TICTACTOE AI

TicTacToe AI Documentation

Name: Lukas Zürcher  
Klasse: M4C

Inhaltsverzeichnis

[In the beginning 2](#_Toc168784297)

[Research 2](#_Toc168784298)

[Theory 2](#_Toc168784299)

[My research 5](#_Toc168784300)

[Attempts 5](#_Toc168784301)

[The first attempt 5](#_Toc168784302)

[The new attempt 8](#_Toc168784303)

[Reflection 10](#_Toc168784304)

[Results and Interpretation 10](#_Toc168784305)

[Self-reflection and what to do better next time 13](#_Toc168784306)

[Sources 14](#_Toc168784307)

# In the beginning

As we received the assignment, I began thinking about possible answers to the question: How can we train a Neural Network to learn a deterministic game like tic-tac-toe? While pondering this, I recalled some of the research I conducted for my matura project. I also consulted ChatGPT, as I had an idea in mind and wanted to explore other possible solutions. On one hand, I knew that the idea was to write an algorithm that estimates good moves, take another automated approach to create the dataset, or even find one online and adjust it so our programs could read it. On the other hand, I knew that reinforcement learning could do the job, even if it seemed a bit overkill. After asking ChatGPT and receiving the term "q-learning," I became curious and started researching that topic. My findings were interesting enough that I decided to take this approach, partially because I had tried it before and given up, and partially because it seemed to be a very interesting topic. As I read through blog posts and articles, looked at some GitHub code, and read some forums, I concluded that this would be my approach.

# Research

## Theory

Q-learning is used in reinforcement learning and is a very interesting concept. The point of reinforcement learning is to get an AI to learn based on experiences rather than a predefined dataset. The advantages are that the AI learns in a more human-like manner and doesn’t need a fully labeled dataset. The disadvantages, in turn, are that it is more susceptible to errors, and training takes longer and is more difficult since the data used for training partly comes from the AI itself. This also makes it hard to identify where the flaws come from.

Q-learning, in particular, needs a few components: a way to select actions, a way to observe the influence of these actions, and a function to get the Q-values. The selection of actions is usually done via epsilon-greedy, which is just a decreasing chance of using random actions. For example, at the start, there may be 100% random actions, and after a number of iterations, they are 70% random, with the other 30% based on the prediction of the model. This model prediction consists of the Q-values for every action. Assuming we also consider invalid actions, a Tic-tac-toe board always has 9 actions. To make a prediction, a model needs an input, which in this case is a representation of the board state.

Now, how the learning works is where things get interesting. The function for Q-learning is:Ein Bild, das Text, Schrift, Reihe, weiß enthält.

Automatisch generierte Beschreibung

The Q-value for the current state is the Q-value that the model predicts, plus the reward, plus the maximum possible reward for the chosen action. This function is based on the Bellman equation, which is the part in brackets at the end. Therefore, this function updates the Q-value for a given state-action pair by adding the reward plus the maximum obtainable reward. The gamma is the discount factor, and it defines how much emphasis there should be on future rewards. The reward is just the reward for the action, and if the action is terminal, the Q-value is usually just set to the observed reward. In this function, the weighted reward is taken, but I have also seen programs where only the Bellman part of the equation is used.

To get the Q-values to converge, one tries to minimize the Temporal Difference error (short TD error), which is the reward plus gamma times the estimate minus the actual Q-values. Double Q-learning works similarly but with one difference: there are two models trained on the same problem that help update each other. The function for Q-learning uses the maximum of the estimated future values, which can be problematic due to overestimation. In the YouTube video I watched, it was explained like this (adjusted to the metric system):

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungIf we measure 100 people who weigh 100 kg ± 1 kg, we would statistically get 50 people over 100 kg and 50 people under 100 kg. The specifics don’t matter much, but what matters is that the maximum value will almost certainly be over 100, and if we estimate 100 people, the chances of getting the true value of 100 kg are very low; instead, it will be maybe 100.7 kg, which is an overestimate. The second network is there to counter this. If the highest weight of 100 people is 100.9 kg and it’s the 40th person, we take that 40 and use the second network to get the weight of the 40th person in this different set. Now there is no overestimate because the person in the second group is as likely to be over 100 kg as below 100 kg. Hasselt wrote the Double Q-learning algorithm like this: In Double Q-learning, we have two networks that are used to update each other with a function similar to the one used for single Q-learning. Here, one network gets updated with the TD error, but instead of using the maximum estimate of the model, we use the value of the maximum index of the other model. Another important aspect to mention is that the estimates for the next states are usually done by a target model. The target model gets its weights updated to the main model weights every n iterations, which makes training more stable. If the target model and the main model predict the exact same thing, the Q-values have most likely converged and will change no more, making this a good measurement for convergence.

Up until now, I have only talked about DQN and Double DQN. Before this, there is also the ability to do Q-learning with a table. Instead of involving target models and weight updating, it is also possible to do Q-learning with a table that has rows for each state and columns for each action that can be taken. This approach can be analyzed very easily and also gives quite good results, sometimes even better than those of neural networks if enough data is present.

The problem is that for a Tic-tac-toe game, this table won’t get too large since there are only 3^9 possible board states. There are 9 possible cells with 3 different values: either player 1, player 2, or empty. For each field, there can be an action for a total of 9, so if every board is played with all cells filled, even if it terminates after 5 moves, there is a maximum of 3^9 rows and 9 columns. This totals 3^9 \* 9 cells in our table, which is 59,049 cells. If we use half-precision (float16), which takes 16 bits or 2 bytes per cell, we get a table size of 118,098 bytes or 0.118 Megabytes, which is negligible for modern computers. Even with duplicate data like 1,000 times, the file would still be pretty small.

The problem is such a table won’t work for problems only slightly bigger. If we consider the extremes of chess, for example, and consider illegal moves, we get a number of cells so big it wouldn’t be possible to store in our universe. Even with only legal moves, it would still be way too big. Neural networks counter this by not needing so much space since they learn a policy from data and not a table. They are much more flexible and dynamic, able to react to situations they have never seen before.

## My research

To write my program, I read many blog posts, questions, and articles, as well as the papers by Hasselt. I compiled everything into sources but did not include references in my text because I can't remember exactly where each piece of knowledge came from, as the research happened both during and before the project.

My biggest problem was, and still is, that I never found a code example that I could fully understand. I read code for the OpenAI Gym environment since it is well documented, as well as code from different blog authors, but I found many variations and couldn’t quite combine them effectively. Despite extensive research, I am still making a mistake that I can't identify.

# Attempts

## The first attempt

I first got to work by writing a TicTacToe class that would handle the game. All of this takes place in the main branch of my GitHub repository. Firstly, this game engine should hold a representation of the game board. Secondly, it should also contain various methods to interact with the board. There needs to be a function for performing an action; the OpenAI Gym environments call this function step().

The next two methods are responsible for returning the action and observation spaces. The action space should contain all possible (valid or invalid) actions, and the observation space should return a board representation. In my code, the methods that start with get\_ return parts of these spaces. The get\_all\_actions() and get\_invalid\_actions() functions provide a way to get the action space, while the other get\_ functions are responsible for accessing the observation space in my code. The check\_board\_condition() method is there to determine if someone has won the game, if it’s a draw, or if it should continue.

Now that the game engine can perform actions, provide an action space, and an observation space, it also needs a reset method to start the game anew. With this, the TicTacToe class is complete and can be used in the Networks.py script.

The Networks.py script contains two classes: the QNetwork class and the Training class. Despite the convention of one file containing one class, I didn’t change this because I made an error when I tried to move the Training class to train.py. The QNetwork class is responsible for creating the Q-learning agent, and the Training class handles the training process. The constructor for the QNetwork class is rather large due to the many variables that accumulated over the course of writing the script. All values for the network parameters are saved in a separate variables.py script so I could later save and load different training configurations. I also tried saving the parameters in a JSON file, but I had some trouble with the saving and loading function, so I decided to use a Python file to avoid wasting more time on this issue.

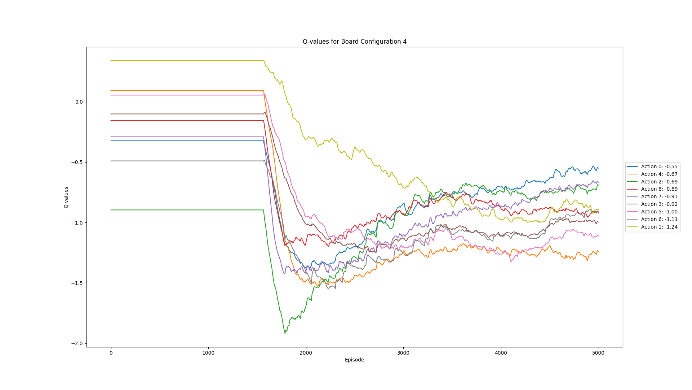
The model is created in the create\_model() method, which handles the configuration of the model according to the parameters. I went with a sequential model and dense layers together with an input layer. The ReLU activation function seemed to be very popular for reinforcement learning, and the mean squared error is also widely used. Both ChatGPT and some GitHub repositories I looked at during my research used them. I also chose the Adam optimizer since it is well-regarded for such tasks.

The get\_qs() method returns the predicted Q-values for a given state, and the choose\_action() method chooses an action to perform using epsilon-greedy. The two update\_ methods are responsible for updating epsilon according to the decay factor and adding a new transition to the replay memory. The update\_replay\_memory() method interacts with the game engine to get the current state, select an action, perform the action, get the new board state, and then check the outcome. After all is done, a transition is saved to the replay memory of the agent. A transition contains the observation space before an action, the observation space after the action, the action itself, the immediate reward of that action, and whether the action led to a terminal state.

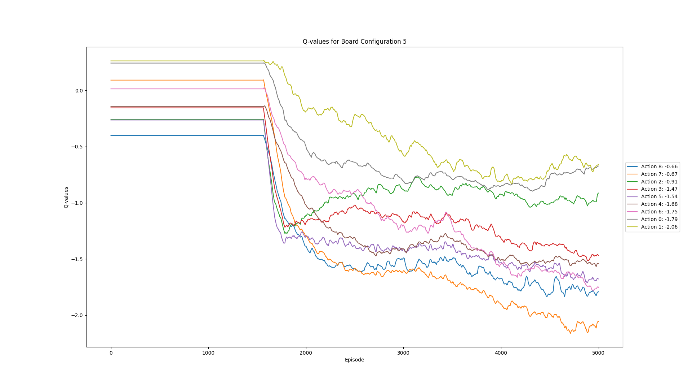
Moving on to the Training class, it is responsible for handling the training process, and its constructor holds the instances of both player agents and the environment, along with the length of the training. The train\_on\_batch() method only runs after the specified minimal memory size is reached to prevent errors from having a batch size larger than the available data and to ensure a diverse set of experiences to start the training process. The method takes a random sample from the replay memory and extracts all values from it. I used a for loop to go over all the transitions in a batch, but the reference I used for many parts of that script (saved as reference2.py in earlier commits on the main branch of my repository) used enumerate to go over the batch, so I had to adapt some of my code because I don’t fully understand how enumerate works.

Arguably, the most important part of this method, or even the entire script, is the Q update rule. I tried many variations since I never found an exact reference for training a DQN (Deep Q Network) for this specific game, but in the end, I used the one from Wikipedia. The last if-else statement at the end of the code updates the target model with the weights of the actual model.

The plot\_metrics() method comes from ChatGPT, and the run\_training() method is responsible for handling the training over many episodes. There are some test boards to monitor the values over time for the same boards, and inside the for loop, the training process happens. I also added tqdm to get a progress bar to see my training speed without setting verbose to 1 during model fitting. Inside that for loop, the test boards get evaluated each episode, and the while loop generates a new game each episode to add to the replay memory. After the while loop, the train\_on\_batch() method gets called on both agents and trains one batch of experience, and then the Q-values for the test boards get visualized with matplotlib. This visualization part also comes from ChatGPT, but the colors of the graphs it generates don’t match the colors of the values depicted in the legend, and I never found out why. This first attempt didn’t work as I expected because the results were unstable. If we look at the graphs for the test boards, we can see that the values fluctuate a lot.

Ein Bild, das Text, Diagramm, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Diagramm, Reihe, Karte enthält.

Automatisch generierte Beschreibung

First, the values are linear since there is no training because the minimal memory size isn’t reached. When the memory size is reached, the changes are drastic, and when they become slower, they are unstable. One thing to note is that the value order at the end shows some similarities between the boards, such as 0 being the best move in 3 out of these 4 example boards. The predicted values aren’t what they should be, and when I looked at the curves, it was evident that something was going wrong.

After conducting more tests, I came up with the final results that are now in the repository, but they are far from satisfactory. Consequently, I started another attempt to improve the values and make them actually useful.

## The new attempt

For this new attempt, I researched methods to make my training more stable and obtain useful values. The second paper Hasselt published about Q-learning seemed like the way to go for me. I set up a new branch in my repository and started anew. I copied parts of the existing code and wrote new code to accommodate my experiences and new research. I went above and beyond to write code, extending my script significantly. I generated many more tests along the way, adjusted my code multiple times, added new ways to extract data from my trainings, and experimented with different parameters.

The file structure looks the same as with my first attempt. The necessary files are `environment.py`, `variables.py`, and `DoubleNetworks.py`. The biggest change is that I implemented Double Deep Q Networks into my code. The constructor of the agent now also contains a second model besides the normal model, together with a second history and a second memory. The hyperparameters changed so that alpha could also decay and gamma builds up over time with a function that has an asymptote at 1. Other changes include an added counter for wins, losses, draws, and illegal moves for evaluation purposes.

The model creation also improved to implement any technique that could potentially increase the quality of my model. The Adam optimizer got a scheduler for experimentation. The model configuration now includes batch normalization and dropout between all layers to smooth out training, and the dense layers got a kernel initializer and a kernel regularizer. I also added a model summary to see how my models look. Halfway through writing this new attempt, I downgraded TensorFlow to version 2.10 and downgraded the Python version, as well as CUDNN and the NVIDIA toolkit for compatibility. Since TensorFlow 2.10 onward, GPU support was no longer available for native Windows, and I wanted to speed up my training, which could take up to 6 hours for 20,000 episodes, depending on the parameters.

More code-wise changes include updating the target model weights with a factor now, something I also saw several times during my research. The `update\_target\_weights()` method and the tau variable in the constructor are used for this. The `choose\_action()` and `get\_qs()` methods didn’t change much, except that `get\_qs()` now needs to distinguish between the two networks I use. The `choose\_action()` method has another commented version that could also return illegal actions since I really wanted to try everything. A small change is that the model and every related function now use a one-hot encoded representation of the board instead of just the flattened numpy array. The `update\_epsilon()` method became `update\_variables()` to also update alpha and gamma.

The `Training` class now has a lot more code related to logging and evaluating data generated during training, such as the `log\_transition()` method that generates a CSV file with the current variables for debugging. This file can scale to gigabytes, so be careful with the episode size. The `update\_replay\_memory()` method became `play\_move()`, with the addition that it now only returns a transition without the next state. This is used in the new `update\_memory\_with\_game()` method that also distinguishes between the two different networks. This method plays games and adds transitions to memory. A big change here is the way transitions are handled. In my first attempt, I made a grave mistake by not updating the rewards of the other player in case one player wins, loses, or draws. Now, with this new way of handling the memory extension, the reward on every transition is correct. This is something I discovered during testing.

I also found out that the states I feed my agents were wrong. I needed to set the current state as the state where the player makes a move and the next state as the state where the same player can make the next move, not save a state as the next state where the other player has its turn. For example, if the current state is [0,0,0,0,0,0,0,0,0] and the action is (0,0), then the next board shouldn’t be [1,0,0,0,0,0,0,0,0]. Instead, the other player should make a move, maybe (1,1), and then the next state should be [1,0,0,0,-1,0,0,0,0]. This was hard to solve since I had to update my transitions in a bracket manner and then add them all at once, handling all the possible cases. I believe it is solid now. The old `update\_replay\_memory()` is still there as a comment for reference.

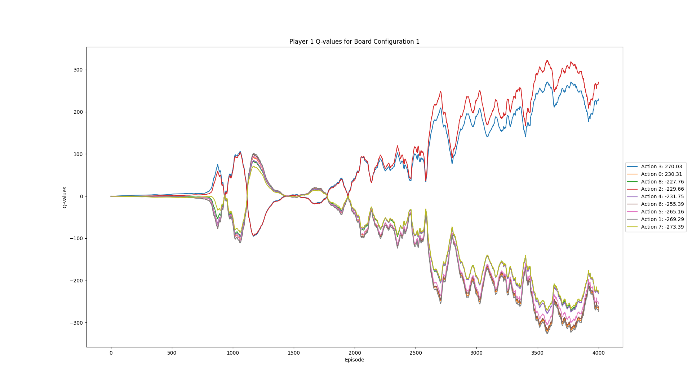
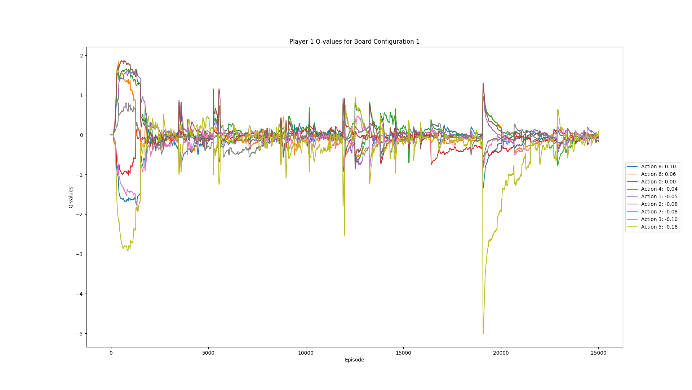
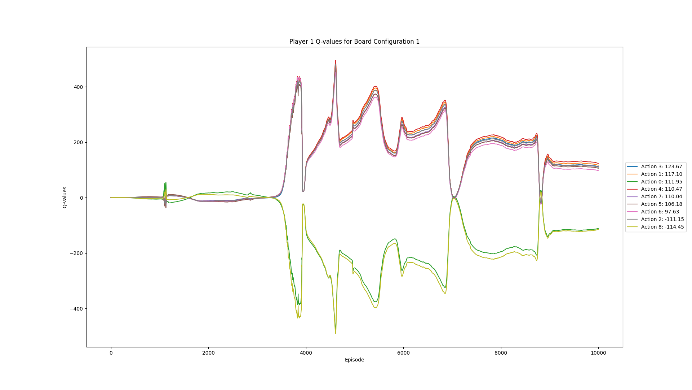
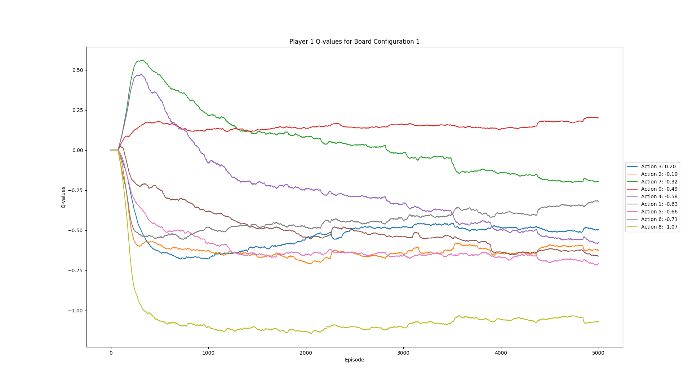
The `train\_on\_batch()` method still does the same as before on a macro level but now also handles logging to the CSV file via the `log\_transition()` method. The biggest change, and the reason why everything now has two networks, is the updated Q-value calculation. The new method doesn’t use the max function anymore but works with two networks to eliminate overestimation and make the training smoother. The `run\_training()` method remains more or less the same, with the addition of new data gathering methods and changes to accommodate the updates to the training structure.

How to run the program changed too. There is now a check for available GPUs, and the loss function can be specified as an input for the agents.

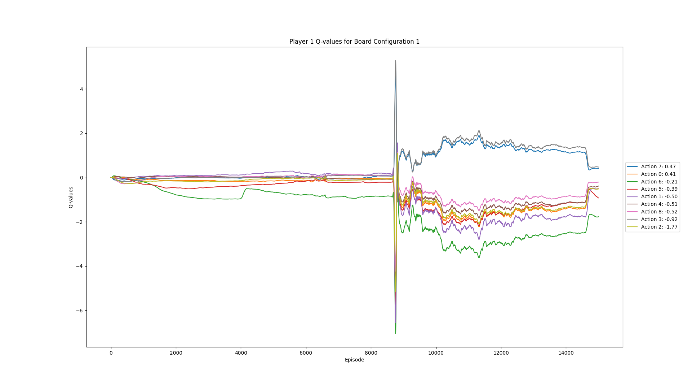
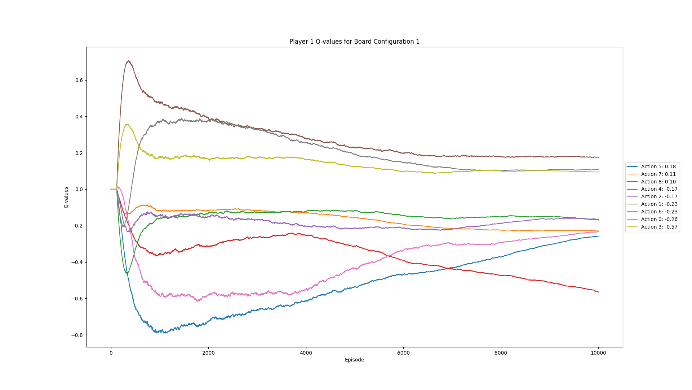
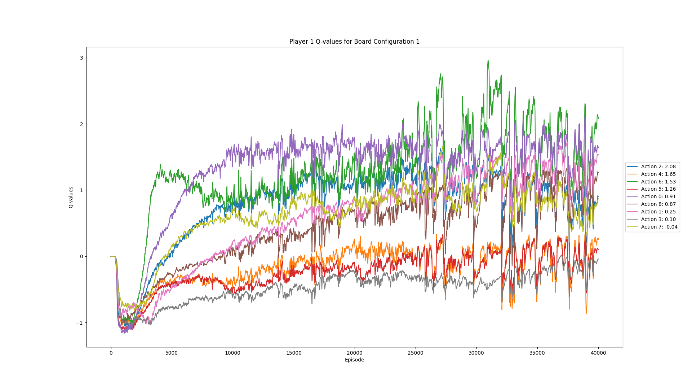
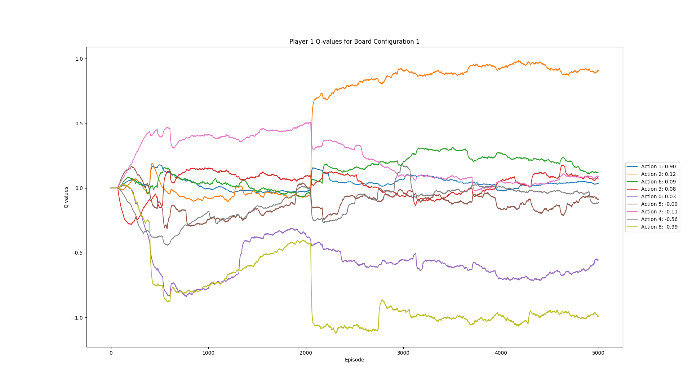
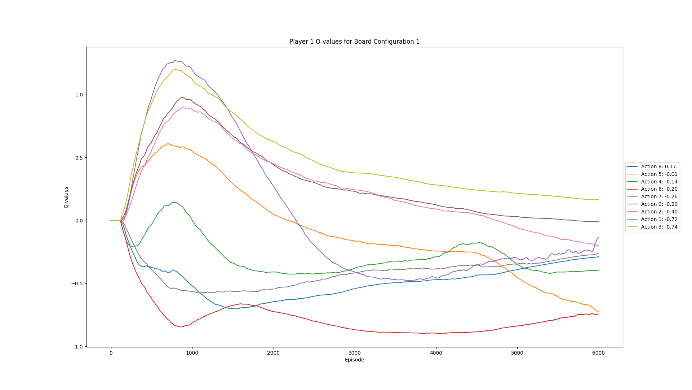
The `environment.py` file still has a `TicTacToe` class with the ability to get the observation space, action space, and interaction method. The only difference here may be the code structure and that there are now three functions to get all actions, only valid actions, and only invalid actions. This is unnecessary since the valid actions are all actions minus the invalid actions, but I wrote a separate function for testing. The biggest change in `environment.py` is that now you can play a game against a model that gets saved at the end of training. It loads both models despite the human having no choice but to be player 1. I planned to adjust this later but didn’t do it in the end because the models are flawed anyway.

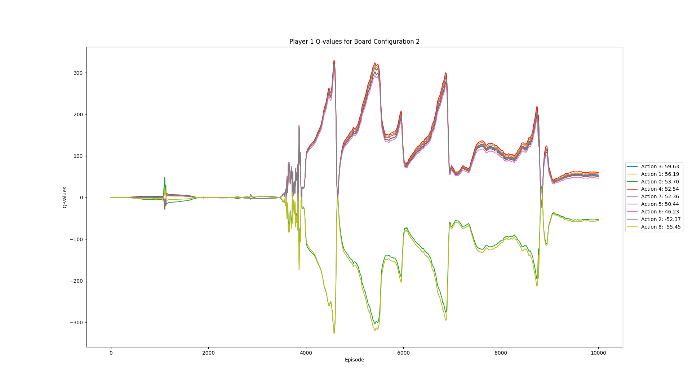
# Reflection

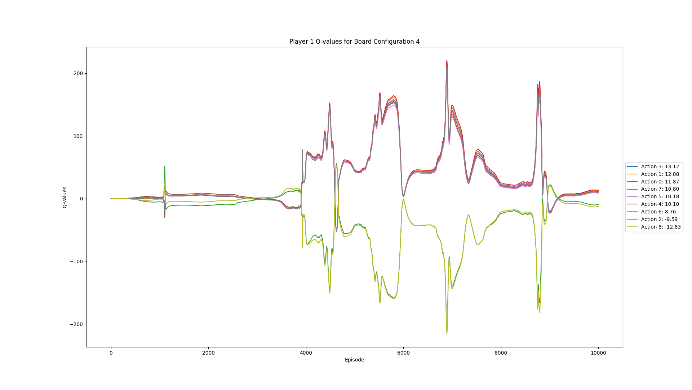
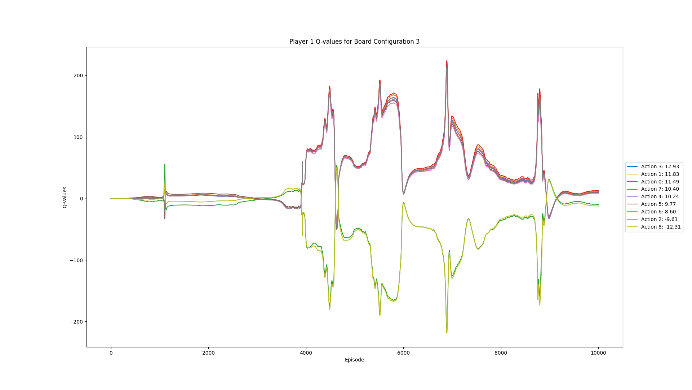
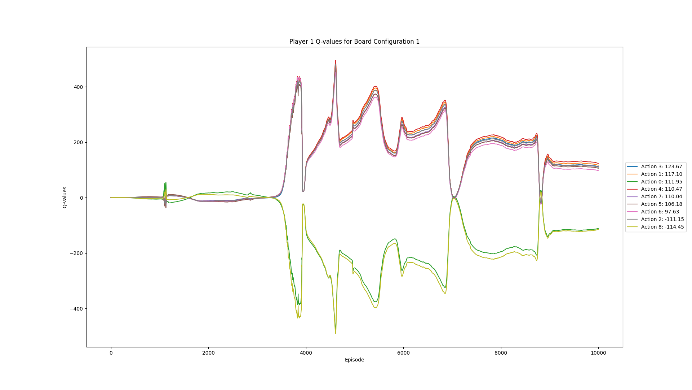
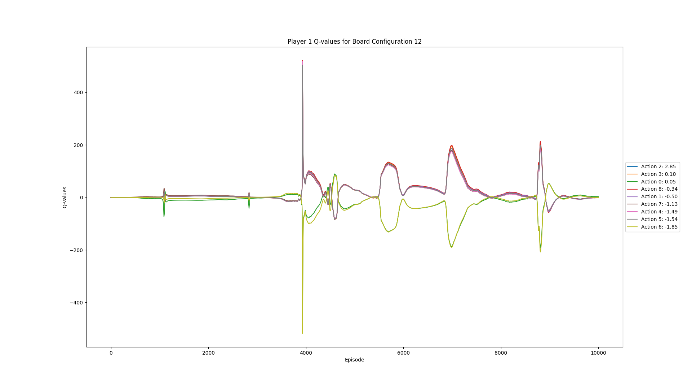
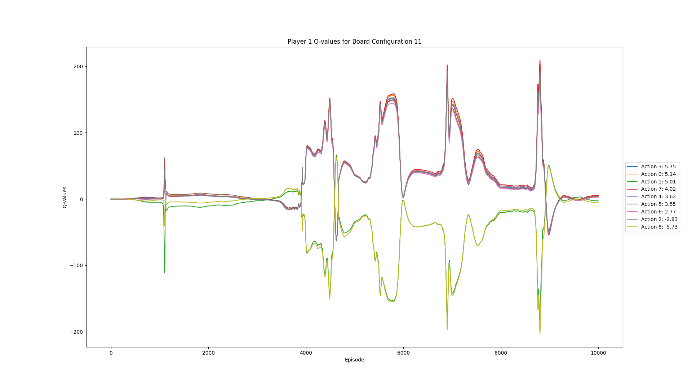
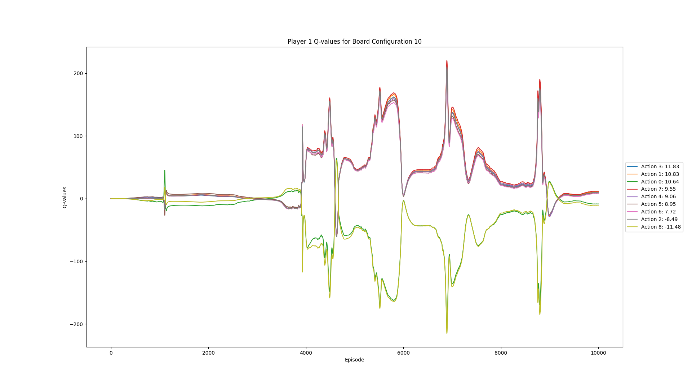
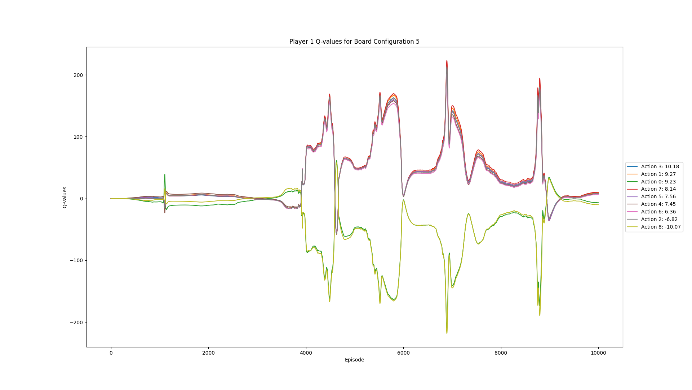
## Results and Interpretation

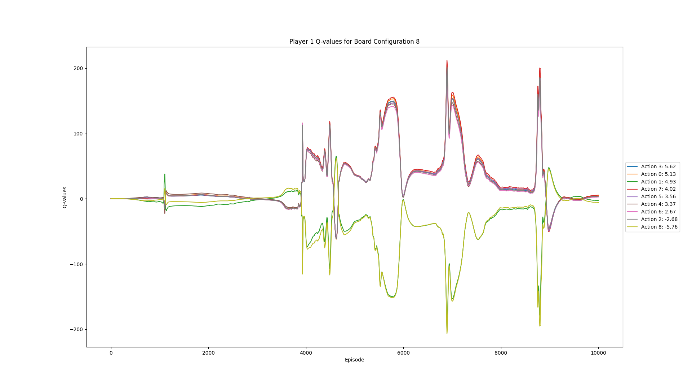
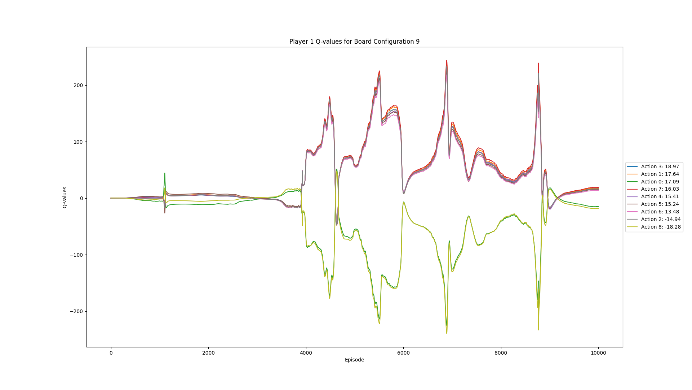
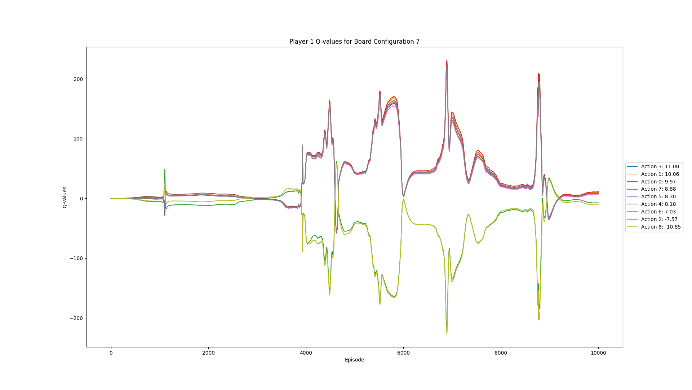
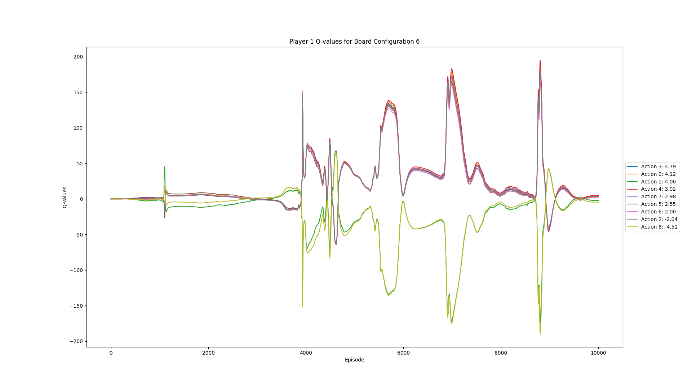
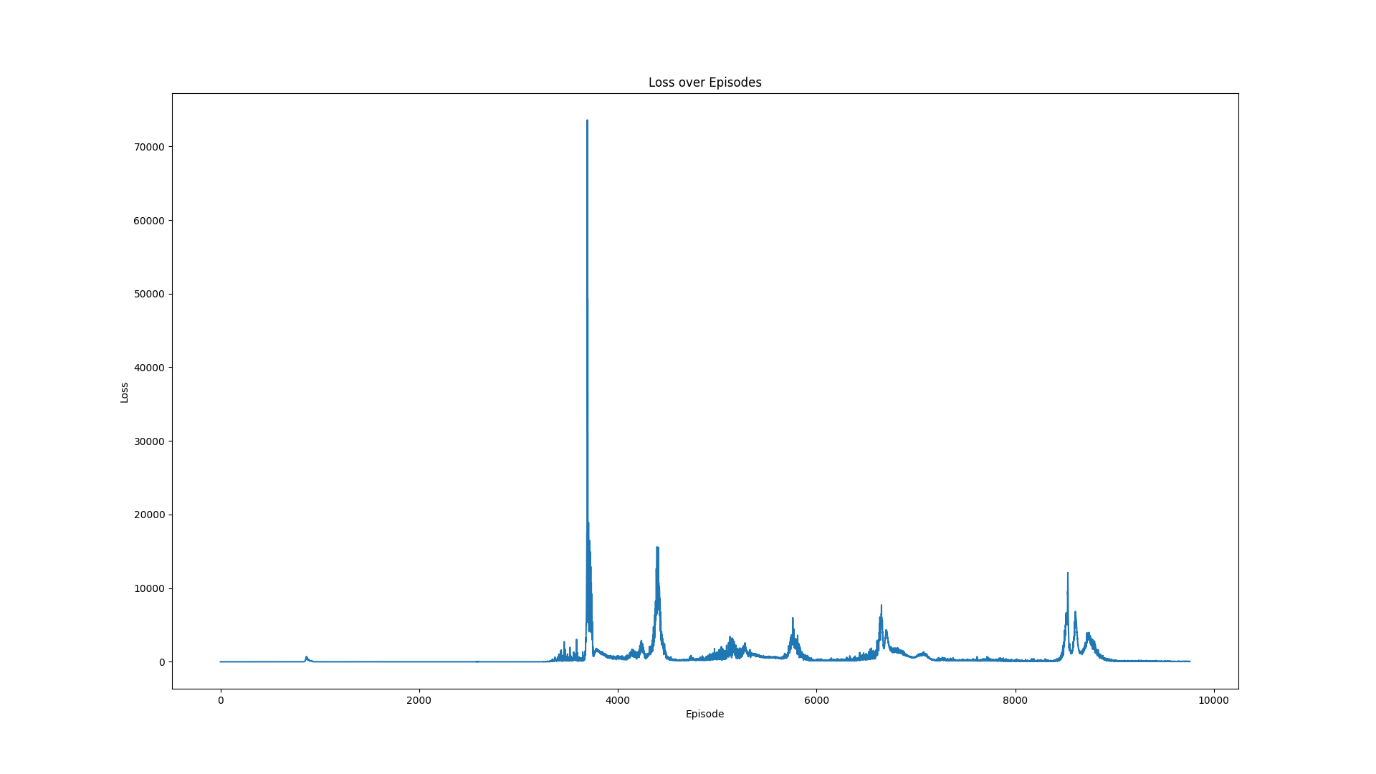
Ein Bild, das Diagramm, Reihe, Text, Karte enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Diagramm, Reihe, Karte enthält.

Automatisch generierte BeschreibungFrom my many tests, I had very different results depending on parameters and the different code versions I used. For example:

These are all images from an empty board and its Q-values for each action. The first board was trained with a single Q-Network and not Double DQN, so the results are very unstable. Later versions used Double DQN. Training became more stable, but depending on the parameters, rewards, configuration, and so on, I got different results. However, the similarities are obvious. Every graph has values that change in a similar way at specific points. In some graphs, the Q-values even change identically down to the smallest detail. The even bigger problem is that such trends can be observed not only over time as I built my program, but some sets also have these similarities between the boards themselves.



These twelve board configurations are all different; some are not even for the same player. On some boards, it's player 2's turn. Regardless, all the values follow a certain pattern. The loss reflects this to some extent, but it doesn’t explain it fully. Another problem is that with mean squared loss or Huber loss, my loss sometimes exceeds eight digits, which should never happen. These sudden loss spikes appear throughout my training, and I have changed nearly every single line of code related to it. Sometimes my data seems to converge on a value, sometimes it’s totally symmetrical, sometimes it predicts the same output for all test data. Very often, I encounter these loss spikes, and sometimes the loss drops to zero in a believable curve, but the values are all over the place or locked in a more or less strict order. Even after manually observing thousands of games being trained, generating gigabytes of log data into CSV files, and about 50 attempts, I don’t know why this keeps happening.

I optimized the model as much as I could with no changes. I altered my input after discovering an error where the next state was a state where the player who received it wasn’t the one who should play. I changed all parameters to extremes, allowed different action spaces, changed the board representation, modified the way the target model gets updated, and experimented with various losses. I used categorical loss and different normal losses, and I changed my network layout multiple times.

The only reason I can think of for why my results have a pattern from before to after all these changes is that I am doing something wrong with my Q-function. When I read one article, it says to use the TD error to update the Q-value; the next says I should use the TD error as loss. The Hasselt paper says what I am currently using is correct, as does Wikipedia. Other GitHub repositories use PyTorch and manual backpropagation. I have seen target networks updated with a tau factor and just set to the model weights. I used different initializers, but sometimes after 50 training games, my Q-values reach values they are not supposed to. Overall my training, they first experience drastic changes, and then the changes subside until suddenly a huge loss spike appears out of nowhere. Maybe I will find the error someday, but as of right now, I have no clue where to even search.

## Self-reflection and what to do better next time

What I think I did wrong is that I went off track with the assignment and didn’t get much help because I did what I was asked to do, but in a manner that even my teachers couldn’t assist me with. I also did way too much research without finding any meaningful sources. Sure, I found a lot and learned a lot, but even after all that, I am still clueless as to why my program is failing.

On the other hand, I was very resilient over the course of these weeks and learned a lot, not just about reinforcement learning but also about the perspective of learning in general, how neural networks learn, and a bit of math too. Through discussions with my classmates, I learned how supervised learning works on the go, despite already knowing the basics. It was and still is a fun journey, but what made it a bit stressful was my lack of time management when it came to the actual work of writing this documentation as well as the organization of my GitHub repository.

I also underestimated how much I would read and research and didn’t start with a program to track my sources. So now, I can’t be sure if everything is there, and I had to search through a lot of my search history and tabs I had open on different devices. Overall, I liked the project despite it being too difficult for our class (in my opinion), and I hope that during university, I will do some projects as interesting as this one.

# Sources

**3cky**, TensorFlow RL Tic-Tac-Toe GitHub Repository, *https://github.com/3cky/tensorflow-rl-tictactoe*

**Aitude**, Supervised vs. Unsupervised vs. Reinforcement Learning, *https://www.aitude.com/supervised-vs-unsupervised-vs-reinforcement*

**Ardra**, Teaching Agents to Play Tic-Tac-Toe Using Reinforcement Learning, *https://medium.com/@ardra4/tic-tac-toe-using-q-learning-a-reinforcement-learning-approach-d606cfdd64a3*

**Atharva Shekatkar**, DinoGameAI\_V2 GitHub Repository, *https://github.com/AtharvaShekatkar/DinoGameAI\_V2*

**Atharva Shekatkar**, Using Q-Learning to Play the Chrome Dinosaur Game, *https://medium.com/@atharvashekatkar1.2/using-q-learning-to-play-the-chrome-dinosaur-game-e836da2bc74a*

**Baeldung**, Epsilon-Greedy Q-Learning, *https://www.baeldung.com/cs/epsilon-greedy-q-learning*

**Codastic**, Tic-Tac-Toe RL GitHub Repository, *https://github.com/codastic/tic-tac-toe-rl*

**Daniel Onugha**, How to Change the Learning Rate of TensorFlow, *https://medium.com/@danielonugha0/how-to-change-the-learning-rate-of-tensorflow-b5d854819050*

**Data Science Stack Exchange**, Q-Learning Neural Network for Tic-Tac-Toe, *https://datascience.stackexchange.com/questions/26559/q-learning-neural-network-for-tic-tac-toe-input-implementation-problem*

**Data Science Stack Exchange**, Reinforcement Learning: Decreasing Loss Without Increasing Reward, *https://datascience.stackexchange.com/questions/37792/reinforcement-learning-decreasing-loss-without-increasing-reward*

**DataCamp**, Loss Function in Machine Learning, *https://www.datacamp.com/tutorial/loss-function-in-machine-learning*

**GeeksforGeeks**, [*https://www.geeksforgeeks.org*](https://www.geeksforgeeks.org)

**Hado van Hasselt**, Deep Reinforcement Learning with Double Q-learning, *https://www.researchgate.net/publication/282182152\_Deep\_Reinforcement\_Learning\_with\_Double\_Q-Learning*

**Hado van Hasselt**, Double Q-learning, *https://papers.nips.cc/paper\_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf*

**JamesQ9**, Tic-Tac-Toe Machine Learning Using Reinforcement Learning GitHub Repository, *https://github.com/jamesq9/Tic-Tac-Toe-Machine-Learning-Using-Reinforcement-Learning*

**Machine Learning Mastery**, Loss Functions in TensorFlow, *https://machinelearningmastery.com/loss-functions-in-tensorflow*

**Mahowald**, Deep Tic-Tac-Toe, *https://mahowald.github.io/deep-tictactoe/*

**MJeremy2017**, Reinforcement Learning Implementation GitHub Repository, *https://github.com/MJeremy2017/reinforcement-learning-implementation*

**Moodle KSA SZ**, Script NN Kapitel 4-4 TensorFlow, *https://moodle.ksasz.ch/pluginfile.php/195842/mod\_resource/content/0/Script\_NN\_Kapitel\_4-4\_Tensorflow.pdf*

**NVIDIA**, CUDA Toolkit, *https://developer.nvidia.com/cuda-toolkit*

**NVIDIA**, cuDNN, *https://developer.nvidia.com/cudnn*

**NumPy**, NumPy Documentation, *https://numpy.org/doc/stable*

**OpenAI**, ChatGPT, [*https://chat.openai.com*](https://chat.openai.com)

**OpenAI**, Gymnasium, *https://gymnasium.farama.org/*

**Python Programming,** Deep Q-Learning Tutorial, *https://pythonprogramming.net/training-deep-q-learning-dqn-reinforcement-learning-python-tutorial/?completed=%2Fdeep-q-learning-dqn-reinforcement-learning-python-tutorial%2F*

**Python Software Foundation**, Python Documentation, *https://docs.python.org*

**Reddit**, Asynchronous One-Step Q-Learning Implementation, *https://www.reddit.com/r/reinforcementlearning/comments/5op1po/asynchronous\_onestep\_qlearning\_implementation/*

**Shakedzy**, Tic Tac Toe GitHub Repository, https://github.com/shakedzy/tic\_tac\_toe

**Simplilearn**, Q-Learning Tutorial, *https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-q-learning*

**Stack Overflow**, Stack Overflow, *https://stackoverflow.com/*

**TensorFlow**, TensorFlow Agents GitHub Repository, *https://github.com/tensorflow/agents*

**TensorFlow**, TensorFlow Documentation, *https://www.tensorflow.org*

**Towards Data Science**, Interactive Q-Learning, *https://towardsdatascience.com/interactive-q-learning-9d9203fdad70*

**Towards Data Science**, Reinforcement Learning: Implement Tic-Tac-Toe, *https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542*

**Tracy Renee**, Using Keras & TensorFlow to Predict Tic-Tac-Toe Game Winnings, *https://tracyrenee61.medium.com/use-keras-tensorflow-to-predict-on-the-winnings-of-a-tic-tac-toe-game-board-232fe9e68075*

**Wikipedia**, Machine Learning, *https://en.wikipedia.org/wiki/Machine\_learning*

**Wikipedia**, Mean Squared Error, *https://en.wikipedia.org/wiki/Mean\_squared\_error*

**Wikipedia**, Q-Learning, *https://en.wikipedia.org/wiki/Q-learning*

**Wikipedia**, Reinforcement Learning, *https://en.wikipedia.org/wiki/Reinforcement\_learning*

**W3Schools**, W3Schools Python Tutorial, [*https://www.w3schools.com/python*](https://www.w3schools.com/python)

**Yashpatel5400**, *https://gist.github.com/yashpatel5400*

**Yauheniya AI**, Visualizing the Q-Learning Algorithm, *https://medium.com/@yauheniya.ai/visualizing-the-q-learning-algorithm-bd283600773f*

**YouTube**, Double DQN,

[*https://www.youtube.com/watch?v=Lb5ADHnRQV8*](https://www.youtube.com/watch?v=Lb5ADHnRQV8)

My code and other data can be found here: https://github.com/DoctorBlackZ33/TicTacToeAI