# Tic tac toe

David Tonderski (davton)

To shorten training, a hash function was created as follows:

$$H(S) = 1 + \sum_{i=1}^{i=9} (S(10-i)+1) \cdot 3^{i-1}, \tag{1}$$

where $S(j)$ is the state of the $j$-th field of board $S$, where an "X" is represented by a "1", an "O" by a "-1", and an empty field by a "0". We have $H([-1,-1,-1,-1,-1,-1,-1,,-1,-1]) = 1, H([-1,-1,-1,-1,-1,-1,-1,-1,0]) = 2, ..., H([1,1,1,1,1,1,1,1,1]) = 19683$. Then, the Q-table was initialized as a 2 by 19683 matrix (there are 19683 possible tic tac toe board states), where each field was a struct with the field 'entry' containing an empty matrix. Whenever a state was encountered, the program checked whether the $H$-th column of the Q-table consisted of structs containing empty matrices. If it did, the program initialized those matrices, where row 1 represented the board state, and row 2 represented the Q values. If the structs didn't contain empty matrices, the program had already encountered those board states before, and the existing Q-values were used. Then, after each iteration, the Q-values were updated according to the rules described in the lecture notes.

The upside of this approach is that the training is much faster, as the program doesn't have to loop through the Q-table to find the existing entry, but instead knows almost instantly where to look. However, the program is less memory-efficient (as the Q-table must be initialized to be able to contain every possible value), and less intuitive. The program was run for 1e5 iterations, as visualised in figure 1. In the beginning, $\epsilon$ was set to 1 (completely random play), but after every $1e4$ iterations, $\epsilon$ was reduced by a factor 0.9 ($\epsilon \leftarrow \epsilon/10$).

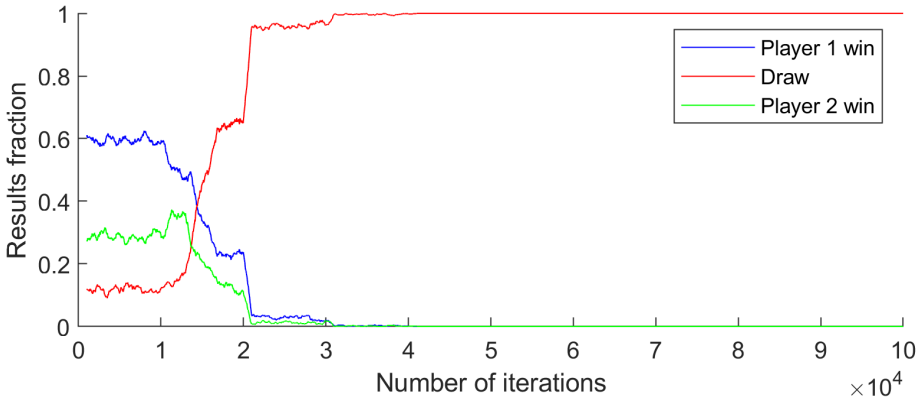


Figure 1: Game results as a function of the number of training iterations.