Solving Connect Four with Deep Q-Learning

Dangel Frederik, Merkel Jonathan and Wegert Jacqueline, *University of Applied Sciences Karlsruhe*

***Abstract*—This paper is about applying Deep Q-Learning to the board game connect four. After a brief introduction to the game and showing why the game is too complex to train a game-strong agent using conventional reinforcement learning (lookup methods), we first lay essential theoretical foundations. Subsequently, similar work is referenced to address similar uses of Deep Q-Learning.**
Based on this fundamentals, the iterative approach of the study is outlined. Finally, the resulting findings are presented. The paper ends with a discussion and presentation of further development opportunities. The code is publicly available on Github.

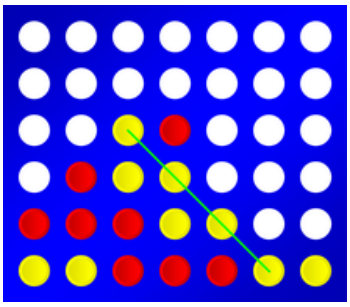

Fig. 1: The connect 4 game board

I. Introduction

In this paper, we present our results that emerged in a practical project at the University of Applied Sciences. We, as a team, participated in the Kaggle Challenge 'Connect X'. The goal of the project was to investigate various influences on the optimal strategy in response to the underlying Markov Decision Process of the well-known board game. Our approach follows modern deep reinforcement learning methods. Connect four is a widely used social game for two players playing against each other. The goal of the game is to achieve a combination of 4 connected coins of one's color in a row, in a column, or in a diagonal. There are considerable possibilities to apply a strategy to increase one's probability of winning, e.g. fields in the center increase the probability of obtaining a 4-coin long row of one's own color. The game board and rules allow for a wide variety of possible end boards, with players able to perform numerous different actions to achieve them.

A. The connect four game

Connect Four is played on a vertical board, the environment, with six rows and seven columns, resulting in 42 playable positions on the board. Each player has 21 pieces, which differ in color. In general, it is decided randomly which player will make the first move - the subsequent ones are made alternately. A move consists of a player dropping his token from the top through a column into the board. It then either falls into the bottom row or lands on another tile. If there are already six pieces in a column, no more can be dropped into it.

The goal of Connect Four is to be the first player to place four of his colored tiles in a row without leaving any gaps between the four tiles on the board. At this point, the game is over and the player who has connected the four tiles wins. When all the pieces have been played and the board is full, the game is considered a draw. Even though the game setup and the rules seem straightforward, we must consider the game complexity. Although the board does not seem large at 42 digits and only two players are involved, the game exceeds what is possible with classical lookup methods of reinforcement learning. The game state complexity is 10^{42} and the game-tree complexity is 10^{14}. The game is often decided in the top rows, because the average game is over after 36 moves [1].

B. Related Work

The first successful combination of Reinforcement Learning, specifically Q-Learning, with methods of Deep Learning let to the term DQN (Deep Q-Learning) and was done by [2]. They used DQN to train the player win Atari games and proved that this model is capable of applying the gained knowledge to completely new problems.
DQN approximates the Q-Value function with a deep (convolutional) neural network. Our implementation follows this architecture. Following papers are noteworthy and drive this groundbreaking work further.

Osband et al. tackled the problem, that efficient exploration is still a major challenge for reinforcement learning. Due to incompatibility between randomized value functions and algorithms with nonlinearly parametrized value functions, they proposed bootstrapped DQN. This enables the combination of deep exploration and deep neural networks for faster learning [3].

Based on recent distributional reinforcement learning, Dabney et al. applied a variant for DQN by using quantile regression to approximate the full quantile function for the Q-Value distribution. Their foundings are, that by reparameterizing a distribution over the state-action space an implicitly defined return distribution is yielded and enhances many risk-sensitive policies. They also demonstrated the performance increase on the Atari 2600 games [4].

When using experience replay, it is common to sample experience transitions uniformly from a replay memory. This approach replays transitions at the same frequency that they were originally experienced but neglect their significance. In their work, Schaul et al. replayed important transitions more frequently, and therefore improved the efficiency of learning. This approach is called prioritized experience replay [5].

Q-Learning is known to overestimate action values under certain conditions. It was not previously known whether such overestimations are common, whether they harm performance, and whether they can generally be prevented. Hasselt et al. showed that DQN suffers from substantial overestimations in some games and proposed Double Q-learning that can be used at scale to successfully reduce this overoptimism, resulting in more stable and reliable learning. It uses the existing architecture of DQN without requiring additional networks or parameters [6].

Often, deep reinforcement learning methods use noise for exploration in the action space. An alternative concept adds noise directly to the agent´s parameters for more consistent exploration and more diverse behavior. In their work, Plappert et al. combined parameter noise with noise in the action space and proved, that both on- and off policy methods profit from this approach [7].

Even if all these alternatives seem good, we decided to implement a DQN and proposed all this related work as future work that could be added to our implementation of DQN for the connect four agent.

C. Approach

To use our DQN approach, we assumed that our reinforcement task follows a finite MDP (Markov Decision Process). For a better understanding, let us at first introduce a few important terms:

- Agent : A 'thing' that executes actions by following a policy.
- Environment e : An interactive interface that takes actions of the agent and emits a new observation and a reward therefore. In the connect four setup the environment is the game board.
- State space S : Every possible situation in the game, including the position of all tiles for both colors.
- Action space A : The set of all allowed actions, so every column that did not exceed it´s range: $\{i|i \in 1, ..., 7\}$.
- Policy π : S $\rightarrow$ A : A mapping from the state space to the action space. It tells the agent which action to take for every state.
- Reward r : A (scalar) value passed from the environment to the agent which can be interpreted as feedback for the last action. In the connect four scenario, the reward is 1, if the player wins and -1 if he looses. Draws are valued at 0.

In order to win the game, the objective for the player is to take a combination of optimal actions to maximize the overall reward. The reward can only be obtained at the end of the game. Therefore Q-Learning approximates the value of every state-action combination and designates this approximation the *Q-Value*. Q-values are learned iterative by updating the current Q-value estimate towards the sum of the observed reward and the maximum Q-value over all actions in the resulting state. The Q-Value function can then be defined as:

$$Q(s_{t+1}|s_t, a) = r(s_t, a) + E(\sum_{k=1}^{\infty} \gamma^k r(s_{t+k}, a_{t+k}))$$

This function defines the Q-value of a particular state s_{t+1}. The agent chooses the action at state s_t such that he maximizes the Q-value at the next state s_{t+1}. Thus we define the optimal policy as follows:

$$\hat{\pi}(s_t) = \arg\max Q(s_{t+1}|s_t, a), a \in A(s_t)$$

With the foundational structure of Q-Learning in mind, Deep Q-Learning is very easily understood as the only difference is a substitution of the Q-Table. The limitation of the Q-Table for environments with large observation spaces is reached very quickly in Q-Learning. According to the online Encyclopedia of Integer Sequences, the state space for connect four is 4,531,985,219,092 which is not feasible to write into a table [8]. This large number is the sum of all possible actions for the 42 possible moves. The following table shows for a few of those 42 moves the number of legal 7x6 positions after n plies.

n	a(n)
0	1
1	7
2	49
3	238
4	1120
5	4263
10	1662623
15	176541259
20	6746155945
25	97266114959
30	410378505447
35	370947887723
40	22695896495
41	7811825938
42	1459332899

Table I: State space for connect 4

Therefore one needs a good approximation of the Q-Values. As algorithms of Deep Learning can take an observation along with a loss function to calculate from the reward, make some sort of generalization about the observation of the environment (according to detect similarity between spaces) in order to choose a single output (or action), so that saving Q-Values in a table becomes superfluous. In summary, to simplify the staggering size of the observation space in the connect four game, we used a standard Q-Learning structure in combination with a Deep Neural Network. To make it more precise, we used a Convolutional Neural Network.

II. Proposed Method

A. Data Preparation

Per default the kaggle environment emits a state representation that consists of a single vector with length 42 and values zero, one or two for field not filled, filled by player one and filled by player two. To bring this in a more detailed view we decided to reshape this into a stack of three 6x7 matrices each one-hot-encoded. The first gives an overview which fields are free, the second show the coins of player one and the third the ones of player two.

B. Network Architecture

Because of the reshaping, the problem is now a multidimensional problem, so a CNN is used as network. The network consists of two Conv2D-Layers. The first layer has 3 input channels and 64 features. The second layer has 64 input and 64 output features. Both layers use kernels with size three and a ReLU activation function.
After the convolution the output is flattened and passed into a fully connected network with three layers where two have each 300 neurons and the last one has seven output neurons, one for each action.

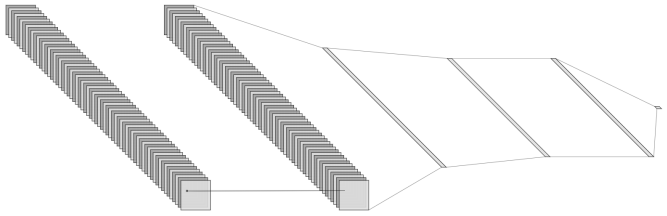

Fig. 2: The architecture of our network

C. Reward Adjustments

During training our network we found out that the emitted reward is crucial to the network performance. Also a submission to the challenge [9] proposed to adjust the reward. We did not chose the same reward system. We also added for some train runs rewards for getting a streak of three coins in a row. The resulting reward system is as follows:

Result	Reward	Reward with streak
Win	10	20
Loss	-10	-20
Invalid Move	-20	-40
Draw	1	1
Step	1/42	1/42 + 3*(no. three in a row)

Table II: Reward system

To prevent the agent from taking actions to prolong the game and getting many stacks of three in a row we decided to double the reward for win, loss and invalid move.
When experimenting with the reward we found out, that our agent seemed to train bad when no or a negative reward is given for each step. We had to at least give the agent a little reward for keeping him playing.

III. Training

In order to train the network we let our agent play versus a random agent provided by the kaggle environment. We also tried to train our model with a pre-trained network trained by us and with a negamax agent using a search algorithm but since the results were not promising at all we chose not to go any further in this direction.

Like in the Atari paper proposed we use an experience replay buffer and a target-network to train our model [2]. The replay buffer is filled with transitions consisting of the previous state, the chosen action, the reward, the current state and the information if the episode is over. When the buffer size of 20.000 transitions is reached the oldest transition will be replaced with the new transition. After enough experiences are collected the policy network is trained and updated after each step with a batch size of 32 random samples from the experience buffer.
The target net and the policy net are synchronized every 50 episodes.
During training actions are chosen with an ϵ-greedy policy with ϵ decaying from 0.9 to 0.1 within 1500 episodes. Per default we also prohibited actions that are invalid, e.g. when the column is already full.
Since playing as player one and player two is slightly different we also switched the position of our agent after every thousand episodes.
As optimizer we used the Adam optimizer with a learning rate of 0.001 and as loss function the mean squared error was used.

A. Training Issues

During training we encountered some issues which led to a poor performance. First of all we observed the Q-Values of the first action. When playing connect four the best move to start with is placing the coin in the middle column. We expected the agent to learn this behaviour. When choosing a bad reward system our agent failed to learn this behaviour. It seems he didn't care about the column at all since all Q-Values converge fast to 0. This behaviour did not improve over various episodes. Due to this we had to choose the reward system like proposed.

IV. Evaluation

For improving our network we ran different training runs each consisting of 20.000 episodes. We found out that the parameters having the most impact on performance were the reward and the discount factor. We ran several experiments with disount factors ranging form 0.4 to 1.0 and using the additional reward system for having three coins in a row. For Evaluation we let our agent play versus the random and the negamax agent.
There was also an additional training run with discount factor 1.0 but allowing invalid moves to test if the agent learns to not make those moves. We chose the discount factor to be one because we suspected that this would help the agent to ignore bad actions since the reward of this action is very bad.

A. Default Reward System

The following table shows the win percentage of the models after different training runs using the default reward system. The agents are facing a random agent or the negamax agent each for 100 times. Even though the agent gets a little bit better versus the random agent with a smaller discount factor

Discount factor	Random	Negamax
0.4	97%	1%
0.5	96%	2%
0.6	92%	3%
0.7	94%	7%
0.8	95%	2%
0.9	86%	3%
1.0	86%	0%

Table III: Win percentage default reward system

it performs best with a medium discount factor of 0.7 versus the negamax agent. Also the agent is unable to win versus the negamax agent when setting the discount to one.

B. Additional Reward for three in a row

For the following training runs the step reward was adjusted to give additional reward when the agent places three coins side by side.

Discount factor	Random	Negamax
0.4	98%	7%
0.5	97%	7%
0.6	98%	4%
0.7	96%	9%
0.8	91%	8%
0.9	78%	6%
1.0	58%	2%

Table IV: Win percentage with streak rewards

When using this configuration the win percentage for lower discount factors get´s slightly better for the random agent but a lot better for the negamax agent. There is also a great performance decrease if the discount factor gets bigger than 0.8. Surprisingly this is more valid for the random opponent than for the negamax. Our agent still manages to win versus the negamax agent even though he performs bad against the random agent.

C. Allow invalid actions

In the following training runs we allowed the agent to take invalid actions during training. For each run a discount of 1.0 was used.
Even though the agent was trained allowing invalid actions we made evaluation runs disallowing those action to compare the performance to the other training runs.

Evaluation Configuration	Random	Negamax
Default Reward Disallow invalid actions	69%	6%
Additional Reward Disallow invalid actions	74%	5%
Default Reward Allow invalid actions	59%	0%
Additional Reward Allow invalid actions	65%	0%

Table V: Win percentage with invalid actions

This table shows how many invalid actions were chosen by the agent during the evaluation versus the random and the negamax enemy.

Evaluation Configuration	Invalid Actions Random	Invalid Actions Negamax
Default Reward Allow invalid actions	11	1
Additional Reward Allow invalid actions	9	6

Table VI: Number of invalid actions

These test runs show that with the chosen training parameters the agent performs almost equally good versus the random agent and slightly better versus the negamax agent compared to the runs where invalid actions were permitted and the discount was set to 1.0. Unfortunately the agent fails to learn to not make invalid moves. When invalid moves are allowed during testing he not only performs bad versus both agents he also still makes moves that are not allowed.
Since the other training runs performed well with a discount of 0.7 further investigations with this training configuration can be made.

D. Training With Limited Episodes

Due to a bug in the Replay-Buffer the first train runs were not as good as the presented results. With some unit tests we found out that once the Replay-Buffer is full the newest transition will be overridden instead of the oldest. Because of this the agent could only learn from older and presumably worse transitions.

Configuration	Random	Negamax
Discount: 0.7 Default Reward	93%	3%
Discount: 0.7 Streak Reward	90%	4%

Table VII: Win percentage with Replay-Buffer bug

Even though the agent did only learn from older episodes he learned enough to perform well during the tests. Because of this smaller training runs might be sufficient enough to train a good agent.

E. Evolution During Training

The following chart presents how the reward of the train batches evolved during training. It is comparing the runs with a discount of 0.7 for each reward system. The orange plot uses the reward system with additional reward for a streak and the purple plot the default reward system. The plot is smoothed with a smoothing of 0.8. The reward of the

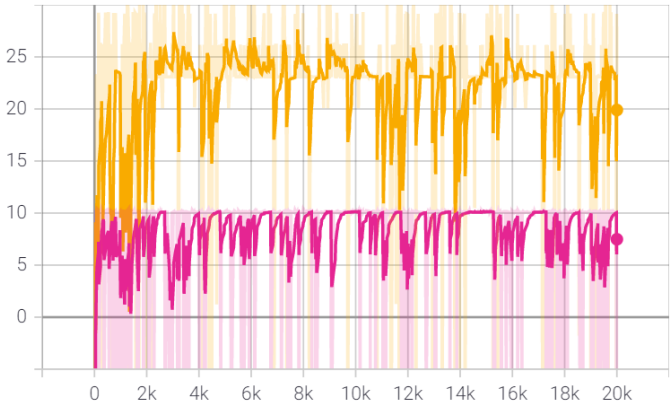

Fig. 3: Reward during training

purple plot has more plateaus so training with this reward system gives more stable results. The additional rewards during training are making the agent trying to maximize the reward and getting more streaks. This can be concluded since the reward for winning is 20 but it still drops very often. This leads to the assumption that for further experiments the reward system must be adjusted to make winning fast more important than getting a lot of streaks and try to win afterwards.

The last figure shows how the win percentage of the agents evolved during training. The evaluation took place every 500 episodes.
The orange and the purple charts describe the runs with the fixed Experience-Replay and the blue and grey plots the ones with a limited amount of new episodes. For both run groups it can be observed that even during training the reward system with the streak rewards are better in terms of winning versus the random agent.

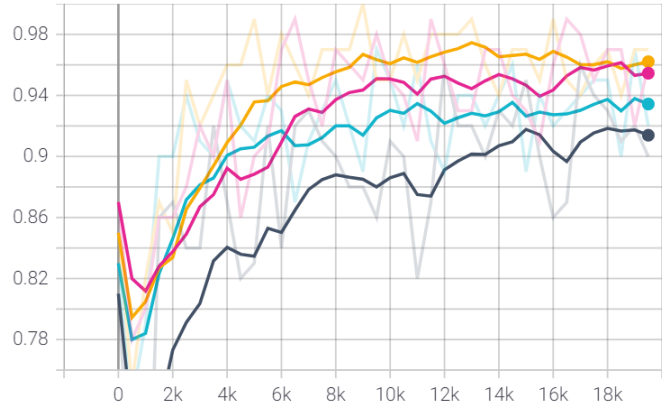

Fig. 4: Win percentage during training

It can also be seen that the peak during training was before the 20.000 episodes. This reinforces the assumption that lesser episodes might be sufficient or even better. Even though this has to be tested for playing the negamax or even better agents.

V. Conclusion And Further Development

In this paper we proposed a model using DQN for training and playing Connect Four versus a random and a negamax agent which is using a search algorithm. We tried different training configurations and found out that changing the reward system and adjusting the discount factor is crucial for the performance of the policy. The reward system with additional reward for getting a streak of three coins seems to slightly improve the agent. Future test runs can exploit this further with giving also rewards for getting a streak of two. It should also be highlighted that when giving additional rewards for playing the game (e.g. getting reward for a streak or just a move) the agent should also be forced to chose shorter games in order to prevent draws versus better agnets than random.
We also showed that with the chosen configuration it was not possible for the network to learn to only make valid moves. This topic can be investigated further and combined with experiments having more training episodes or a different network architecture using for example a smaller network.
Another approach to improve the general performance of the network is to use a pre-trained network as opponent for training after having trained for a period of time. This can also be extended to train versus the negamax agent as soon as it is good enough to generate a sufficient amount of winning episodes versus the negamax. Driving the investigations further, one can also apply the approaches presented in the related work subsection and combine it with our proposed method.

Overall you can say that our model performs pretty well on the task compared to other submissions with DQN to the kaggle challenge which most of don't win versus the negamax agent at all.

Acknowledgment

The authors would like to thank the University of Applied Sciences Karlsruhe for providing the GPU resources and also Prof. Dr. Patrick Baier for his support.

References

[1] Stefan Edelkamp and Peter Kissmann. "Symbolic Classification of General Two-Player Games". In: *KI 2008: Advances in Artificial Intelligence*. Ed. by Andreas R. Dengel et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 185–192. ISBN: 978-3-540-85845-4.

[2] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[3] Ian Osband et al. "Deep Exploration via Bootstrapped DQN". In: *CoRR* abs/1602.04621 (2016). arXiv: 1602.04621. URL: http://arxiv.org/abs/1602.04621.

[4] Will Dabney et al. “Implicit Quantile Networks for Distributional Reinforcement Learning”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 1096–1105. URL: http://proceedings.mlr.press/v80/dabney18a.html.

[5] Tom Schaul et al. *Prioritized Experience Replay*. cite arxiv:1511.05952Comment: Published at ICLR 2016. 2015. URL: http://arxiv.org/abs/1511.05952.

[6] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[7] Matthias Plappert et al. “Parameter Space Noise for Exploration”. In: *CoRR* abs/1706.01905 (2017). arXiv: 1706.01905. URL: http://arxiv.org/abs/1706.01905.

[8] John Tromp. *A212693 Number of legal 7 X 6 Connect-Four positions after n plies.* URL: https://oeis.org/A212693.

[9] *Deep Reinforcement Learning*. https://www.kaggle.com/alexisbcook/deep-reinforcement-learning. Accessed: 2021-06-27.