Tabular Q-Learning 3x3 2048

We applied a Q-learning algorithm to the 3x3-reduced form of the game 2048. We followed the implementation for Georg Wiese (https://github.com/georgwiese/2048-rl) to program the game logic. For the implementation of the Q-learning algorithm we followed the tutorial by Arthur Juliani (https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0).

The game 2048

For implementing the tabular Q-learning algorithm we reduced the original 4x4 form of the game to a 3x3 form with the purpose to reduce the state space size. While the number of actions (right, left, up, down) is fixed, the number of possible states raises to a combination of the number of tiles (4x4 or 3x3) to the power of $log2\,(n) + 1$ where n is the highest possible tile value. So in the original game, assuming that the game ends with 2048, there are potentially 16^12 or approx. 18’000 billions possible states. We thus limited the playground to a 3x3 matrix and the highest tile to 64, resulting in approx. 500’000 possible states. We set the reward to be provided when a certain tile/number (here 64) is achieved. When the agent achieves this tile, he receives a reward of +100 and the game is over. For every move the agent receives a punishment of -1. Everything else is as in the original form of the game.

Tabular Q-Learning

We implemented a tabular Q-learning algorithm. In this case, Q-Learning is a table for every state (row) and action (column) possible. Within each cell we learn a value for how good it is to take an action within a given state. In our 3x3 game we have a state space of approx. 500’000 and 4 actions. This gives us a 9^6x4 table of Q-values. At the beginning, all values equal zero. Then, we observe rewards in every episode and update the Q-values in the table. We update the Q-Values according to following code:

Q(s,a)= Q(s,a) + learning Rate * (r + y (max Q(s’,a’) - Q(s,a))

Where s=state, a=action, r=reward, y=discount factor, maxQ(s’,a’)= maximal expected future reward for the next state.

We defined a discount factor of 0.95 and a learning rate of 0.8. A low discount rate (0.05) makes sense, because the nature of the game includes no discounting since the reward falls for the final state. The learning rate defines how much we update based on what we have last seen. We chose a rather high learning rate (=0.8), because new learning should have a rather high impact on older state values.
For indexing the states of the 3x3 game we proceeded as follow: Each cell of the game can achieve a theoretical maximum of 64 (=2^6) and a minimum of 0 (empty cell). We took the

logarithm on the base 2 of each field with a value or zero for empty cells to get a value between 0 and 6 for each cell. Then we summed up the values of the cells and multiplied each field with the factor 10^x (where x indicates the location of the cell, top left = 8, bottom right = 0). Here an example:

2	8	32
64	2	0
0	4	32

Calculating the logarithm on the base 2 we get:

1	3	5
6	1	0
0	2	5

Then we sum up the values as follows: 1*10^8 + 3*10^7 +5*10^6 + 6*10^5 + 1*10^4 + 0*10^3 + 0*10^2 + 2*10^1 + 5*10^0 = 135’610’025 and get the following state index of the example above.
For constructing the Q-table, each state (each state index as described above) gets assigned 4 actions (Up, Down, Left, Right). It comes as little surprise, that the Q-table is huge and requires billions of episodes and big computing power to achieve significant results.

Results

The goal of our project is to implement reinforcement learning algorithms on the game 2048. We decided to compare the performance of tabular Q-learning and deep Q-learning. From the beginning, we were conscious of the fact that tabular Q-learning the game 2048 creates a huge state space. For the purpose of reducing this state space we implemented a 3x3 format of the game and decreased the goal of the game to a lower tile.
We began by defining the goal of the game in achieving the tile 256. We did not find any significant result. Most probably, because the tabular Q-learning requires too many episodes (billions of episodes to reach every possible state at least one time). Running our code for 3 days did not help. We tried to achieve some learning results by further decreasing the goal of the game (down until 64) but still did not get any results. We still cannot distinguish the algorithms’ performance from randomness.
Summarizing, the tabular Q-learning requires too much computing power for the game 2048 (even in a strongly reduced form). Implementing the Deep Q-learning algorithm to the game appears more promising.

The trained algorithm has been saved in the file “q_values_20170110.dat”. Because of the size of the resulting Q-Table, the file size is approx. 32GB and we were not able to submit it via OLAT. Since the results do not show anything special to see, transferring this file to you might not be necessary. Nevertheless, should it be relevant for you, please contact Florian to transfer it to you via cloud. Once you have the file, please read the instruction on the line 57-58 in the file “qLearning.py” to run the game taking these results into account.