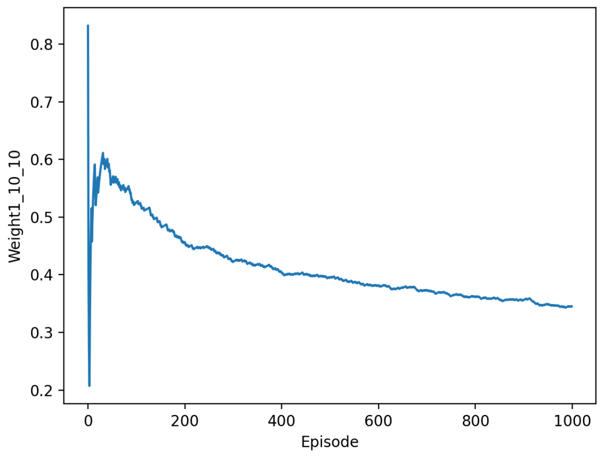


Fig. 8. Sarsa: Fixed Exploding Gradients

VI. Conclusion

We developed two models that learned to play a simplified version of chess, using Sarsa and Q-learning respectively. We experimented with several variations in the administration of reward and in state representation. We found that a change in representation by using one-hot encoding and dropping three features that were a linear combinations of other features lead to a significant increase in performance. Using Sarsa administering a reward ten times higher for winning and using tuned parameters lead to the best performance, with a victory rate over all training episodes of 0.974 and an average number of moves required of 3.228.

Acknowledgment

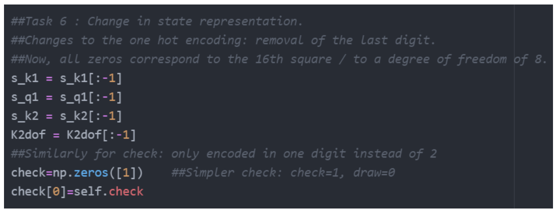
We would like to thank Prof. Eleni Vasilaki and Dr. Luca Manneschi for delivering this course in a dynamic and interactive way, and for their significant support during the final project and the labs.

References

[1] Arabnejad, Hamid, et al. ”A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling.” 2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID). IEEE, 2017.
[2] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.
[3] http://www.cs.toronto.edu/ hinton/coursera/lecture6/lec6.pdf
[4] Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3-4):293–321, 1992.
[5] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. In International Conference on Machine Learning, pages 3061–3071. PMLR, 2020.
[6] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. International Conference on Learning Representations (ICLR), 2015.
[7] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. Distributed prioritized experience replay. In International Conference on Learning Representations (ICLR), 2018.
[8] Material of the lecture ”Introduction to Reinforcement Learning” by Pr. Eleni Vasilaki. Available at http://staffwww.dcs.shef.ac.uk/people/E.Vasilaki/introduction-to-reinforceme.html

Appendix

[1] Code snippet of the reduced one-hot-encoding, used to change the representation of the state used as input for the neural network. The last digit of every element of the features is removed, which result in a state with the shape 1x53.

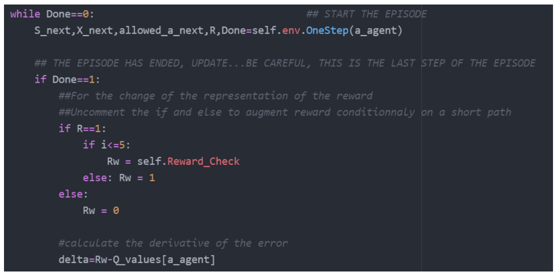


```
##Task 6 : Change in state representation.
##Changes to the one hot encoding: removal of the last digit.
##Now, all zeros correspond to the 16th square / to a degree of freedom of 8.
s_k1 = s_k1[:-1]
s_q1 = s_q1[:-1]
s_k2 = s_k2[:-1]
K2dof = K2dof[:-1]
##Similarly for check: only encoded in one digit instead of 2
check=np.zeros([1])    ##Simpler check: check=1, draw=0
check[0]=self.check
```

Fig. 9. Reduced one-hot-encoding

[2] Code snippet of the implementation of the increased reward (Reward=2) only if the agent finds a checkmate in 5 steps or less. The reward is not modified separately than in ”Chess_env.py” to be able to keep a consistent average reward.

[3] Code snippet of the implementation of RMSprop in backpropagation with ReLU activation function.

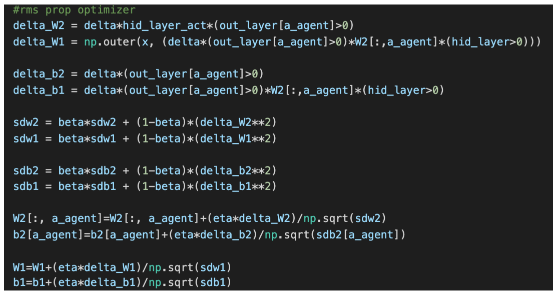


```
while Done==0:                                        ## START THE EPISODE
    S_next,X_next,allowed_a_next,R,Done=self.env.OneStep(a_agent)

    ## THE EPISODE HAS ENDED, UPDATE...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE
    if Done==1:
        ##For the change of the representation of the reward
        ##Uncomment the if and else to augment reward conditionnaly on a short path
        if R==1:
            if i<=5:
                Rw = self.Reward_Check
            else: Rw = 1
        else:
            Rw = 0

        #calculate the derivative of the error
        delta=Rw-Q_values[a_agent]
```

Fig. 10. Reduced one-hot-encoding



```
#rms prop optimizer
delta_W2 = delta*hid_layer_act*(out_layer[a_agent]>0)
delta_W1 = np.outer(x, (delta*(out_layer[a_agent]>0)*W2[:,a_agent]*(hid_layer>0)))

delta_b2 = delta*(out_layer[a_agent]>0)
delta_b1 = delta*(out_layer[a_agent]>0)*W2[:,a_agent]*(hid_layer>0)

sdw2 = beta*sdw2 + (1-beta)*(delta_W2**2)
sdw1 = beta*sdw1 + (1-beta)*(delta_W1**2)

sdb2 = beta*sdb2 + (1-beta)*(delta_b2**2)
sdb1 = beta*sdb1 + (1-beta)*(delta_b1**2)

W2[:, a_agent]=W2[:, a_agent]+(eta*delta_W2)/np.sqrt(sdw2)
b2[a_agent]=b2[a_agent]+(eta*delta_b2)/np.sqrt(sdb2[a_agent])

W1=W1+(eta*delta_W1)/np.sqrt(sdw1)
b1=b1+(eta*delta_b1)/np.sqrt(sdb1)
```

Fig. 11. RMSprop optimizer in Backpropagation