# Introduction to Reinforcement Learning

*Institute for Informatics (IfI)*
*University of Zürich (UZH)*
Zürich, Switzerland (CH)

***Abstract*—This Paper shows the result of two reinforcement algorithms in the environment of a simple chess game. Both SARSA and Q-Learning were used to win the game and achieve checkmate. Furthermore, the paper discusses the differences between the two algorithms, analyzes the results and gives an in dept analysis of the Beta ($\beta$) and Gamma ($\gamma$) parameter.**

## I. Introduction

This paper is structured as follows. It presents the basic intuition of Q-Learning and SARSA first. In a second step it presents the neural network, which was used in the implementation reinforcement algorithms in this paper. Then it goes into more detail, by talking about the implementation of Q-Learning. It then presents the Double Q-Learning algorithm and shows different outcomes for different hyperparameters. The last section then shows the SARSA algorithm and draws a conclusion. The underlying code for this paper can be found on Github and Gitlab.

## II. Methodology

In this section we have a closer look at the SARSA and Q-Learning Reinforcement algorithms.

The pseudo code used in this paper uses the same notation as the code in the Jupyter Notebook. The Q-values are calculated using a Neural Network instead of a table. Since the Neural Network only takes the current state ($X$) as input and produces the Q-values for all possible actions as a vector output, the Q-values are displayed as $Q\text{-}values(X)$ instead of $Q\text{-}value(X, A)$. When it is unclear to which action a $Q\text{-}value$ is referring to, we state it explicitly (e.g. $max(Q\text{-}value)$ or $Q - value[A\text{-}next]$). This represents a zero vector with the $Q\text{-}value$ at the place of the action.

### A. Q-Learning

**Algorithm 1** Q-Learning

```
1: $Q\text{-}values \leftarrow Initialize()$
2: for Episode in [Episodes] do
3:     $X \leftarrow Env.Initialize$
4:     while Game not finished do
5:         $A \leftarrow policy(Q\text{-}values(X))$
6:         $R, X\text{-}next \leftarrow Env.step(A)$
7:         $Q\text{-}values(X) \leftarrow Q\text{-}values(X) + \eta \cdot [R + \gamma \cdot \max Q\text{-}values(X\text{-}next) - Q\text{-}values(X)]$
8:         $X \leftarrow X\text{-}next$
```

In a first step we initialize the Q-values. Since we use a Deep Neural Network, this is done by initializing the weights of the network. Then we start the learning process by playing the game of chess multiple times (episodes). For each game played we use the Q-values to determine the action for a certain state S of the game (In the code S is represented by the vector X). With the action one can determine the action to the next state and collect a reward. In case that the game is not finished the Q-values for the next state ($S\text{-}next$) are calculated by using the action which returns the highest possible Q-value (Line 7 in pseudo code). This is one difference to other Reinforcement algorithms, since this Q-value might be different from the policy's Q-value for $S\text{-}next$. To summarize, Q-Learning is an off-policy algorithm, meaning it takes the best possible option for the next state to update [1]. The algorithm then updates the neural network (Q-values) by using back-propagation (II-D). After the back-propagation, the $S\text{-}next$ state becomes the current state $S$.

If there is no next state due to the fact, that the game has finished, the same update rule applies. In our notation, the $\max Q\text{-}values(X\text{-}next)$ position would be a one-hot-encoded vector with the Q-value at the position of the action.

### B. SARSA

SARSA is the acronym for State, Action, Reward, State (next State), Action (next Action) [2]. This hints towards, the fact that SARSA uses the action determined by the policy to update the neural network. The pseudocode in 2 gives a more detailed view.

**Algorithm 2** SARSA

```
1: $Q\text{-}values \leftarrow Initialize()$
2: for Episode in [Episodes] do
3:     $X \leftarrow Env.Initialize$
4:     while Game not finished do
5:         $A \leftarrow policy(Q\text{-}values(X))$
6:         $R, X\text{-}next \leftarrow Env.step(A)$
7:         $A\text{-}next \leftarrow policy(Q\text{-}values(X\text{-}next))$
8:         $Q\text{-}values(X) \leftarrow Q\text{-}values(X) + \eta \cdot [R + \gamma \cdot Q\text{-}values(X\text{-}next)[A\text{-}next] - Q\text{-}values(X)]$
9:         $X \leftarrow X\text{-}next$
10:        $A \leftarrow A\text{-}next$
```

As before, we initialize the Q-values and start the episode loop. In each episode (game) we start with a fresh game and therefore need to initialize the game. While the game is not finished, we do the following weight update process. First we calculate an action according to policy. Then we use this action to change the environment and get a reward and the next state. With this next state ($X$-$next$) we can calculate the next action ($A$-$next$) according to our policy. This is a difference to Q-learning, where we always used the action with the highest Q-value, respectively a greedy action. With the $A$-$next$ we can then determine the $Q$-$values(X$-$next)$ of $A$-$next$ and use it in the update formula.

### C. Differences SARSA vs. Q-Learning

Some differences were already mentioned. SARSA for example is an on-policy algorithm, meaning it uses the policy to determine $A$-$next$ which then leads to $Q$-$values(X$-$next)[A$-$next]$. Q-learning on the other hand uses $\max Q$-$values(X$-$next)$. Hence the Q-value does not depend on the policy. Consequently, Q-learning is an off-policy algorithm. If SARSA uses a greedy only policy, SARSA and Q-Learning are the same [1].

Due to this max-function Q-learning is more aggressive and takes more risks when learning and should result in a more optimal result, while SARSA prefers less optimal solutions with less risks [1]. In the long run, the differences become minor since epsilon is decaying to 0.

Furthermore, it is important to notice that off-policy algorithms, such as Q-Learning can suffer from certain instabilities.

Both methods use many training examples to converge to a optimal solution in complex environments with many actions and states. In case a table approach is used, meaning, for each state all actions are stored in a "memory" table, memory overload can occur. To mitigate this problem, developers can use Neural Networks to match states to actions.

To reiterate, if an optimal solutions is preferred then Q-Learning is the way to go. In case of costly failure, SARSA should be chosen since it gives a less optimal but safer solution.

### D. Neural Network

For all training runs, the same neural network was chosen. This section will briefly talk about it's functionality and design choices. The graph below shows our two layer neural network (two weight matrices).

The inputs consist of 58 states + 1 additional position for the bias. The output of the neural network are the Q-values for the respective 32 actions. Before, the state is passed through the $forward$ function one uses the $add\_column\_for\_bias$ function to add the numeric value 1 to the state vector. This will lead to a weight (bias) being added to each hidden neuron. After the first matrix multiplication the bias position is then again manually set to 1 so that another bias can be added to the output neurons.

In the $backward$ function we back-propagate the loss and update the weights according to the gradient. The overall update formula is as following:

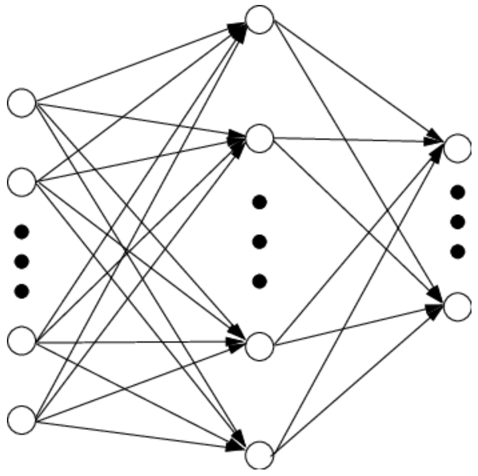


Fig. 1: Two layer Neural Net based on [3]

$$\theta^{new} = \theta^{old} + \eta * [R + \gamma * \max_{a'}(Q\text{-}values(X\text{-}next, a'; \theta^{old}) - Q\text{-}values(X, a; \theta^{old})] \nabla_\theta Q\text{-}values(X, a, \theta^{old})$$

Fig. 2: Reshaped update function based on [4]

Therefore, we expand the reward vector to the size of the output layer. Then compute the delta depending on whether there is a next state or not. Once the delta vector is calculated we need to select the delta at the right index, since we only want to update the weights of a specific action. The one-hot-encoded vector of that action (1 at the place of the action) is provided to the function. Therefore, we multiply the one-hot-vector with the delta to get the final term to back-propagate. Finally we update the weights accordingly in the $update\_weights$ function. These principles are explained in more detail in VIII.

## III. Q-Learning Implementation

The Q-Learning algorithm follows the pseudo code structure from II-A. The following pre-set parameters were used:

TABLE I: Parameter Settings

| **Parameter Name** | ***Value*** |
|---|---|
| Nr. Hidden Neurons | 200 |
| Nr. Episodes | 30'000 |
| Gamma ($\gamma$) | 0.85 |
| Eta ($\eta$) | 0.0035 |
| Beta ($\beta$) | 0.00005 |
| Epsilon ($\epsilon$) | 0.2 |

The Q-learning code was run multiple times to see the effects of different seeds. In the results one can very clearly see that Q-learning sometimes suffers from unstable behavior. The reward rose in the beginning but in some cases it dropped and oscillated around a certain reward level.

Figure 3 shows, that the reward over the episodes (smoothed with an exponential moving average) continuously rises. We see that the reward rises pretty quickly in the beginning and

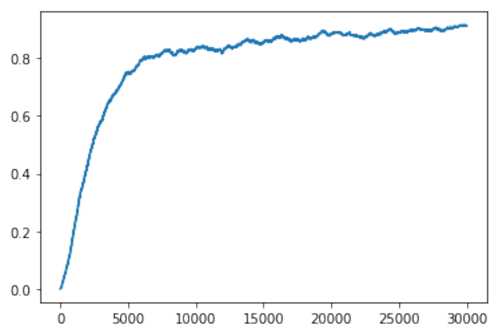


Fig. 3: Reward Q-Learning with parameters from I

then slows its ascend. Overall, Q-Learning achieves a max of 91% reward during this learning phase. In the illustration 4 we see the steps over the time of training. In the beginning the model was lucky and achieved a low step count. Then it rose pretty fast since the model hasn't learned much yet. With continuing training the action count drops to 12 and starts to oscillate around that value. A reason for such a high step count could be the instabilities of Q-Learning as well as that the minimization of steps is not a direct part of the optimization formula. Nevertheless, Q-learning achieves a high reward.

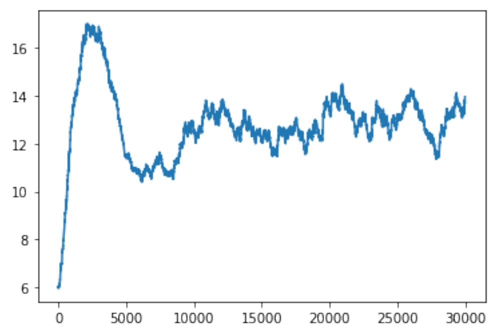
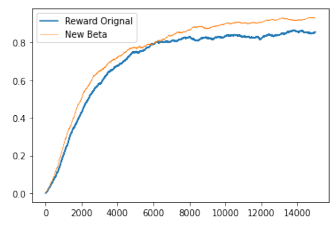


Fig. 4: Steps Q-Learning with parameters from I

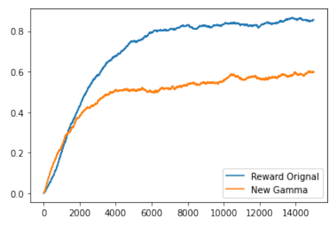
## A. *Parameter change*

The Gamma parameter ($\gamma$) is responsible for discounting future Q-values during the update process. It's value ranges from 0 to 1. The closer to 1 the less the future Q-values are discounted. In Graph 5b we compare the reward of QLearning algorithm with a new Gamma of 0.03 to the output in 3 with a Gamma of 0.85. This means the future Q-values are discounted more, hence the model updates slower. Very interesting is the result in 5d where we see the optimal drop of number of steps.

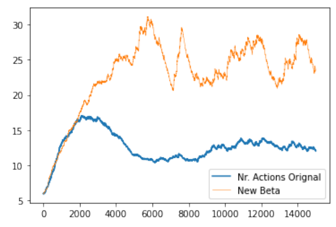
The Beta parameter ($\beta$) decays the Epsilon ($\epsilon$). This means the more episodes go by, the more epsilon decays, which means the algorithm starts exploiting faster. Since we tested the change on Q-Learning, Beta only has an impact on line 5 of II-A. In 5a we see that the reward increases faster, due to the increase in the Beta. We can therefore state that a higher Beta leads to more greedy behavior, which finally results in slightly higher rewards. It is also interesting to see in 5c that



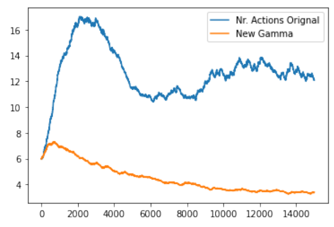
(a) Reward: beta = 0.0005



(b) Reward: Gamma = 0.03



(c) Nr. Actions: Beta = 0.0005



(d) Nr. Actions: Gamma = 0.003

Fig. 5: Comparison of original Reward and Step size to model output with adapted parameters

the number of steps is on average higher and oscillates more than previously.

## IV. Double Q-Learning Implementation

The results of the Q-learning implementation in III were a little disappointing due to the fact that the step count did not declined as much as in SARSA (V). Therefore I implemented another Q-Learning algorithm: Double Q-learning (3). Double Q-Learning is also an off-policy algorithm, but it uses two networks to get the max action. This helps avoid overestimation from which the original algorithm suffers [5].

**Algorithm 3** Double Q-Learning based on [5]

```
1:  $Q\text{-}values\text{-}A \leftarrow Initialize()$
2:  $Q\text{-}values\text{-}B \leftarrow Initialize()$
3:  for Episode in [Episodes] do
4:      $X \leftarrow Env.Initialize$
5:      while Game not finished do
6:          $A \leftarrow policy(Q\text{-}values\text{-}A(X) and Q\text{-}values\text{-}B(X))$
7:          $R, X\text{-}next \leftarrow Env.step(A)$
8:          $Randomly update A or B$
9:          if $Update A$ then
10:             $Action\text{-}A \leftarrow argmax(Q\text{-}values\text{-}A(X\text{-}next))$
11:             $Q\text{-}values\text{-}A(X) \leftarrow Q\text{-}values\text{-}A(X) + \eta \cdot [R + \gamma \cdot Q\text{-}values\text{-}B(X\text{-}next)[Action\text{-}A] - Q\text{-}values\text{-}A(X)]$
12:         else
13:             $Action\text{-}B \leftarrow argmax(Q\text{-}values\text{-}B(X\text{-}next))$
14:             $Q\text{-}values\text{-}B(X) \leftarrow Q\text{-}values\text{-}B(X) + \eta \cdot [R + \gamma \cdot Q\text{-}values\text{-}A(X\text{-}next)[Action\text{-}B] - Q\text{-}values\text{-}B(X)]$
15:         $X \leftarrow X\text{-}next$
```

When one looks at the same resulting graphs, we can state that double Q-learning (6) is not able to grow its reward as fast as the simple Q-learning algorithm. This might be due to the two networks, which makes training slower. But we can see that even after the initial reward burst the reward continues to grow in a steady pace, whereas in Q-Learning the later improvements are minor 3.

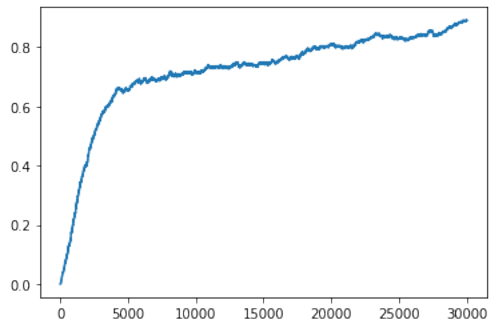


Fig. 6: Reward Double Q-Learning with parameters from I

With double Q-learning we can also observe a steep decline in steps in comparison to the algorithm before. Therefore, we achieved a better step count with the DQ-algorithm over 30’000 Episodes.

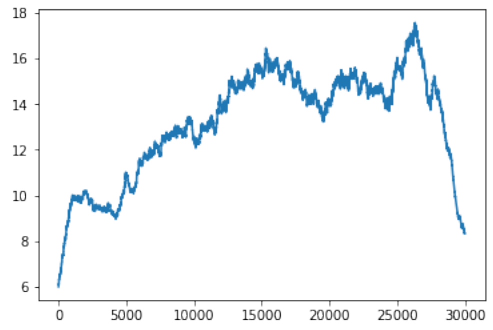


Fig. 7: Steps Double Q-Learning with parameters from I

## V. SARSA Implementation

The second algorithm implementation of the task is SARSA. The implementation follows the pseudo code outline in 2 and uses the parameters defined in I. We can observe in 8 that SARSA achieves a very high reward at around 96%. Furthermore, we can see that the step counter in the beginning rises due to the fact the initialization got lucky. After the initial rise we can observe the expected behavior. The number of steps declines gracefully over the episodes and settles on the low level of around 2 steps on average.

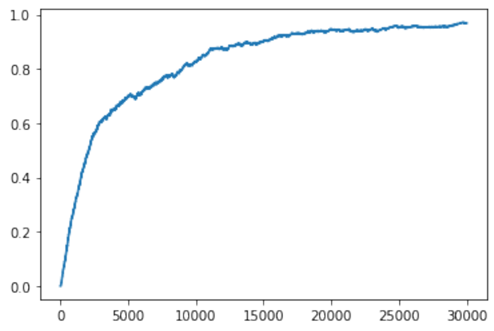


Fig. 8: Reward SARSA with parameters from I

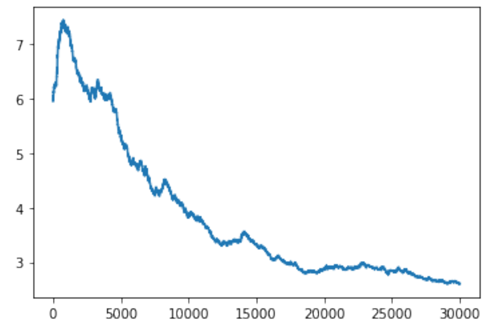


Fig. 9: Steps SARSA with parameters from I

If we compare the results with the previous graphs of the Q-learning algorithm we can state the following points. Both achieve an equally high reward. Overall, the reward graphs 3 and 8 look very similar. This makes sense, since SARSA becomes more and more greedy over time due to the epsilon decay. Therefore, it behaves similar to the Q-Learning algorithm in later stages. But one can recognize a different effect in the number of steps. Both algorithms rise in the beginning due to optimal weight initialization (same initialization) but the step count of SARSA decreases continuously while Q-Learning seems to oscillate around a certain value.

## VI. Conclusion

Both, SARSA and Q-Learning achieve high rewards in the chess environment. Additionally, Double Q-Learning also delivers very good results. Of all the algorithms, only SARSA reduces the step counter significantly. Q-Learning and Double Q-Learning show this behavior only for specific seeds or as we have seen with different parameters (Gamma).

## VII. Reproducibility

The code is available on Github and Gitlab. I recommend using Gitlab, since the output of all executed fields is shown.

To reproduce the results, one has to clone the repository and execute the main file (Assignment_Final_Code). Thanks to the seed, the results (graphs) will be the same.

## References

[1] Viet Hong Tran Duon, “Intro to reinforcement learning: temporal difference learning, SARSA vs. Q-learning,” Medium Feb. 2021, Visited: Mar 2022, www.IntroToReinforcementLearning.com

[2] Adesh Gautam, “Introduction to Reinforcement Learning (Coding SARSA) — Part 4,” Medium Jul. 2018, Visited: Mar 2022, www.CodingSarsa.com

[3] Tzafestas Spyros, Konstantinos Blekas, “Hybrid Soft Computing Systems: A Critical Survey with Engineering Applications,” ResearchGate Jan. 2000, Visited: Mar 2022, www.ResearchGate.net

[4] Lawrence Carin, David Carlson, Timothy Dunn, Kevin Liang, “Introduction to Machine Learning,” Duke University, Visited: Mar 2022, www.DukeUniversityCoursera.net

[5] Hado van Hasselt, “Double Q-learning ,” Multi-agent and Adaptive Computation Group Centrum Wiskunde & Informatica, 2010, www.DoubleQLearningPaper.com

## VIII. Appendix

### A. Neural Network Dimensions

The neural network follows the annotation/methodology of the Deep Learning Lecture by Manuel Günther (UZH). Therefore, we use an additional neuron in both input and hidden layer with the value one. The dimensions become clear in the initialization 1. To add this additional Neuron we have to modify the input (X) to the neural network as well as the hidden layer. 2 shows the modification of X in the *add_column_for_bias* function. 3 on the other hand shows how to change the Neuron in the hidden layer to 1.

```
    import numpy as np

     # initialize the weights
    self.W1 = np.random.randn(size_hidden_layer
+1, input_layer_size + 1) * np.sqrt(1 / (
input_layer_size+1))
    self.W2 = np.random.randn(output_layer_size,
 size_hidden_layer+1) * np.sqrt(1 / (
size_hidden_layer+1))
```

Listing 1: Initialization of layers

```
    # in the function: add_column_for_bias
    # since the input to the NN is (X.shape + 1
(for bias)) we need to add 1
    X = np.vstack((np.array([[1]]), X))
```

Listing 2: Add Neuron to input

```
    # set first row to 1 (bias is in the weight
matrix)
    h1[0,:] = 1
```

Listing 3: Add Neuron to hidden layer

### B. Back-propagation

Another function which might be complicated to grasp is the back-propagation function. In the first step we create a column vector which has the same reward at each place of the vector

```
    # create a vector of reward ()
    reward_vector = np.tile(reward, (self.
output_layer_size, 1))
```

Listing 4: Create Reward Vector

Depending on whether a next state exists or not we calculate the delta according to the pseudo code update rule.

```
    # check if there is Q_value for the next
step does exist (Game is not over yet)
    # Q_next is calculated according to the
learning algo:
    # Q-Learning: Q_next(X) = max(Q-values of
the next state (X))
    # SARSA:  Q_next(X) = policy(Q-values of the
 next state (X), epsilon)
    if Q_next is not None:

        # formula from the slides
        delta = reward_vector + self.gamma*
Q_next - Qvalues

    # if the game is over there is no Q_next
    else:
        delta = reward_vector - Qvalues
```

Listing 5: Calculate Delta

Since we only want to update one specific action (the action chosen by the policy at state X) we need to multiply the delta vector with a one-hot-encoded vector to get the delta value only at the index of the specific action.

```
    # only calculate the gradient for the action
 which was taken
    delta = delta*choosen_action
```

Listing 6: Only update the chosen action

The next step includes calculating the gradient through back-propagation to then update the weights according to the formula in 2.

```
    # gradient of loss W2
    grad_W2 = np.dot(grad_a2, self.h1.T)

    # gradient of loss W1
    grad_W1 = np.dot((np.dot(self.W2.T, grad_a2)
 * self.heaviside(self.a1) ), x.T)

    # update the weights
    self.update_weights(grad_W1, grad_W2)
```

Listing 7: Compute Gradients and update weights

### C. Double Q-Learning

Since the training loops follow the guidelines of the assignment and are well explained by the pseudo code abov,e they are not explained in detail in this section. Nevertheless, the code of the double Q-Learning algorithm will be presented in 8.

```
     # Update(A) or Update (B)
        if np.random.rand() < update_A:
            # we update Q_values of A

            # calculate the next Q_values for B
            #  QA(X_next, a = parameter)
            Q_next_A = neural_network_A.forward(
X_next, True)

            # arg max on action -> set epsilon
to 0 -> therefore it is greedy and we get max
value
            # then we can extract
            one_hot_action_star, action_star =
policy.get_choosen_action(Q_next_A, allowed_a,
0)

            Q_next_B_action_star =
neural_network_B.forward(X_next, True)[
action_star]

            # update the Q values of A with the
Q_values of B
            neural_network_A.backward(R,
Q_values_A, Q_next_B_action_star, X,
one_hot_action)
```

```

            else:
                # we update Q_values of B....

```

Listing 8: Compute Gradients and update weights

In the beginning we randomly decide whether to update network A or B. Next we calculate the next Q-values according to network A. Then we need to get the index of the action which would return the highest Q-value for the network A. We do this by setting epsilon to 0 in the $get\_choosen\_action$ function. We can then update the neural network according to the update rule in 3