Reinforcement Learning Assignment

Charlotte Eder
Department of Informatics
University of Zürich
Zürich, Switzerland
charlotte.eder@uzh.ch

***Abstract*—Reinforcement learning studies how artificial systems can learn to optimize their behaviour by simulating an agent that interacts with an environment [3]. In this paper, we first discuss the two most common reinforcement learning methods, Q-Learning and SARSA. Then we combine these two methods with a neural network to make an agent learn how to play a simplified version of chess.**

***Index Terms*—reinforcement learning, neural network, SARSA, Q-Learning**

I. Introduction

Reinforcement Learning is about an agent interacting with an environment that tries to learn an optimal policy to solve a specific problem [3]. Reinforcement Learning algorithms are applied in a multitude of domains such as natural sciences, social sciences, and engineering. Also, with the rise of deep learning methods, the integration of reinforcement learning methods and neural networks has become more prominent in recent times under the name deep reinforcement learning [1]. This is why for this paper we explore the two most common reinforcement learning methods, Q-Learning and SARSA, and combine them with a neural network to solve the problem of playing a simplified version of chess.

II. Task 1: Q-Learning and SARSA

Both Q-Learning and SARSA are solution methods used in reinforcement learning to find optimal policies for agent-environment systems [2]. For this, the methods estimate the value function v_π (the *prediction problem*) and propose a way to find optimal policies π_* (the *control problem*). They do this by combining ideas from temporal difference learning, dynamic programming and Monte Carlo methods [3].

A. SARSA

SARSA is an on-policy temporal difference control method. For the *control problem*, SARSA follows a pattern of generalized policy iteration combined with temporal difference learning methods.

In a first step, SARSA learns the action-value function. This is done by estimating $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a. This can be done using the following formula for the action-values:

$Q(S_t, A_t)$ is the expected reward for the state-action pair of S and A. $Q(S_t, A_t)$ can be estimated based on the current estimation of $Q(S_t, A_t)$, the estimation of the expected reward R and based on the next state-action pair reward $Q(S_{t+1}, A_{t+1})$ as well as some fine-tuning parameters α and γ. The first three terms also give the method its name SARSA. If S_t describes a terminal state, $Q(S_{t+1}, A_{t+1})$ is set to zero. Also, it is important to assume that the state-action pairs form a Markov chain.

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

Based on this formula, an on-policy control algorithm can be designed (see fig. 1).

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Fig. 1. SARSA Algorithm

As in all on-policy methods, we continually estimate q_π for the behavior policy π. Over time we change π toward greediness with respect to q_π [3].

B. Q-Learning

Q-Learning is an off-policy temporal difference control method. For the *control problem*, the simplest form of Q-Learning, one-step Q-learning, uses the following formula to calculate the expected rewards:

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

Differently from SARSA, the learned action-value function Q directly approximates the optimal action-value function q∗ independently of the policy. This also simplifies the following algorithm (Fig. **??**):

C. Differences between Q-Learning and SARSA

There is one big difference between Q-Learning and SARSA which is that SARSA is an on-policy method whereas

Identify applicable funding agency here. If none, delete this.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Fig. 2. Q-Learning Algorithm

Q-Learning is an off-policy method. This means, that SARSA decides all actions based on the current policy π being followed. Q-Learning, on the other hand, directly approximates the optimal action-value function q$*$ independent of the policy. Assume we apply SARSA and Q-Learning to the same problem with ϵ-greedy strategy. The goal of the problem is to reach the final destination which lies beneath a cliff (see Fig. 3.

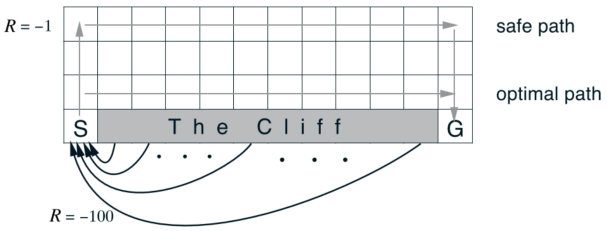

Fig. 3. The Gridworld Problem **??**

Q-Learning will learn the optimal route which follows right along the cliff. This will result in occasionally falling off the cliff because of the variance of the ϵ-greedy strategy. Since SARSA takes the action selection into account, it learn the longer but safer path away from the cliff [3].

D. Advantages and Disadvantages

The advantage of SARSA is that this method also takes into account the current policy and the action selection. This makes SARSA a good choice for problems that involve a big penalty near the optimal path. A disadvantage of SARSA is that it only learns the optimal policy if a strategy is chosen that decreases the ϵ in the ϵ-greedy strategy. Q-Learning on the other hand directly learns the optimal policy, which makes it a suitable choice if the learning process has to go fast. However, the disadvantage of Q-Learning is that it generates lower results during the training process when high penalties near the optimal path are involved [3].

III. Methods

The main logic of the reinforcement learning agent is implemented in the NeuralNetworkQAS() class. This class contains the hyper-parameters for the reinforcement learning methods, a neural network to store the q-values and functions implementing the reinforcement learning methods as well as exponential moving average smoothing. To make the results of the analysis reproducible, the class also automatically creates the same seeds for all random variables used.

A. Task 3: Implementation of SARSA

The implementation of SARSA is based on the pseudo-code in Fig. 1. For a given number of episodes, a game of chess is created via the given Chess_env.Initialise_game method. Also, to make the results reproducible, the game-creation method is given a seed to fix all random numbers that occur during the game creation. Then, the next action to take is chosen based on the q-values. This happens by giving the current state of the game self.X to the neural network that contains the q-values. In the predict() function, the neural network then makes a forward pass of with the input self.X and calculates the q-values for all actions. Then, with a probability of 1 - ϵ, the action with the highest q-values out of the allowed actions (= allowed chess game moves) is returned. with a probability of ϵ, a random action out of the allowed actions is returned. After that, for all the moves of the game (see also Fig. 8, the chosen action A is taken and the game move simulated via the function env.OneStep(A). If the game has finished, $Q(S_{t+1}, A_{t+1})$ is set to zero. Otherwise, $Q(S_{t+1}, A_{t+1})$ is calculated via the predict function of the neural network. Then, the weights of the neural network have to be updated based on the reward. This is done by using the error term $\eta(R + Q(S,A) - \gamma(Q(S_{t+1}, A_{t+1}))x2$ where x2 is a vector with the length of all possible choices for action A. It contains 1 for the action the neural network recommended to take and 0 for all the other actions. This error term is then used as input for the back-propagation algorithm of the neural network to update the weights of the neural network. Lastly, the action to take is updated for the next game move.

B. Task 5: Implementation of Q-Learning

The implementation of Q-Learning is based on the pseudo-code in Fig. 1. Like for the implementation for SARSA, a game of chess is created for a given number of episodes. Then for each game move (see also Fig. 9, an action is chosen and executed based on the state of the game via the predict function. Then, differently from SARSA, not the result of the predict function for the next move is chosen but the optimal value $maxQ(S', a)$. The error term is calculated based on the formula $\eta(R + Q(S,A) - \gamma(maxQ(S_{t+1}, a))x2$ and given as an input to the back-propagation function. Lastly, the state gets updated.

C. Original Code Parts

First of all, the code of the labs has been rearranged to fit into the NeuralNetworkQAS class. This helps to reuse the code for the analysis and to avoid duplication. Then, the code of Lab one has been modified to take the chess board with the positioning of the figures as input. Also, the output layer had to be modified to output the action to take for the next chess move. Furthermore, it was not possible to use a max() function to get the q_value with the highest output. Rather, a conditional output function had to be implemented (see Fig. 11) that takes into account all the allowed actions and chooses the maximum q_value based on these allowed actions. Moreover,

Q-learning and the exponential moving average function had to be implemented from scratch (see Fig. 10).

IV. Results

Unfortunately, there must be a mistake somewhere in the program, which is why the agent does not get better over time. The most probable explanation is that either the calculation of the error term for the action values is wrong (input for the back-propagation) or that there is some mistake in the back-propagation itself. Therefore, for the following questions, the answers are based on how the plots should look like instead of how they actually look.

A. Task 3: Implementation of SARSA

For SARSA, the average reward per game should augment over time and form some sort of sigmoid curve that converges to a value near 1. The average amount of moves per game should decline since the agent is getting faster at putting the opponent into checkmate. In reality, reward function does not change and stays around 0.1 (Fig. 4). The average moves per game are slightly augmenting over time (Fig. 5).

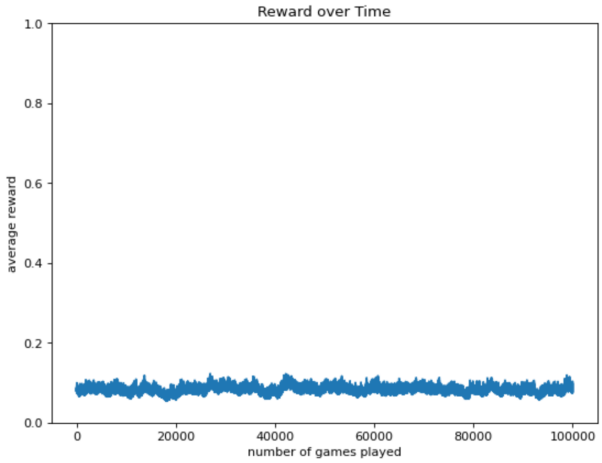

Fig. 4. Reward over time for SARSA

B. Task 4: Changing Parameters

For this task, the discount factor γ and the speed of decaying parameter β are varied. When β is augmented a little bit (e.g. beta=0.0001), the algorithm visits enough actions to do thorough exploring but converges quicker since there is from earlier on more focus on exploitation instead of exploration. However, when β is set even higher (e.g. beta=0.01), there is not enough exploration anymore and the algorithm gets stuck with non-optimal solutions. The discount factor γ defines how much weight is put on rewards that lay in the far away future compared to those in the immediate future. When γ is set to a lower value of (e.g. gamma=0.5) less weight is put on the immediate future and more onto rewards that are further away. In the case of chess, this probably leads to better results since we are focused on a reward that is far away (the checkmate) and there are not dangers in the immediate future like getting

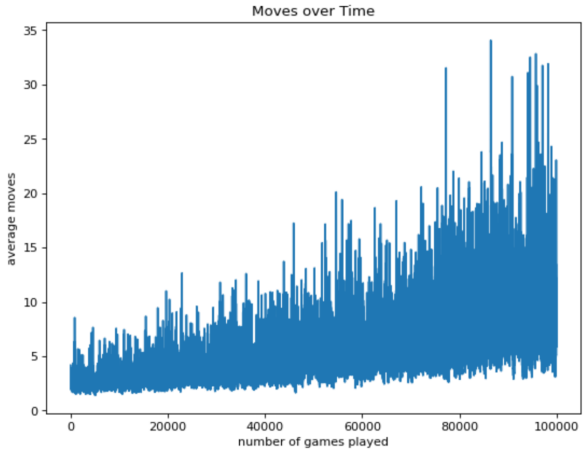

Fig. 5. Moves over time for SARSA

beaten by the opponent and receiving a big negative reward. On the other hand, if γ is set very high (e.g. gamma=1.0) and all the weight is put on the immediate reward, the learning process might be slower since the agent only takes immediate actions into account and does not have foresight on how the immediate actions will affect the future.

C. Task 5: Implementation of Q-Learning

For Q-learning, the plots look similar to SARSA except that the curve converges to a lower level because of the e-greedy strategy. The amount of moves per game should decline since the agent is getting faster at putting the opponent into checkmate. In reality, the plot for average rewards per game and average moves per game is a straight line (Fig. 6). However, the average moves per game are increasing over time (Fig. 7).

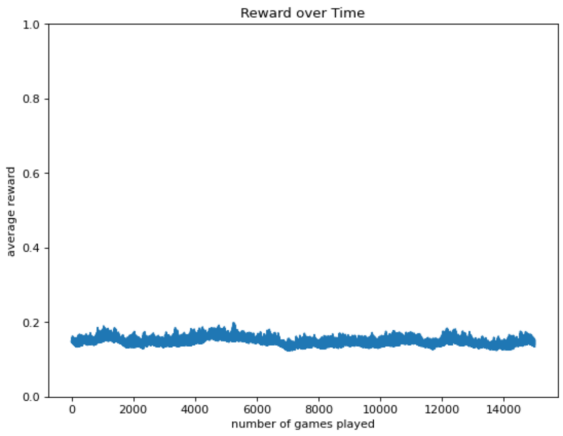

Fig. 6. Reward over time for Q-learning

V. Conclusion

In this paper, we had a look into the two most common reinforcement learning solution methods Q-Learning and SARSA. We used them together with a neural network to build an agent

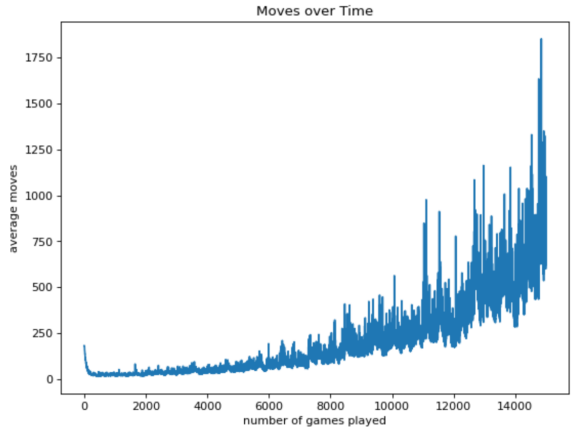

Fig. 7. Moves over time for Q-Learning

that learns how to play a simplified version of chess. However, the agent did not get better over time at playing chess, probably because there is some mistake in the function that calculates the error term or in the back-propagation algorithm.

References

[1] Li, Y. (2017). *Deep reinforcement learning: An overview.* arxiv. https://doi.org/10.48550/arXiv.1810.06339 (25. March 2022)

[2] Zhao, D., Wang, H., Shao, K., and Zhu, Y. (2016). Deep reinforcement learning with experience replay based on SARSA. *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, (pp. 1-6).

[3] Sutton, R. S., and Barto, A. G. (2014). *Reinforcement Learning: An Introduction* (2. ed). The MIT Press.

VI. Appendix

```
# As long as the game has not finished = for all moves of the game
while Done==0:
    allowed_indices,_=np.where(self.allowed_a==1)
    if A not in allowed_indices:
        break
    self.S_ap,self.X_ap,self.allowed_a,R,Done=env.OneStep(A)
    moves_per_game += 1

    if Done==0:
        # Choose A_ap from X_ap (-> S_ap written differently), but do not take the action
        # save the x2 from the old predict
        x2_old = self.x2
        q_s_ap_a_ap , A_ap = self.predict(self.X_ap, self.allowed_a, self.seeds[n])
    else:
      q_ap = 0
    # Update Q-Values
    reward_per_game += R
    # backpropagation
    # expected - actual
    # Compute the error signal -> this is probably wrong?
    e_n = self.eta*((R+q_s_a)-self.gamma*q_s_ap_a_ap)*x2_old
    self.backpropagation(e_n)
    if Done==0:
        #Update state and action
        self.S_ap,self.X,self.allowed_a,R,Done=env.OneStep(A_ap)
```

Fig. 8. The core loop of SARSA

```
while Done==0:
    # Choose A from initial state X (==S)
    q1,A = self.predict(self.X, self.allowed_a, self.seeds[n])
    # Take action A, observe R, S', X'
    self.S_ap,self.X_ap,self.allowed_a,R,Done=env.OneStep(A)
    # Collect rewards
    reward_per_game += R
    if Done==0:
        # go to S'
        _, A_ap = self.predict(self.X_ap, self.allowed_a, self.seeds[n])
    else:
      q_max = 0
    # find optimal Q(S_ap, a)
    q_max = np.max(self.h2)
    # Update Neural Net
    e_n = self.eta*((R+q1)-self.gamma*q_max)*self.x2
    self.backpropagation(e_n)
    # replace state
    self.X = self.X_ap
    moves_per_game += 1
```

Fig. 9. The core loop of Q-Learning

```
def ema(self, alpha, interval_size, time_values):
    current_value = time_values[interval_size]
    if interval_size <= 0:
        return current_value
    else:
        return alpha*current_value + (1-alpha)*self.ema(alpha, interval_size-1, time_values[:interval_size])

def exponential_moving_average(self, alpha, interval_size, list_to_smooth):
    # alpha between zero and one
    ema_list = []
    # first element gets not smoothed
    first_values_list = np.empty(interval_size)
    mean_for_first_values = np.mean(list_to_smooth[interval_size:])
    first_values_list.fill(mean_for_first_values)
    ema_list += list(first_values_list)

    for index, element in enumerate(list_to_smooth[interval_size:]):
        ema_list.append(self.ema(alpha, interval_size, ema_list[len(ema_list)-interval_size:]+[element]))
    return ema_list
```

Fig. 10. Recursive exponential moving average

```
def select_largest_index_from_allowed(self, allowed_a):
    allowed_numerical,_=np.where(allowed_a==1)
    max_value = 2.2250738585072014e-308
    max_index = np.random.permutation(allowed_numerical)[0]

    for q_value, is_allowed, index in zip(self.h2, allowed_a, range(0, len(self.h2))):
        if is_allowed[0] == 1:
            if q_value > max_value:
                max_value = q_value
                max_index = index

    return max_value, max_index
```

Fig. 11. Conditional softmax