Exploring Deep Reinforcement Learning Methods with Connect-4

Marlon Dammann, **Johannes Nowack**, and **Pelin Kömürlüoğlu**

Deep Reinforcement Learning
Summer Semester 2022
Universität Osnabrück

Abstract

Games offer a vast variety of rules and settings that create an excellent opportunity to develop reinforcement learning methods. Connect-4 has also gotten into the focus of this research area and many researchers have tested different methods for AI to play Connect-4. However, we found no research to compare the performances in board sizes different than the regular size, 6x7. This project aims to investigate the performances of agents that were implemented using three different methods - Minimax, DQN, and NEAT - in different grid sizes. While in the regular size grid the NEAT and the Minimax agents perform very similar, in the rest of the grid sizes the Minimax agent demonstrated the best performance.

1 Introduction

Games provide interesting problems for a wide range of Reinforcement Learning (RL) techniques to tackle. The availability of various video game environments allows researchers to improve methods to solve problems in a more efficient manner. On the other hand, the complexity of board games is comparable to real-life problems, moreover, testing enables different AI techniques and human players to compete against each other [3, 10]. They also have more or less simple rules in comparison to reality, however, the settings generate complex problem-solving and decision-making tasks [8]. The diversity and also the simplicity to some degree of the rules and board configurations offers numerous challenges for AI to overcome and due to this reason, several games have been the focus of AI research, such as othello, backgammon, chess [1], and even Settlers of Catan [10] with the aim of developing new methods or improving the current ones.

In this project, we implement RL algorithms to play Connect-4. Connect-4 is a simple 2-player turn-taking game. The game is played on a 6x7 size (vertical x horizontal) grid and there are 21 checkers for each color, red and yellow. The turn consists of putting a checker into one of the columns of the grid and the checker falls down to the lowest row available.

The checkers will stack up on each other in the grid creating numerous combinations of positions, resulting in 4,5 x 10^{12} board positions [8]. The objective of the game is to be the first player to make a row of 4 checkers of the same color and this row can be horizontal, vertical, or diagonal [7]. We have a DQN, a Minimax, and a NEAT agent. We have decided to train DQN and NEAT agents, and implemented a baseline Minimax agent. We aim to evaluate the DQN- and NEAT models against baseline methods and compare the performances of agents in a simple environment of Connect-4 that has multiple size settings.

2 Background

2.1 Tree search algorithms

One of the classical paradigms for finding an optimal choice in each state of a game is the application of a tree search algorithm. A plethora of such algorithms exist, some of the most prominent are Minimax, Alpha-Beta Pruning, and Monte-Carlo Tree Search.

The general idea of tree search is to build a tree with the current state as root node and from there on create branches corresponding to moves in the game to child nodes. A heuristic value, or a true value if known, is then applied to each node either recursively from the leaf nodes up or iteratively whenever a new child node is created, and the result is propagated upwards. Evaluation of a given node's state can either be performed using a custom evaluation function for the game or by statistical playouts given a playout policy. Another option would be to only use true, game-theoretical, values and to assign a neutral value to all nodes with unknown true value. After a given number of iterations or once a certain depth has been fully evaluated a move among the children of the root node is chosen to be played [1].

For this project Connect-4 has the advantage of full knowledge, meaning the entire state can be observed, and fully known and deterministic transition rules which enable easy simulation of the game for tree search.

In Dabas et al. Minimax with Alpha-Beta pruning, and Monte-Carlo Tree Search have been used to compete against a trained Double Deep Q-Network (DDQN) and some models which were not trained by themselves. The limiting factor chosen in for the tree search algorithms was a maximum depth and a limited search time. The most successful agent was DDQN. However, with given search time MCTS showed an improved performance,as well as the Minimax with Alpha-Beta Pruning when the depth level was adjusted to 2 [2].

Thill et al. were among the first to develop an agent trained with N-tuples to succeed in the game Connect-4. They have also showed the performances of Temporal Difference Learning (TDL) agents using the Minimax as a referee [8]. Minimax is a good baseline for evaluation the performance of other Connect-4 agents which is why we have chosen to use Minimax as our tree search baseline for evaluation our model .

2.2 Deep Q-Networks

Deep Q-Networks (DQNs) are used to approximate the value function of state-action pairs when state or action spaces reach a size that would make storing their entirety in a

Q-table exceed available memory or make storage impossible if one of the state space is continuous.

Agents making use of DQNs can train them using samples they gathered from exploring an environment. When making use of an Experience Replay Buffer, those samples are stored in the buffer and used for training for example via Q-learning. In order to encourage an agent to explore their environment (and thus avoiding sticking to local optima), a technique called epsilon-greedy can be used. With that, each action an agent takes is either random or follows the highest value action as suggested by the DQN given the current state. Epsilon-decay may be used to have a decreasing Epsilon over the course of the training leading to frequent random actions initially and lessening exploration during advanced stages of training.

During training, the problem of moving targets arises when using the same DQN for estimation of state-action values and target values as every training update slightly changes the target. To meet that, two DQNs can be used where one acts as the state-action value estimation network, the other as for target calculation during network updates. When delaying the target network, only the value-DQN is directly trained, the target network gets its parameters copied from the value-DQN with a delay.

2.3 Neuroevolution

Neuroevolution (NE) is the application of genetic algorithms to evolve artificial neural networks. It works as a form of training for network weights and in some cases additionally as a means to develop and improve network topology. Biologically inspired, NE maintains a population of genomes, which encode a neural network in its entirety. Genomes get evaluated by a user-defined fitness function, e. g. by measuring the represented neural network's performance on a task. Low-performing genomes are discarded, while better ones are kept, mutated (by little random changes in some of their values) and recombined (by combining parts of two genomes). The resulting genomes form a new generation undergoing the same procedure - a process which is repeated until for example a threshold in fitness of one of the population's genomes is reached. When using the feature of speciation, genomes are grouped into species by similarity and several species form the population. Speciation is a method to strengthen exploration, as it allows lower-performing genomes to evolve before being exposed to selection amongst the higher-performing genomes. NEAT (NeuroEvolution of Augmenting Topologies) [6] is a common method of NE featuring non-fixed structures of neural networks while performing the neuroevolution. NEAT was shown to be able to challenge basic reinforcement learning methods in some tasks [6] An implementation for using NEAT in python is provided by the module neat-python [5].

3 Methods

3.1 Environment

We created an environment for Connect-4 which was largely inspired by the OpenAI Gym's environment classes [4]. Although there were several environments available for the standard grid choice, we opted to write one which allows for specification of a custom grid choice larger than six by seven. We chose the default size for our NEAT model and our first DQN model and then also created one model each for the grid sizes eight by nine, ten by eleven, twelve by thirteen.

We chose to represent a given state observation to be returned of shape (M, N, 3) in which MxN (rows x columns) refers to the grid size. Our feature dimensions contain binary information on the location of the player tokens and on whose turn it currently is. The first feature dimension contains the information where a token for player A is, the second feature dimension, correspondingly, for player B, the third feature dimension is filled with zeros or ones, given the current player whose turn it is.

Actions in the environment are taken by passing an integer corresponding to the column in which to add a piece. Therefore, the action space is discrete and ranges from zero to N-1, since indexing starts at zero. Actions which are illegal are handled by selecting the next unfilled column to the right which enables for less trouble with illegal actions. After each action it is checked whether the following state is terminal.

For the rewards we chose to assign a reward of 100 to the winner, minus 100 to the loser, and, in case of a draw 10 for each player. Additionally, each player receives a reward of 1 for taking any action which does not lead to a terminal state. Giving that step reward is a tradeoff, as taking more moves until a terminal state is beneficial for the player who looses in the end (at least from a training's perspective), but not so for the player who wins the game. And the outcome is unclear during the game. Yet, for training, we think that giving a loosing agent an information about how long they withstood an opponent is more valuable than it is harmful to give a winning agent a wrong information on the value of the game's length.

3.2 Game-theoretical Minimax

The baseline we decided upon is a custom Minimax implementation adjusted to fit both our environment and our specifications. A node class and a Minimax class which contains the search tree were created. The node class contains a move, a value, the index of the parent and children nodes, and two Boolean values named proven and visited. Since a state's data allocation is much larger than simply storing a move and using it to re-traverse the environment from a starting state, we opted to store moves in each node and not the environment object. Additionally, we also decided against using an evaluation function, all node values which are not game-theoretically proven are assumed to be neutral and have a value of zero.

We also modified Minimax to allow for multithreading and included proven rules which were taken from Winands et al. These rules allow us to propagate the known true values of nodes upwards in case they are present. The first rule states that a terminal node won by either player automatically makes the parent node's value known as a loss by the opponent, assuming an exploitative policy by the player. Additionally, the second rule states that a node's value is known as a win for player A if all children result in player B losing. These rules were implemented in our tree search to enable for fast branch ignoring and to be used in the actual minimax operation when propagating values up the tree [9].

Our algorithm starts by creating all nodes up to the maximum depth. From there on it will work its way downward to the maximum depth and, from all nodes in the current depth, traverse down to one terminal node. This traversal is used to find terminal nodes fast and to prevent accessing nodes twice. Only nodes which have visited set to false will be examined and proven nodes automatically count as visited. If a terminal node is found the first proven rule is applied and all nodes beneath that node are set to visited. In this manner eventually, and whilst avoiding visiting any terminal chain twice, every value at the maximum depth or, if a node is terminal before the maximum depth, the proven value at current depth, is collected. Once all these values were collected the Minimax operation can be applied to propagate all proven values up as much as possible. The general assumption here is of course that the player tried to maximize its score and the opponent tries to minimize the player's score.

Although a larger maximum search depth is possible for our algorithm we have chosen to limit it to a depth of three since, depending on the branching factor of the Connect-4 grid size, a larger depth may result in a search time which is too long for our purposes.

3.3 DQN model and training

Our Deep Q-Network models consist of four convolutional layers, each has 32 filters, a 3x3 kernel, and uses the ReLU activation function. After the convolution we apply global average pooling followed by flattening. Finally, we pass the flattened tensor through a final dense layer with a unit size of N, the number of columns in the Connect-4 grid and apply the sigmoid activation function.

For the Experience Replay Buffer we chose a buffer size of 32.000 single batch elements. Each element appended will contain the input state, the action taken in the input state, the resulting reward, the resulting successor state which is the state after the opponent has also played their move, and an integer tensor which contains the terminal information, converted from Boolean value. For training on the buffer, we converted 32 randomly chosen single episode batches from the buffer into a 32-batch used for training.

In training we made use of the method of a delayed target network with a delay of two training episodes. For the directly trained DQN, we applied the Adam optimizer with a learning rate of 0.002 and used the Mean Square Error as our loss function between targets and predictions of our DQN and our target network on the buffer elements.

The training consists of two steps. Firstly, the episode step in which we collect 32 full episode trajectories which we append to the buffer; the episodes are played in parallel using the multi-threading module in python. Secondly, the training step in which we collect 32 randomly selected elements from our buffer to use for training the DQN. In each epoch we did the episode trajectory collection 50 times followed each by ten training steps which amount to a total of 320 buffer elements used for training. Depending on the turn count, which can vary from the minimum of seven to the maximum of M times N turns, the number of collected trajectories can vary between 224- and 32-times MN, therefore, our exact sampling ratio varies. Initially, before starting the first epoch, we also do ten-episode steps to collect a total of at least 2240 elements following the aforementioned formula.

In our training we additionally included an exploration factor epsilon which we iteratively decreased with epsilon decay. A proportion of moves played in each episode corresponding to this epsilon were randomly chosen. We decided to have a starting epsilon of 0.9 which we let decay to a minimum of 0.1. The decay factor was 0.998 to the power 32, our batch size and was used to update epsilon after each epoch. The discount factor used for computing the return in the training step is set to 0.99.

The number of epochs which we trained for each model varied, additionally we also had to start the training from latest epoch with an empty buffer due to a RAM overflow error which could not be identified. The exact number of epochs for each model and the epoch in which we restarted the buffer are the following. Our six by seven model ran a total of 71 epochs and was resumed with empty buffer from epochs 52 onwards. The second model for the eight by nine grid ran a total of 50 epochs and was resumed once on epoch 27. Thirdly, the ten by eleven model ran 50 epochs, resumed at epoch 26. Lastly, the twelve by thirteen model ran 26 epochs and was resumed from the twentieth epoch. The motivation between the last model's low epoch count was negligible loss improvement and the motivation behind running the smallest model for 71 epochs was to improve it further since it appeared to stagnate.

3.4 NEAT Model

The agent that is relying on genetic algorithms to learn how to play connect-4 uses NEAT to evolve an artificial neural network. The network gets the current state of the game as an input and evaluates it to determine which move the make. Each move from the action space gets a value this way, the highest value indicates the move for the agent to take.

Differently to the DQN model, an observation of a game state for the neuroevolution agent is represented by only one grid, each cell having either a 0 for no placed checker, the values 1 and 2 for the two players respectively. This decision was made to reduce the crucial size of the networks for NEAT to work with, thus improving speed and possibly performance of the algorithm.

Initially, the population consists of 160 genomes representing one neural network each. The agent supports the standard connect-4 grid of 6x7 only, though would be suitable for adaptation to other grid sizes, each then requiring a separate evolution run with possibly individual parameters.

The networks have 42 input nodes. Then there are 7 output nodes and at the start, one hidden layer of 50 nodes. The connections are initially established with a sparsity of 0.4 in a feed-forward fashion between the layers. Nodes as well as connections can (and will) be deleted or added during evolution while not maintaining the clear layer separation, with the possible result of some input neurons getting directly connected to output neurons and hidden neurons connected to each other.

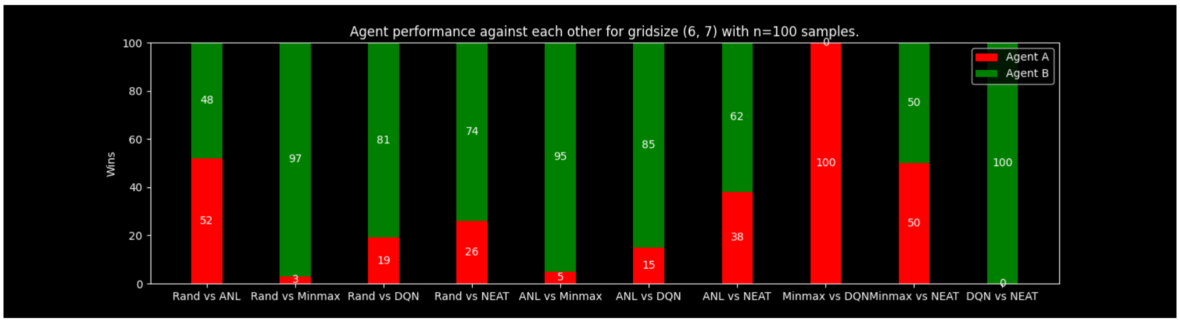
The evolution is separated into two stages, each using a different method to evaluate the networks' fitness. During the first stage, in each iteration (generation) of the neuroevolution every network from the population plays the game 20 times against a randomly playing opponent. The fitness is calculated by accumulating the environment's default rewards for the respective population's network divided by the number of games played (20). By evaluating each network multiple times, variation in the outcome resulting from the opponent's random moves is reduced.

The second stage of evolution includes a form of fixed self-play by introducing the best performing network from the first evolution stage as an opponent, plus one either playing a move suggested the newly introduced self-play-agent or a random move adding to the variance of game states to which the agent is exposed. The list of opponents gets completed by The AvoidNextLoss agent as well the already used randomly playing agent. The fitness function to evaluate performances now has the networks to evaluate face each of those four agents four times per iteration, otherwise it is similar to the first evolution stage. Other than that, the threshold to species-assignment is lowered as the initial variance of the population is expected to decrease after the first evolution series.

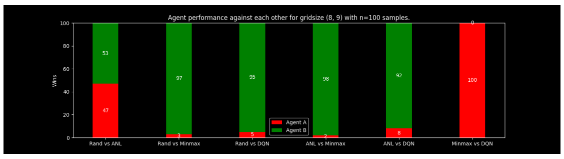
4 Results

4.1 Comparison of agents

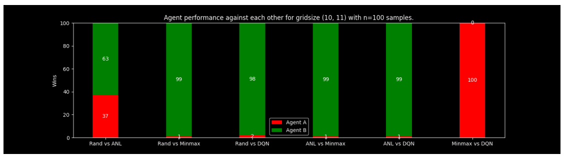
Our different agents competed in playing connect-4 against each other on boards of sizes 6x7, 8x9, 10x11, and 12x13, with the exception of the neuroevolution agent (NEAT Agent) which was only concepted for a board size of 6x7. Additionally, a random agent and an agent that just prohibited losing by the opponent's next move were implemented and included in the performance evaluation. Each match of two agents was performed 100 times. Figure 1 shows the outcome of the competition.



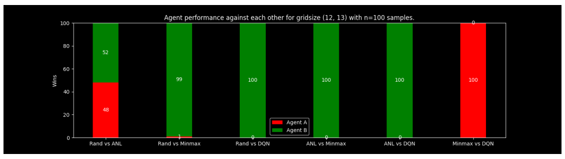
(a) 6x7 Grid Size



(b) 8x9 Grid Size



(c) 10x11 Grid Size



(d) 12x13 Grid Size

Figure 1: Average Return of the DQN model on different grid sizes

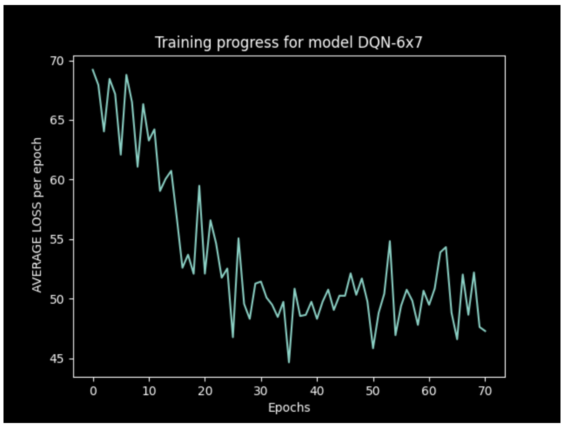
The Minimax agent (Minmax) was the most successful agent on all grid sizes. On the basic 6x7 grid, the Neuroevolution agent (NEAT) achieves a draw in direct confrontation with the Minimax agent, but has worse performances against the other agents. The DQN agent (DQN) outperforms the Random agent (Rand) as well as the Avoid Next Loss agent

(ANL) but doesn't achieve a win in the games played against the Neuroevolution agent. While both outperform Random and ANL agents, the DQN interestingly performs better against both than the NEAT agent. Also note that the Random agent surprisingly performs in a close draw when playing against ANL.

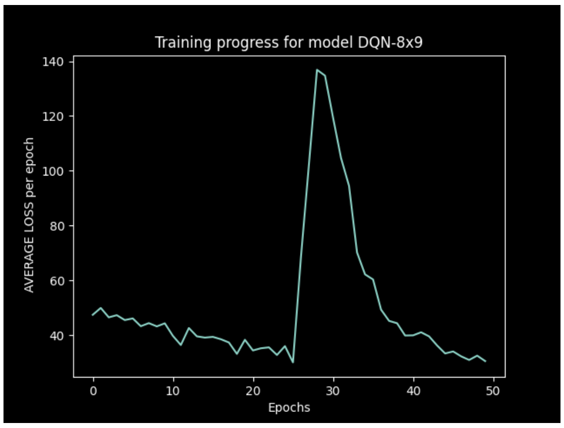
During the games played on the larger grids, the Neuroevolution agent doesn't take part in the competition, the remaining agents perform tendencially as they did on the basic grid. A closer look has to be taken on the DQN agent, which significantly increases the winning rates against both Random and ANL agents, even winning every match against those on the 12x13 grid. The Minimax agent also further increases its already high performance against all other agents as the grids get larger.

4.2 Training progression

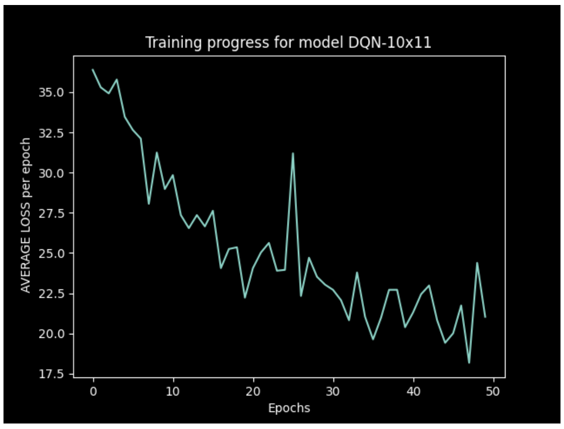
4.2.1 Deep Q-Network



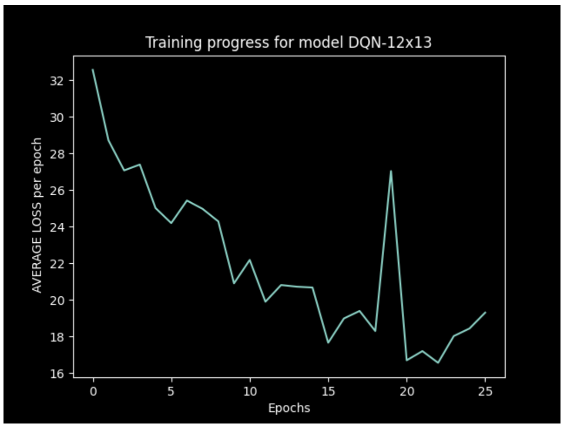
(a) 6x7 Grid Size



(b) 8x9 Grid Size



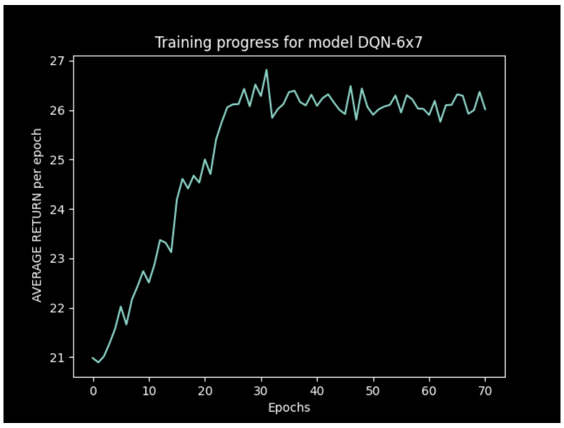
(c) 10x11 Grid Size



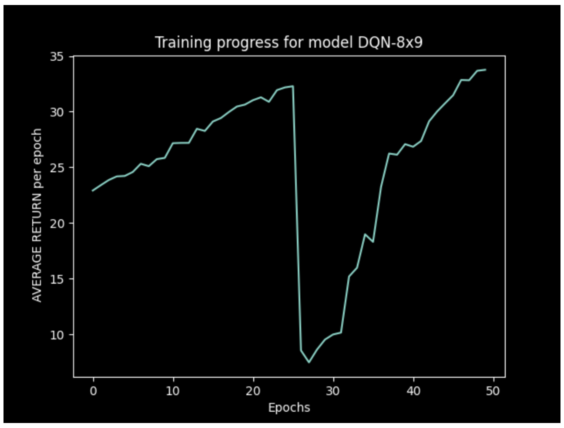
(d) 12x13 Grid Size

Figure 2: Average Loss of the DQN model on different grid sizes

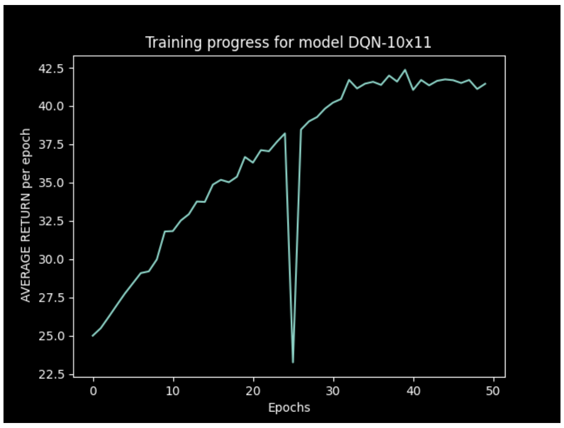
To get a measure on training progression for the Deep Q-Network, we track the network's averaged loss for each training episode. The average return is tracked as well, though it isn't as much an appropriate measure. Since the network is trained by self-play it is always facing an equally strong opponent in itself. Nevertheless, we included a small reward for every time an action is taken and thus a move is made in the game. Due to that, higher rewarded games in self-play correlate with those games lasting longer. From personal experience, we can assume that with two strong opponents, a game will last until most of the board is filled with checkers, with two beginners, games tend to not last as long.



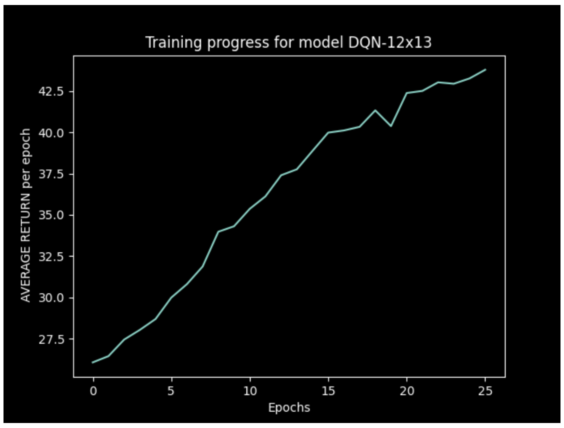
(a) 6x7 Grid Size



(b) 8x9 Grid Size



(c) 10x11 Grid Size



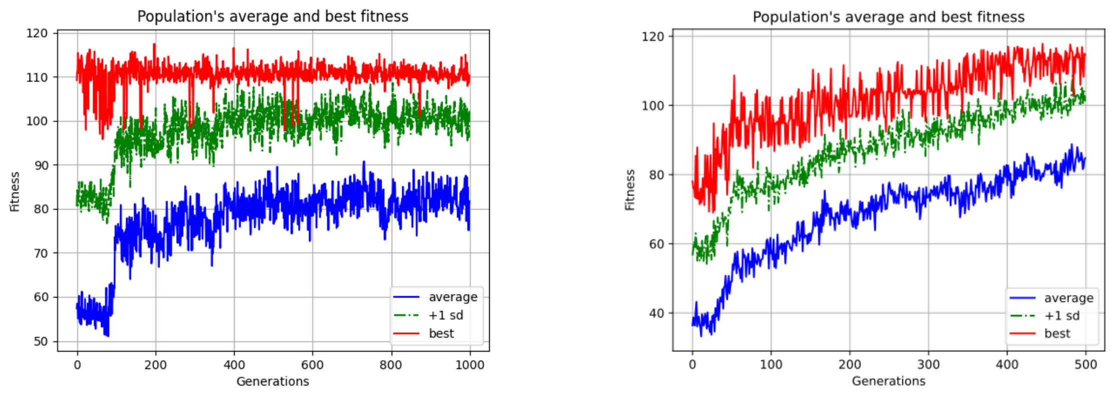
(d) 12x13 Grid Size

Figure 3: Average Return of the DQN model on different grid sizes

Figure 3 and Figure 2 show progressions of loss and return during training for different grid sizes. While there is quite a bit of oscillation in the loss value progression, the trend goes downwards with all board sizes, indicating agent improvement. The loss improvements slowed down during training and eventually came to a halt. Note that the heavy outliers during training on the three bigger grids in roughly the middle/third quarter are due to the mentioned fact that training had to be restarted. While we were being able to use the halfway trained weights to continue with, the Experience Replay Buffer restarted empty.

The average episode return correlates with the trend of the loss, albeit as a negative correlation. Interestingly, the oscillations of the loss are mirrored in a much smaller fashion by the return, resulting in smooth progression.

4.2.2 Neuroevolution



(a) 1st evolution stage   **(b)** 2nd evolution stage

Figure 4: Fitness progression during NEAT evolution

Figure 4 shows the fitness progressions during neuroevolution. On the left side the course of the first stage of evolution is shown, the right shows it during the second stage. The red graphs indicates the fitness of the iteration (generation)'s best performing network, the blue ones show the population's average fitness. The fitness equals the averaged return of a game as defined by the environment. To be able to gauge the population's variation, the standard deviation added to the population's average fitness is presented by green graphs.

During both evolution stages, general upward trends in the different fitness measures can be observed. Note that the fitness values shown in the graphs represent the population after selection, resulting in positive rewards.

While playing against a random agent in the first stage, the best performing network is already winning most of the games (for interpretation of the values as an averaged game return, see the reward setting of the environment). The confidence in securely beating the random agent increases during this stage of evolution, most other oscillations of the red graph are due to the step-rewards from games of differing length. A prominent feature is the steep fitness gain in the average population's fitness. This is happening because of the only briefly mentioned speciation feature in combination with the initially large size of the population. A large population will lead to a high initial variance in genomes, resulting in many different species. Species have a stagnation criterion, which leads to extinction of the species if for a certain number of generations, fitness of the species' best performing genome isn't improved. This results in a large number of the population's initial species dying out at the same time.

The graphs are mostly stagnating after around 500 or 600 generations, so the population is ready to go through the second stage of evolution. The focus is now on the right part of the figure. Here, maximum fitness and average fitness correlate more obvious with each other. The fitness values start significantly lower than they did for the first evolution stage, which is the result of the changed fitness function that now includes among others the best performing agent of the first evolution stage. A continuous increase in the population's average fitness as well as its maximum fitness can be observed. Whilst increase is slowing, stagnation doesn't happen until the end of the evolution stage, rendering the decision to only let evolution run for 500 generations a regrettable one.

Nevertheless, when facing the other agents, the Neuroevolution agent shows considerable performance. Interestingly, when competing against the Random agent, the winning rate is quite a bit lower than what would be expected, at least after completion of the first stage of evolution. Playing against the wider range of opponents will have contributed to the agent letting go of some of its previously learned strategies.

5 Conclusion

In this project, we have implemented three different reinforcement learning methods, DQN, Minimax and NEAT to play the game Connect-4 and evaluated their performances against each other and two additional agents, random and ANL. Apart from the previous literature, we have evaluated the performance of the agents in different grid sizes of the game. We have seen that on the regular size the Minimax and the NEAT performs equally well, however, in the other settings, the Minimax performs the best against DQN, random agent, and the avoid-loss agent.
With NEAT being a somewhat curious technique from a Deep Reinforcement Learning's point of view, we were surprised positively about its outcome in regards to the project. Yet, it is in its core a Reinforcement Learning technique as well.

This project does not only reflect the acquired knowledge from the contents of the Deep Reinforcement Learning course, but also goes further and explores additional topics. To actualize this project, we used our knowledge on Markov-Decision Processes (MDPs) to define Connect-4 as an MDP to progress with the building of the environment, and on Q-Learning to model a network to play Connect-4. For the enrichment of the skills, two more agents making use of Minimax and Neuroevolution were added to the project.

References

[1] Oleg Arenz. “Monte carlo chess”. In: (2012).

[2] Mayank Dabas, Nishthavan Dahiya, and Pratish Pushparaj. “Solving Connect 4 Using Artificial Intelligence”. In: *International Conference on Innovative Computing and Communications*. Ed. by Ashish Khanna et al. Singapore: Springer Singapore, 2022, pp. 727–735. ISBN: 978-981-16-2594-7.

[3] Imran Ghory. “Reinforcement learning in board games”. In: *Department of Computer Science, University of Bristol, Tech. Rep* 105 (2004).

[4] *Make your own custom environment*. URL: `https://www.gymlibrary.dev/content/environment_creation/`.

[5] Alan McIntyre et al. *neat-python*. URL: `https://github.com/CodeReclaimers/neat-python`.

[6] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: `10.1162/106365602320169811`.

[7] *the original game of connect* $4_2$009. 2009.

[8] Markus Thill, Patrick Koch, and Wolfgang Konen. “Reinforcement Learning with N-tuples on the Game Connect-4”. In: *Parallel Problem Solving from Nature - PPSN XII*. Ed. by Carlos A. Coello Coello et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 184–194. ISBN: 978-3-642-32937-1.

[9] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. “Monte-Carlo Tree Search Solver”. In: *Computers and Games*. Ed. by H. Jaap van den Herik et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 25–36.

[10] Konstantia Xenou, Georgios Chalkiadakis, and Stergos Afantenos. “Deep reinforcement learning in strategic board game environments”. In: *European Conference on Multi-Agent Systems*. Springer. 2018, pp. 233–248.