Tackling Chess with Deep Reinforcement Learning

Introduction to Reinforcement Learning (Spring 2022). Source code available at: https://github.com/TwoDigitsOneNumber/IntroRL_ChessAssignment

van den Bergh Laurin
Institute of Informatics
University of Zurich
Zurich, Switzerland
laurin.vandenbergh@uzh.ch — Matriculation number: 16-744-401

***Abstract*—We explored the use of three deep reinforcement learning methods for training agents to play a simplified version of chess. All algorithms, SARSA, Q-Learning and DQN were able to learn successful strategies. DQN was able to overcome the shortcomings of Q-Learning and provides an off-policy method with performance comparable to the on-policy method SARSA.**

***Index Terms*—deep reinforcement learning, chess, temporal-difference methods**

I. Introduction

In this assignment we explore three different deep reinforcement learning algorithms to learn how to play a simplified version of chess, which can be thought of as a special instance of an endgame. These three algorithms are SARSA, Q-Learning and DQN[1]. We will first provide a general look over our methodology (see Section II) and later discuss the result obtained in our experiments (see Section III).

The focus of this report lies on comparing these three algorithms in theory and in practise on the chess endgame environment. We further explored the impact of the hyper-parameters β and γ, which represent the speed of the decaying trend for the learning rate and the discount factor respectively.

Throughout the report we indicate in footnotes which task a particular section is referring to in terms of answering the task. We do this as the solutions to certain tasks are spread throughout multiple sections, e.g. task 3 is answered in Section II and Section III. Even though this assignment was not solved in a group, we decided to also answer some of the "group only" and we stick to the numbering of the assignment in order to avoid confusion.

II. Methods

A. Environment

This version of chess takes place on a 4 by 4 board and can be thought of as a specific version of an endgame where the agent has a king and a queen, and the opponent has only a king. Since this game can only end in a win for the agent or in a draw, it is the agent's goal to learn how to win the game and avoid draws. For all experiments considered, the agent will be given a reward of 1 for winning, 0 for drawing, and 0 for all intermediate steps.

This chess setting, and chess in general, fulfills the Markov property and therefore justifies the use of the temporal difference methods used in this assignment.

B. SARSA and Q-Learning[2]

1) Temporal-Difference Methods: SARSA and Q-Learning are two very related model-free types of temporal-difference (TD) algorithms for learning expected rewards, also known as Q-values, when rewards are not immediate and possibly sparse. The learning takes place via interaction with an environment through trial and error. These Q-values are in general represented by an action-value function Q and, for finitely many state-action pairs (s, a), can be considered as a Q-table where each state-action pair, (s, a), maps to a single Q-value, thus providing an estimate of the quality of any given state-action pair (s, a). In this assignment however we use neural networks to approximate the action-value function, which outputs the Q-values for all possible actions for any given state. This helps to avoid computing large Q-tables. All algorithms explored in this assignment, including DQN, require the environment to fulfill the Markov property.

2) On-policy vs. Off-policy: SARSA and Q-Learning address the temporal-credit-assignment problem [1], that is, trying to attribute future rewards to previous actions. These future rewards get discounted with the hyper-parameter γ (see Section III-C). Both algorithms repeatedly choose and take actions in the environment according to some policy π, e.g. an ϵ-greedy policy.

However, this is where they differ. SARSA is an on-policy algorithm, which means that consecutive actions are chosen according to the same policy π, even during the update step of the Q-values, which leads to the update rule:

$$Q_\pi(s,a) \leftarrow Q_\pi(s,a) + \eta(r + \gamma Q_\pi(s_{t+1}, a') - Q_\pi(s,a))$$

for some future action a' chosen according to policy π.

Q-learning, on the other hand, is an off-policy algorithm, which means that it takes its actions a according to its policy π, but during the update steps it assumes a greedy policy, i.e. optimal play, for future actions a'. Q-Learning has the update rule:

$$Q_\pi(s,a) \leftarrow Q_\pi(s,a) + \eta(r + \gamma \max_{a'} Q_\pi(s_{t+1}, a') - Q_\pi(s,a)).$$

[1]SARSA serves as answer to task 3 and DQN serves as answer to task 5. Q-Learning is an additional method beyond what was asked.

[2]Answer to task 1.

3) Advangages and Disadvantages: This leads to one of Q-Learning's major advantages: Because of Bellman's optimality equation, Q-Learning is guaranteed to learn the values for the optimal policy, i.e. $Q_*(s,a) = \max_\pi Q_\pi(s,a)$, regardless of the policy used to train it, and in a greedy stetting will take the optimal actions, at least if it was trained sufficiently. However, this can in certain cases mean that the online performance of Q-Learning will be worse than the one from SARSA, as Sutton et al. [2] demonstrate with their "gridworld" example "Cliff Walking". Our chess game is a similar situation, because a win and a draw can be very close, thus during exploration Q-Learning can accidentally create a draw because it is going for the optimum when exploiting. Q-Learning is however relatively unstable and the parameters can even diverge when it is combined with non-linear function approximators [3], making the guarantee to learn the optimal policy irrelevant.

SARSA will learn to take a safer path, because it keeps its policy in mind when updating the Q-values, i.e. it keeps in mind that it will explore in future actions. This has the advantage that SARSA in general tends to explore more than Q-Learning.

C. Experience Replay[3]

Experience replay is a technique proposed by Lin [4] to speed up the training process for reinforcement learning algorithms by reusing past experiences for future training. This is analogous to the human ability to remember past experiences and learn from them even after the fact. The past experiences are stored in a replay memory of fixed size at each time step t as a tuple $e_t = (s_t, a_t, r_t, s_{t+1})$. This essentially allows us to transform the learning process from online learning to mini-batch learning, where a batch of experiences e_j is randomly sampled for each update step. Experience replay can only be used in combination with off-policy algorithms, because otherwise the current parameters determine the next sample and create unwanted feedback loops [3], [5].

Experience replay provides many benefits over online Q-Learning, especially when neural networks are used to approximate the action-value function. First, it enables the agent to learn from past experiences more then once, leading to increased data efficiency and faster convergence [4], [5]. Second, since for each update step past experiences are sampled randomly, the correlations between the individual actions are reduced, which then reduces the variance of the updates [5]. This leads to the experience samples e_j being closer to i.i.d. and thus guaranteeing better convergence when using optimization algorithms such as stochastic gradient descent as most convergence proofs assume i.i.d. data.

D. Deep Q-Networks (DQN)[4]

A first version of the DQN algorithm was proposed by Mnih et al. [5] and combined experience replay with Q-learning, where a neural network was used as a non-linear function approximator for the action-value function. Mnih et al. [3] later improved upon the method and presented the DQN algorithm, as it is known today, where they address the problem of the Q-values $Q_\pi(s,a)$ being correlated to the target values $y = r + \gamma \max_{a'} Q_\pi(s',a')$ because they are generated using the same neural network. In the DQN algorithm they separated the Q-network from the target network and only update the target network every C steps, which helps to break this correlation and combat diverging network parameters.

Since DQN uses experience replay, we essentially transform the reinforcement learning task to a supervised learning task. Therefore a suitable loss function for the neural network is needed. Mnih et al. [3] used a squared loss of the temporal-difference error, also known as delta: $\delta = y - Q_\pi(s,a)$.

E. Experiments

In order to address all tasks, we divided the tasks into several independent experiments. First, we conducted seeded runs[5] for all three algorithms using seed 21 for reproducibility, which was chosen a-priori. These seeded runs serve as examples to compare the algorithm's online performance qualitatively. The seeds are used such that the weights of all neural networks are instantiated identically for all algorithms and they subsequently serve as seeds for any random number used during training. This makes sure that all agents start with the same initial conditions and that the results are reproducible (see Section V-A). All algorithms were run for 100000 episodes using identical model architecture and hyper-parameters (see Section II-F).

Since the seeded runs are heavily influenced by the choice of the seed, we could end up with anything between a very lucky and well performing seed, or with a very unlucky one. Also the interpretation of the seeded runs is more difficult as we just have one run for each algorithm. Therefore, we decided to perform a simulation study and complete 30 non-seeded runs for each algorithm in order to get a better idea of how the algorithms perform on average. For computational reasons we limited these runs to 40000 episodes as we realized with test runs that by then most of the training progress has already taken place.

To analyze the impact of the hyper-parameters β and γ[6] we trained 49 agents with different combinations for β and γ but keeping all other hyper-parameters and model architecture identical. We chose SARSA for this experiment as we found it to have very low variance between its unseeded runs, which makes it an ideal candidate for comparing individual runs. These runs are seeded identically to the seeded runs mentioned above.

F. Implementation and Hyper-parameters

We implemented all algorithms from scratch according to Sutton et al. [2] (SARSA and Q-Learning[7]) and Mnih et al. [3] (DQN[8]). For the implementation see file `neural_net.py`

[3]Answer to "group only" task 2.

[4]Answer to task 5: Describing the used method.

[5]Answers to task 3 and 5.

[6]Answer to task 4.

[7]SARSA as answer to task 3 and Q-Learning as additional algorithm.

[8]Answer to task 5.

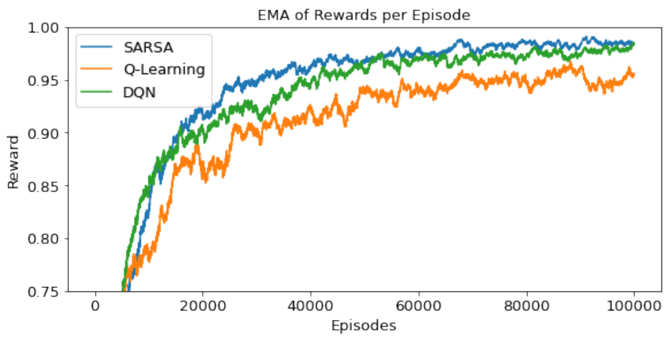

Fig. 1: Exponential moving average of the rewards achieved during training for 100000 episodes with identical hyper-parameters, weight initialization and model architecture.

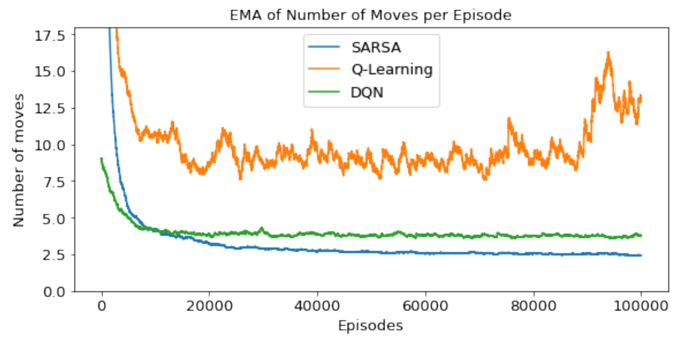

Fig. 2: Exponential moving average of the number of moves per episode achieved during training for 100000 episodes with identical hyper-parameters, weight initialization and model architecture.

on GitHub or Listing 1. All algorithms use a neural network with 58 input neurons, 200 hidden neurons and 32 output neurons, not including the biases for the input and hidden layer. The neural network automatically adds a constant input for the bias and the hidden layer. The implementation treats the biases like any other weights and thus they are part of any matrix multiplication. We used a ReLU activation function for the hidden layer and no activation on the output layer. The weights were initialized using Glorot initialization [6], such that the weights are sampled from a normal distribution with mean 0 and variance $\frac{2}{n_{\text{in}}+n_{\text{out}}}$, where n_{in} and n_{out} denote to the number of input and output neurons of the respective layer. This helped preventing exploding gradients for the most part.

For all experiments we used the default hyperparameters provided in the `Assignment.ipynb` file unless otherwise noted (see Table I). For DQN we updated the weights of the target network after every $C = 10$ steps, as most games take fewer steps than that. We used a replay memory of size 100000 and a batch size of 32.

Parameter	**Value**
Nr. input neurons	58+1
Nr. hidden neurons	200+1
Nr. output neurons	32
Initial exploration probability ϵ_0	0.2
Learning rate η	0.035
Decay rate of ϵ, β	0.00005
Discount factor γ	0.85

TABLE I: Common hyper-parameters shared by all algorithms.

III. Results

A. Seeded Runs[9]

The rewards and number of moves for the seeded runs are depicted in Figures 1 and 2 respectively. Since the curves are very noisy, we smoothed them using an exponential moving average (EMA) with a weight on the most recent observation of $\alpha = 0.001$.

As expected, the online performance of Q-Learning in terms of the rewards is generally lower than the rewards for SARSA but they converge slowly as ϵ decreases (Figure 1). Also in Figures 1 and 2 we can see that Q-Learning experiences instable learning behavior as both plots are a lot more noisy and at about 20000 and 90000 episodes the rewards decrease for some period. SARSA and DQN don't show this behavior.

Even though the number of steps is not punished, all agents still learn to reduce the number of steps over time, as they do not give rewards and their goal is to take actions that do. SARSA seems to do the best job at this, which perhaps is caused by its tendency to explore more and find better strategies. Q-Learning however seems to struggle to reduce the number of steps it takes.

As suggested by Mnih et al. [3], [5], DQN[10] was able to overcome the downsides of Q-Learning which lead to an online performance which is comparable to that of SARSA in terms of reward and number of moves it achieved, and also in terms of the stability during training. It did however not learn to reduce the amounts of steps as much as SARSA.

B. Simulation Study (Non-seeded Runs)

We can confirm that the qualitative results from the seeded runs reasonably well represent the average case. The only notable exception being Q-Learning, for which most runs performed equally well to the seeded run, but some runs experienced huge increases in the number of steps, which influenced the average run dramatically, leading to an average of about 23 moves per episode after 40000 episodes.

We observed that DQN and SARSA show very comparable learning curves with DQN showing slightly faster convergence in the first 5000 episodes. SARSA showed the lowest variance in all runs and seems to be a very stable algorithm. Q-Learning on the other hand showed clear signs of divergence as for some runs the rewards consistently dropped while the number of moves consistently increased. This shows that the measures

[9] Answer to task 3 (SARSA and Q-Learning as additional algorithm) and 5 (DQN).

[10] Answer to task 5 for comparing DQN to SARSA and Q-Learning.

taken by Mnih et al. [3] to combat the disadvantages of Q-Learning worked and increased the stability as well as the convergence speed. We were able to verify that the gradients of the Q-Learning agents were a lot less stable than the gradients of the other agents. However, using the Glorot initialization [6] helped prevent exploding gradients from occurring.

We also found out that, unsurprisingly, the effective training time is mainly dependent on the number of steps an algorithm takes per episode. This leads to Q-Learning having by far the longest training time, especially when the parameters diverge and the number of steps increase. DQN and SARSA have relatively short training times, with SARSA being the fastest.

We can conclude that the seeded runs in our initial experiment truthfully represent the average run and therefore some level of inference is justified.

C. Hyper-parameters[11]

Figure 3(a) depicts the rewards and number of moves per episode as a function of β and γ. We can see that the reward increases monotonically as γ is increases, suggesting that a value of $\gamma \in [0.80, 1)$ should be chosen for almost all values of β. This intuitively makes sense, as we have very sparse rewards and want the agent to "backpropagate" this reward through its sequence of actions. The left plot of Figure 3 suggests that reducing γ to a value in $[0.5, 0.8]$ can teach the SARSA agent to not reduce the number of steps. Intuitively this makes sense, as the only reward will be "backpropagated" less to earlier states and thus the agent will move faster towards setting the opponent's king checkmate.

The hyper-parameter β controls how fast the exploration probability ϵ will decay and therefore controls how the agent will tackle the exploration-exploitation problem. We can not see a clear relationship between β and the rewards, apart from $\beta = 0$ being an inferior choice for all values of γ. In Figure 3(b) we can see that the number of steps taken by the agent decreases drastically when increasing γ from very low levels, but this effect seems larger for larger values of β. We can however see that there is a slight, but possibly insignificant, peak in the rewards around $\beta = 5 \cdot 10^{-3}$. In summary, for reasonably chosen values of γ the choice of β seems to not have much of an influence for training periods of around 40000 episodes.

IV. Conclusion

We are aware that the performance of the individual algorithms could be improved by tuning the hyper-parameters, however, this was not explicitly asked for and the focus on this assignment lies on the comparison of these algorithms from a theoretical and practical perspective.

For any deep reinforcement learning method the choice of suitable hyper-parameters for the task is crucial and can have large impacts on the training outcome. In our case, the default parameters provided to us performed very well so no need for much further consideration was necessary.

[11] Answer to task 4.

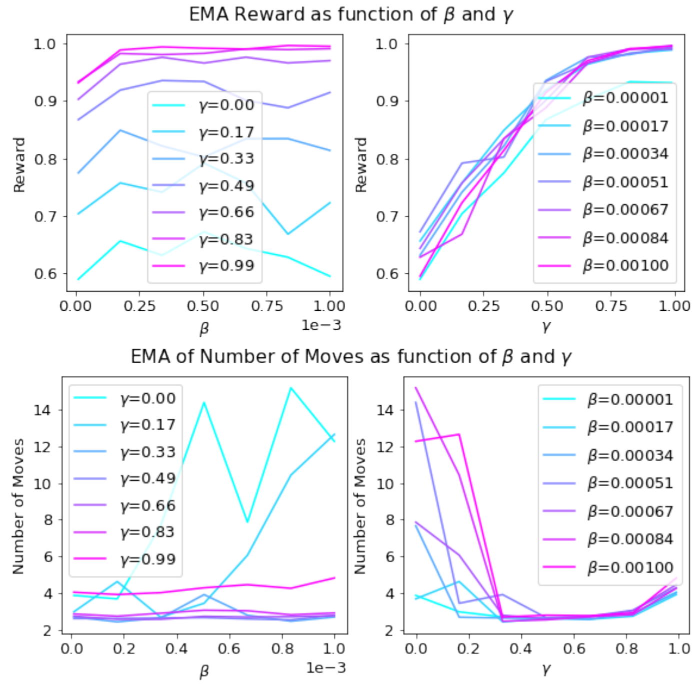

Fig. 3: Rewards and number of moves as functions of the the speed of the decaying trend β and the discount factor γ after training a SARSA agent for 40000 episodes.

All three algorithms were able to learn to play the simplified version of chess to a very high degree even without hyper-parameter tuning. SARSA proved to be the most stable algorithm, which was confirmed to be the general case with 30 non-seeded runs. Q-Learning suffers from some instabilities when training, but DQN was able to overcome all of the problems of Q-Learning and provides an off-policy method that can learn with high stability, fast convergence and a low training time comparable to SARSA. Since DQN is an off-policy method, it comes with the added advantage that it will learn an optimal policy, similar to Q-Learning.

References

[1] R. S. Sutton, "Temporal credit assignment in reinforcement learning," Ph.D. dissertation, 1984.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[3] 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," *Nat.*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[4] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, no. 3–4, p. 293–321, may 1992. [Online]. Available: https://doi.org/10.1007/BF00992699

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[6] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International*

Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics, 2010.

V. Appendix

A. Reproducibility

In order to reproduce the results presented in this report we provide the code on GitHub in the repository https://github.com/TwoDigitsOneNumber/IntroRL_ChessAssignm Along this we provide a conda environment `environment.yaml` which can be used to recreate the exact environment we used. We recommend to run the file `Assignment_Train_Algorithms.ipynb` before the files `Assignment_Compare_Algorithms.ipynb` and `Assignment_Hyperparameter_Influence.ipynb`, as the latter use files generated by the former. However, even on fast hardware running the former file takes between 5-6 hours, so we provid all necessary intermediate outputs in the repository as well.

B. Code Excerpts

```python
# import libraries
from types import MethodDescriptorType
import numpy as np
from tqdm.notebook import tqdm
import os
import json
import time
import random
from collections import namedtuple, deque

# import from files
from Chess_env import *



# ===== Epsilon-greedy Policy =====

def EpsilonGreedy_Policy(Qvalues, allowed_a, epsilon
    ):
    """
    returns: tuple
        an action in form of a one-hot encoded
            vector with the same shapeensions as
            Qvalues.
        an action as decimal integer (0-based)

    Assumes only a single state, i.e. online
        learning and NOT (mini-)batch learning.
    """
    # get the Qvalues and the indices (relative of
        all Qvalues) for the allowed actions
    allowed_a_ind = np.where(allowed_a==1)[0]
    Qvalues_allowed = Qvalues[allowed_a_ind]


    # ------------ epsilon greedy ------------

    # draw a random number and compare it to epsilon
    rand_value = np.random.uniform(0, 1, 1)

    if rand_value < epsilon:  # if the random number
        is smaller than epsilon, draw a random
        action
        action_taken_ind_of_allwed_only = np.random.
            randint(0, len(allowed_a_ind))
    else:  # greedy action
        action_taken_ind_of_allwed_only = np.argmax(
            Qvalues_allowed)

    # get index of the action that was chosen (
        relative to all actions, not only allowed)
    ind_of_action_taken = allowed_a_ind[
        action_taken_ind_of_allwed_only]


    # ------------ create usable output ------------

    # get the shapeensions of the Qvalues
    N_a, N_samples = np.shape(Qvalues)  # N_samples
        must be 1

    # initialize all actions of binary mask to 0
    A_binary_mask = np.zeros((N_a, N_samples))
    # set the action that was chosen to 1
    A_binary_mask[ind_of_action_taken,:] = 1

    return A_binary_mask, ind_of_action_taken



# ===== activation functions and it's derivatives
    ======

# relu and its derivative
def relu(x):
    return np.maximum(0,x)

def heaviside(x):
    return np.heaviside(x,0)

# sigmoid and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def gradient_sigmoid(x):
    return sigmoid(x) * (1 - sigmoid(x))

# tanh and its derivative
def tanh(x):
    return np.tanh(x)

def gradient_tanh(x):
    return 1 - np.tanh(x)**2

# identity and its derivative
def identity(x):
    return x

def const(x):
    return np.ones(x.shape)


def act_f_and_gradient(activation_function="relu"):
    if activation_function == "relu":
        return relu, heaviside
    elif activation_function == "sigmoid":
        return sigmoid, gradient_sigmoid
    elif activation_function == "tanh":
        return tanh, gradient_tanh
    else:  # identity and constant 1
        return identity, const



# ===== Replay Memory for Experience Replay (with
    DQN) =====

Transition = namedtuple('Transition', ("state", "
    action", "reward", "next_state", "done"))

class ReplayMemory(object):
    def __init__(self, capacity):
```

```python
        self.memory = deque(maxlen=capacity)

    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        # if less data than batch size, return all data
        if len(self) < batch_size:
            batch_size = len(self)
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)



# ===== Neural Network ======

class NeuralNetwork(object):

    def __init__(self, N_in, N_h, N_a, activation_function_1="relu", activation_function_2=None, method="qlearning", seed=None, capacity=100_000, C=100):
        """
        activation functions: "relu", "sigmoid", "tanh", None
        methods: "qlearning", "sarsa", "dqn"
        """
        self.D = N_in  # input dimension (without bias)
        self.K = N_h   # nr hidden neurons (without bias)
        self.O = N_a   # nr output neurons (letter O, not digit 0)

        # store method and seed
        self.method = method
        self.seed = seed

        if self.method == "dqn":
            self.capacity = capacity
            self.replay_memory = ReplayMemory(capacity)
            self.C = C

        # set activation function and gradient function
        self.act_f_1_name = activation_function_1
        self.act_f_2_name = activation_function_2
        self.act_f_1, self.grad_act_f_1 = act_f_and_gradient(activation_function_1)
        self.act_f_2, self.grad_act_f_2 = act_f_and_gradient(activation_function_2)


        # initialize the weights and biases and set grobal seed
        np.random.seed(self.seed)

        # self.W1 = np.random.randn(self.K+1, self.D+1)/np.sqrt(self.D+1)  # standard normal distribution, shape: (K+1, D+1)
        # glorot/xavier normal initialization
        # self.W1 = np.random.randn(self.K+1, self.D+1)*np.sqrt(2/ (self.D+1 + self.K+1))  # standard normal distribution, shape: (K+1, D+1)
        self.W1 = np.random.standard_normal((self.K+1, self.D+1))*np.sqrt(2/ (self.D+1 + self.K+1))  # standard normal distribution, shape: (K+1, D+1)
        # self.W1 = np.random.randn(self.K+1, self.D+1)  # standard normal distribution, shape: (K+1, D+1)

        # self.W2 = np.random.randn(self.O, self.K+1)/np.sqrt(self.K+1)  # standard normal distribution, shape: (O, K+1)
        # glorot/xavier normal initialization
        self.W2 = np.random.standard_normal((self.O, self.K+1))*np.sqrt(2/ (self.K+1 + self.O))  # standard normal distribution, shape: (O, K+1)
        # self.W2 = np.random.randn(self.O, self.K+1)  # standard normal distribution, shape: (O, K+1)

        if self.method == "dqn":
            self.W1_target = np.copy(self.W1)
            self.W2_target = np.copy(self.W2)


    def forward(self, x, target=False):
        """
        x has shape: (D+1, 1) (constant bias 1 must be added beforehand added)
        target: if True, use the weights of the target network

        returns:
            last logits (i.e. Qvalues) of shape (O, 1)
        """

        if target == True:
            W1 = np.copy(self.W1_target)
            W2 = np.copy(self.W2_target)
        else:
            W1 = np.copy(self.W1)
            W2 = np.copy(self.W2)

        # forward pass/propagation
        a1 = W1 @ x
        h1 = self.act_f_1(a1)
        h1[0,:] = 1  # set first row (bias to second layer) to 1 (this ignores the weights for the k+1th hidden neuron, because this should not exist; this allows to only use matrix multiplication and simplify the gradients as we only need 2 instead of 4)
        a2 = W2 @ h1
        h2 = self.act_f_2(a2)
        return a1, h1, a2, h2


    def backward(self, R, x, Qvalues, Q_prime, a1, h1, a2, gamma, future_reward, action_binary_mask):
        """
        backward for methods "qlearning" and "sarsa"

        x has shape (D+1, 1) (constant bias 1 must be added beforehand)
        set future_reward=True for future reward with gamma>0, False for immediate reward.
        Q_prime must be chosen according to the method on x_prime (on- or off-policy)
        """
```

```
        # backward pass/backpropagation
        # compute the gradient of the square loss with respect to the parameters

        # ===== compute TD error (aka delta) =====

        # make reward of shape (O, 1)
        R_rep = np.tile(R, (self.O, 1))
        if future_reward:  # future reward
            delta = R_rep + gamma*Q_prime - Qvalues  # -> shape (O, 1)
        else:  # immediate reward
            delta = R_rep - Qvalues  # -> shape (O, 1)

        # update only action that was taken, i.e. all rows apart from the one corresponding to the action taken (action index) are 0
        delta = delta*action_binary_mask


        self.compute_gradients(delta, a1, h1, a2, x)
        self.update_parameters(self.eta)


    def backward_dqn(self, batch, gamma):
        """
        backward for method "dqn"
        """

        # ===== compute targets y and feature matrix X =====

        # turn batch into individual tuples, numpy arrays, or lists
        states = batch.state
        rewards = np.array(list(batch.reward))
        actions = np.array(list(batch.action))
        next_states = list(batch.next_state)
        dones = np.array(list(batch.done))

        # compute targets y and feature matrix X
        y = np.zeros((self.O, len(dones)))
        for j in np.arange(len(dones)):
            if dones[j]:  # if done, set y_j = r_j
                y[actions[j], j] = rewards[j]
            else:
                # compute Q_prime
                Q_target = self.forward(next_states[j], target=True)[-1]
                y[actions[j], j] = rewards[j] + gamma*np.max(Q_target)


        # convert states to feature matrix X
        X = np.hstack((states))


        # ===== compute TD error (aka delta) =====

        a1, h1, a2, Qvalues = self.forward(X)
        delta = y - Qvalues  # -> shape (O, batch_size)

        self.compute_gradients(delta, a1, h1, a2, X)
        self.update_parameters(self.eta)


    def compute_gradients(self, delta, a1, h1, a2, x):
        # ===== compute gradient of the loss with respect to the weights =====

        # common part of the gradient  TODO: check dimensions
        self.dL_da2 = delta * self.grad_act_f_2(a2)

        # gradient of loss wrt W2
        self.dL_dW2 = self.dL_da2 @ h1.T

        # gradient of loss wrt W1
        self.dL_dW1 = ( (self.W2.T @ self.dL_da2) * self.grad_act_f_1(a1) ) @ x.T



    def update_parameters(self, eta):

        # gradient clipping

        # dL_dW1_norm = np.linalg.norm(self.dL_dW1)
        # if dL_dW1_norm >= self.gradient_clip:
        #     self.dL_dW1 = self.gradient_clip * self.dL_dW1 / dL_dW1_norm

        # dL_dW2_norm = np.linalg.norm(self.dL_dW2)
        # if dL_dW2_norm >= self.gradient_clip:
        #     self.dL_dW2 = self.gradient_clip * self.dL_dW2 / dL_dW2_norm

        # update W1 and W2
        self.W2 = self.W2 + eta * self.dL_dW2
        self.W1 = self.W1 + eta * self.dL_dW1




    def train(self, env, N_episodes, eta, epsilon_0, beta, gamma, alpha=0.001, gradient_clip=1, batch_size=32, run_number=None):
        """
        alpha is used as weight for the exponential moving average displayed during training.
        batch_size is only used for the DQN method.
        """

        # add training hyper parameters
        self.N_episodes = N_episodes
        self.eta = eta
        self.epsilon_0 = epsilon_0
        self.beta = beta
        self.gamma = gamma
        self.alpha = alpha
        self.gradient_clip = gradient_clip
        self.batch_size = batch_size


        training_start = time.time()

        try:

            # initialize histories for important metrics
            self.R_history = np.full([self.N_episodes, 1], np.nan)
            self.N_moves_history = np.full([self.N_episodes, 1], np.nan)
            self.dL_dW1_norm_history = np.full([self.N_episodes, 1], np.nan)
            self.dL_dW2_norm_history = np.full([self.N_episodes, 1], np.nan)

            # progress bar
            episodes = tqdm(np.arange(self.N_episodes), unit="episodes")
            ema_previous = 0
```

```
        n_steps = 0

        for n in episodes:

            epsilon_f = self.epsilon_0 / (1 + beta * n)    ## DECAYING EPSILON
            Done = 0                                        ## SET DONE TO ZERO (BEGINNING OF THE EPISODE)
            i = 1                                           ## COUNTER FOR NUMBER OF ACTIONS

            S, X, allowed_a = env.Initialise_game()         ## INITIALISE GAME
            X = np.expand_dims(X, axis=1)                   ## MAKE X A TWO DIMENSIONAL ARRAY
            X = np.copy(np.vstack((np.array([[1]]), X)))  # add bias term

            if self.method == "sarsa":
                # compute Q values for the given state
                a1, h1, a2, Qvalues = self.forward(X)  # -> shape (O, 1)

                # choose an action A using epsilon-greedy policy
                A_binary_mask, A_ind = EpsilonGreedy_Policy(Qvalues, allowed_a, epsilon_f)  # -> shape (O, 1)


            while Done==0:                                  ## START THE EPISODE

                if (self.method == "qlearning") or (self.method == "dqn"):
                    # compute Q values for the given state
                    a1, h1, a2, Qvalues = self.forward(X)  # -> shape (O, 1)

                    # choose an action A using epsilon-greedy policy
                    A_binary_mask, A_ind = EpsilonGreedy_Policy(Qvalues, allowed_a, epsilon_f)  # -> shape (O, 1)


                # take action and observe reward R and state S_prime
                S_prime, X_prime, allowed_a_prime, R, Done = env.OneStep(A_ind)
                X_prime = np.expand_dims(X_prime, axis=1)
                X_prime = np.copy(np.vstack((np.array([[1]]), X_prime)))  # add bias term

                n_steps += 1

                if self.method == "dqn":

                    # store the transition in memory
                    self.replay_memory.push(X, A_ind, R, X_prime, Done)

                    # sample a batch of transitions
                    transactions = self.replay_memory.sample(self.batch_size)
                    # turn list of transactions into transaction of lists
                    batch = Transition(*zip(*transactions))

                    # backward step and parameter update
                    self.backward_dqn(batch, self.gamma)

                # update Q values indirectly by updating the weights and biases directly

                if Done==1:  # THE EPISODE HAS ENDED, UPDATE...BE CAREFUL, THIS IS THE LAST STEP OF THE EPISODE

                    if (self.method == "qlearning") or (self.method == "sarsa"):
                        # compute gradients and update weights
                        self.backward(R, X, Qvalues, None, a1, h1, a2, None, future_reward=False, action_binary_mask=A_binary_mask)

                    # store history
                    # todo: record max possible reward per episode
                    self.R_history[n] = np.copy(R)  # reward per episode
                    self.N_moves_history[n] = np.copy(i)  # nr moves per episode

                    # store norm of gradients
                    self.dL_dW1_norm_history[n] = np.linalg.norm(self.dL_dW1)
                    self.dL_dW2_norm_history[n] = np.linalg.norm(self.dL_dW2)

                    # compute exponential moving average (EMA) to display during training
                    ema = alpha*R + (1-alpha)*ema_previous
                    if n == 0: # first episode
                        ema = R
                    ema_previous = ema
                    if run_number is not None:
                        episodes.set_description(f"Run = {run_number}; EMA Reward = {ema:.2f}")
                    else:
```

```
                        episodes.set_description(f"EMA Reward = {ema:.2f}")

                    break

                else:  # IF THE EPISODE IS NOT OVER...

                    if self.method == "qlearning":
                        # chose next action off-policy
                        Q_prime = np.max(self.forward(X_prime)[-1])

                    elif self.method == "sarsa":
                        # chose next action on-policy

                        a1_prime, h1_prime, a2_prime, Qvalues_prime = self.forward(X_prime)  # -> shape (N_a, 1)

                        # chose next action and save it
                        A_binary_mask_prime, A_ind_prime = EpsilonGreedy_Policy(Qvalues_prime, allowed_a_prime, epsilon_f)

                        # get Qvalue of next action
                        Q_prime = Qvalues_prime[A_ind_prime]


                    if (self.method == "qlearning") or (self.method == "sarsa"):
                        # backpropagation and weight update
                        self.backward(R, X, Qvalues, Q_prime, a1, h1, a2, self.gamma, future_reward=True, action_binary_mask=A_binary_mask)


                    # NEXT STATE AND CO. BECOME ACTUAL STATE...
                    if self.method == "sarsa":
                        A_binary_mask = np.copy(A_binary_mask_prime)
                        A_ind = np.copy(A_ind_prime)
                        a1 = np.copy(a1_prime)
                        h1 = np.copy(h1_prime)
                        a2 = np.copy(a2_prime)
                        Qvalues = np.copy(Qvalues_prime)
                    S = np.copy(S_prime)
                    X = np.copy(X_prime)
                    allowed_a = np.copy(allowed_a_prime)

                    i += 1  # UPDATE COUNTER FOR NUMBER OF ACTIONS

                    if (self.method == "dqn") and (n_steps % self.C == 0):
                        # update target network every C steps
                        self.W1_target = np.copy(self.W1)
                        self.W2_target = np.copy(self.W2)


            training_end = time.time()
            self.training_time_in_seconds = training_end - training_start

            return None


        except KeyboardInterrupt as e:
            # return nothing
            training_end = time.time()
            self.training_time_in_seconds = training_end - training_start

            return None


    def save(self, name_extension=None):
        # create directory for the model
        name = f"{self.method}_{self.act_f_1_name}_{self.act_f_2_name}"
        if name_extension is not None:
            name += f"_{name_extension}"

        path = f"models/{name}"
        if not os.path.isdir(path): os.mkdir(path)
        print(f"saving to: {path}")

        # save weights
        np.save(f"{path}/W1.npy", self.W1)
        np.save(f"{path}/W2.npy", self.W2)

        # save training history
        np.save(f"{path}/training_history_R.npy", self.R_history)
        np.save(f"{path}/training_history_N_moves.npy", self.N_moves_history)
        np.save(f"{path}/training_history_dL_dW1_norm.npy", self.dL_dW1_norm_history)
        np.save(f"{path}/training_history_dL_dW2_norm.npy", self.dL_dW2_norm_history)

        # save training parameters and other general info
        params = {
            "method": self.method,
            "N_episodes": self.N_episodes,
            "eta": self.eta,
            "epsilon_0": self.epsilon_0,
            "beta": self.beta,
            "gamma": self.gamma,
            "alpha": self.alpha,
            # "gradient_clip": self.gradient_clip,
            "seed": self.seed,
            "D": self.D,
            "K": self.K,
            "O": self.O,
            "training_time_in_seconds": self.training_time_in_seconds
        }
        if self.method == "dqn":
            params["capacity"] = self.capacity
```

```python
            params["batch_size"] = self.batch_size
            params["C"] = self.C
        with open(f"{path}/training_parameters.json", "w") as f:
            json.dump(params, f)


def load_from(method, act_f_1, act_f_2, name_extension=None):

    # read values and store in neural network instance
    name = f"{method}_{act_f_1}_{act_f_2}"
    if name_extension is not None:
        name += f"_{name_extension}"

    path = f"models/{name}"
    # print(f"loading from: {path}")

    # initialize neural network
    nn = NeuralNetwork(0,0,0, activation_function_1=act_f_1, activation_function_2=act_f_2, method=method)

    # network weights
    nn.W1 = np.load(f"{path}/W1.npy")
    nn.W2 = np.load(f"{path}/W2.npy")

    # network training history
    nn.R_history = np.load(f"{path}/training_history_R.npy")
    nn.N_moves_history = np.load(f"{path}/training_history_N_moves.npy")
    nn.dL_dW1_norm_history = np.load(f"{path}/training_history_dL_dW1_norm.npy")
    nn.dL_dW2_norm_history = np.load(f"{path}/training_history_dL_dW2_norm.npy")

    # network training parameters
    with open(f"{path}/training_parameters.json", "r") as f:
        params = json.load(f)

        # set parameters to the network instance
        nn.method = params["method"]
        nn.N_episodes = int(params["N_episodes"])
        nn.eta = float(params["eta"])
        nn.epsilon_0 = float(params["epsilon_0"])
        nn.beta = float(params["beta"])
        nn.gamma = float(params["gamma"])
        nn.alpha = float(params["alpha"])
        # nn.gradient_clip = float(params["gradient_clip"])
        try:
            nn.seed = int(params["seed"])
        except:
            nn.seed = params["seed"]
        nn.D = int(params["D"])
        nn.K = int(params["K"])
        nn.O = int(params["O"])
        nn.training_time_in_seconds = float(params["training_time_in_seconds"])

        if nn.method == "dqn":
            nn.capacity = int(params["capacity"])
            nn.batch_size = int(params["batch_size"])
            nn.C = int(params["C"])


    if nn.method == "dqn":
        nn.W1_target = np.copy(nn.W1)
        nn.W2_target = np.copy(nn.W2)

    return nn
```

Listing 1: Object oriented implementation of the neural networks, which can be instantiated with specifications for the model architecture and a method: “sarsa”, “qlearning” or “dqn”. The training loop will adapt automatically.

```python
import numpy as np
import matplotlib.pyplot as plt

def moving_average(a, n=3) :
    steps = len(a)-n
    ma = np.full(steps, np.nan)
    for i in range(steps):
        ma[i] = np.mean(a[i:i+n])
    return ma, np.arange(steps)


def exponential_moving_average(array, alpha=0.001):
    """
    Calculate exponential moving average of an array
    """
    ema = np.full(len(array), np.nan)
    ema[0] = array[0]
    for i in range(1, len(array)):
        ema[i] = alpha * array[i] + (1 - alpha) * ema[i-1]
    return ema


def save_avg_statistics(histories, method):
    # unpack histories
    R_histories = [history[0] for history in histories]
    N_moves_histories = [history[1] for history in histories]
    training_times = [history[2] for history in histories]
    layer1_gradient_norms_histories = [history[3] for history in histories]
    layer2_gradient_norms_histories = [history[4] for history in histories]

    # turn into numpy arrays
    R_histories = np.hstack(R_histories)
    N_moves_histories = np.hstack(N_moves_histories)
    training_times = np.hstack(training_times)
    layer1_gradient_norms_histories = np.hstack(layer1_gradient_norms_histories)
    layer2_gradient_norms_histories = np.hstack(layer2_gradient_norms_histories)

    # compute mean and standard deviation for each row of the histories
    R_mean = np.mean(R_histories, axis=1)
    R_std = np.std(R_histories, axis=1)

    N_moves_mean = np.mean(N_moves_histories, axis=1)
    N_moves_std = np.std(N_moves_histories, axis=1)

    layer1_gradient_norms_mean = np.mean(layer1_gradient_norms_histories, axis=1)
    layer1_gradient_norms_std = np.std(layer1_gradient_norms_histories, axis=1)

    layer2_gradient_norms_mean = np.mean(layer2_gradient_norms_histories, axis=1)
    layer2_gradient_norms_std = np.std(layer2_gradient_norms_histories, axis=1)

    # save to file
```

```python
    np.save(f"statistics/{method}_R_mean.npy", R_mean)
    np.save(f"statistics/{method}_R_std.npy", R_std)

    np.save(f"statistics/{method}_N_moves_mean.npy", N_moves_mean)
    np.save(f"statistics/{method}_N_moves_std.npy", N_moves_std)

    np.save(f"statistics/{method}_training_times.npy", training_times)

    np.save(f"statistics/{method}_layer1_gradient_norms_mean.npy", layer1_gradient_norms_mean)
    np.save(f"statistics/{method}_layer1_gradient_norms_std.npy", layer1_gradient_norms_std)

    np.save(f"statistics/{method}_layer2_gradient_norms_mean.npy", layer2_gradient_norms_mean)
    np.save(f"statistics/{method}_layer2_gradient_norms_std.npy", layer2_gradient_norms_std)


def load_avg_statistics(method):
    R_mean = np.load(f"statistics/{method}_R_mean.npy")
    R_std = np.load(f"statistics/{method}_R_std.npy")

    N_moves_mean = np.load(f"statistics/{method}_N_moves_mean.npy")
    N_moves_std = np.load(f"statistics/{method}_N_moves_std.npy")

    training_times = np.load(f"statistics/{method}_training_times.npy")

    layer1_gradient_norms_mean = np.load(f"statistics/{method}_layer1_gradient_norms_mean.npy")
    layer1_gradient_norms_std = np.load(f"statistics/{method}_layer1_gradient_norms_std.npy")

    layer2_gradient_norms_mean = np.load(f"statistics/{method}_layer2_gradient_norms_mean.npy")
    layer2_gradient_norms_std = np.load(f"statistics/{method}_layer2_gradient_norms_std.npy")

    return R_mean, R_std, N_moves_mean, N_moves_std, training_times, layer1_gradient_norms_mean, layer1_gradient_norms_std, layer2_gradient_norms_mean, layer2_gradient_norms_std


def printable_name(method):
    if method == "sarsa":
        return "SARSA"
    elif method == "qlearning":
        return "Q-Learning"
    elif method == "dqn":
        return "DQN"
    else:
        return None
```

Listing 2: Helper functions used throughout the implementation of the neural network and the notebooks, where the experiments were conducted.